上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

C语言与ASM以及线性汇编初步

- DSP 培训课件之五

上海交大-TI 联合DSP实验室

主要内容

- C语言、ASAM和线性汇编三种编程工具的比较
- 指令集概述
- C语言编程初步和实验--学习开发工具的使用，C代码性能的优化的使用（变量声明—两种变量访问方式；C优化器选项；Intrinsics；字访问）
- ASAM汇编语言初步和实验—学习用汇编语言编写简单程序
- 线性汇编—编写C可调用的汇编程序

TI DSP培训以及技术服务简介

上海交大BME-美国德州仪器联合DSP实验室成立于2007年，是国内最权威的TI技术服务于培训机构。实验室有TI（C6000，C2000，C5000，达芬奇，多核DSP）全系列开发平台，提供DSP ，MSP430等技术培训与技术服务，项目合作等。培训内容有

1）CCS开发环境精解与实例；

2）DSP/SYS BIOS 实例；

3）C6000/C5000/C2000全系列DSP架构以及汇编，C语言，混合编程等；

4）HPI，EMIF，EDMA，Timer等外设；

5）C6416、DM642， C6678多核EVM开发平台实例；

6）Bootloader 原理以及实例等。

常年开班，三人以上集体报名8折优惠，学生5折。

联系电话：13651621236（牛老师），

邮件报名：jhniu@sjtu.edu.cn ，niujinhai@yahoo.com.cn

颁发TI授权的培训证书

三种开发工具的比较

------C、ASM、线性汇编

三种开发工具的比较

- TI的DSP软件设计可采用有C/C++语言（.c）、汇编语言(.asm)和线性汇编(C语言和汇编语言的混合编程，.)。
- 用Ｃ语言开发应用程序优缺点：

  优点：易于开发和维护，用C语言书写接近自然语言，可读性强、利于理解；可移植性强；不容易发生流水线冲突；有大量现存算法可用；适用于的执行效率相对较低，不能满足实时性的要求。线性汇人机界面的开发。

  缺点：代码量大；程序效率较低；优化代码存在一定困难。一般用C语言设计应用程序的总体框架、解决人机接口和对速度效率要求不太高的复杂算法。编可把两者优点有效结合起来，设计出性价比最好、开发周期较短、比较复杂的ＤＳＰ系统，已是在C62XX上最流行的编程方法。

三种开发工具的比较

- 用汇编语言开发应用程序的优缺点：

优点：更能发挥系统特点，汇编语言设计出的程序更贴近硬件特性，往往能将硬件效能发挥到极致；代码精练、不易产生冗余、效率高；代码量小。

缺点：可读性差，不利于复杂算法的开发和实现；可移植性差；容易产生流水线冲突；复杂性高、开发周期长。

三种开发工具的比较

- C 语言程序在执行时，先要调用 C 标准库中的初始化程序(入口标号为“_c_int00”)，完成设置之后，才转入用户的主程序 main()运行，而汇编语言程序在执行时直接从用户指定入口开始，常见的入口标号为“start”；
- 由于 CCS 的代码链接器默认支持 C 语言，在编制汇编语言程序时，需要设置链接参数，选择非自动初始化，注明汇编程序的入口地址。

三种开发工具的比较

什么是线性汇编？

线性汇编类似于汇编代码，不同的是线性汇编代码中不需要给出汇编代码必须指出的所有信息，线性汇编代码对这些信息可以进行一些选择，或者由汇编优化器确定。下面是不需要给出的信息：

- 使用的寄存器
- 指令的并行与否
- 指令的延时周期
- 指令使用的功能单元

三种开发工具的比较

- 何时使用线形汇编？

1. 当程序中需要操作与硬件密切相关的设备，而用C语言较难实现时；
2. 当需要绕开C编译器的规定，进行特殊操作时。如：C语言规定程序不能访问代码区，当需要进行类似访问时可用限制较小的汇编语言程序设计；
3. 当需要提高模块的效率（包括空间上和时间上），而C语言程序无法达到要求时。

三种开发工具的比较

1）点积的C语言代码

```
int dotp(short a[], short b[])
{
    int sum, i;
    sum = 0;

    for(i=0; i<100; i++)
        sum += a[i] * b[i];

    return(sum);
}
```

三种开发工具的比较

2）ASM语言代码

```
        LDH     .D1     *A4++,A2        ; load ai from memory
        LDH     .D1     *A3++,A5        ; load bi from memory
        MPY     .M1     A2,A5,A6        ; ai * bi
        ADD     .L1     A6,A7,A7        ; sum += (ai * bi)
        SUB     .S1     A1,1,A1         ; decrement loop counter
[A1]    B       .S2     LOOP            ; branch to loop
```

- 使用的逻辑单元
  - Load (LDH and LDW) instructions must use a .D unit.
  - Multiply (MPY and MPYSP) instructions must use a .M unit.
  - Add (ADD and ADDSP) instructions use a .L unit.
  - Subtract (SUB) instructions use a .S unit.
  - Branch (B) instructions must use a .S unit.

三种开发工具的比较

- 3）非并行的ASAM代码

```
        MVK     .S1     100, A1         ; set up loop counter
        ZERO    .L1     A7              ; zero out accumulator
LOOP:
        LDH     .D1     *A4++,A2        ; load ai from memory
        LDH     .D1     *A3++,A5        ; load bi from memory
        NOP     4                       ; delay slots for LDH
        MPY     .M1     A2,A5,A6        ; ai * bi
        NOP                             ; delay slot for MPY
        ADD     .L1     A6,A7,A7        ; sum += (ai * bi)
        SUB     .S1     A1,1,A1         ; decrement loop counter
   [A1] B       .S2     LOOP            ; branch to loop
        NOP     5                       ; delay slots for branch
;   Branch occurs here
```

- 4）并行的ASAM代码

```
        MVK     .S1     100, A1         ; set up loop counter
||      ZERO    .L1     A7              ; zero out accumulator
LOOP:
        LDH     .D1     *A4++,A2        ; load ai from memory
||      LDH     .D2     *B4++,B2        ; load bi from memory
        SUB     .S1     A1,1,A1         ; decrement loop counter
   [A1] B       .S2     LOOP            ; branch to loop
        NOP     2                       ; delay slots for LDH
        MPY     .M1X    A2,B2,A6        ; ai * bi
        NOP                             ; delay slots for MPY
        ADD     .L1     A6,A7,A7        ; sum += (ai * bi)
;   Branch occurs here
```

三种开发工具的比较

非并行和并行ASAM代码性能比较

ode Example		100 Iterations	Cycle Coun
xample 5–9	Fixed-point dot product nonparallel assembly	2 + 100 × 16	1602
xample 5–10	Fixed-point dot product parallel assembly	1 + 100 × 8	801

5）线性汇编代码

```
        LDW     *a++,ai_i1      ; load ai & a1 from memory
        LDW     *b++,bi_i1      ; load bi & b1 from memory
        MPY     ai_i1,bi_i1,pi  ; ai * bi
        MPYH    ai_i1,bi_i1,pi1 ; ai+1 * bi+1
        ADD     pi,sum0,sum0    ; sum0 += (ai * bi)
        ADD     pi1,sum1,sum1   ; sum1 += (ai+1 * bi+1)
[cntr]  SUB     cntr,1,cntr     ; decrement loop counter
[cntr]  B       LOOP            ; branch to loop
```

三种开发工具的比较

完整的线性汇编代码

```
          .global _dotp

_dotp:    .cproc    a, b

          .reg      sum, sum0, sum1, a, b
          .reg      ai1:ai, bi1:bi, pi, pi1

          MVK       50,cntr          ; cntr = 100/2
          ZERO      sum0             ; multiply result = 0
          ZERO      sum1             ; multiply result = 0

LOOP:     .trip 50
          LDDW      *a++,ai1:ai      ; load ai & ai+1 from memory
          LDDW      *b++,bi1:bi      ; load bi & bi+1 from memory
          MPYSP     ai,bi,pi         ; ai * bi
          MPYSP     ai1,bi1,pi1      ; ai+1 * bi+1
          ADDSP     pi,sum0,sum0     ; sum0 += (ai * bi)
          ADDSP     pi1,sum1,sum1    ; sum1 += (ai+1 * bi+1)
 [cntr]   SUB       cntr,1,cntr      ; decrement loop counter
 [cntr]   B         LOOP             ; branch to loop

          ADDSP     sum,sum1,sum0    ; compute final result

          .return sum

          .endproc
```

