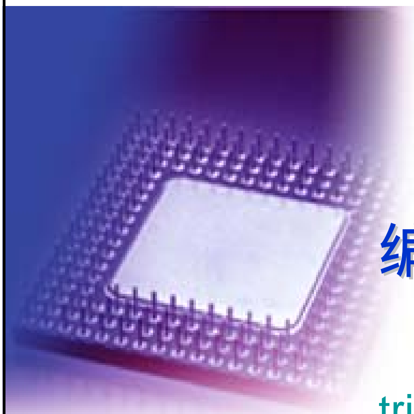
CNASIC



第 三 讲
编译，汇编，链接

凌 明

trio@seu.edu.cn

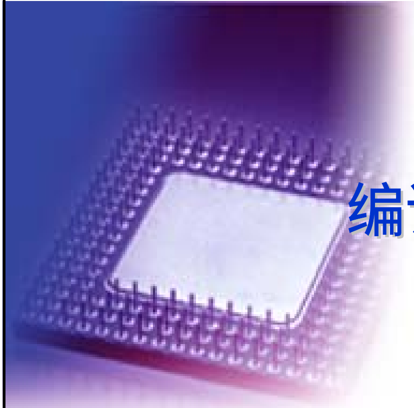
东南大学国家专用集成电路系统工程技术研究中心

CNASIC

目 录

- 编译、汇编与连接
- 项目中的文件组织
  - 项目中文件的依赖关系
  - Make
- 为ARM编写高效的C语言代码
  - 基本的C数据类型
  - C循环结构
  - 寄存器分配
  - 函数调用
  - 结构体的安排
  - 移植问题

