中国科学技术大学
University of Science and Technology of China

2.2 程序的编译过程

王超

中国科学技术大学计算机学院
2023年春

GTC 2022-2023

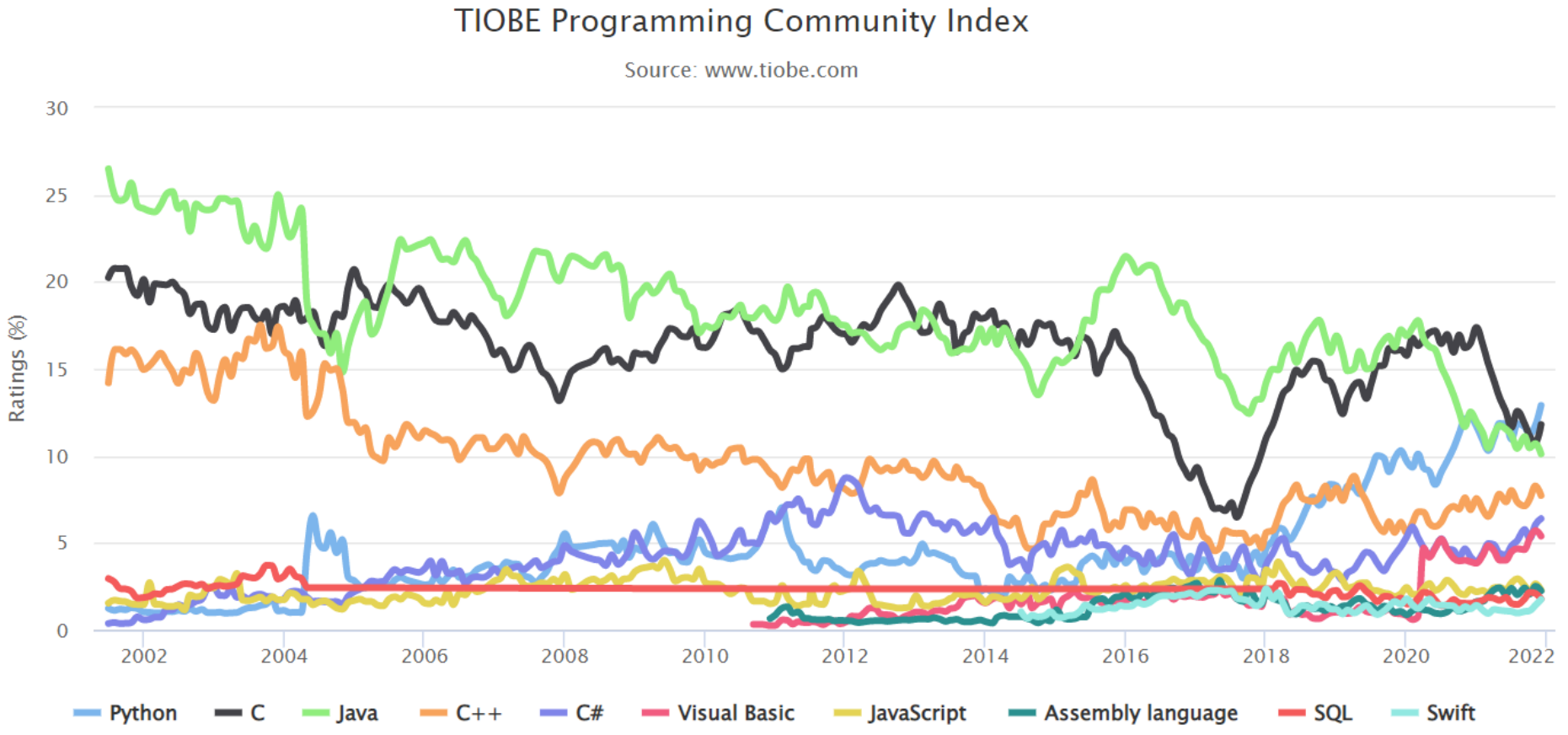
https://www.nvidia.cn/gtc-global/keynote/

硬件、软件的发展

- 包括硬件的迭代、编程模型和编程语言的发展

NVIDIA Accelerator Specification Comparison			
	H100	A100 (80GB)	V100
FP32 CUDA Cores	16896	6912	5120
Tensor Cores	528	432	640
Boost Clock	~1.78GHz (Not Finalized)	1.41GHz	1.53GHz
Memory Clock	4.8Gbps HBM3	3.2Gbps HBM2e	1.75Gbps HBM2
Memory Bus Width	5120-bit	5120-bit	4096-bit
Memory Bandwidth	3TB/sec	2TB/sec	900GB/sec
VRAM	80GB	80GB	16GB/32GB
FP32 Vector	60 TFLOPS	19.5 TFLOPS	15.7 TFLOPS
FP64 Vector	30 TFLOPS	9.7 TFLOPS (1/2 FP32 rate)	7.8 TFLOPS (1/2 FP32 rate)
INT8 Tensor	2000 TOPS	624 TOPS	N/A
FP16 Tensor	1000 TFLOPS	312 TFLOPS	125 TFLOPS
TF32 Tensor	500 TFLOPS	156 TFLOPS	N/A
FP64 Tensor	60 TFLOPS	19.5 TFLOPS	N/A
Interconnect	NVLink 4 18 Links (900GB/sec)	NVLink 3 12 Links (600GB/sec)	NVLink 2 6 Links (300GB/sec)
GPU	GH100 (814mm2)	GA100 (826mm2)	GV100 (815mm2)
Transistor Count	80B	54.2B	21.1B
TDP	700W	400W	300W/350W
Manufacturing Process	TSMC 4N	TSMC 7N	TSMC 12nm FFN
Interface	SXM5	SXM4	SXM2/SXM3
Architecture	Hopper	Ampere	Volta

编程语言指数走势(2002-2022)



https://www.tiobe.com/

编程语言的发展趋势

中国科学技术大学 University of Science and Technology of China

TIOBE Index for December 2021

December Headline: Will C# become the programming language of 2021?

Next month, the TIOBE programming language of the year will be announced. This award is given to the programming language that has had the highes increase in ratings in 2021. At the moment, C# is by far the most likely candidate for this title. It is interesting to note that C# has never won the "TIOBE index programming language of the year award" during its 21 years of existence, although it has been in the top 10 for the last 2 decades. Let's see what happens next month! Other interesting moves in the TIOBE index this month are Swift (from #14 to #10), R (from #15 to #11), and Kotlin (from #33 to #26). *-- Paul Jansen CEO TIOBE Software*

The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found here.

Dec 2021	Dec 2020	Change	Programming Language	Ratings	Change
1	3	↑	Python	12.90%	+0.69%
2	1	↓	C	11.80%	-4.69%
3	2	↓	Java	10.12%	-2.41%
4	4		C++	7.73%	+0.82%
5	5		C#	6.40%	+2.21%
6	6		Visual Basic	5.40%	+1.48%
7	7		JavaScript	2.30%	-0.06%
8	12	⇈	Assembly language	2.25%	+0.91%
9	10	↑	SQL	1.79%	+0.26%
10	13	↑	Swift	1.76%	+0.54%
11	9	↓	R	1.58%	-0.01%
12	8	⇊	PHP	1.50%	-0.62%
13	23	⇈	Classic Visual Basic	1.27%	+0.56%
14	11	↓	Groovy	1.23%	-0.30%
15	15		Ruby	1.16%	-0.01%
16	18	↑	Delphi/Object Pascal	1.14%	+0.27%
17	32	⇈	Fortran	1.04%	+0.59%
18	14	⇊	Perl	0.96%	-0.24%
19	16	↓	Go	0.95%	-0.19%
20	17	↓	MATLAB	0.92%	-0.18%

Programming Language Hall of Fame

The hall of fame listing all "Programming Language of the Year" award winners is show has the highest rise in ratings in a year.