三种开发工具的比较

- C代码

```
y = a * b
```

代码效率低

- 使用Intrinsics的C代码

```
y = _mpy (a, b)
```

- 嵌入汇编

```
asm ("MPY A0, A1, A2")
```

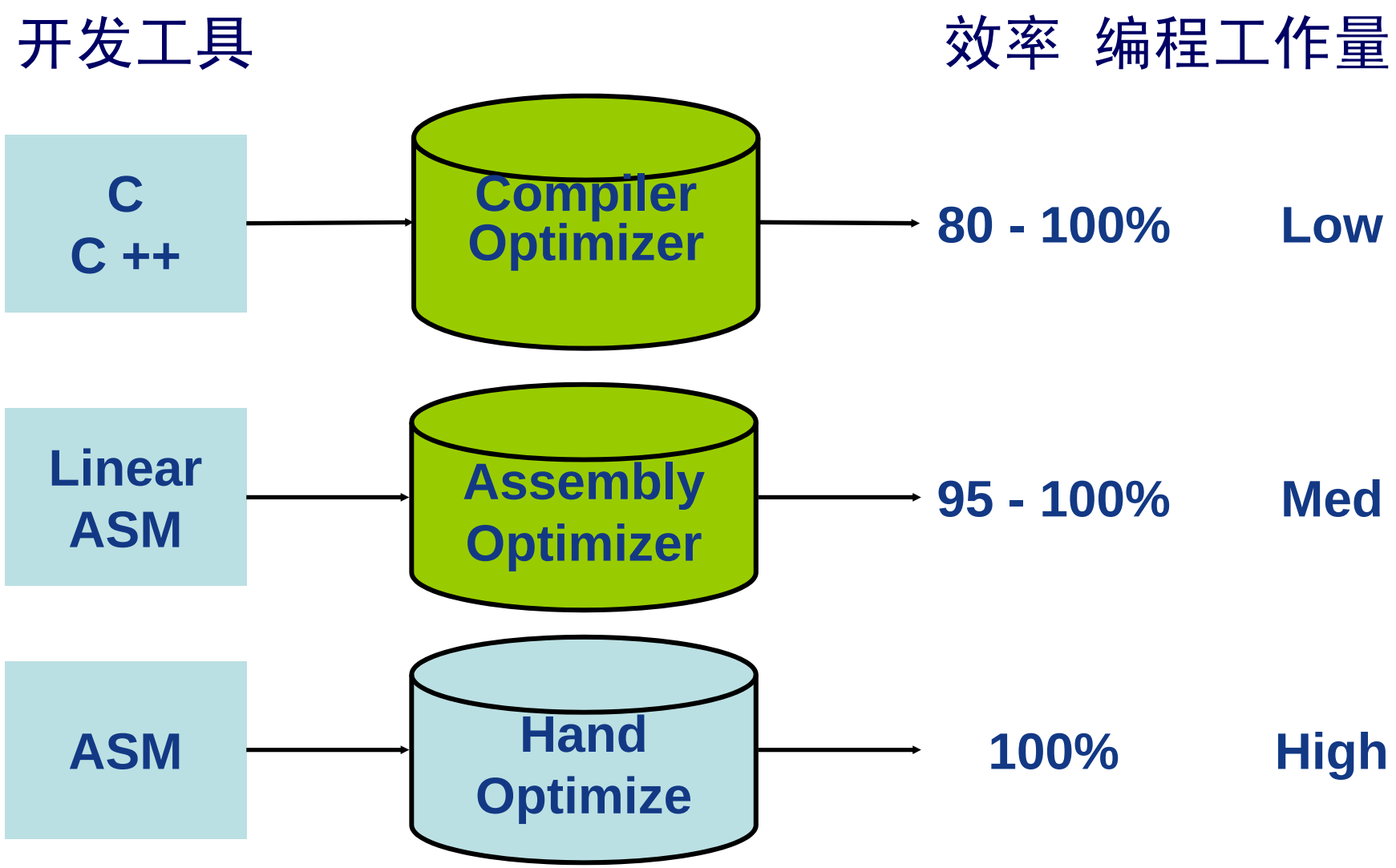
容易破坏C环境

- 汇编代码

```
MPY A0, A1, A2   ; a, b, y
```

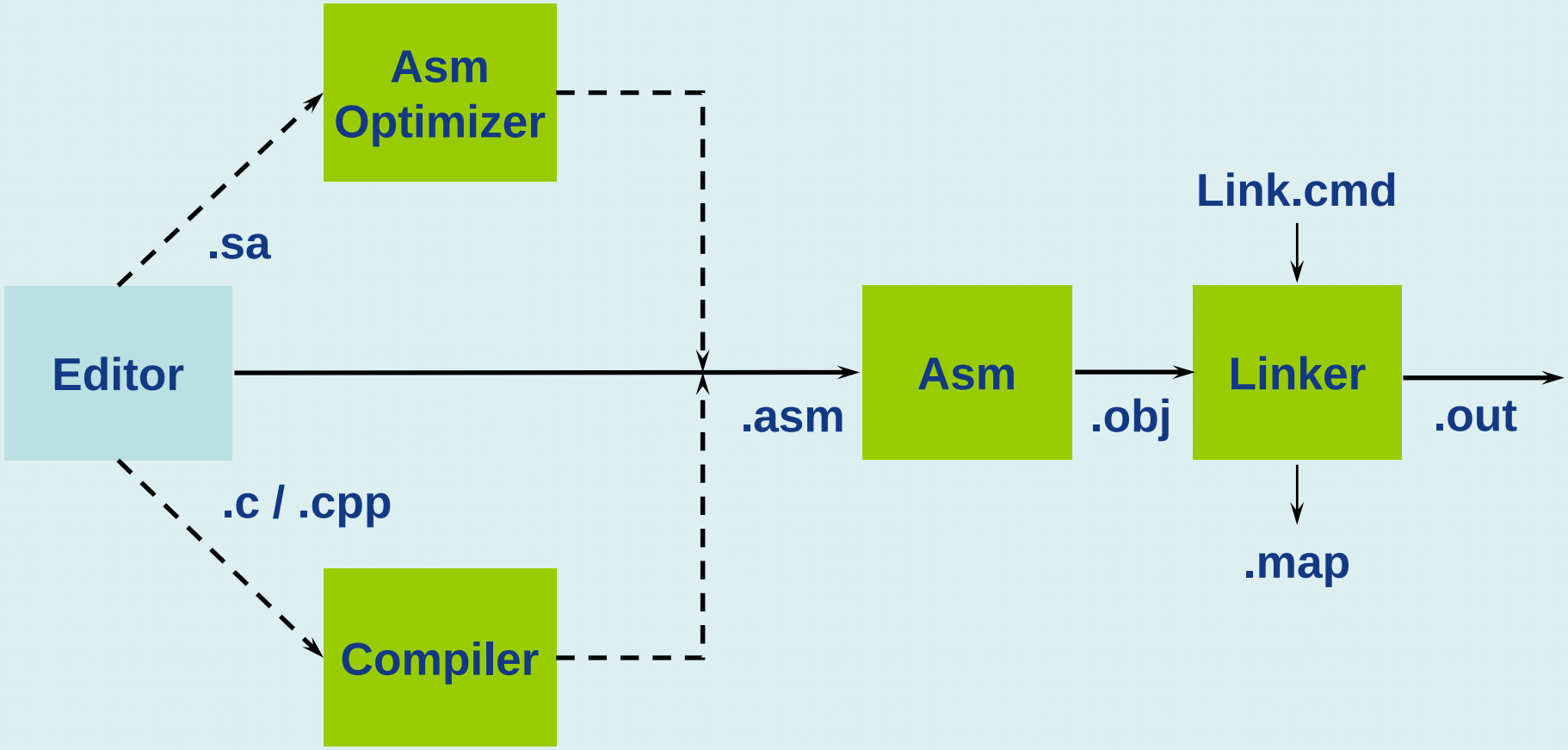
编程工作量大

三种开发工具的比较


开发工具
效率
编程工作量
C
C ++
Compiler
Optimizer
80 - 100%
Low
Linear
ASM
Assembly
Optimizer
95 - 100%
Med
ASM
Hand
Optimize
100%
High