CNASIC



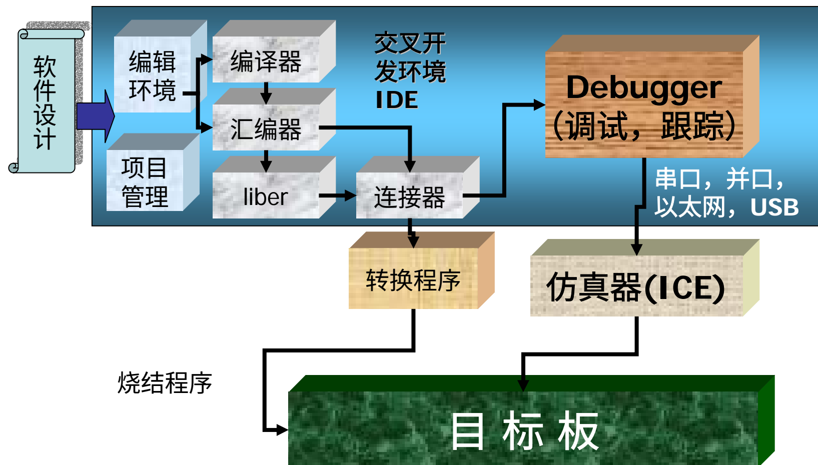
编译、汇编与连接

www.cnasic.com

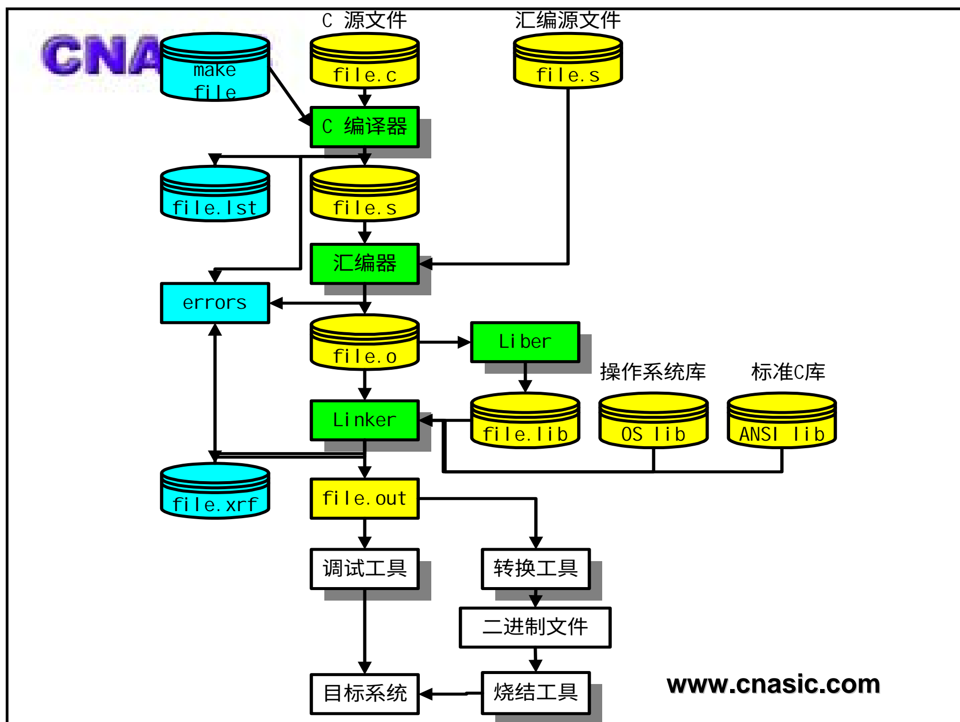
CNASIC

面向嵌入式系统的软件开发环境

（传统的开发环境）



www.cnasic.com

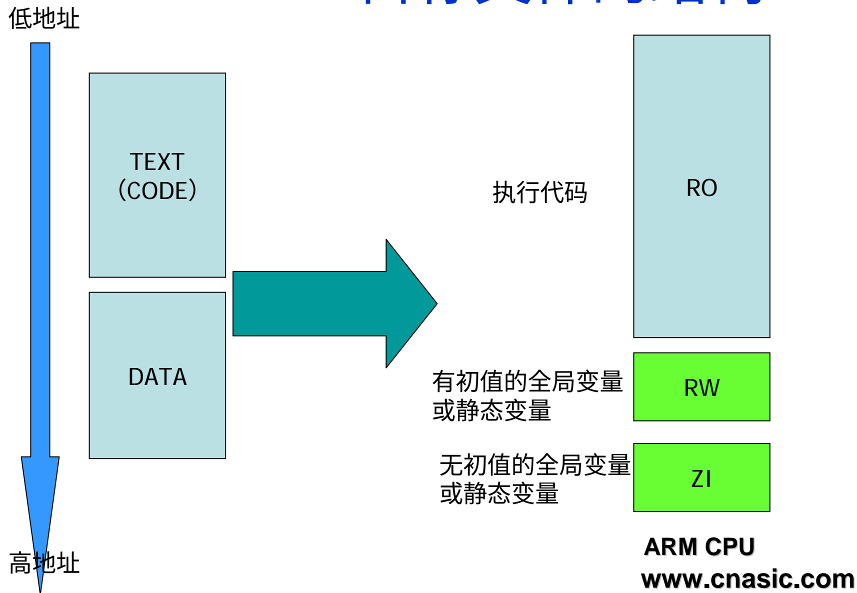


CNASIC

编译器和汇编器的作用

- 编译器将C文件转换成为汇编文件
- 汇编器将汇编文件转换成为二进制指令流 *.o文件（目标文件）
- 每个目标文件是独立编址的，也就是说每个目标文件的第一条指令都从相同的地址开始存放

www.cnasic.com



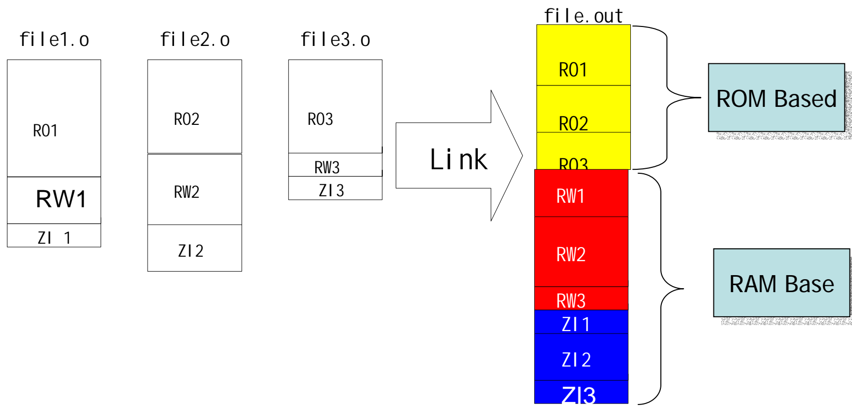
CNASIC

连接器的作用

- 将多个目标文件或库文件按照各文件中段进行统一编址
- 生成一个完整的统一的地址印象
- 嵌入式系统中一般生成一个绝对地址印象（非PIC）
- 在有MMU的系统中可以为每个任务单独分配一个地址空间