Year	Winner
2020	Python
2019	C
2018	Python
2017	C
2016	Go
2015	Java
2014	JavaScript
2013	Transact-SQL
2012	Objective-C
2011	Objective-C
2010	Python
2009	Go
2008	C
2007	Python
2006	Ruby
2005	Java
2004	PHP
2003	C++

编程语言受欢迎情况

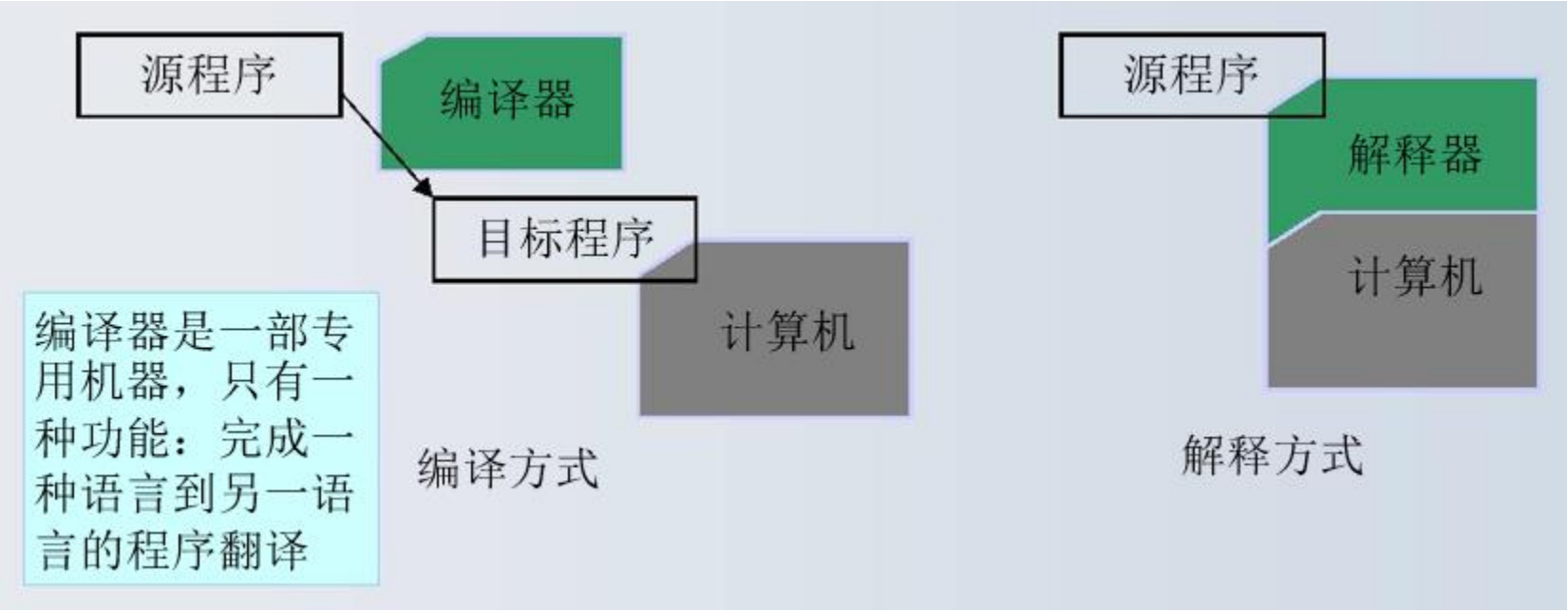
Very Long Term History

To see the bigger picture, please find below the positions of the top 10 programming languages of many years back. Please note that these are *average* positions for a period of 12 months.

Programming Language	2021	2016	2011	2006	2001	1996	1991	1986
C	1	2	2	2	1	1	1	1
Python	2	5	7	8	20	27	-	-
Java	3	1	1	1	2	15	-	-
C++	4	3	3	3	3	2	2	5
C#	5	4	4	7	11	-	-	-
Visual Basic	6	13	-	-	-	-	-	-
JavaScript	7	7	10	10	9	20	-	-
PHP	8	6	5	5	10	-	-	-
Assembly language	9	10	-	-	-	-	-	-
SQL	10	-	-	-	35	-	-	-
Fortran	20	26	29	21	25	7	3	8
Ada	28	30	17	17	18	11	4	2
Lisp	32	27	13	13	16	8	7	3
(Visual) Basic	-	-	8	4	4	3	8	6

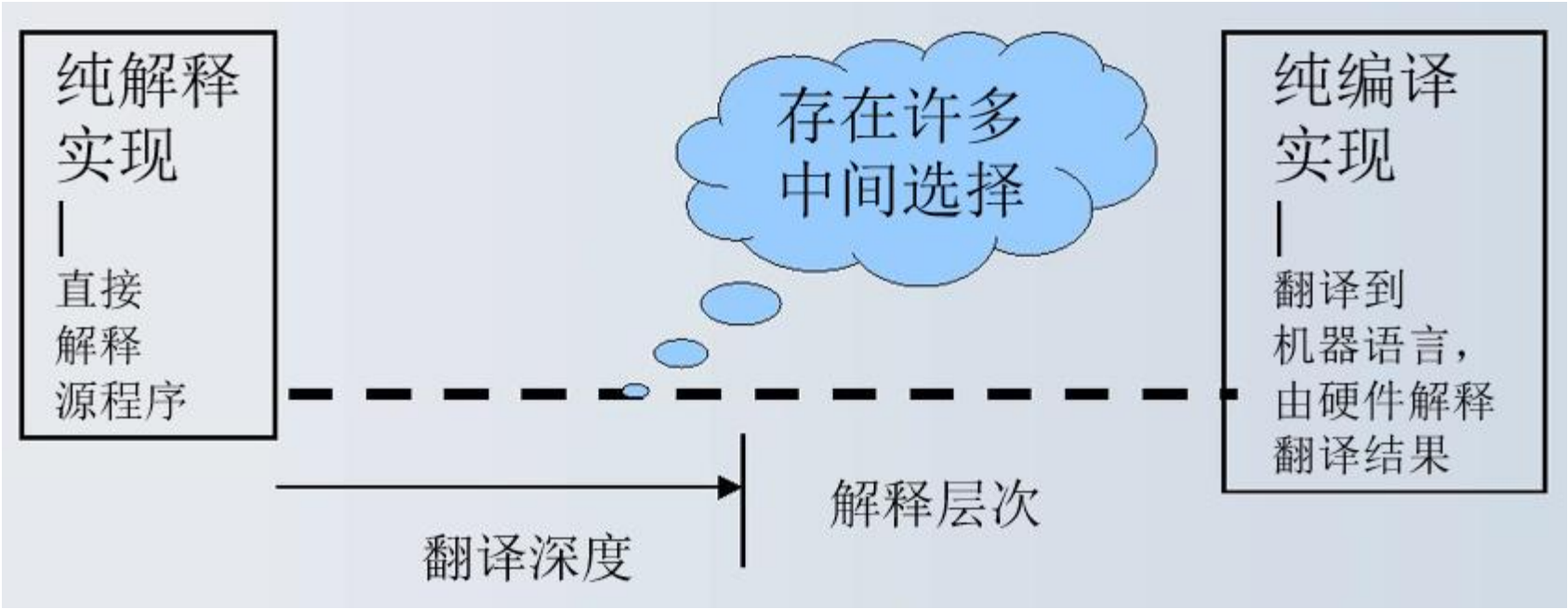
编程语言实现的方式

- （高级）语言的实现有两种方式，编译和解释。
  - ✓编译：把源程序编译为机器语言目标程序后执行
  - ✓**解释：在目标机器上实现一个源语言的解释器，由这个解释器直接解释执行源语言程序**