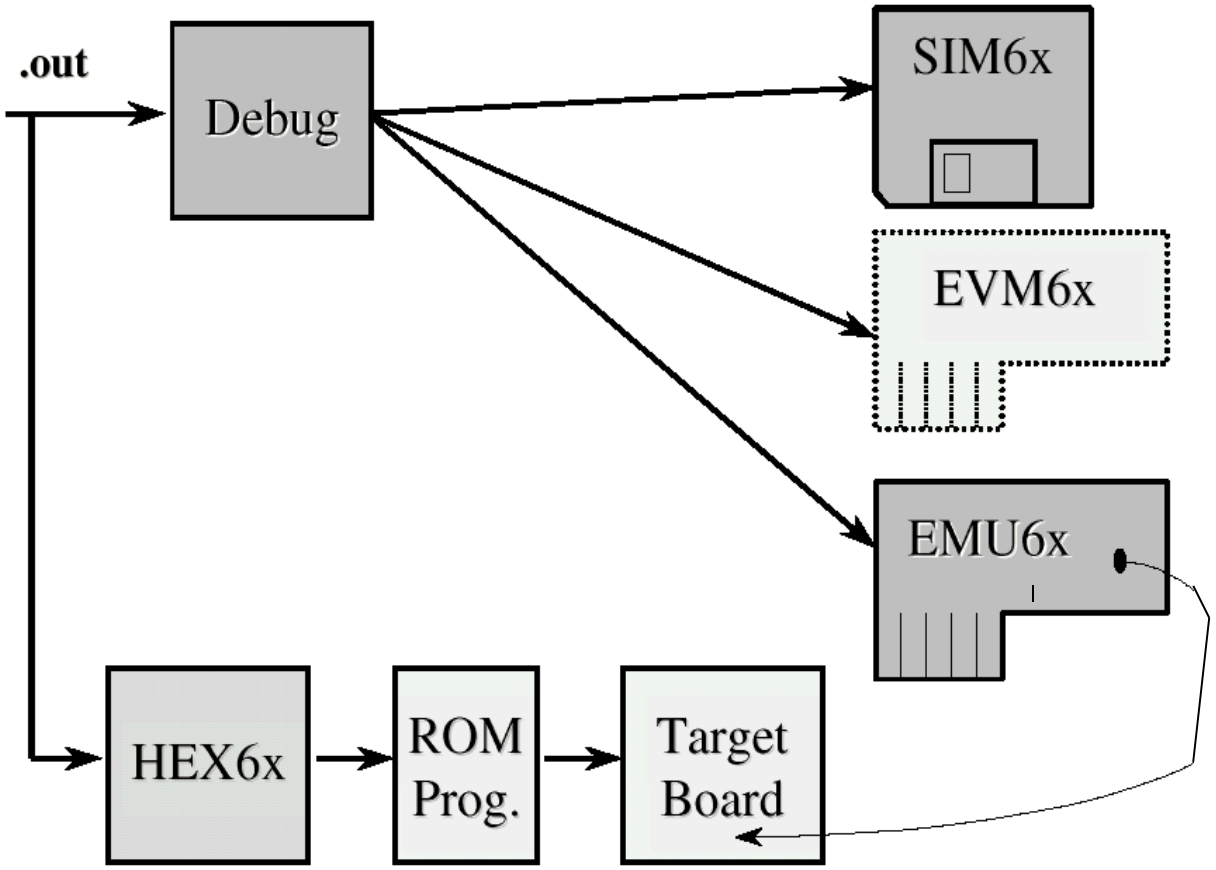
三种开发工具的比较

软件工具流程



三种开发工具的比较

硬件工具流程



仅软件

含DSP
PCI卡

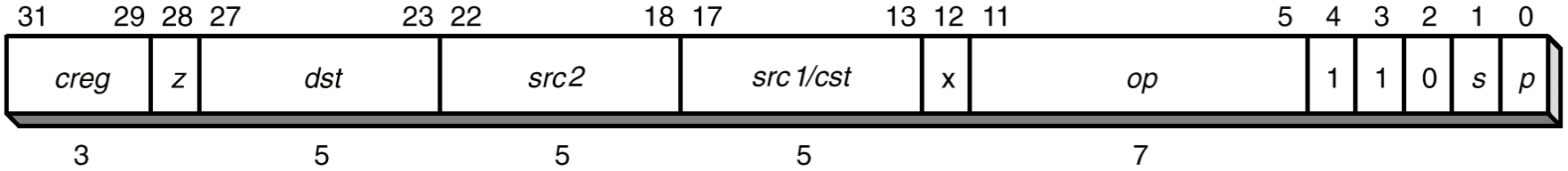
ISA卡
无DSP

指令集概述

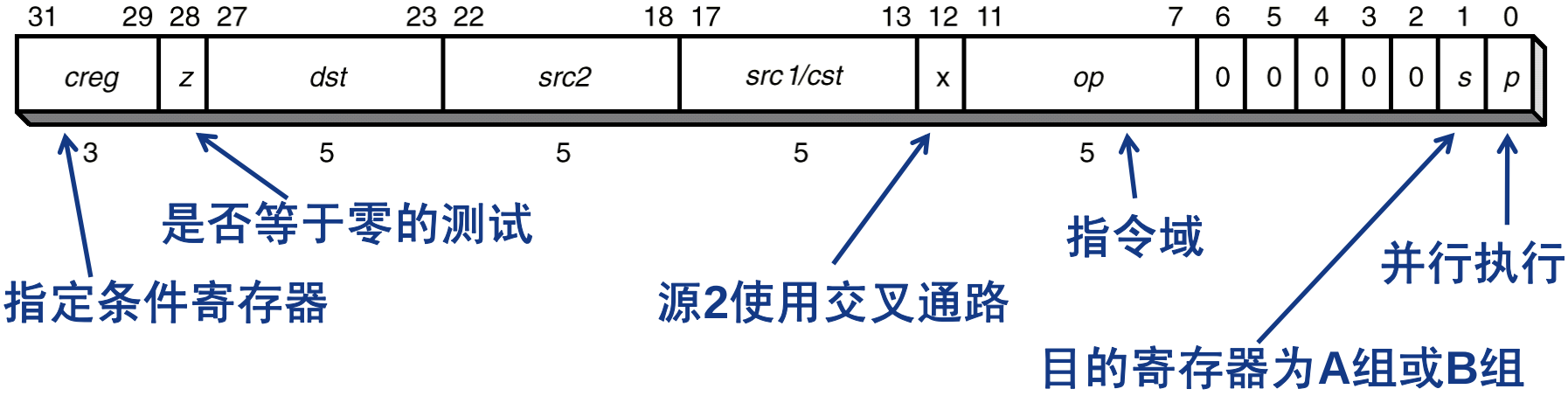
指令集概述

操作码映射（.L/.M）

Operations on the .L unit



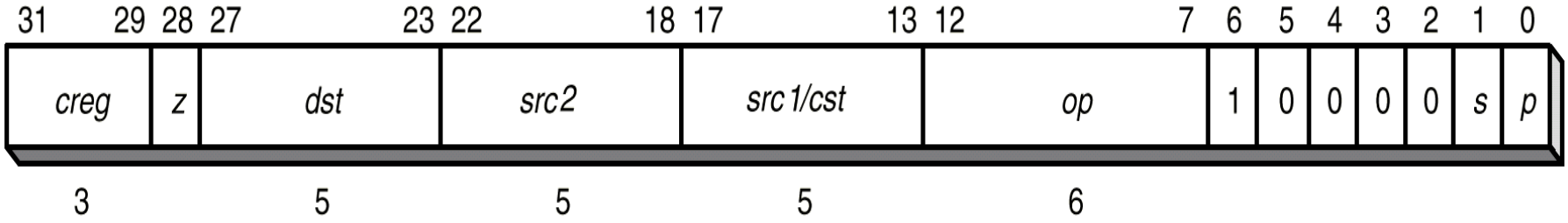
Operations on the .M unit



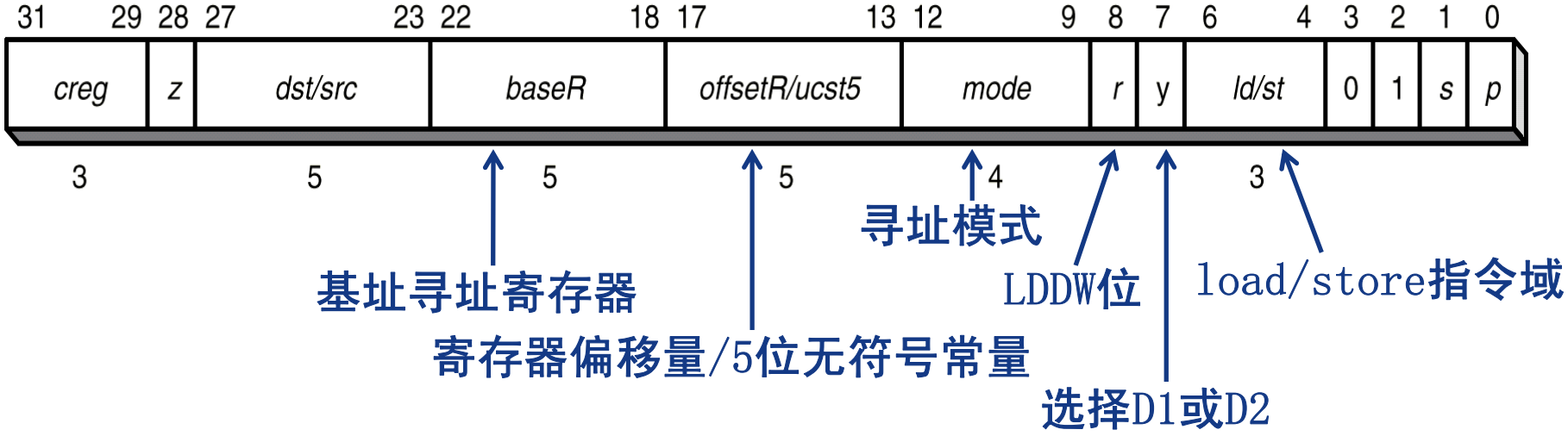
指令集概述

操作码映射（.D）

Operations on the .D unit

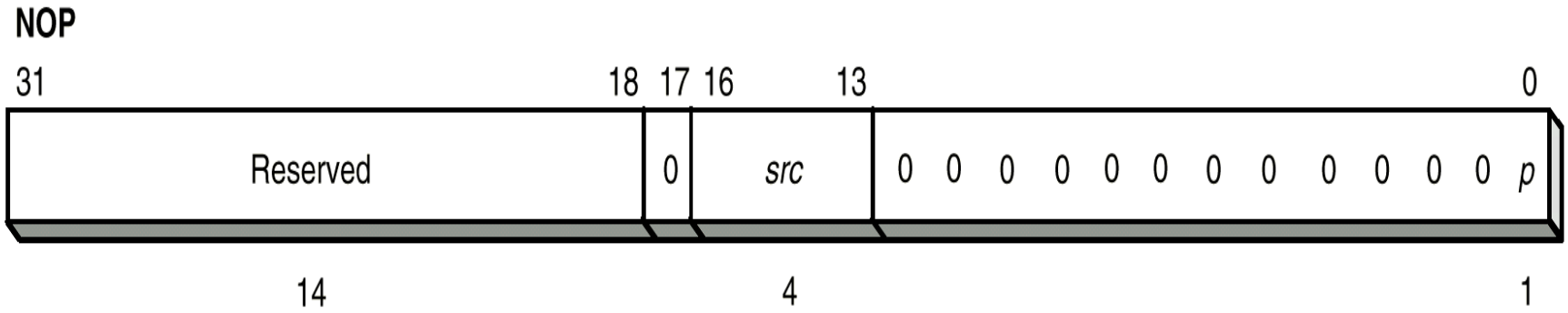


Load/store baseR + offsetR/cst on the .D unit



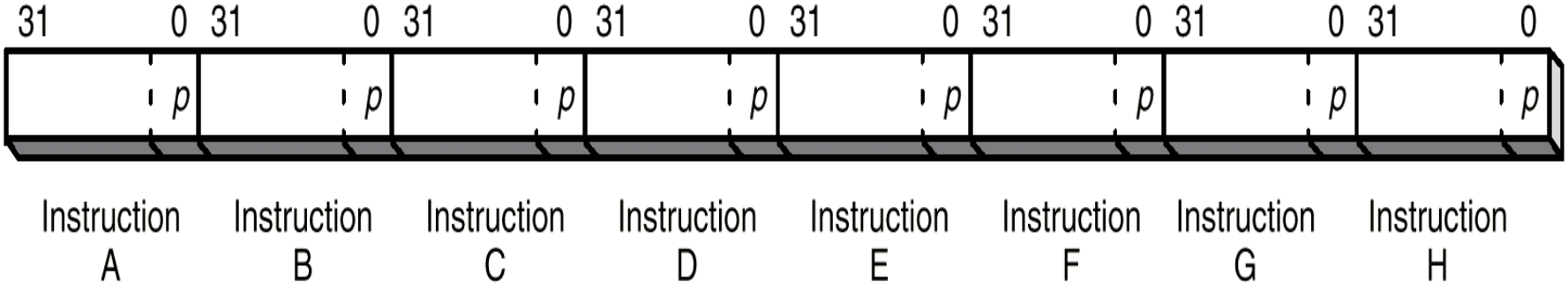
指令集概述

操作码映射（NOP）



指令集概述

并行操作

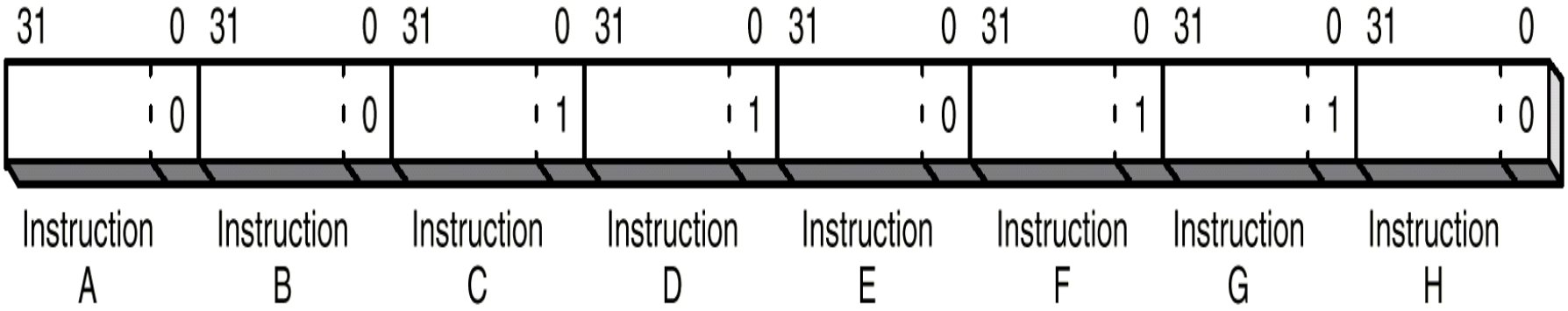


取指包的基本格式

- 取指包:八条32bit指令;
- 执行包:并行执行的所有指令。执行包中的每一条指令使用的功能单元必须各不相同;
- 每条指令的并行执行位(*p*位)控制本条指令是否与取指包中的其他指令并行执行：p =1 与下条指令并行；p =0 下条指令在当前指令的下个周期执行。

指令集概述

例子：取指包的部分并行p位模式



Cycle/Execute Packet	Instructions		
1	A		
2	B		
3	C	D	E
4	F	G	H

指令集概述

条件操作

Specified Conditional Register	creg			z
Bit	31	30	29	28
Unconditional	0	0	0	0
Reserved	0	0	0	1
B0	0	0	1	z
B1	0	1	0	z
B2	0	1	1	z
A1	1	0	0	z
A2	1	0	1	z
Reserved	1	1	x	x

- z=1，进行零测试
- z=0，进行非零测试
- $creg$=0，z=0，意味着指令将无条件地执行。

指令集概述

C62xx指令集（根据操作类型分类）

Arithmetic
ABS
ADD
ADDA
ADDK
ADD2
MPY
MPYH
MPYHL
MPYLH
SMPY
SMPYH
SADD
SAT
SSUB
SUB
SUBA
SUBC
SUB2

Logical
AND
CMPEQ
CMPGT
CMPLT
NOT
NEG
OR
SHL
SHR
SSHL
XOR

Other
IDLE
MV
MVC
NOP
ZERO

Load/Store
LD
MVK
MVKH
ST

Bit Mgmt
CLR
EXT
LMBD
NORM
SET

Program Ctrl
B
B IRP
B NRP
B reg

指令集概述

C62xx指令集（根据功能单元分类）

.S Unit	
ADD	MVKH
ADDK	NOT
ADD2	NEG
AND	OR
B	SET
B IRP	SHL
B NRP	SHR
B reg	SSHL
CLR	**SUB**
EXT	SUB2
MV	XOR
MVC	ZERO
MVK	

.L Unit	
ABS	NEG
ADD	OR
AND	SADD
CMPEQ	SAT
CMPGT	SSUB
CMPLT	**SUB**
LMBD	SUBC
MV	XOR
NORM	ZERO
NOT	

.M Unit	
MPY	MPYLH
MPYH	SMPY
MPYHL	SMPYH

.D Unit	
ADD	ST
ADDA	**SUB**
LD	SUBA

Other	
NOP	IDLE

指令集概述

C62xx指令集（根据执行周期分类）

Single Cycle		
ABS	SUB2	XOR
ADD	AND	MV
ADDA	CMPEQ	MVC
ADDK	CMPGT	ZERO
ADD2	CMPLT	MVK
SADD	NOT	MVKH
SAT	NEG	CLR
SSUB	OR	EXT
SUB	SHL	LMBD
SUBA	SHR	NORM
SUBC	SSHL	SET