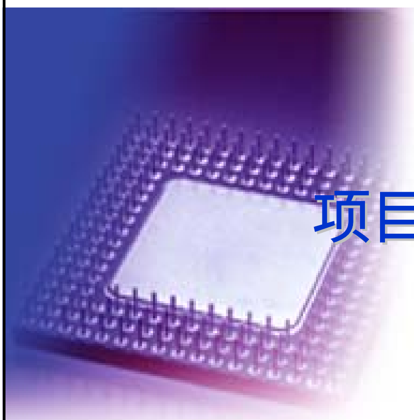
CNASIC

连接器的作用



www.cnasic.com

CNASIC

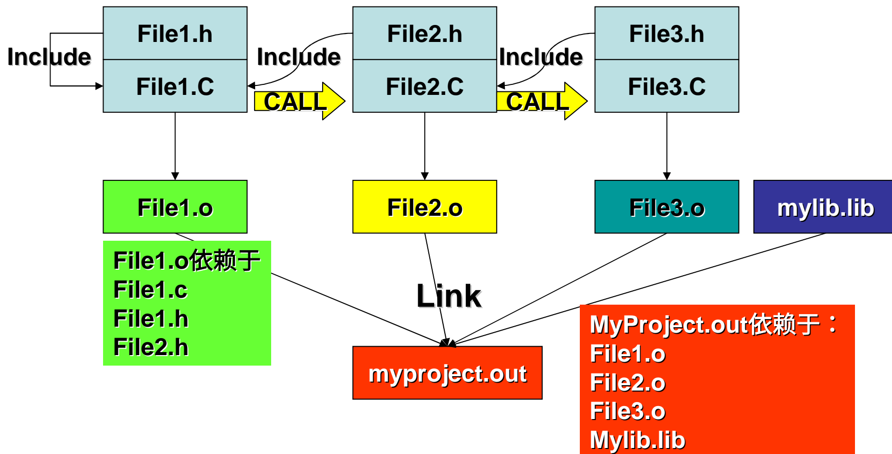


项目中的文件组织

www.cnasic.com

CNASIC

一个简单的项目-myproject



CNASIC

如何生成myproject?

- 编译File1.C->File1.o
- 编译File2.C->File2.o
- 编译File3.C->File3.o
- 连接File1.o＋ File2.o＋ File3.o＋mylib.lib

问题：
1、如果不仅仅是这几个文件，而是上百个文件怎么办？一个一个编译？

回答：
只要编译修改过的文件就可以了。但是如果我们修改了File2.h, 是不是要重新编译File1.C? 有没有办法解决这种复杂的相互依赖关系？

回答：
MAKE文件
IDE中的项目管理

再问？如果我们修改了File2.C,而没有修改File2.h, 是否要重新编译File1.C?

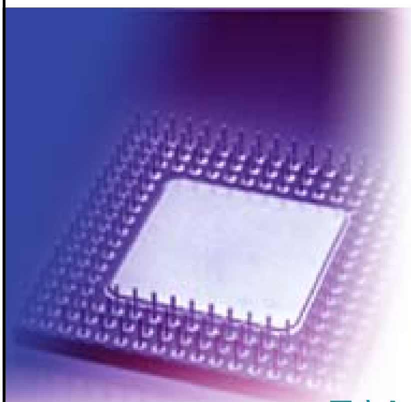
www.cnasic.com

CNASIC

MAKE文件

- MAKE实际上是一个批处理程序，该程序通过解释特定格式的MAKE脚本（MAKE文件），完成一个项目相关文件的编译，汇编与连接
- MAKE脚本一般描述了整个项目中各个文件的相互依赖关系，MAKE通过调用脚本中指定编译器，汇编器和连接器，按照项目个文件的依赖关系进行处理。
- 常见的MAKE程序：
  - MS nmake
  - Gcc make

CNASIC



MAKE文件阅读

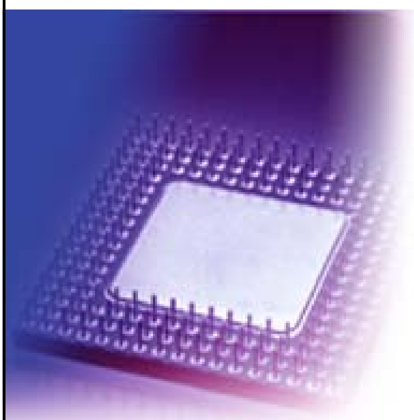
E:\ASIX OS\platform-20030124\platform\build\drball\PDA.MAK