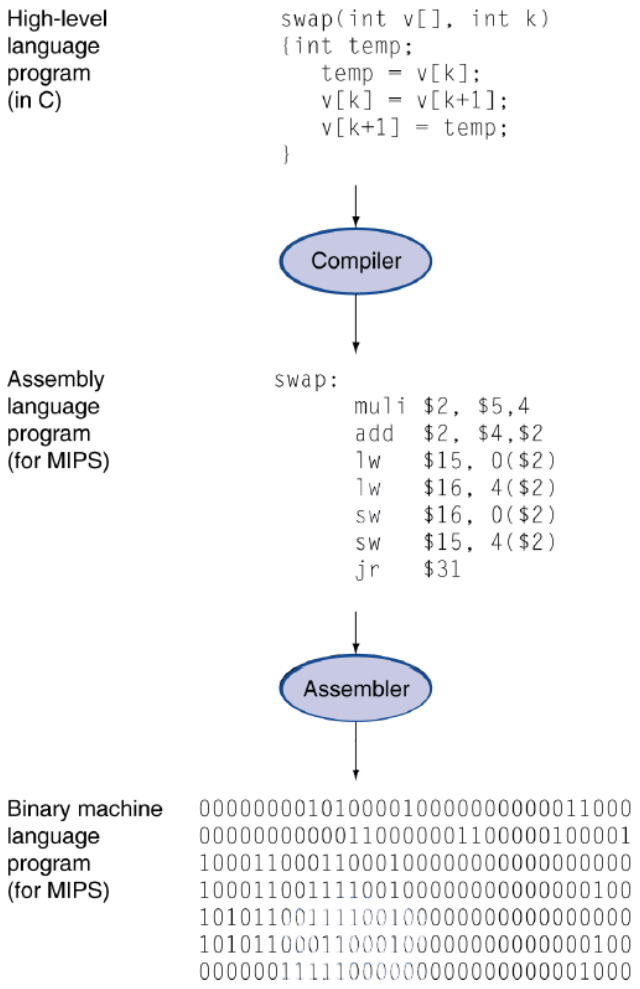
编程语言实现的方式

- 语言实现方式多种多样，纯粹的编译或纯粹的解释只是两个极端
  - ✓ **C、Fortran** 语言的常见实现方式可以认为是比较纯粹的编译方式
  - ✓ 早期的 BASIC，DOS 的 bat 文件，现在有些脚本语言实现，采用的基本上是纯粹的解释方式
  - ✓ Java等中间层次语言（字节码、语法树等形式）

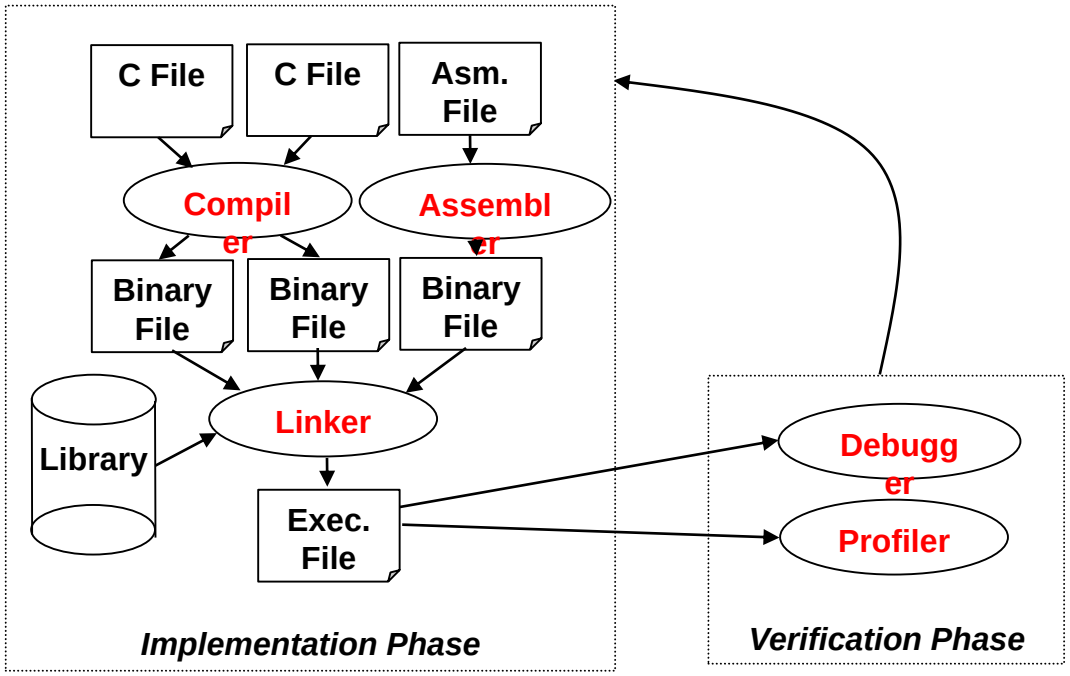


编译：High Level to Assembly

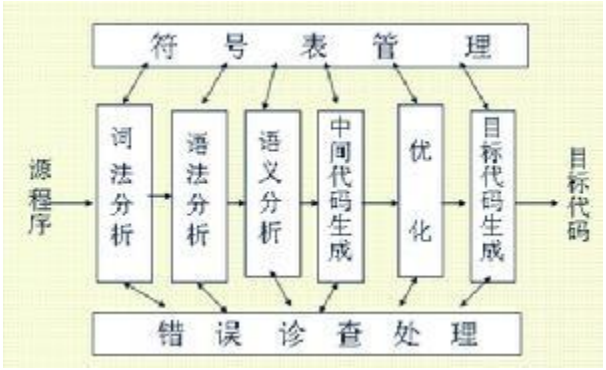
- High Level Lang (C, C++, Java, etc.)
  - ✓ Statements
  - ✓ Variables
  - ✓ Operators
  - ✓ Methods, functions, procedures
- Assembly Language
  - ✓ Instructions
  - ✓ Registers
  - ✓ Memory