Multiply	
MPY	PMYLH
MPYH	SMPY
MPYHL	SMPY

Load
LD

Store
ST

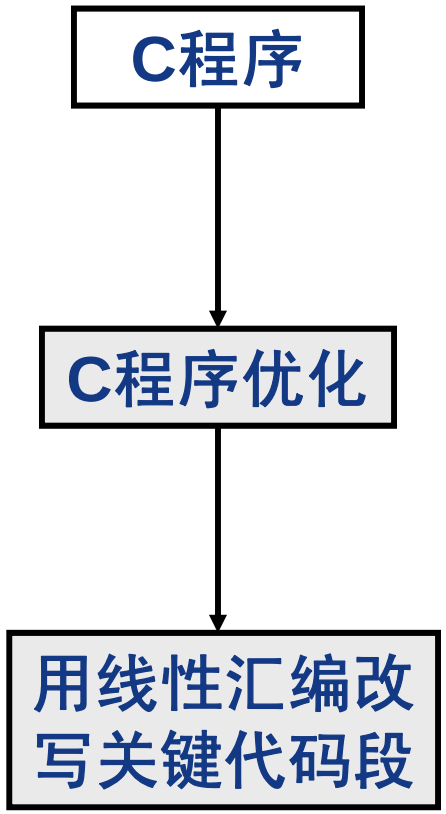
Branch
B

Other
NOP
IDLE

C语言编程初步

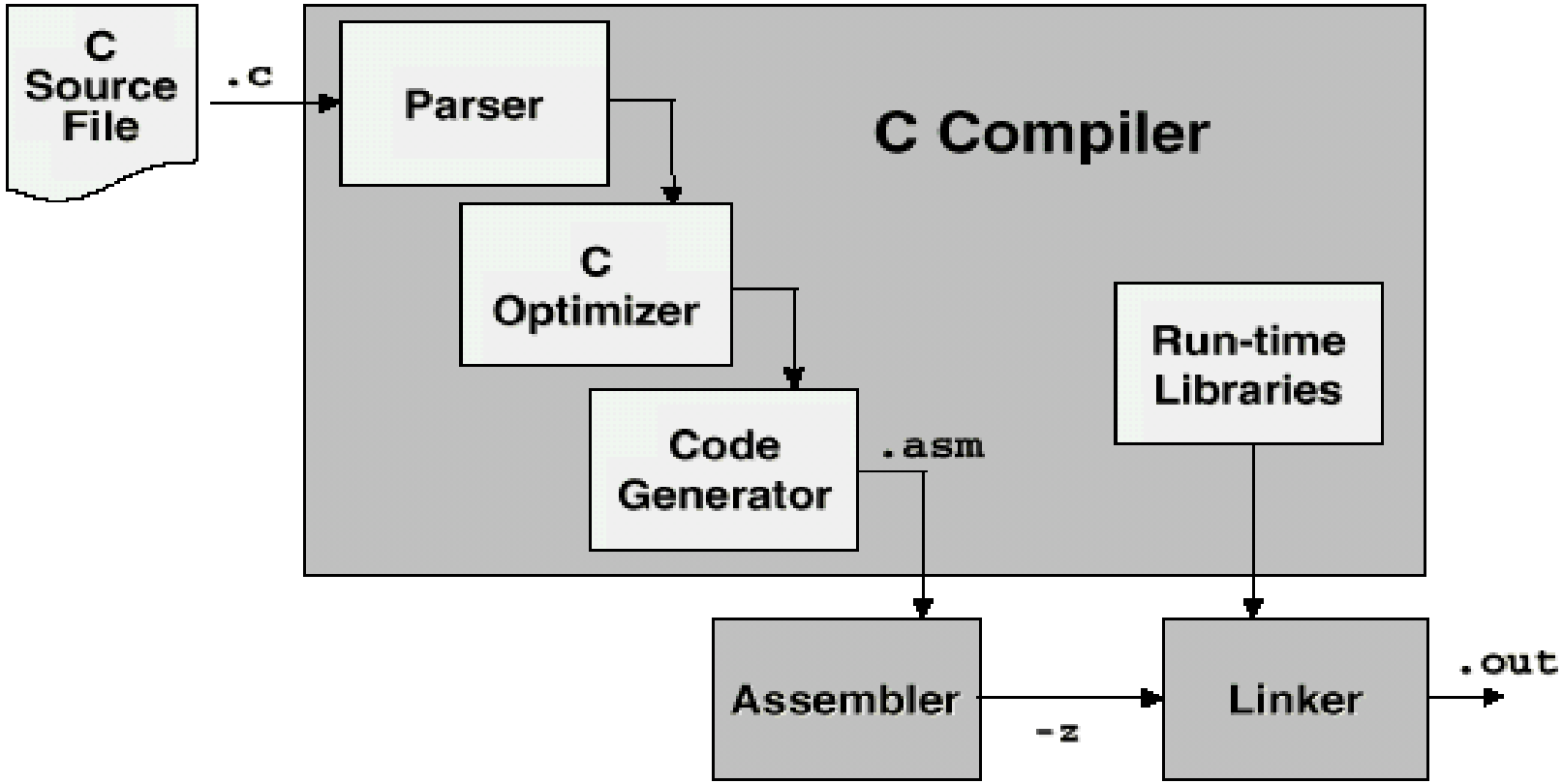
C语言编程初步

建议的程序开发流程



C语言编程初步

C的代码产生工具

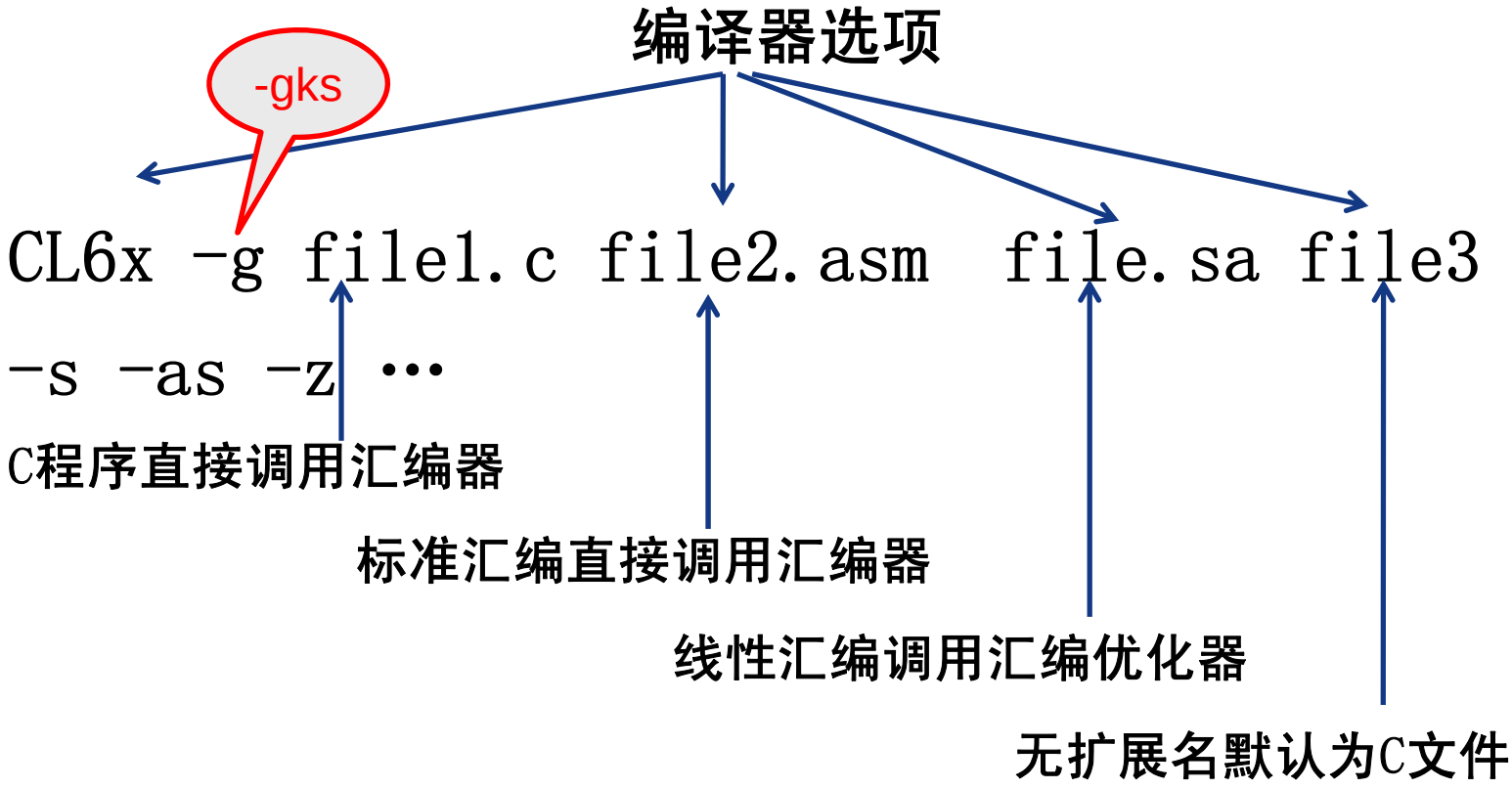


C语言编程初步

编译命令

建工程时可直接调用在CCS中进行编译、汇编和链接工具，也可在CCS外用DOS Shell程序直接调用：

C16x [options] [files]



常用的编译选项

选项	功能	备注
-g	使能符号调试	编译选项
-s	在 C 编译器生成的.asm 文件中，使能 C 语句和汇编指令交叉列表	编译选项
-on	启动 C 优化器的最高级别优化	编译选项
-pm	与-o3 合用，使能程序级优化	编译选项
-al	生成汇编器列表文件	汇编选项
-as	保留汇编符号，用于调试	汇编选项
-z	启动连接器	连接选项
-o	可执行输出文件	连接选项
-m	映射文件名(.map)	连接选项
-l	运行支持库	连接选项

C语言编程初步

连接器选项

链接器可用下面语句调用：

```
CL6x -g -s file.c -z link.cmd -o file.out -l
   rts6201.lib
```

运行支持库

-z调用连接器

连接器命令文件

输出文件名

C语言编程初步

C62xx C 数据类型

Type	Size	Representation
char, signed char	8 bits	ASCII
unsigned char	8 bits	ASCII
short	**16 bits**	**2s complement**
unsigned short	16 bits	binary
int, signed int	**32 bits**	**2s complement**
unsigned int	32 bits	binary
long, signed long	40 bits	2s complement
unsigned long	40 bits	binary
enum	32 bits	2s complement
float	32 bits	IEEE 32-bit
double	64 bits	IEEE 64-bit
long double	64 bits	IEEE 64-bit
pointers	32 bits	binary

注意：在32位计算机上C语言的long代表Size是32bits

C语言编程初步

变量声明

- 对局部变量的访问

在堆栈内分配存储空间；

用堆栈首地址作首基地址，用指针*+B15(disp)来访问；

堆栈分配在默认段.stack。

.cinit	变量初值表
.const	常量
.text	代码
.bss	全局变量和静态变量
.stack	堆栈空间
.sysmem	用于存储器分配函数
.far	Far变量

C语言编程初步

变量声明

- 全局变量/静态变量—两种访问形式

1.默认的访问方式—Near变量

```
int n;
main()
{ ...
  n +=...
}
```

编译后的汇编输出和访问方式

```
                    .bss _n, 4, 4
ldw.d1              *+DP(_n), A0
```

在.bss内分配地址一条指令访问

2.Far变量

```
far int n;
main()
{ ...
  n += ...
}
```

编译后的汇编输出和访问方式

```
_n      .usect .far,2,2
mvk             _n, A1
mvkh            _n, A1
ldw.d1          *+A1, A0
```

在.far内分配地址三条指令访问

C语言编程初步

变量声明

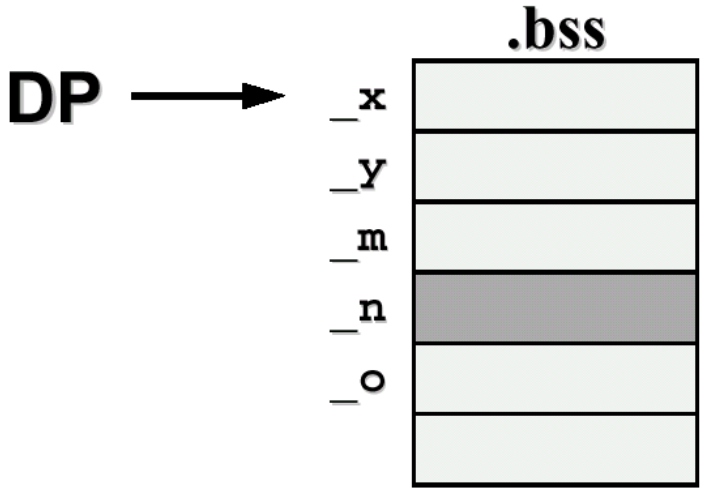
- Near变量的生成和使用

C语言

```
int x;
int y;
int m, n, o;
```

汇编语言

```
-------------
.bss _x, 4, 4
.bss _y, 4, 4
.bss _m, 4, 4
.bss _n, 4, 4
.bss _o, 4, 4
```




```
DP  .set   B14
    ldw .D2    *+DP(_n),Reg
```

相对偏移地址

```
LDW  .D2     *+B15(12),     Reg
```

C语言编程初步

变量声明

- Near/Far变量--例子

```
short  a;
int    b;
far short x;
#pragma DATA_SECTION (y, "myVariables");
#pragma DATA_SECTION (z, "myVariables");
int    y[32];
short  z;
```

声明

```
      .global _a,_b,_x,_y,_z
      .bss _a,2,4
      .bss _b,4,4

_x    .usect ".far",2,4
_y    .usect "myVariables",32*4,4
_z    .usect "myVariables",   2,4
```

C变量名加下划线

编译输出

C语言编程初步

变量声明总结

- 局部变量在堆栈段.stack分配地址，用一条指令访问。
- 采用near形式声明全局变量，变量分配在数据段.bss，用一条指令访问。
- 采用far形式声明全局变量，变量分配在数据段.far或用户自定义数据段，用三条指令访问，应尽量避免采用

C语言编程初步

C优化器

- 经过C优化器的优化，以及其它C语言优化后，C代码效率可达70—80%
- 用优化选项启动

优化器选项

优化选项	作用	优化级别
-o0	优化寄存器的使用	低
-o1	本地优化	↓
-o2或-o	全局优化	高
-o3	文件级优化	有软件流水功能

C语言编程初步

C优化器

- 与优化有关的其它编译选项

建议使用

-pm　　与-o3合用，进行程序级优化

-mt　　程序中没有数据aliasing

-x2　　函数内联

Aliasing
两个指针指向同一个变量，或一个指针修改后指向另外一个变量

不要使用

-ml　　大模式(使得.bss段内的变量都按far方式访问)

-g　　符号调试

-s，-ss，-os　　C编译器生成的汇编文件内，C语句作为注释出现

C语言编程初步

C优化器

- 使用步骤-建议

1. 不带优化选项进行编译(功能验证)

```
cl6x -g -s file.c -z
```

2. 用优化选项-o2进行编译(-o2是与符号调试兼容的最高优化级别)

```
cl6x -g -o file.c -z
```

3. 用最高级别优化选项进行编译

以上每个步骤都需要进行功能验证

C语言编程初步

C优化器

3. Intrinsics

它是直接与C62xx汇编指令相对应的特殊内联函数，没有函数调用开支。常见的Intrinsics列表如下：

Intrinsics	
_add2	_norm
_clr	_sadd
_ext/u	_set
_lmbd	_smpy
_mpy	_smpyh
_mpyh	_sshl
_mpylh	_ssub
_mpyhl	_subc
_nassert( )	_sub2
	_sat

对应汇编指令 .trip

加法、减法、乘法位域操作、long转换为int

Intrinsics的特点

- 函数参数使用C变量名(不是寄存器)，与C环境兼容；
- 不增加C的编程工作量；
- 代码效率与汇编相同。

C语言编程初步

- C优化器

4. 字访问

字访问优化方法：

1） 利用32位字访问16位数据(三种方法)

① 联合Union

② 强制类型转换

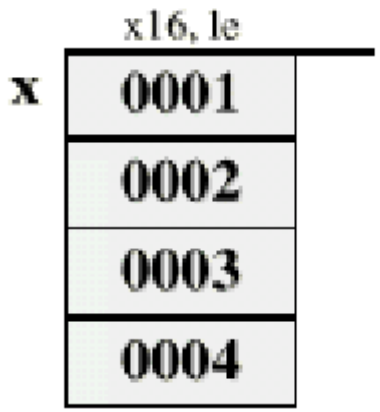
③ 把数据直接定义为32位字

2）. 利用Intrinsics完成数值运算（_mpy, _mpyh, _add2, _sub2〕

C优化器

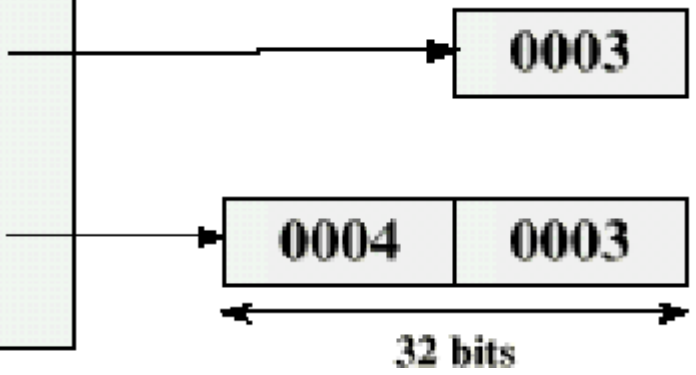
4. 字访问------联合Union

```
typedef union
{
  short s[4];
  int   w[2];
} data_union;
data_union  x = {1,2,3,4};
short       y;
int         z;
```



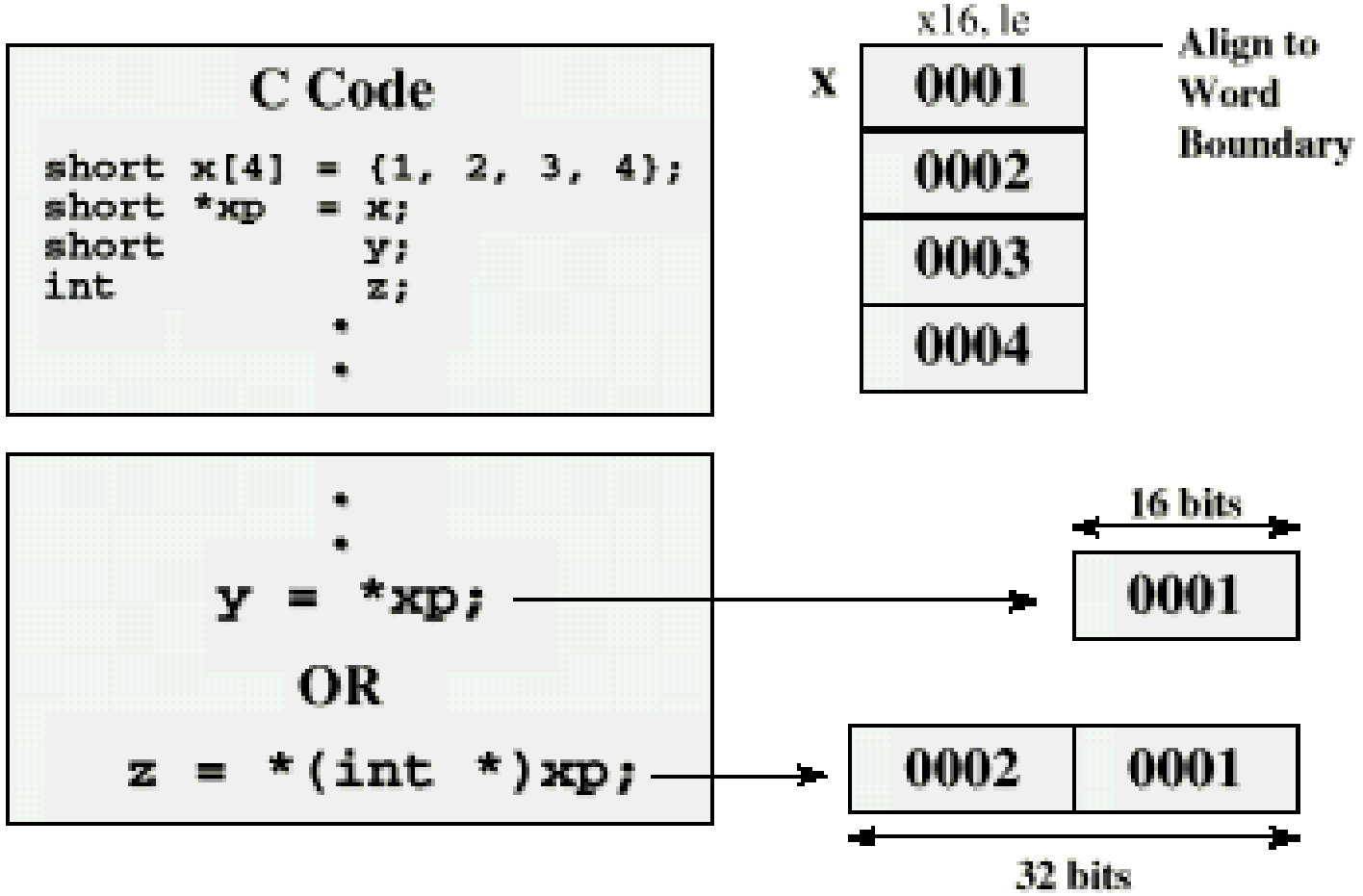
必须按照word数据定界

```
        :
        :
y = x.s[2]
   OR
z = x.w[1]
```