CNASIC

IDE 中的项目管理

- 现代的IDE环境一般不需要程序员自己编写MAKE脚本，系统引入Project的概念，自动维护文件之间的依赖关系，大大方便了程序员的工作。

CNASIC



为ARM编写高效的C语言代码

CNASIC

编译器相关是一个问题！

- 不同的编译器对于数据类型的约定可能不同
  - Char, short, int, long 分别表示什么？
- 不同的编译器在处理函数调用的传参与返回值可能不同
- 不同的编译器在局部变量的处理上可能不同
- 不同的编译器在组织结构体的时候，存储器的布局可能不同
- 不同的编译器在缺省状态下的优化选项可能不同

CNASIC

ARM体系结构中的LDR和STR指令

体系结构	指令	功能
Pre V4	LDRB	装载一个无符号8位数据
	STRB	存储一个有/无符号8位数据
	LDR	装载一个有/无符号32位数据
	STR	存储一个有/无符号32位数据
V4	LDRSB	装载一个有符号8位数据
	LDRH	装载一个无符号16位数据
	LDRSH	装载一个有符号16位数据
	STRH	存储一个有/无符号16位数据
V5	LDRD	装载一个有/无符号64位数据
	STRD	存储一个有/无符号64位数据

装载8位或16位数据必须做符号位扩展，这需要额外的时间

www.cnasic.com

CNASIC

ARM C编译器的数据类型映射

- Char 无符号8位
- Short 有符号16位
- Int 有符号32位
- Long 有符号32位
- Long long 有符号64位

CNASIC

局部变量的数据类型

```
Int checksum(int *data)
{
   char i;
   int sum  = 0;
   for (i = 0; i < 64; i++)
   {
       sum += data[i];
    }

    return sum;
}
```

```
Checksum
    MOV  r2, r0                  ; r2 = data
    MOV  r0, #0                  ; sum = 0
    MOV  r1, #0                  ; i = 0
Checksum_loop
    LDR   r3,[r2, r1, LSL #2]    ; r3 = data[i]
    ADD   r1, r1, #1             ;  r1 = i + 1
    AND   r1,r1,#0xff            ; i = (char)r1
    CMP   r1,#0x40               ; I < 64 ?
    ADD   r0, r3, r0             ; sum += r3
    BCC   checksum_loop
    MOV   pc, r14                ; return sum
```

这段代码的问题:
1, 所有的ARM 寄存器都是32位
2, 所有的堆栈入口也是32位
3,编译器必须显示地处理 i = 255 时的情况，对于char 而言 255 加1 的结果是 0 !

CNASIC

局部变量的数据类型

```
short checksum(short *data)
{
   unsigned int  i;
   short sum  = 0;
   for (i = 0; i < 64; i++)
   {
      sum = (short)( sum+data[i] );
    }

    return sum;
}
```

```
Checksum
    MOV  r2, r0                 ; r2 = data
    MOV  r0, #0                 ; sum = 0
    MOV  r1, #0                 ; i = 0
Checksum_loop
   ADD  r3,r2,r1,LSL #1         ;r3 = &data[i]
   LDRH r3,[r3,#0]              ;r3=data[i]
   ADD   r1,r1,#1               ;i++
```

```
short checksum(short *data)
{
   unsigned int  i;
   int sum  = 0;
   for (i = 0; i < 64; i++)
   {
      sum += *( data++ );
    }
   //将数据类型转换放在循环外
   return (short)sum;
}
```

假设数据包中的数据是16位
注意：缺省情况下 sum + data[i]的结构是32位整新，因此需要进行显示的数据类型的强制转换

CNASIC

函数的参数类型

```
Int wordinc (int a)
{return a+1;
}
```

→

```
wordinc
ADD a1,a1,#1
MOV PC,LR
```

```
Int shortinc(short a)
{return a+1;
}
```

→

```
wordinc
ADD a1,a1,#1
MOV a1,a1,LSL #16
MOV a1,a1,ASR #16
MOV PC,LR
```

```
Int charinc(char a)
{return a+1;
}
```

→

```
wordinc
ADD a1,a1,#1
AND a1,a1,#&ff
MOV PC,LR
```

CNASIC

循环的处理

```
int fact1(int limit)
{
        …
        for(i=1;i<=limit;i++)
        {       fact = fact*i;
        }
        …
}
```

```
int fact2(int limit)
{
        …
        for(i= limit;i ! =0t;i--)
        {       fact = fact*i;
        }
        …
}
```

```
Fact1
        …
        0x000010:MUL R2,R1,R2
        0x000014:ADD R1,R1,#1
        0x000018:CMP R1,R0
        0x00001c:BLE 0x10
        …
```

```
Fact1
        …
        0x000010:MUL R0,R1,R0
        0x000014:SUBS R1,R1,#1
        0x000018:BNE 0x10
        …
```