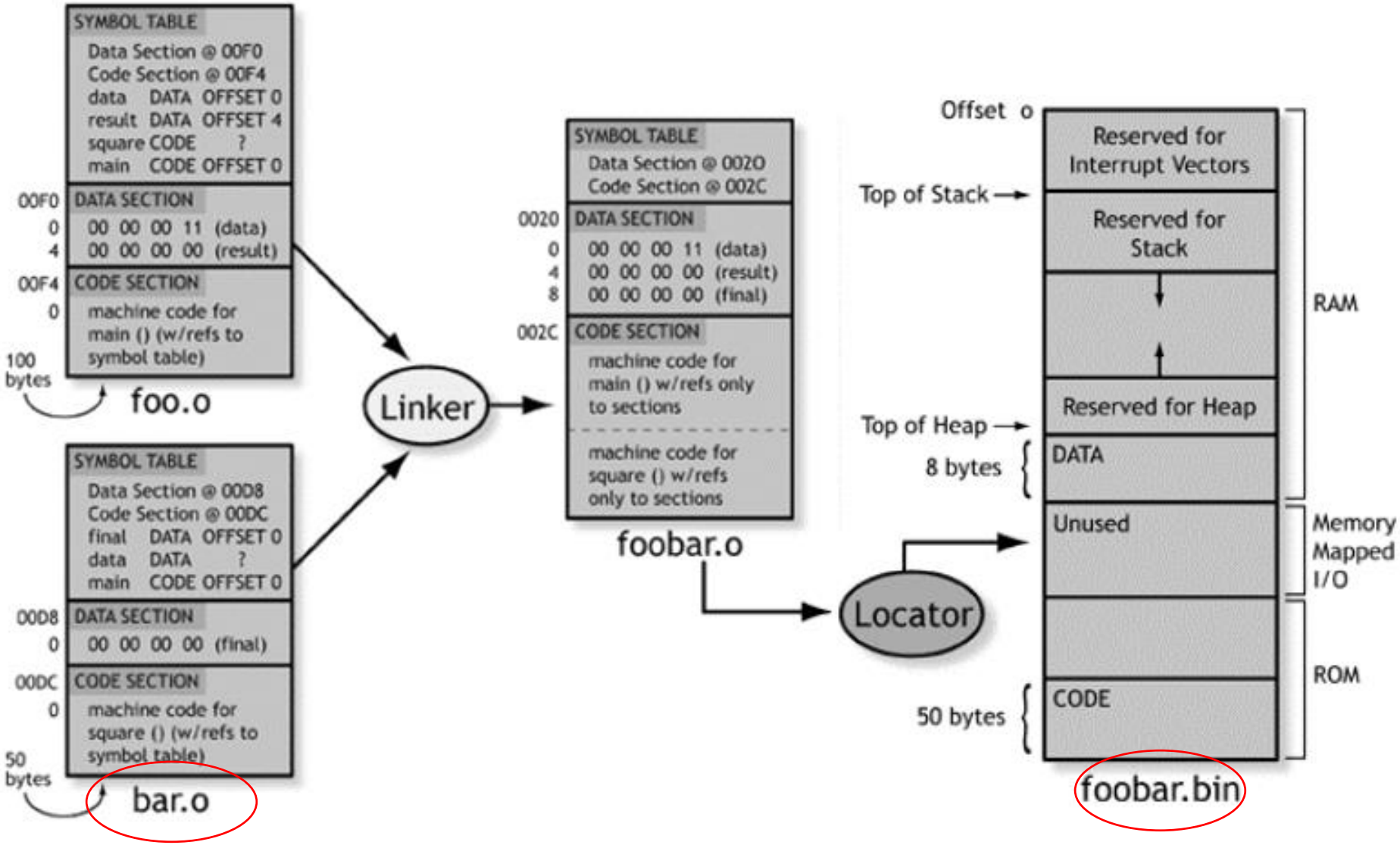
程序开发流程
Program Development Process



- *Implementation Phase*
  - Editor
  - Compilers
    - Cross compiler
      - Runs on one processor, but generates code for another
  - Assemblers
  - Linkers
- *Verification Phase*
  - Debuggers
  - Profilers



链接与重定位


SYMBOL TABLE
Data Section @ 00F0
Code Section @ 00F4
data DATA OFFSET 0
result DATA OFFSET 4
square CODE ?
main CODE OFFSET 0
00F0
0
4
DATA SECTION
00 00 00 11 (data)
00 00 00 00 (result)
00F4
0
CODE SECTION
machine code for main () (w/refs to symbol table)
100 bytes
foo.o
SYMBOL TABLE
Data Section @ 00D8
Code Section @ 00DC
final DATA OFFSET 0
data DATA ?
main CODE OFFSET 0
00D8
0
DATA SECTION
00 00 00 00 (final)
00DC
0
CODE SECTION
machine code for square () (w/refs to symbol table)
50 bytes
bar.o
Linker
SYMBOL TABLE
Data Section @ 0020
Code Section @ 002C
0020
0
4
8
DATA SECTION
00 00 00 11 (data)
00 00 00 00 (result)
00 00 00 00 (final)
002C
CODE SECTION
machine code for main () w/refs only to sections
machine code for square () w/refs only to sections
foobar.o
Locator
Offset o
Top of Stack
Top of Heap
8 bytes
50 bytes
Reserved for Interrupt Vectors
Reserved for Stack
Reserved for Heap
DATA
Unused
CODE
RAM
Memory Mapped I/O
ROM
foobar.bin

中国科学技术大学
University of Science and Technology of China

例子：程序采用GCC编译的过程

王超

中国科学技术大学计算机学院
2023年春

程序编译的过程=>GCC例子

- GCC=一个C编译器?
  其实GCC = GNU Compiler Collection
- 目前，GCC可以支持多种高级语言，如
  - ✓C、C++
  - ✓ADA
  - ✓Object C
  - ✓JAVA
  - ✓Fortran
  - ✓PASCAL

GCC下的工具

- cpp — 预处理器
  GNU C编译器在编译前自动使用cpp对用户程序进行预处理
- gcc — 符合ISO等标准的C编译器
- g++ — 基本符合ISO标准的C++编译器
- gcj — GCC的java前端
- gnat — GCC的GNU ADA 95前端

GNU Tools—gcc

- gcc是一个强大的工具集合，它包含了预处理器、编译器、汇编器、链接器等组件。它会在需要的时候调用其他组件。
  输入文件的类型和传递给gcc的参数决定了gcc调用具体的哪些组件。
- 对于开发者，它提供的足够多的参数，可以让开发者全面控制代码的生成，这对嵌入式系统级的软件开发非常重要

  gcc --help

Linux


Mandriva


xandros


CentOS


slackware


debian


suse


KNOPPIX


ubuntu


FreeBSD


gentoo linux


MEPIS

gcc使用举例（1）源程序

emacs - testsse.c

File Edit Options Buffers Tools C Cscope Help

```
//testsse.c

#include <stdio.h>

int main()
{
  int i,j;
  i=0;
  j=0;
  i=j+1;
  printf("Hello SSE");
  printf("i=j+1=%d\n",i);
}
```

gcc使用举例（2）编译和运行


```
[root@stc-38fadca7df dir]# ls
testsse.c
[root@stc-38fadca7df dir]# gcc -o testsse testsse.c
[root@stc-38fadca7df dir]# ls
testsse  testsse.c
[root@stc-38fadca7df dir]# ./testsse
Hello SSEi=j+1=1
[root@stc-38fadca7df dir]#
```

编译

运行

gcc的编译过程

- 一般情况下，c程序的编译过程为
  1、预处理
  2、编译成汇编代码
  3、汇编成目标代码
  4、链接

1、预处理

- 预处理：使用-E参数
  输出文件的后缀为“.cpp”

```
gcc –E –o gcctest.cpp gcctest.c
```

- 使用wc命令比较预处理后的文件与源文件，可以看到两个文件的差异

```
[root@stc-38fadca7df dir]# gcc -E -o testsse.cpp testsse.c
[root@stc-38fadca7df dir]# ls
testsse.c  testsse.cpp
[root@stc-38fadca7df dir]# wc testsse.c testsse.cpp
   13     15    131 testsse.c
  943   2128 18053 testsse.cpp
  956   2143 18184 总计
```

预处理

行数 单词数 字节数

emacs - testsse.c

File Edit Options Buffers Tools C Cscope Help

```c
//testsse.c

#include <stdio.h>

int main()
{
  int i,j;
  i=0;
  j=0;
  i=j+1;
  printf("Hello SSE");
  printf("i=j+1=%d\n",i);
}
```

emacs - testsse.cpp

File Edit Options Buffers Tools C++ Cscope Help

```
# 134 "/usr/include/bits/types.h" 3 4
# 1 "/usr/include/bits/typesizes.h" 1 3 4
# 135 "/usr/include/bits/types.h" 2 3 4


__extension__ typedef __u_quad_t __dev_t;
__extension__ typedef unsigned int __uid_t;
__extension__ typedef unsigned int __gid_t;
__extension__ typedef unsigned long int __ino_t;
__extension__ typedef __u_quad_t __ino64_t;
__extension__ typedef unsigned int __mode_t;
__extension__ typedef unsigned int __nlink_t;
__extension__ typedef long int __off_t;
__extension__ typedef __quad_t __off64_t;
__extension__ typedef int __pid_t;
__extension__ typedef struct { int __val[2]; } __fsid_t;
__extension__ typedef long int __clock_t;
__extension__ typedef unsigned long int __rlim_t;
__extension__ typedef __u_quad_t __rlim64_t;
__extension__ typedef unsigned int __id_t;
__extension__ typedef long int __time_t;
__extension__ typedef unsigned int __useconds_t;
__extension__ typedef long int __suseconds_t;

__extension__ typedef int __daddr_t;
__extension__ typedef long int __swblk_t;
__extension__ typedef int __key_t;
```