C优化器

4. 字访问------强制类型转换



C语言编程初步

C优化器

4. 字访问------把数据直接定义为32位字

C Code

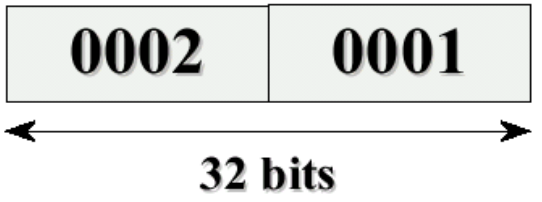
```
int x[2] = {0x00020001, ...};
int *xp  = x;
int      z;

         .
         .
```

x16, le

x	0001
	0002
	0003
	0004

```
         .
         .
   z = *xp;
```



C优化器

4. 字访问------小结

- 用union方式需要对调用函数和被调用函数进行修改
- 用强制类型转换，只需要修改被调用函数
- 直接定义为32字，影响程序可读性

实验

- **实验内容：** 两个数组点积运算
- **实验目的：**
  - 掌握编译过程、C优化器的使用
  - 熟悉Simulator开发环境
  - 学习程序性能测试方法

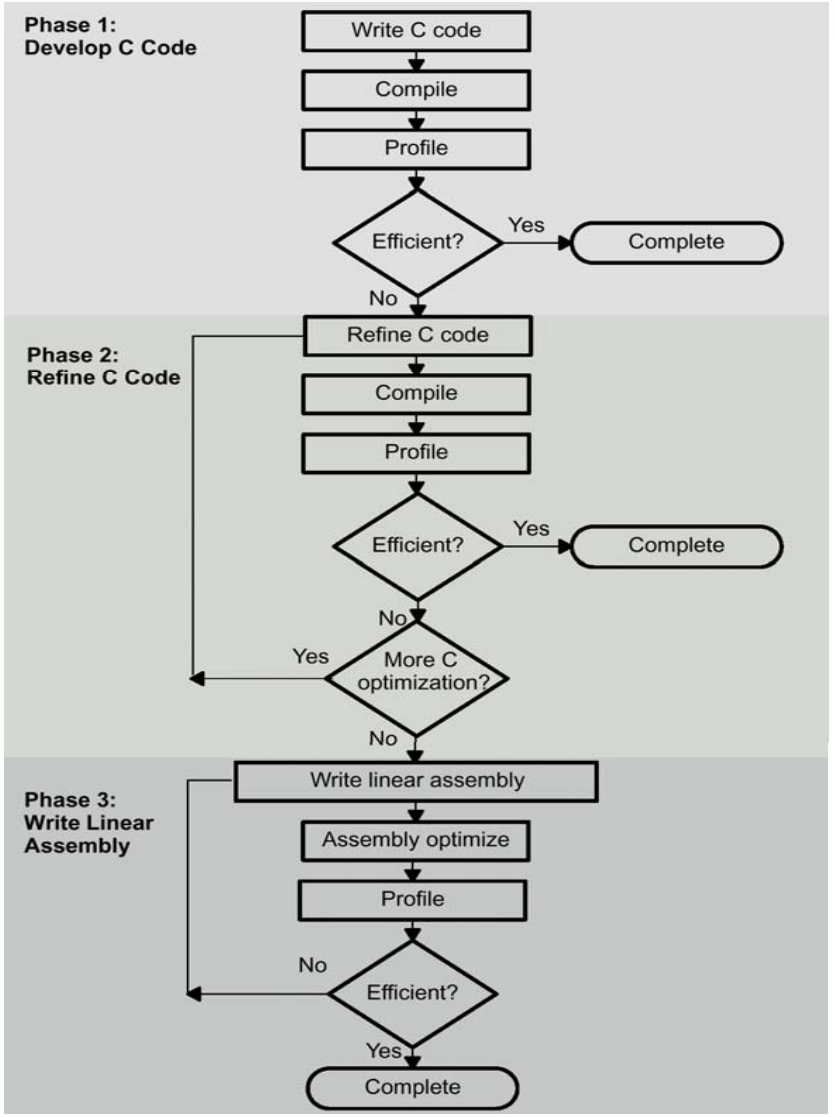
实验：两个数组点积运算

$$Y = \sum_{n=1}^{40} a_n * x_n$$

运算的两个基本指令

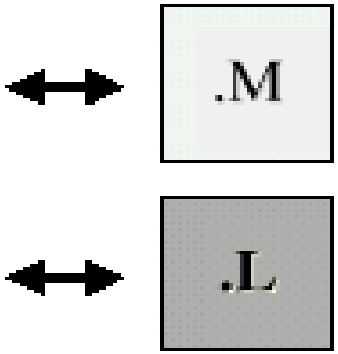
- Multiply
- Add

代码开发流程



实验：两个数组点积运算

- 相加（.L单元）

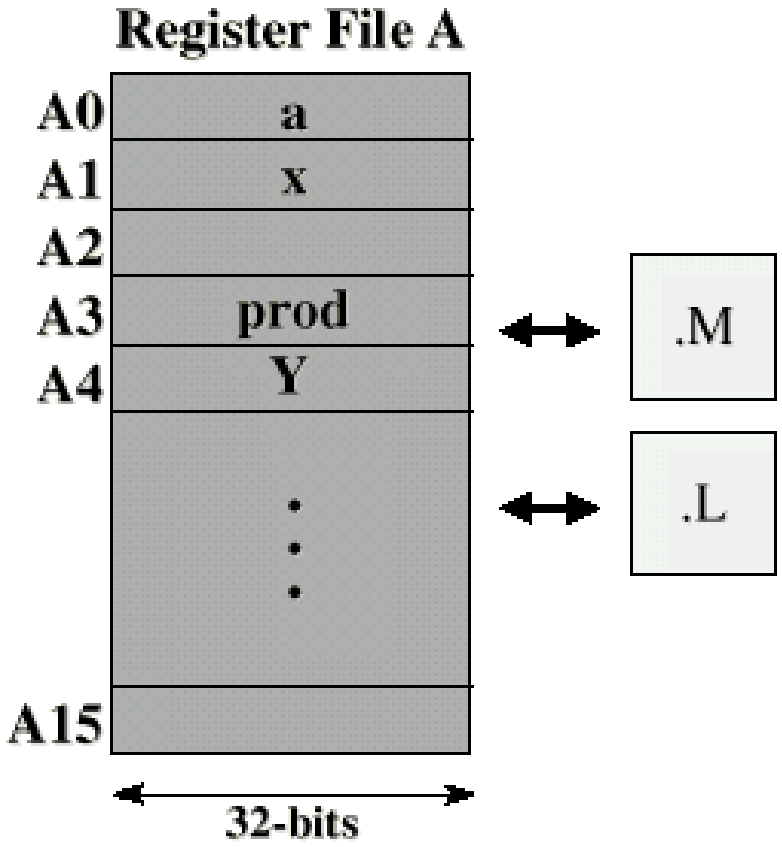


$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
MPY    .M    a, x, prod
ADD    .L    Y, prod, Y
```

实验：两个数组点积运算

- 寄存器组—A

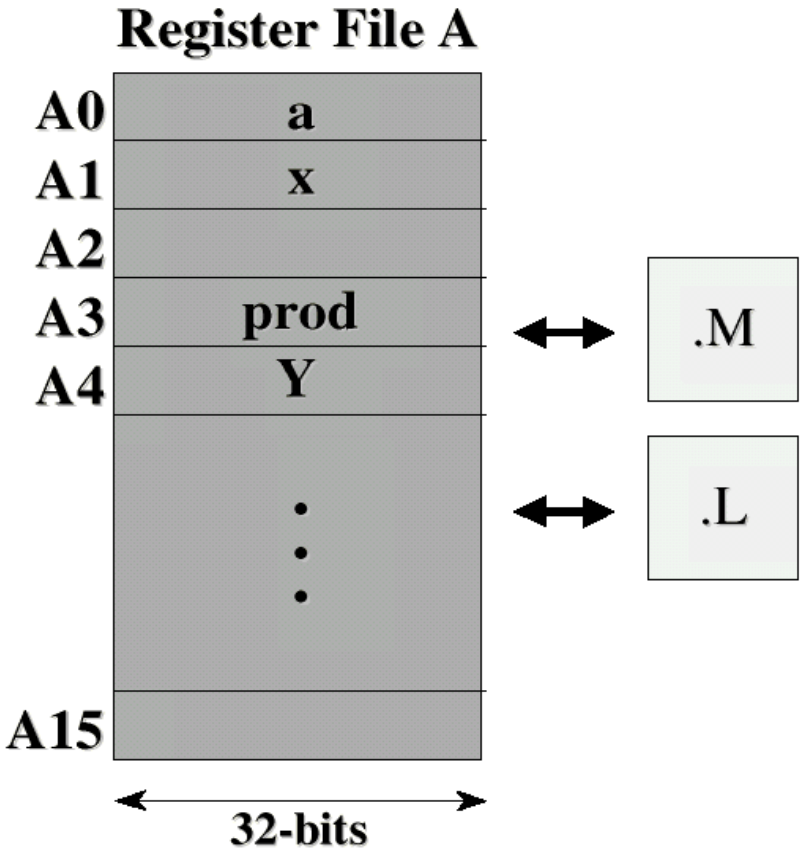


$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
MPY   .M   a, x, prod
ADD   .L   Y, prod, Y
```