CNASIC

循环的展开

```
Int checksum(int *data, unsigned int N )
{
   int sum  = 0;
   do
   {
       sum += *(data++);
       sum += *(data++);
       sum += *(data++);
       sum += *(data++);
       N  - = 4;

   } while  ( N != 0 );

   return sum;
}
```

```
Checksum
    MOV  r2, #0                  ; sum = 0
Checksum_loop
    LDR    r3,[r0],#4            ;r3 = *( data++ )
    SUBS  r1,r1,#4               ; N -= 4 and set flag
    ADD     r2, r3, r2           ;sum += r3
    LDR     r3 ,[r0],#4          ;r3 = *( data++ )
    ADD     r2, r3, r2           ;sum += r3
    LDR     r3 ,[r0],#4          ;r3 = *( data++ )
    ADD     r2, r3, r2           ;sum += r3
    LDR     r3 ,[r0],#4          ;r3 = *( data++ )
    ADD     r2, r3, r2           ;sum += r3
    BNE   checksum_loop
    MOV     r0, r2               ; r0 = sum
    MOV     pc, r14              ; return sum
```

每次循环需要在循环体外增加2条指令：减法指令减少循环计数，分支指令。减法需要1个周期，分支需要3个周期，我们可以通过减少循环次数来降低这些额外的开销。

关于循环展开的问题

- 只有当循环展开对提高程序的整体性能非常重要时，才可以作循环展开；否则得益不大反而会增加代码尺寸，甚至会因为替换掉cache中更重要的代码而降低总体性能

CNASIC

寄存器的分配

- 编译器会试图对C函数中的每一局部变量分配一个寄存器。如果几个局部变量不会交叠使用，编译器会对他们分配相同的寄存器。
- 当局部变量多于可用寄存器时，编译器会将这些变量分配在堆栈中。

www.cnasic.com

CNASIC

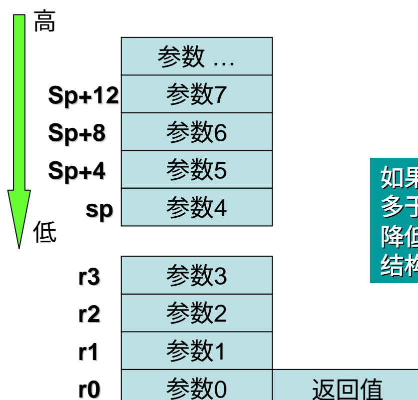
ATPCS（ARM Thumb过程调用标准）

- r0~r3 用于传参，r0用于返回值
- r4~r11 通用变量寄存器，
- r12 临时过渡寄存器
- r13 堆栈指针
- r14 连接寄存器
- r15 PC
- **所以函数内的局部变量最好不要超过12个**

www.cnasic.com

CNASIC

关于传参



如果一个函数的参数多于4个，C++的参数多于3个，其他的参数会通过堆栈进行传递，降低效率，因此将多个相关的参数组织到一个结构体中，传递该结构的指针，是一个好办法

www.cnasic.com

CNASIC

通过寄存器传参实现C调汇编

从C中直接调用汇编函数

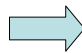
```
Extern void strcopy(char *d,const char *s)

Int main(void)
{
        const char *src = "Source";
        char dest[10];
        …
        strcopy(dest,src);
}
```

```
AREA
StrCopy,CODE,READONLY
EXPORT strcopy

Strcopy
        LDRB R2,[R1],#1
        STRB R2,[R0],#1
        CMP R2,#0
        BNE strcopy
        MOV PC,LR

END
```


CNASIC

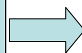
结构体的安排

```
Struct {
    char  a;
    int   b;
    char  c;
    short d;
}
```



	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

```
Struct {
    char  a;
    char  c;
    short d;
    int b;
}
```



	+3	+2	+1	+0
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]

把所有的8位元素安排在结构体的最前面
依次安排 16位，32位，64位
数组和较大的元素安排结构的最后

CNASIC

移植的问题

- Char 类型 （有符号还是无符号？）
- Int 类型(16位还是32位？)
- 不对齐的数据指针
  - 把一个char *指针转换成 int *

CNASIC

总结

- 对局部变量，函数参数和返回值要使用signed 或unsigned int类型。这样可以避免类型转换，而且可高效地使用ARM的32位数据操作指令
- 最高效的循环设计是count down to 0 的do-while循环
- 展开重要的循环来减少循环开销
- 尽可能把函数的参数限制在4个以内
- 不要使用位域，可以用掩码和逻辑操作来代替
- 避免除法
- 避免数据边界的不对齐

www.cnasic.com