--:-- testsse.cpp 1:16下午 0.30 (C++ Abbrev)--L76--C0--

2、编译成汇编代码

- 预处理文件→汇编代码

  1）使用-x参数说明根据指定的步骤进行工作，cpp-output指明从预处理得到的文件开始编译

  2）使用-S说明生成汇编代码后停止工作

```
gcc -x cpp-output -S -o gcctest.s gcctest.cpp
```

- 也可以直接编译到汇编代码

```
gcc -S gcctest.c
```

```
[root@stc-38fadca7df dir]# gcc -x c
[root@stc-38fadca7df dir]# ls
testsse.c  testsse.cpp  testsse.s
[root@stc-38fadca7df dir]#
```

emacs - testsse.c

File Edit Options Buffers Tools C Cscope Help

```c
//testsse.c

#include <stdio.h>

int main()
{
  int i,j;
  i=0;
  j=0;
  i=j+1;
  printf("Hello SSE");
  printf("i=j+1=%d\n",i);
}
```

dir - 文件浏

文件(F) 编辑(E) 查看(V) 转到(G) 书签(B) 帮助(H)

emacs - testsse.s

File Edit Options Buffers Tools Help

```
        .file   "testsse.c"
        .section        .rodata
.LC0:
        .string "Hello SSE"
.LC1:
        .string "i=j+1=%d\n"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $36, %esp
        movl    $0, -12(%ebp)
        movl    $0, -8(%ebp)
        movl    -8(%ebp), %eax
        addl    $1, %eax
        movl    %eax, -12(%ebp)
        movl    $.LC0, (%esp)
        call    printf
        movl    -12(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC1, (%esp)
        call    printf
```

--:-- testsse.s 1:18下午 0.11 (Assembler)--L1--C0

3、编译成目标代码

- 汇编代码→目标代码

  gcc –x assembler –c gcctest.s

- 直接编译成目标代码

  gcc –c gcctest.c

- 使用汇编器生成目标代码

  as –o gcctest.o gcctest.s

```
[donger@donger gcctest]$ ls
gcctest.c  gcctest.s
[donger@donger gcctest]$ gcc -x assembler -c gcctest.s
[donger@donger gcctest]$ ls
gcctest.c  gcctest.o  gcctest.s
[donger@donger gcctest]$
```

汇编代码→目标代码

```
[donger@donger gcctest]$ ls
gcctest.c
[donger@donger gcctest]$ gcc -c gcctest.c
[donger@donger gcctest]$ ls
gcctest.c  gcctest.o
[donger@donger gcctest]$
```

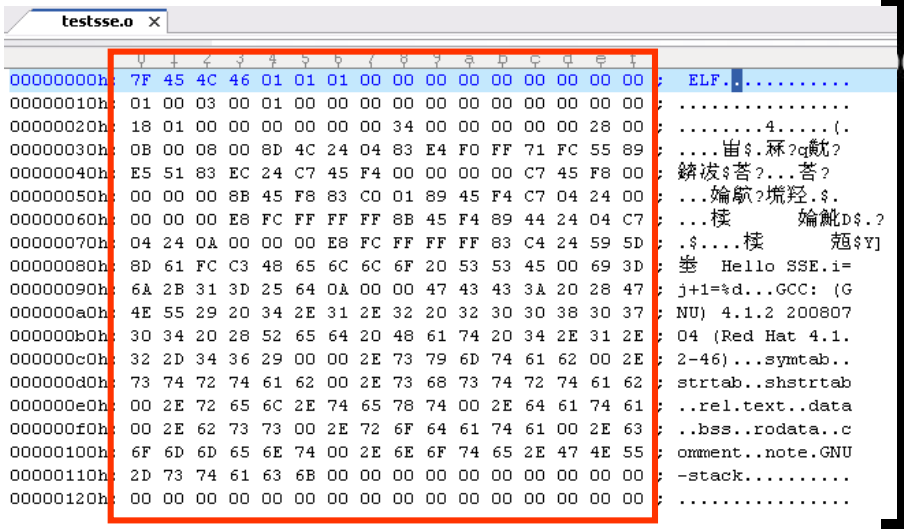
直接编译成目标代码

```
[donger@donger gcctest]$ ls
gcctest.c  gcctest.s
[donger@donger gcctest]$ as -o gcctest.o gcctest.s
[donger@donger gcctest]$ ls
gcctest.c  gcctest.o  gcctest.s
[donger@donger gcctest]$
```

使用汇编器

testsse.o

- UltraEdit



- Objdump

```
[root@stc-38fadca7df dir]# objdump -d testsse.o

testsse.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  0xfffffffc(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   83 ec 24                sub    $0x24,%esp
  11:   c7 45 f4 00 00 00 00    movl   $0x0,0xfffffff4(%ebp)
  18:   c7 45 f8 00 00 00 00    movl   $0x0,0xfffffff8(%ebp)
  1f:   8b 45 f8                mov    0xfffffff8(%ebp),%eax
  22:   83 c0 01                add    $0x1,%eax
  25:   89 45 f4                mov    %eax,0xfffffff4(%ebp)
  28:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  2f:   e8 fc ff ff ff          call   30 <main+0x30>
  34:   8b 45 f4                mov    0xfffffff4(%ebp),%eax
  37:   89 44 24 04             mov    %eax,0x4(%esp)
  3b:   c7 04 24 0a 00 00 00    movl   $0xa,(%esp)
  42:   e8 fc ff ff ff          call   43 <main+0x43>
  47:   83 c4 24                add    $0x24,%esp
  4a:   59                      pop    %ecx
```

4、编译成执行代码

- 目标代码→执行代码

  gcc –o gcctest gcctest.o

- 直接生成执行代码

  gcc –o gcctest gcctest.c

```
[donger@donger gcctest]$ ls
gcctest.c  gcctest.o
[donger@donger gcctest]$ gcc -o gcctest gcctest.o
[donger@donger gcctest]$ ls
gcctest  gcctest.c  gcctest.o
[donger@donger gcctest]$
```

目标代码→执行代码

```
[donger@donger gcctest]$ ls
gcctest.c
[donger@donger gcctest]$ gcc -o gcctest gcctest.c
[donger@donger gcctest]$ ls
gcctest  gcctest.c
[donger@donger gcctest]$
```