实验：两个数组点积运算

- 寄存器取代变量



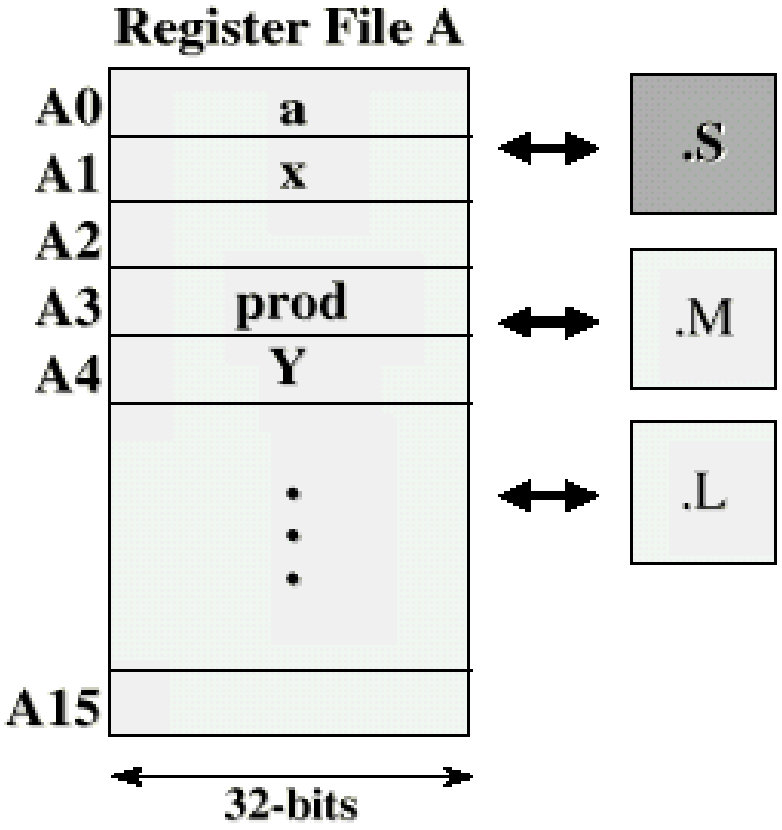
$$Y = \sum_{n=1}^{40} a_n * x_n$$

MPY	.M	A0, A1, A3
ADD	.L	A4, A3, A4

实验：两个数组点积运算

建立循环

1. 添加跳转指令和循环标号



$$Y = \sum_{n=1}^{40} a_n * x_n$$

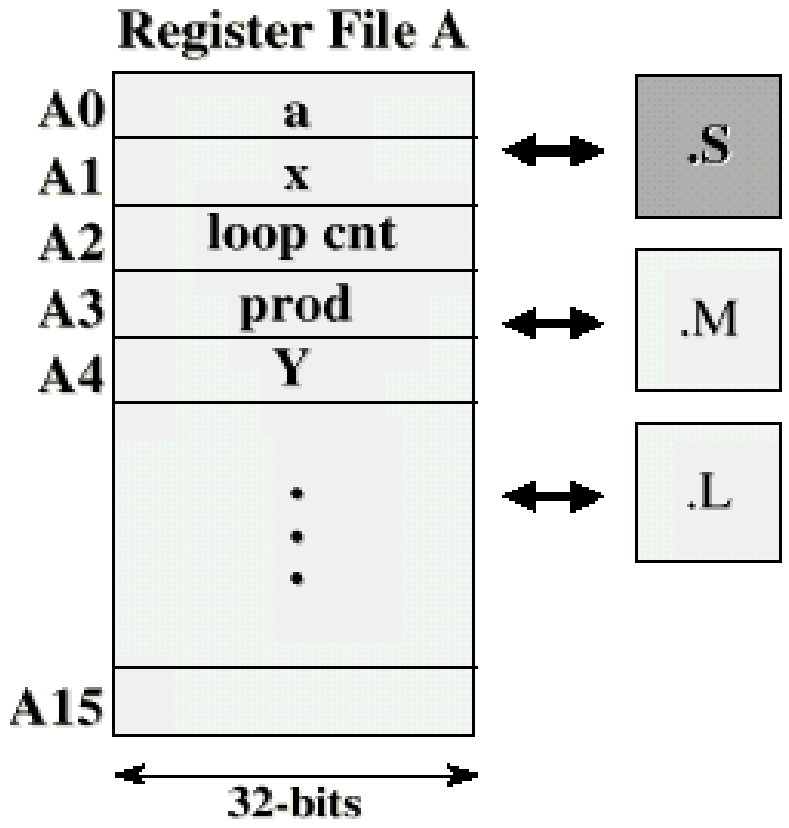
```
loop:
        MPY     .M      A0, A1, A3
        ADD     .L      A4, A3, A4

        B       .S      loop
```

实验：两个数组点积运算

建立循环

2. 设定一个循环计数器



$$Y = \sum_{n=1}^{40} a_n * x_n$$

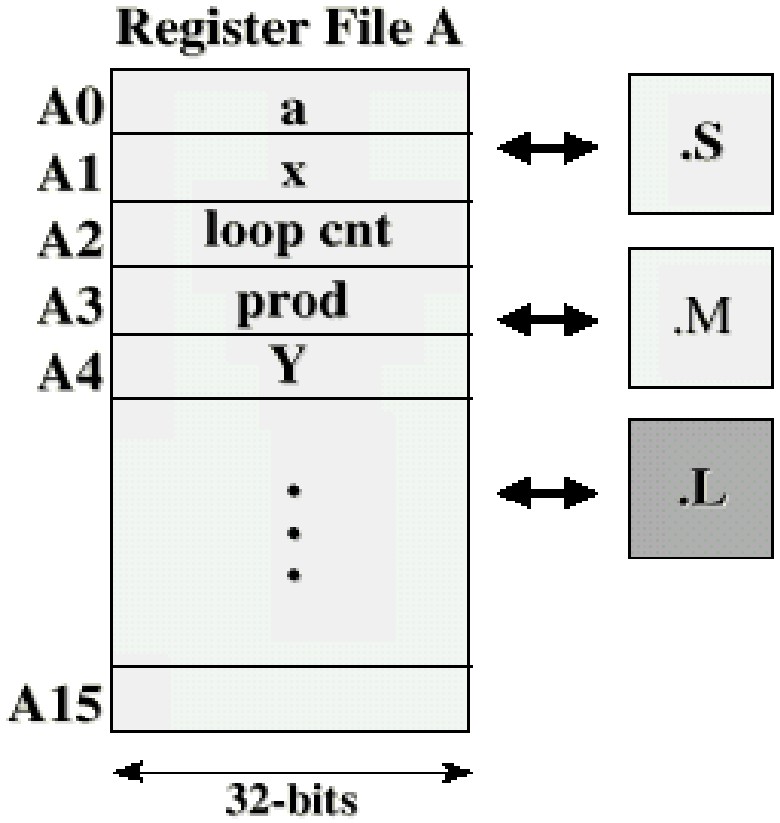
```
        MVK   .S    40, A2
loop:
        MPY   .M    A0, A1, A3
        ADD   .L    A4, A3, A4

        B     .S    loop
```

实验：两个数组点积运算

建立循环

3. 添加递减循环计数指令



$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
        MVK   .S    40, A2
loop:
        MPY   .M    A0, A1, A3
        ADD   .L    A4, A3, A4
        SUB   .L    A2, 1, A2
        B     .S    loop
```

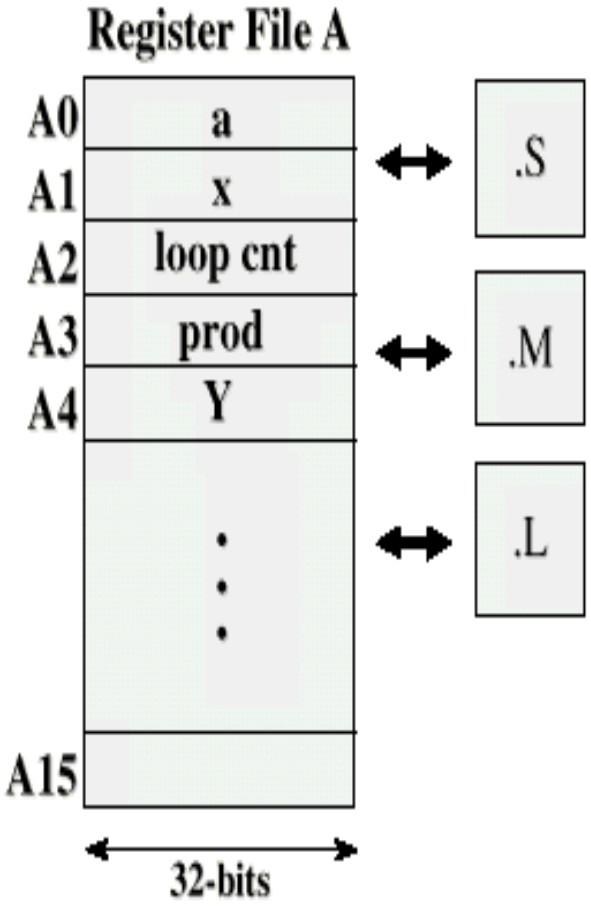
实验：两个数组点积运算

建立循环

4. 给出基于循环计数值的跳转条件

所有指令都是根据下面条件寄存器的值为0或非0条件地执行:A1、A2、B0、B1、B2。

代码语法	指令执行条件
[A2]	A2 ≠ 0
[!A2]	A2=0

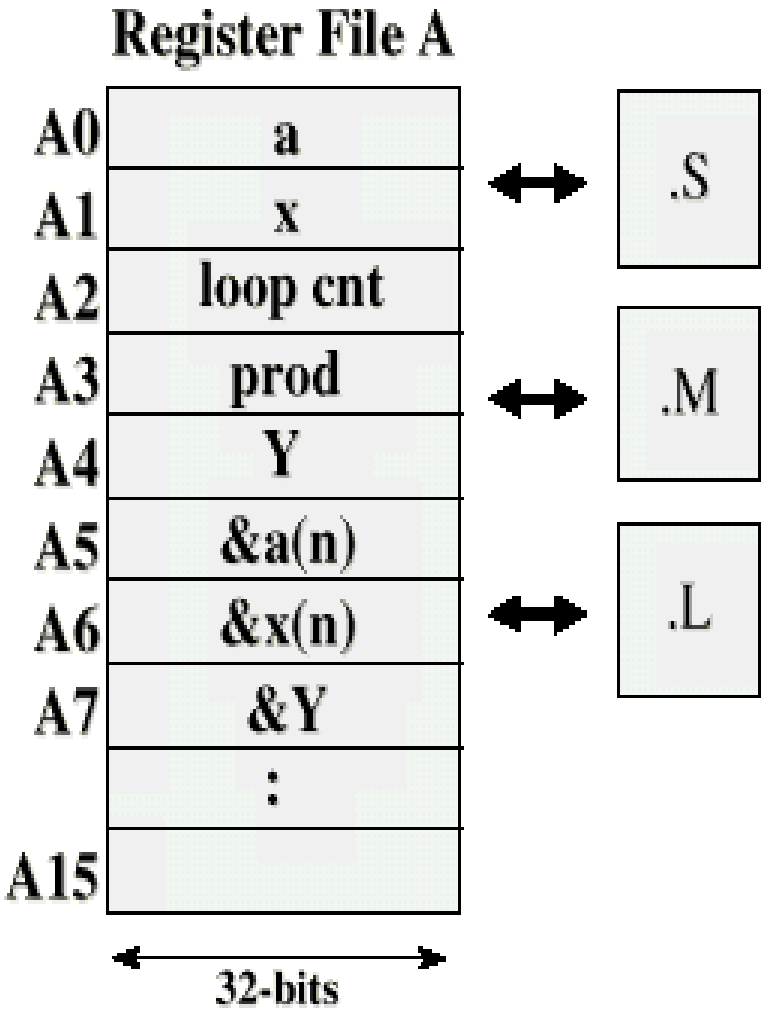


$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
        MVK   .S    40, A2
loop:
        MPY   .M    A0, A1, A3
        ADD   .L    A4, A3, A4
        SUB   .L    A2, 1, A2
[A2]    B     .S    loop
```

实验：两个数组点积运算

- 设将数值读入寄存器



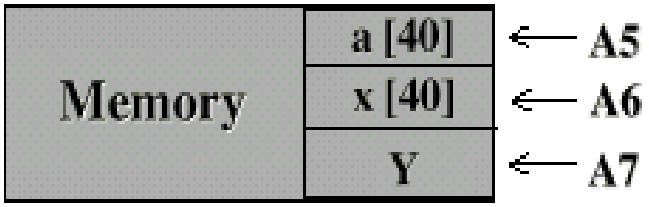
如何读取a和x？

1. a、x和Y在存储器中
2. 建变量指针

```
A5 = &a
A6 = &x
A7 = &Y
```

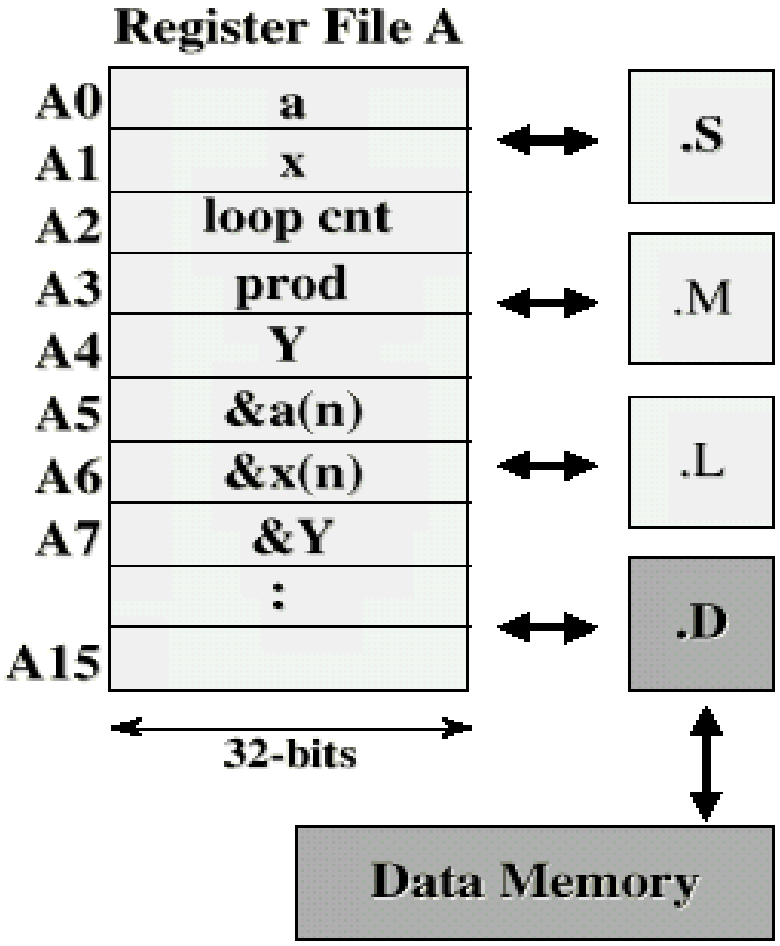
3. load/store中使用指针

```
LD    *A5, A0
LD    *A6, A1
ST    A4, *A7
```



实验：两个数组点积运算

- 读取/存入（.D单元）



$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
        MVK   .S   40, A2
loop:   LD    .D   *A5, A0
        LD    .D   *A6, A1
        MPY   .M   A0, A1, A3
        ADD   .L   A4, A3, A4
        SUB   .L   A2, 1, A2
[A2]    B     .S   loop
        ST    .D   A4, *A7
```

实验：两个数组点积运算

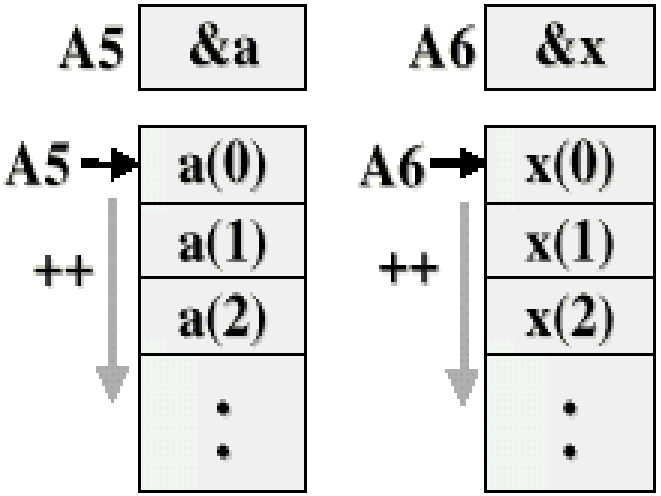
读取指针

1. 地址是常数，因此使用指令MVK： `MVK .S a, A5`
2. MVK指令所移的位数：16 bit
3. 表示一个完整地址的位数：32 bit
4. 一个地址读入寄存器必须使用两条指令，例如：