直接生成执行代码

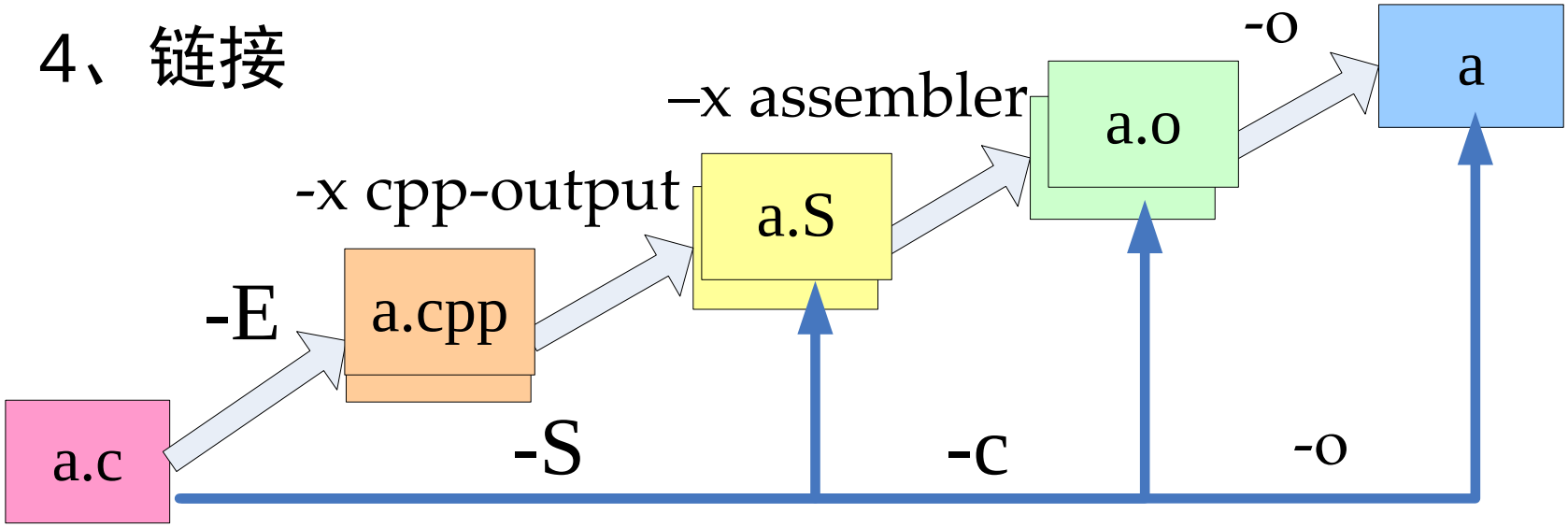
testsse.o V.S. testsse

```
[root@stc-38fadca7df dir]# objdump -d testsse.o

testsse.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  0xfffffffc(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   83 ec 24                sub    $0x24,%esp
  11:   c7 45 f4 00 00 00 00    movl   $0x0,0xfffffff4(%ebp)
  18:   c7 45 f8 00 00 00 00    movl   $0x0,0xfffffff8(%ebp)
  1f:   8b 45 f8                mov    0xfffffff8(%ebp),%eax
  22:   83 c0 01                add    $0x1,%eax
  25:   89 45 f4                mov    %eax,0xfffffff4(%ebp)
  28:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  2f:   e8 fc ff ff ff          call   30 <main+0x30>
  34:   8b 45 f4                mov    0xfffffff4(%ebp),%eax
  37:   89 44 24 04             mov    %eax,0x4(%esp)
  3b:   c7 04 24 0a 00 00 00    movl   $0xa,(%esp)
  42:   e8 fc ff ff ff          call   43 <main+0x43>
  47:   83 c4 24                add    $0x24,%esp
  4a:   59                      pop    %ecx
```

```
testsse:     file format elf32-i386

Disassembly of section .init:

08048250 <_init>:
 8048250: 55                    push   %ebp
 8048251: 89 e5                 mov    %esp,%ebp
 8048253: 83 ec 08              sub    $0x8,%esp
 8048256: e8 79 00 00 00        call   80482d4 <call_gmon_start>
 804825b: e8 00 01 00 00        call   8048360 <frame_dummy>
 8048260: e8 fb 01 00 00        call   8048460 <__do_global_ctors_aux>
 8048265: c9                    leave
 8048266: c3                    ret
Disassembly of section .plt:

080482b0 <_start>:
 80482b0: 31 ed                 xor    %ebp,%ebp
 80482b2: 5e                    pop    %esi
 80482b3: 89 e1                 mov    %esp,%ecx
 80482b5: 83 e4 f0              and    $0xfffffff0,%esp
 80482b8: 50                    push   %eax
 80482b9: 54                    push   %esp
 80482ba: 52                    push   %edx
 80482bb: 68 e0 83 04 08        push   $0x80483e0
 80482c0: 68 f0 83 04 08        push   $0x80483f0
 80482c5: 51                    push   %ecx
 80482c6: 56                    push   %esi
 80482c7: 68 84 83 04 08        push   $0x8048384
 80482cc: e8 b7 ff ff ff        call   8048288 <__libc_start_main@plt>
 80482d1: f4                    hlt
 80482d2: 90                    nop
 80482d3: 90                    nop

08048384 <main>:
 8048384: 8d 4c 24 04           lea    0x4(%esp),%ecx
 8048388: 83 e4 f0              and    $0xfffffff0,%esp
 804838b: ff 71 fc              pushl  0xfffffffc(%ecx)
 804838e: 55                    push   %ebp
 804838f: 89 e5                 mov    %esp,%ebp
 8048391: 51                    push   %ecx
 8048392: 83 ec 24              sub    $0x24,%esp
 8048395: c7 45 f4 00 00 00 00  movl   $0x0,0xfffffff4(%ebp)
 804839c: c7 45 f8 00 00 00 00  movl   $0x0,0xfffffff8(%ebp)
 80483a3: 8b 45 f8              mov    0xfffffff8(%ebp),%eax
 80483a6: 83 c0 01              add    $0x1,%eax
 80483a9: 89 45 f4              mov    %eax,0xfffffff4(%ebp)
 80483ac: c7 04 24 b0 84 04 08  movl   $0x80484b0,(%esp)
 80483b3: e8 e0 fe ff ff        call   8048298 <printf@plt>
 80483b8: 8b 45 f4              mov    0xfffffff4(%ebp),%eax
 80483bb: 89 44 24 04           mov    %eax,0x4(%esp)
 80483bf: c7 04 24 ba 84 04 08  movl   $0x80484ba,(%esp)
```

总结

- 程序的编译->执行过程
  1、预处理
  2、编译成汇编代码
  3、汇编成目标代码
  4、链接



gcc的高级选项

- -Wall：打开所有的警告信息

```
[donger@donger gcctest]$ ls
gcctest.c
[donger@donger gcctest]$ gcc -Wall -o gcctest gcctest.c
gcctest.c: 在函数‘main’中：
gcctest.c:13: 警告：在有返回值的函数中，控制流程到达函数尾
[donger@donger gcctest]$
```

根据警告信息检查源程序


```c
//gcctest.c

#include <stdio.h>

int main()
{
        int i,j;
        i=0;
        j=0;
        i=j+1;
        printf("Hello World!\n");
        printf("i=j+1=%d\n",i);
}
```

Main函数的返回值为int

在函数的末尾应当返回一个值

修改源程序


```c
//gcctest.c

#include <stdio.h>

int main()
{
        int i,j;
        i=0;
        j=0;
        i=j+1;
        printf("Hello World!\n");
        printf("i=j+1=%d\n",i);
        return 0;
}
```

```
[donger@donger gcctest]$ ls
gcctest.c
[donger@donger gcctest]$ gcc -Wall
-o gcctest gcctest.c
[donger@donger gcctest]$ ls
gcctest  gcctest.c
[donger@donger gcctest]$
```

优化编译

- 优化编译选项有：
  - ✓-O0
    缺省情况，不优化
  - ✓-O1
  - ✓-O2
  - ✓-O3

  不同程度的优化
  - ✓等等

gcc的优化编译举例（1）
考虑如下的源代码


```
//mytest.c
#include <stdio.h>
#include <math.h>

int main()
{
        int i,j;
        double k,k1,k2,k3;
        k=0.0; k1=k2=k3=1.0;
        for (i=0;i<50000;i++)
                for (j=0;j<50000;j++)
                {
                        k+=k1+k2+k3;
                        k1 += 0.5;
                        k2 += 0.2;
                        k3 = k1+k2;
                        k3 -= 0.1;
                }
        return 0;
}
```

```
[donger@donger gcctest]$ ls
gcctest.c  mytest.c
[donger@donger gcctest]$ gcc -O0 -o m0 mytest.c
[donger@donger gcctest]$ gcc -O1 -o m1 mytest.c
[donger@donger gcctest]$ gcc -O2 -o m2 mytest.c
[donger@donger gcctest]$ gcc -O3 -o m3 mytest.c
[donger@donger gcctest]$ ls
gcctest.c  m0  m1  m2  m3  mytest.c
[donger@donger gcctest]$
```

不同的优化编译选项

gcc的优化编译举例（2）

使用time命令统计程序的运行

中国科学技术大学
University of Science and Technology of China

```
[donger@donger gcctest]$ ls
gcctest.c  m0  m1  m2  m3  mytest.c
[donger@donger gcctest]$ time ./m3

real    0m2.756s
user    0m2.658s
sys     0m0.042s
[donger@donger gcctest]$ time ./m2

real    0m2.733s
user    0m2.643s
sys     0m0.037s
[donger@donger gcctest]$ time ./m1

real    0m1.829s
user    0m1.767s
sys     0m0.022s
[donger@donger gcctest]$ time ./m0

real    0m40.808s
user    0m39.632s
sys     0m0.337s
[donger@donger gcctest]$
```