```
MVK   .S   a, A5
MVKH  .S   a, A5
```

实验：两个数组点积运算

关于指针



第1次循环后，A4为：a(0) * x(0)

第2次循环如何访问a(1)和x(1)

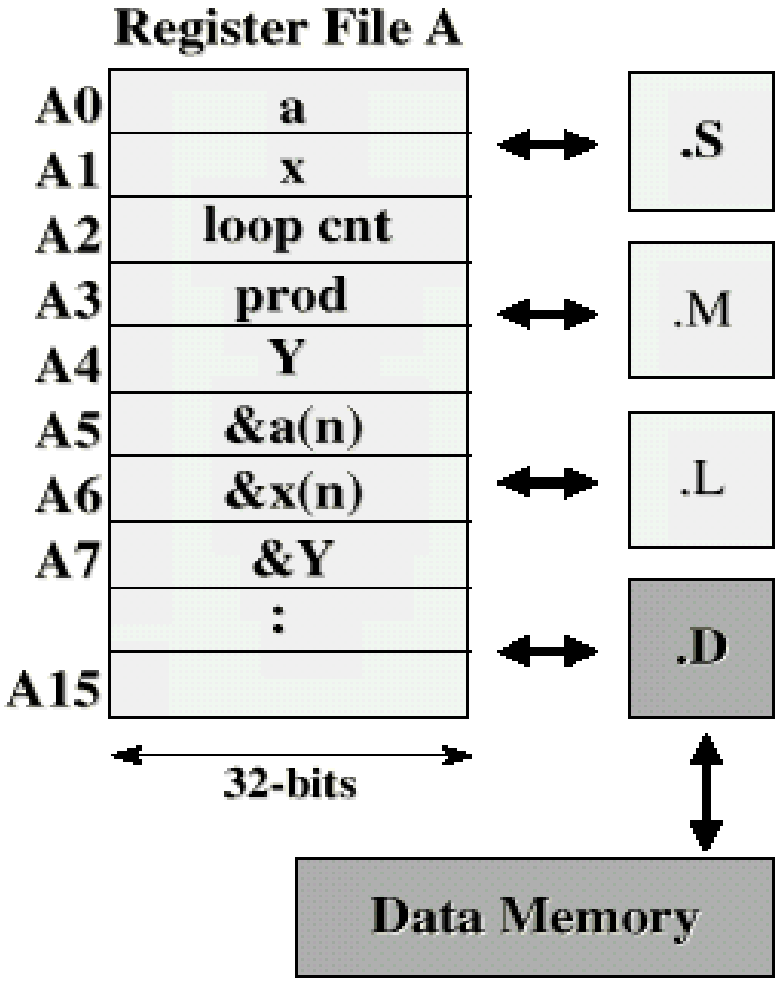
```
LD    .D    *A5++, A0
LD    .D    *A6++, A1
```

$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
        MVK   .S    40, A2
loop:   LD    .D    *A5, A0
        LD    .D    *A6, A1
        MPY   .M    A0, A1, A3
        ADD   .L    A4, A3, A4
        SUB   .L    A2, 1, A2
[A2]    B     .S    loop
        ST    .D    A4, *A7
```

实验：两个数组点积运算

递增指针

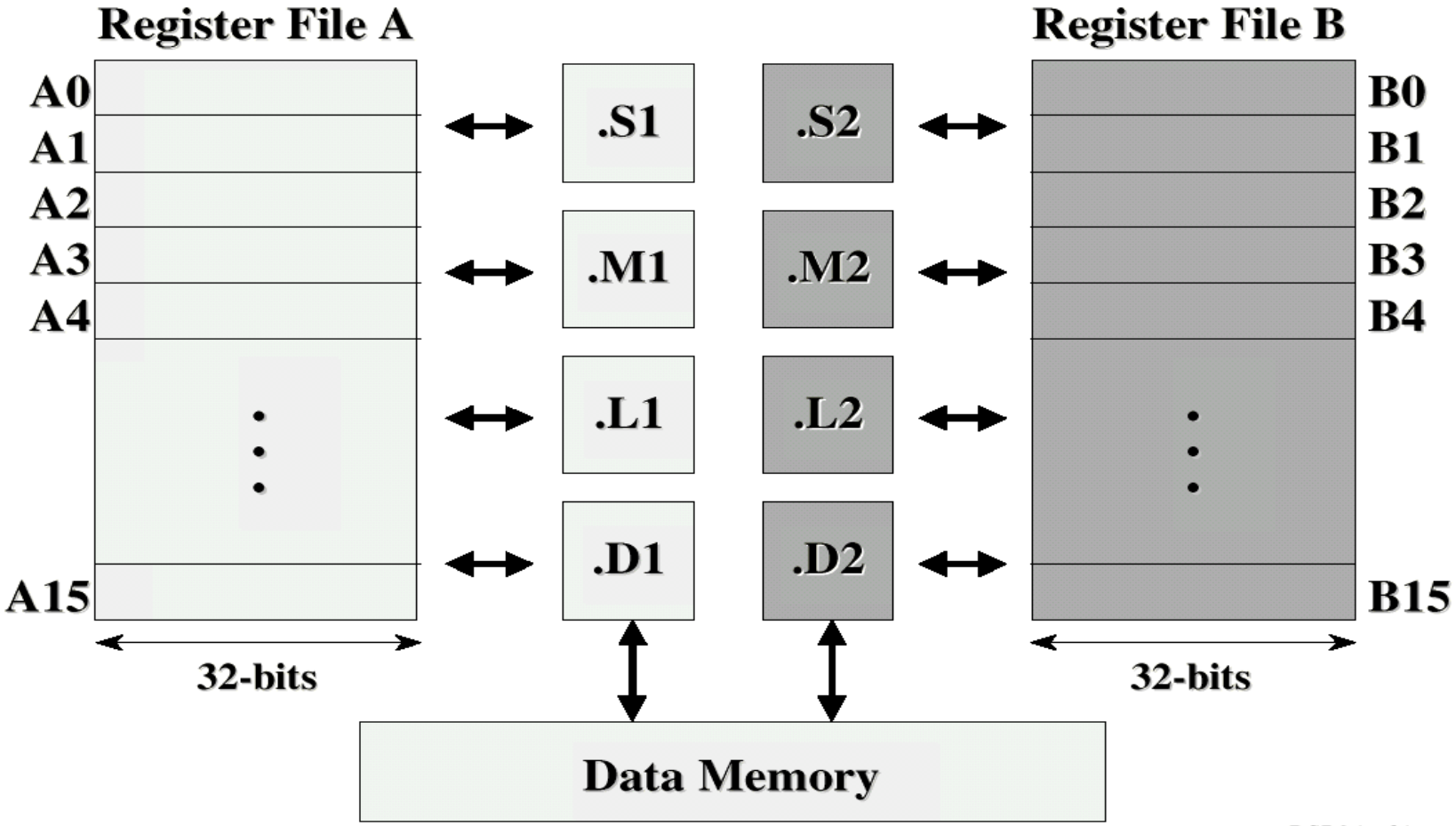


$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
        MVK   .S    40, A2
loop:   LD    .D    *A5++, A0
        LD    .D    *A6++, A1
        MPY   .M    A0, A1, A3
        ADD   .L    A4, A3, A4
        SUB   .L    A2, 1, A2
[A2]    B     .S    loop
        ST    .D    A4, *A7
```

实验：两个数组点积运算

- 另一套功能单元和寄存器



实验：两个数组点积运算

- 代码复习，只使用A侧

$$Y = \sum_{n=1}^{40} a_n * x_n$$

```
        MVK   .S1   40, A2          ; A2 = 40, loop count
loop:   LD    .D1   *A5++, A0       ; A0 = a(n)
        LD    .D1   *A6++, A1       ; A1 = x(n)
        MPY   .M1   A0, A1, A3      ; A3 = a(n) * x(n)
        ADD   .L1   A3, A4, A4      ; Y = Y + A3
        SUB   .L1   A2, 1, A2       ; decrement loop count
[A2]    B     .S1   loop            ; if A2 ≠ 0, branch
        ST    .D1   A4, *A7         ; *A7 = Y
```

汇编语言初步

汇编语言初步

学习内容

- 汇编代码的结构
- 汇编程序的构成
- 编写简单算法：y = mx+b
- C:\CCStudio_v3.3\docs\PDF\spru187n.pdf 第8.4节，讲了混合编程的调用关系例子

汇编语言初步

汇编代码的构成

注意：在输入汇编程序时，除标号以外的程序行都要以一个空格或Tab制表字符开始

```
label:     ||   [cond]   instruction   .unit
                operand   ;comment
```

- label: 标号：代码或变量地址
- ||: 条件寄存器
- [cond]: 条件寄存器
- instruction: 指令：
  - 助记符(mnemonic)
  - 伪指令(directive)
- operand: 操作数：
  - 寄存器
  - 常量
  - 指针
- ;comment: 功能单元(可选)
- .unit: 注释

```
x   .int   10
      MPY .M1   A1, A3, A7
    || ADD .L1x A2, B2, A5
```

汇编语言初步

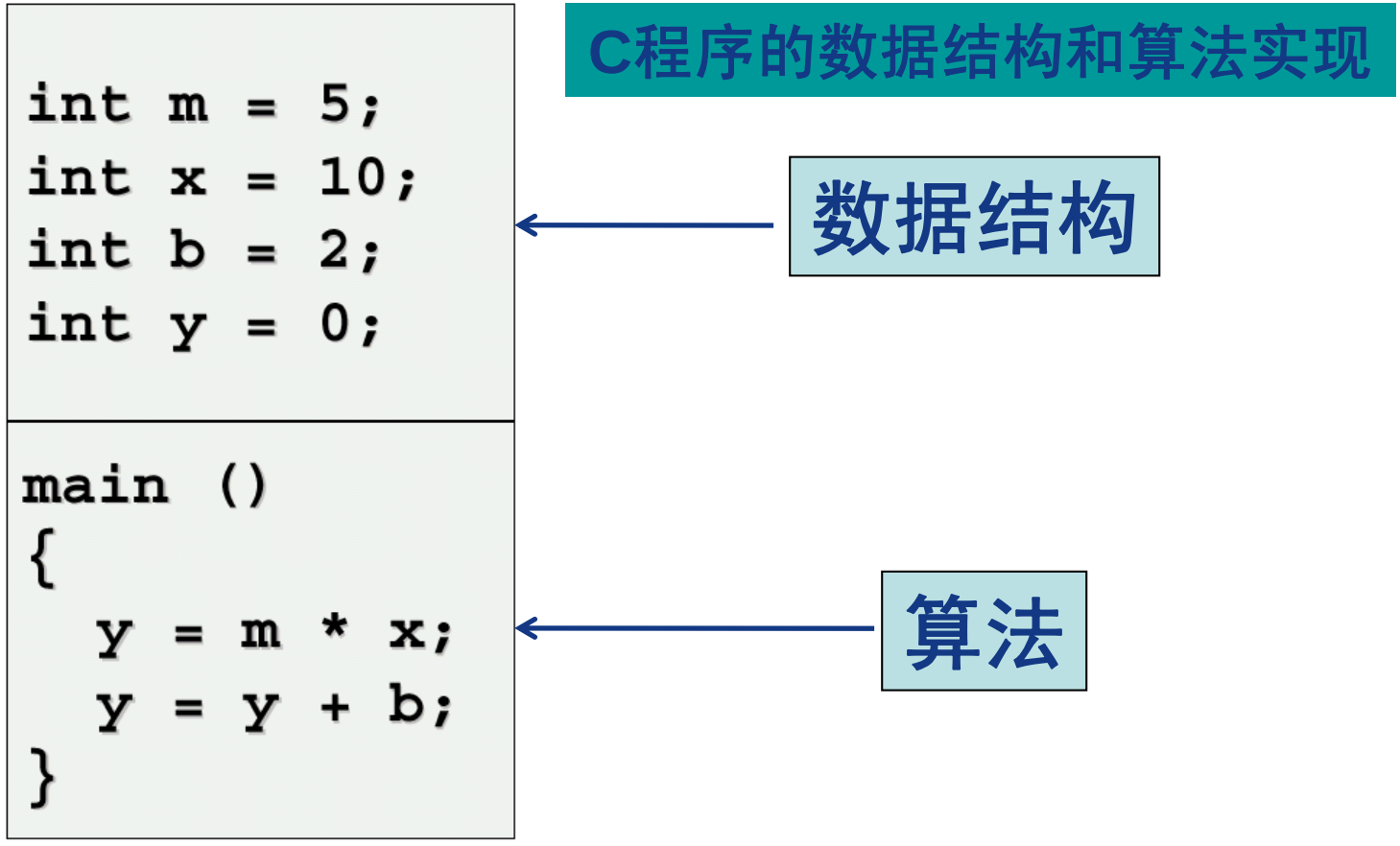
常用的伪指令

汇编指令	说明
.sect	定义一个代码段或数据段
.usect	定义一个未初始化数据段
.int .long .word	定义和初始化32位变量
.short .half	定义和初始化16位变量
.byte	定义和初始化8位变量

在C语言里long是40位，在汇编语言里long是32位

汇编语言初步

- 汇编程序的构成---程序＝数据结构＋算法



汇编语言初步

汇编程序-----数据结构