GNU tools

- GNU tools和其他一些优秀的开源软件可以完全覆盖上述类型的软件开发工具。为了更好的开发嵌入式系统，需要熟悉如下一些软件
  - ✓GCC
  - ✓Binutils—辅助GCC的主要软件
  - ✓Gdb
  - ✓make
  - ✓cvs

二、GNU binutils

- binutils是一组二进制工具程序集，是辅助GCC的主要软件，它主要包括

1.addr2line 把程序地址转换为文件名和行号。在命令行中给它一个地址和一个可执行文件名，它就会使用这个可执行文件的调试信息指出在给出的地址上是哪个文件以及行号。

2.ar 建立、修改、提取归档文件。归档文件是包含多个文件内容的一个大文件，其结构保证了可以恢复原始文件内容。

3. **as** 是GNU汇编器，主要用来编译GNU C编译器gcc输出的汇编文件，他将汇编代码转换成二进制代码，并存放到一个object文件中，该目标文件将由连接器ld连接
4. **C++filt**解码C++符号名，连接器使用它来过滤 C++ 和 Java 符号，防止重载函数冲突。
5. **gprof** 显示程序调用段的各种数据。
6. **ld** 是连接器，它把一些目标和归档文件结合在一起，重定位数据，并链接符号引用，最终形成一个可执行文件。通常，建立一个新编译程序的最后一步就是调用ld。

7. **nm** 列出目标文件中的符号。
8. **objcopy**把一种目标文件中的内容复制到另一种类型的目标文件中.
9. **objdump** 显示一个或者更多目标文件的信息。使用选项来控制其显示的信息。它所显示的信息通常只有编写编译工具的人才感兴趣。
10. **ranlib** 产生归档文件索引，并将其保存到这个归档文件中。在索引中列出了归档文件各成员所定义的可重分配目标文件。
11. **readelf** 显示elf格式可执行文件的信息。

12.size 列出目标文件每一段的大小以及总体的大小。默认情况下，对于每个目标文件或者一个归档文件中的每个模块只产生一行输出。

13.strings 打印某个文件的可**打印字符串**，这些字符串最少4个字符长，也可以使用选项-n设置字符串的最小长度。默认情况下，它只打印目标文件初始化和可加载段中的可打印字符；对于其它类型的文件它打印整个文件的可打印字符，这个程序对于了解非文本文件的内容很有帮助。

14.strip **丢弃**目标文件中的全部或者特定符号。

15. **libiberty** 包含许多GNU程序都会用到的函数，这些程序有： getopt, obstack, strerror, strtol 和 strtoul.
16. **libbfd** 二进制文件描述库.
17. **libopcodes** 用来处理opcodes的库, 在生成一些应用程序的时候也会用到它, 比如objdump. Opcodes是文本格式可读的处理器操作指令.

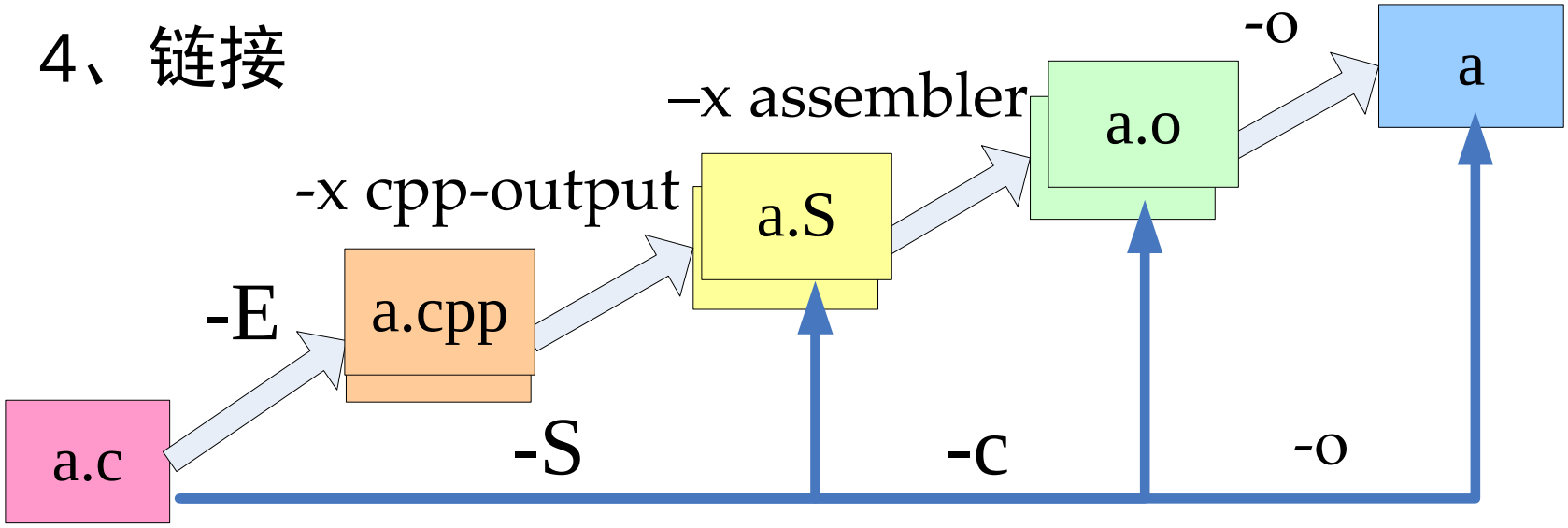
三、其他GNU工具

- Gdb—调试器
- GNU make－－软件工程工具
- diff，patch－－补丁工具
- CVS－－版本控制系统

参考 《Linux操作系统分析》课程中的 GNU Tools

总结

- 程序的编译->执行过程
  1、预处理
  2、编译成汇编代码
  3、汇编成目标代码
  4、链接



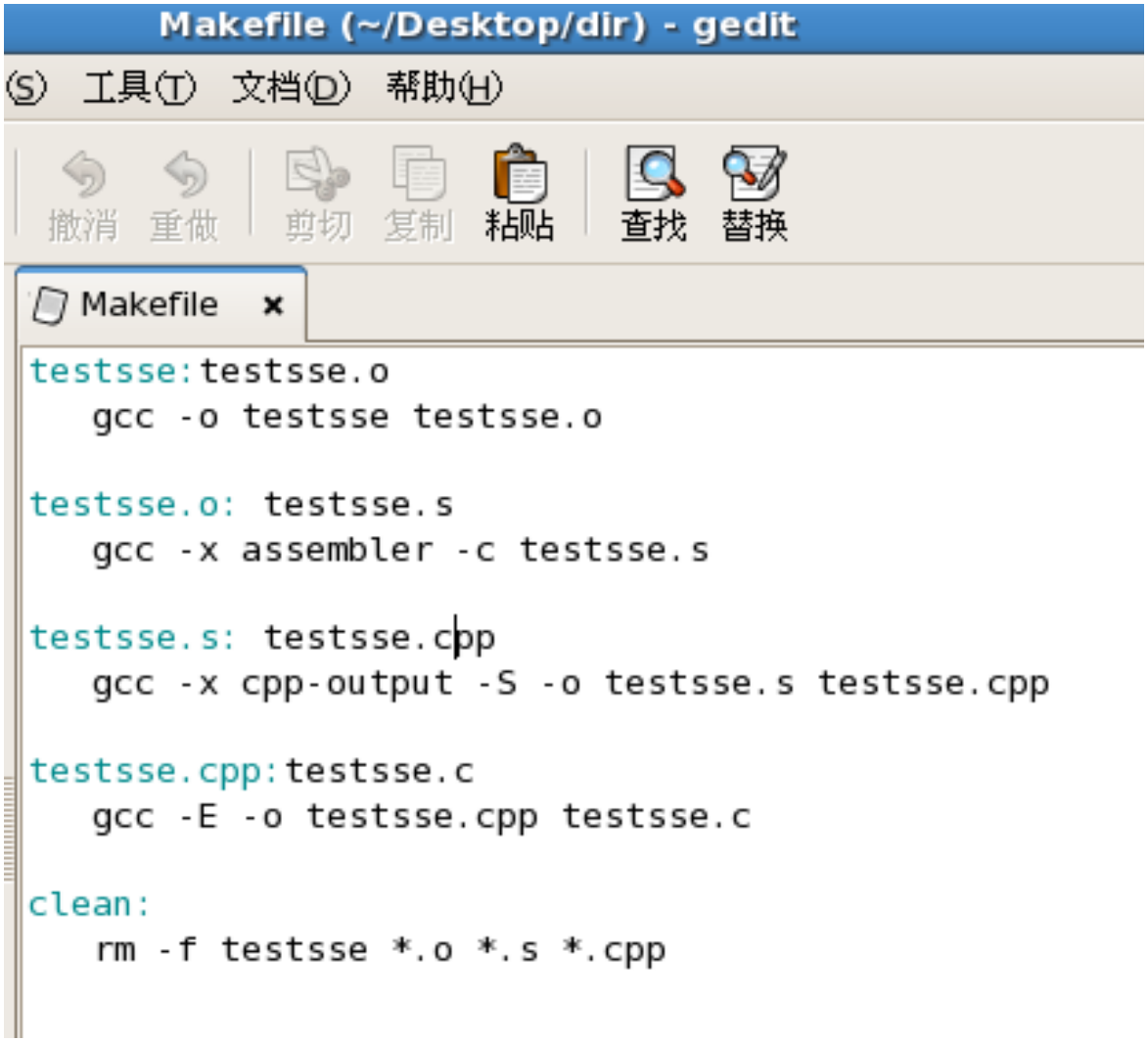
中国科学技术大学
University of Science and Technology of China

例子2. 使用MakeFile编译

王超

中国科学技术大学计算机学院

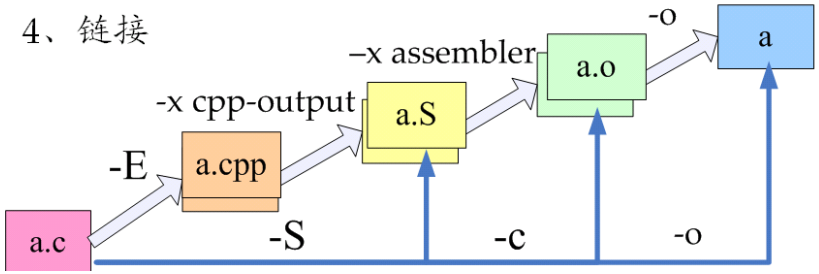
Makefile



依赖关系

```
[root@host-c-236 dir]# ls
Makefile  testsse.c
[root@host-c-236 dir]# make
gcc -E -o testsse.cpp testsse.c
gcc -x cpp-output -S -o testsse.s testsse.cpp
gcc -x assembler -c testsse.s
gcc -o testsse testsse.o
[root@host-c-236 dir]# ls
Makefile  testsse  testsse.c  testsse.cpp  testsse.o  testsse.s
[root@host-c-236 dir]# ./testsse
Hello SSEi=j+1=1
[root@host-c-236 dir]# make clean
rm -f testsse *.o *.s *.cpp
[root@host-c-236 dir]# ls
Makefile  testsse.c
[root@host-c-236 dir]#
```

- 程序的编译->执行过程
  1、预处理
  2、编译成汇编代码
  3、汇编成目标代码
  4、链接



一个“复杂”的例子


```c
//add.h

int add(int a, int b);
```

```c
//add.c

 int add(int a, int b)
{
    int c;
    c=a+b;
    return c;

}
```

```c
//testsse.c

#include <stdio.h>
#include </root/Desktop/dir/add.h>

int main()
{
  int i,j;
  i=0;
  j=0;
//   i=j+1;
  i=add(j,1);
  printf("Hello SSE");
  printf("i=j+1=%d\n",i);
}
```

Makefile

Makefile | makefile | testsse.c

```
CC = gcc -O1 -Wall

testsse:testsse.o add.o
	$(CC) -o testsse testsse.o add.o

testsse.o: testsse.c
	$(CC) -c testsse.c

add.o: add.c add.h
	$(CC) -c add.c
clean:
	rm -f testsse *.o *.s *.cpp
```

定义

多个文件

```
[root@host-c-236 dir]# ls
add.c  add.h  Makefile  testsse.c
[root@host-c-236 dir]# make
gcc -O1 -Wall -c testsse.c
testsse.c: In function ‘main’:
testsse.c:15: 警告：在有返回值的函数中，控制流程到达函数尾
gcc -O1 -Wall -c add.c
gcc -O1 -Wall -o testsse testsse.o add.o
[root@host-c-236 dir]# ls
add.c  add.h  add.o  Makefile  testsse  testsse.c  testsse.o
[root@host-c-236 dir]# ./testsse
Hello SSEi=j+1=1
[root@host-c-236 dir]#
```

在GCC参数较多（灵活）时、文件较多时极为有效

性能分析

考虑如下源代码

```c
#include<stdio.h>
int main()
{
        int i,j;
        int a = 0;
        for(i=0;i<500;i++)
                for(j=0;j<500;j++)
                        a += 1;
        return a;
}
```

```c
#include<stdio.h>
int main()
{
        int i;
        int a=0;
        for(i=0;i<250000;i++)
                a+=1;
        return a;
}
```

在不考虑编译器优化的情况下，哪个代码执行效率高（时间短），短多少?

```c
#include<stdio.h>
int main()
{
        int i;
        int a=0;
        for(i=0;i<250000;i++)
                a+=1;
        return a;
}
```

```c
#include<stdio.h>
int main()
{
        int i,j;
        int a = 0;
        for(i=0;i<500;i++)
                for(j=0;j<500;j++)
                        a += 1;
        return a;
}
```

```
        .file   "test1.c"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -8(%ebp)  a
        movl    $0, -4(%ebp)  i
        jmp     .L2
.L3:
        addl    $1, -8(%ebp)
        addl    $1, -4(%ebp)
.L2:
        cmpl    $249999, -4(%ebp)
        jle     .L3
        movl    -8(%ebp), %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
        .section        .note.GNU-stack,"",@progbits
```

```
        .file   "test.c"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -12(%ebp)    a
        movl    $0, -4(%ebp)     i
        jmp     .L2
.L5:
        movl    $0, -8(%ebp)     j
        jmp     .L3
.L4:
        addl    $1, -12(%ebp)
        addl    $1, -8(%ebp)
.L3:
        cmpl    $499, -8(%ebp)
        jle     .L4
        addl    $1, -4(%ebp)
.L2:
        cmpl    $499, -4(%ebp)
        jle     .L5
        movl    -12(%ebp), %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
        .section        .note.GNU-stack,"",@progbits
```

如何计算哪个更好？

- 一层循环时,共100W条指令
  - ✓50W addl +25W cmp + 25W JLE
- 二层循环时
  - ✓内层循环共100W条指令
    - 50W addl +25W cmp + 25W JLE
  - ✓外层循环共 2500 条指令
    - 500addl+500cmp+500JLE+500movl+500JMP
- 与指令CPI有关

$$T_{CPU} = CPI \times IC \times T_{CLK}$$

总结

- 了解程序编译的过程
- 通过查看汇编文件判断程序的执行路径
- 性能预评估（不考虑硬件实现与编译优化）

- 思考题（不交）：
- 以GCC为例，代码是如何转换为指令的。
- 调研：GCC的不同优化之间的区别在哪些方面
- 自行实现代码，生成指令序列，计算代码执行时间

"The more we study, the more we discover our ignorance."

by Percy Bysshe Shelley