```
int m = 5;
int x = 10;
int b = 2;
int y = 0;
```

```
main ()
{
   y = m * x;
   y = y + b;
}
```

用汇编语言声明数据结构

```
        .sect "myData"

m       .int    5
x       .int    10
b       .int    2
y       .int    0
```

汇编语言初步

完整的汇编程序构成

```
        .sect  “myData”
   m      .int    5
   x      .int    10
   b      .int    2
   y      .int    0

        .sect  “myCode”
start   LD .D1  *A0,A1
                  .
                  .
                  .
        ST .D1  A7,*A6
end     B         end
          NOP     5
```

汇编语言初步

- 用汇编语言编写y=mx+b

1. 数据取入寄存器：m, x, b存储器—>寄存器
   1a. 初始化数据指针
   1b. 取数据
2. 乘法
3. 加法
4. 存储数据：y寄存器—>存储器

汇编语言初步

用汇编语言编写y=mx+b

1a. 初始化数据指针

32位常量

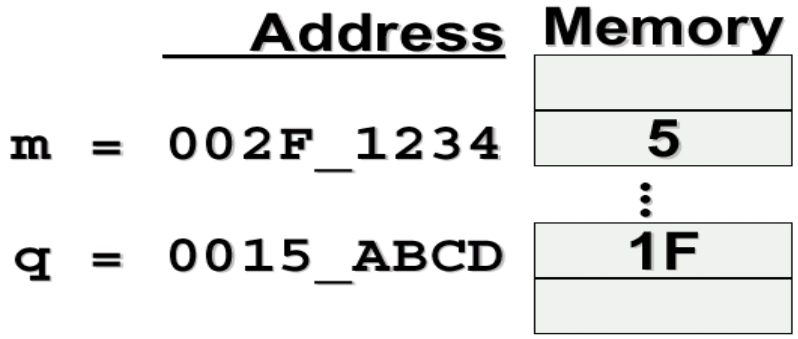
```
        MVK   .S1     m, A0
MVKH.S1       m, A0          ; &m—>A0
        MVK   .S1     x, A2
MVKH.S1       x, A2          ; &x—>A2
        MVK   .S1     b, A4
MVKH.S1       b, A4          ; &b—>A4
```

汇编语言初步

用汇编语言编写y=mx+b

- 指令MVK

MVK把一个16位常数放入寄存器

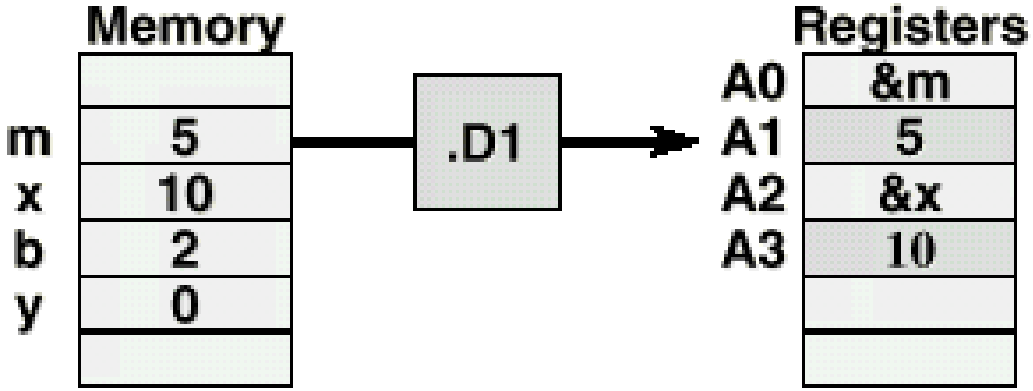


MVK Instruction			Value in A0 (hex)
			FFFF_FFFF
MVK	.S1	m,A0	0000_1234
MVKH	.S1	m,A0	002F_1234
MVK	.S1	q,A0	FFFF_ABCD
MVKLH	.S1	15h,A0	0015_ABCD

汇编语言初步

用汇编语言编写y=mx+b

1b 取数据




```
LDH .D1     *A0, A1     ; 取m
LDH .D1     *A2, A3     ; 取x
LDH .D1     *A4, A5     ; 取b
NOP         4
```

汇编语言初步

用汇编语言编写y=mx+b

- 用LD/ST指令
- 三种Load指令，对应不同长度的数据

  LDW　取32位字(word)
  LDH　取16位半字(short)
  LDB　取8位字节(byte)

  数据取入寄存器后进行符号扩展

- 对无符号数(字节、16位半字)

  LDBU
  LDHU

  无符号扩展

- 指令延迟：四个延迟间隙
- 三个存储指令　　STW　　STH　　STB

汇编语言初步

用汇编语言编写y=mx+b

- 指令延迟间隙

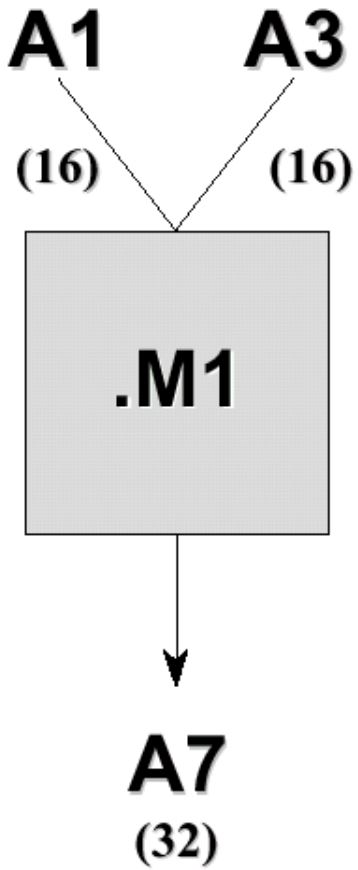
延迟间隙：多周期指令所需要插入的NOP指令个数

指令类型	延迟间隙
单周期	0
乘法	1
Load	4
跳转	5

汇编语言初步

用汇编语言编写y=mx+b

2. 乘法




```
MPY .M1 A1, A3, A7
NOP
```

汇编语言初步

用汇编语言编写y=mx+b

乘法指令

- 四种乘法指令

MPY(U/US/SU)	16LSB×16LSB
MPYH(U/US/SU)	16MSB×16MSB
MPYH(U/S)L(U/S)	16MSB×16LSB
MPYL(U/S)H(U/S)	16LSB×16MSB

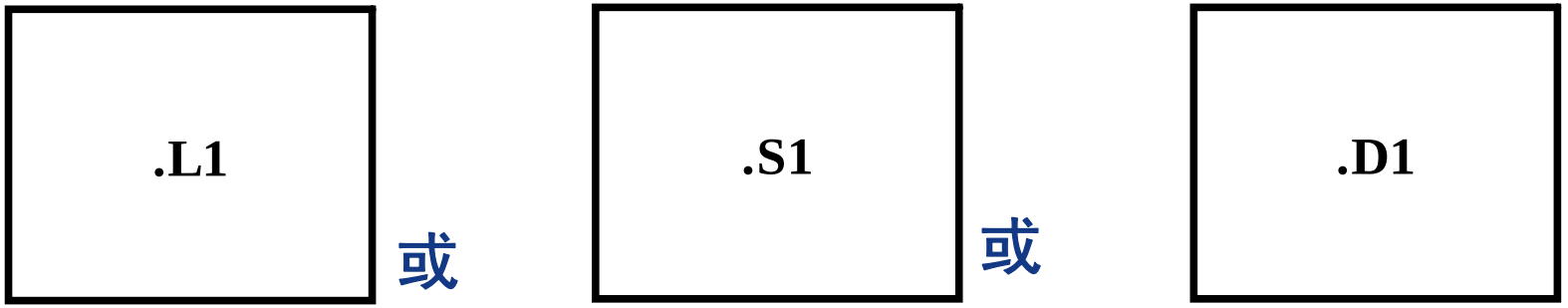
- **指令延迟槽：1**
- **两个乘法单元可以在一个周期内做两次乘法**

汇编语言初步

用汇编语言编写y=mx+b

3. 加法

ADD.?　应该使用哪个功能单元？

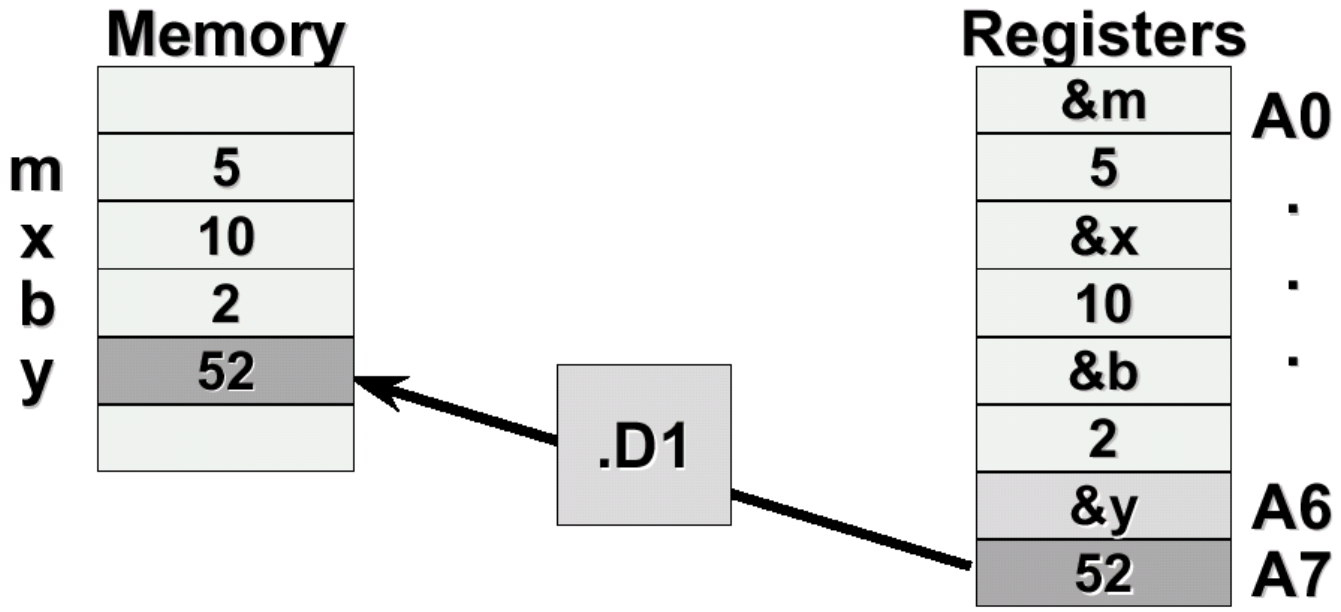



```
ADD .?      A5, A7, A7
```

汇编语言初步

用汇编语言编写y=mx+b

4. 加法




```
STW   .D1   A7,*A6   ; Store result to memory
                     ;      A7 → *A6
```

汇编语言初步

用汇编语言编写y=mx+b

- 完整的y=mx+b汇编程序spru189f.pdf

```
          .title "lab4.asm"
/*定义数据结构*/
          .sect "myData"
m         .short 10
x         .short 5
b         .short 2
y         .short 0
/*算法*/
          .sect "myCode"
/*指针初始化*/
init:     mvk .s1 m,A0      ;A0=&m
          mvkh.s1 m,A0
          mvk .s1 x,A2      ;A2=&x
          mvkh .s1 x,A2
          mvk .s1 b,A4      ;A4=&b
          mvkh .s1 b,A4
          mvk .s1 y,A6                ; A6=&y
          mvkh .s1 y,A6
/*取数据*/
          ldh .d1 *A0,A1              ; A1= m
          ldh .d1 *A2,A3              ; A3=x
          ldh .d1 *A4,A5              ; A5=b
                    nop 4
/*核心算法*/
start:    mpy .m1 A1,A3,A7  ; A7=mx
          nop 1
          add .l1 A5,A7,A7    ; A7=mx+b
          sth .d1 A7,*A6
/*结束循环*/
          b $                 ; endless loop
          nop 5
```

线性汇编

线性汇编

在C程序中使用汇编语句

汇编指令和命令可以嵌入在C程序中，并用asm语句声明，该语句可访问硬件，而只用C语言很难直接访问硬件。

asm （“assembly code”）

编写可被C程序调用的汇编函数

可用extern语句作为函数的外部声明，如： extern int func()

1. 建立C环境
2. 建立汇编环境
3. 编写线性汇编函数
   - 入口代码
   - 算法
   - 出口代码

线性汇编

编写可被C调用的线性汇编函数

1. 建立C环境

1）在C程序前声明线性汇编函数

2）调用汇编函数

```
Parent.C

int child(int, int);    /* prototype */
void main (void)
{   int x = 7;
    int y;
    y = child(x, 5);   /* call function */
}
```

1)声明

2)调用

线性汇编

编写可被C调用的线性汇编函数

2. 建立汇编环境

1）汇编函数的入口地址声明为全局变量，

汇编函数的入口地址—C函数名前加下划线；

2）定义函数入口地址。

1)声明 →

2)定义 →

```
;Child.ASM

          .def   _child   ;make the label_func
                          ;visible to linker

_child:                   ;define function's
          instr           ;entry point
          instr
          instr
          .
          .
```

线性汇编

编写可被C调用的线性汇编函数

3. 编写线性汇编函数

入口代码

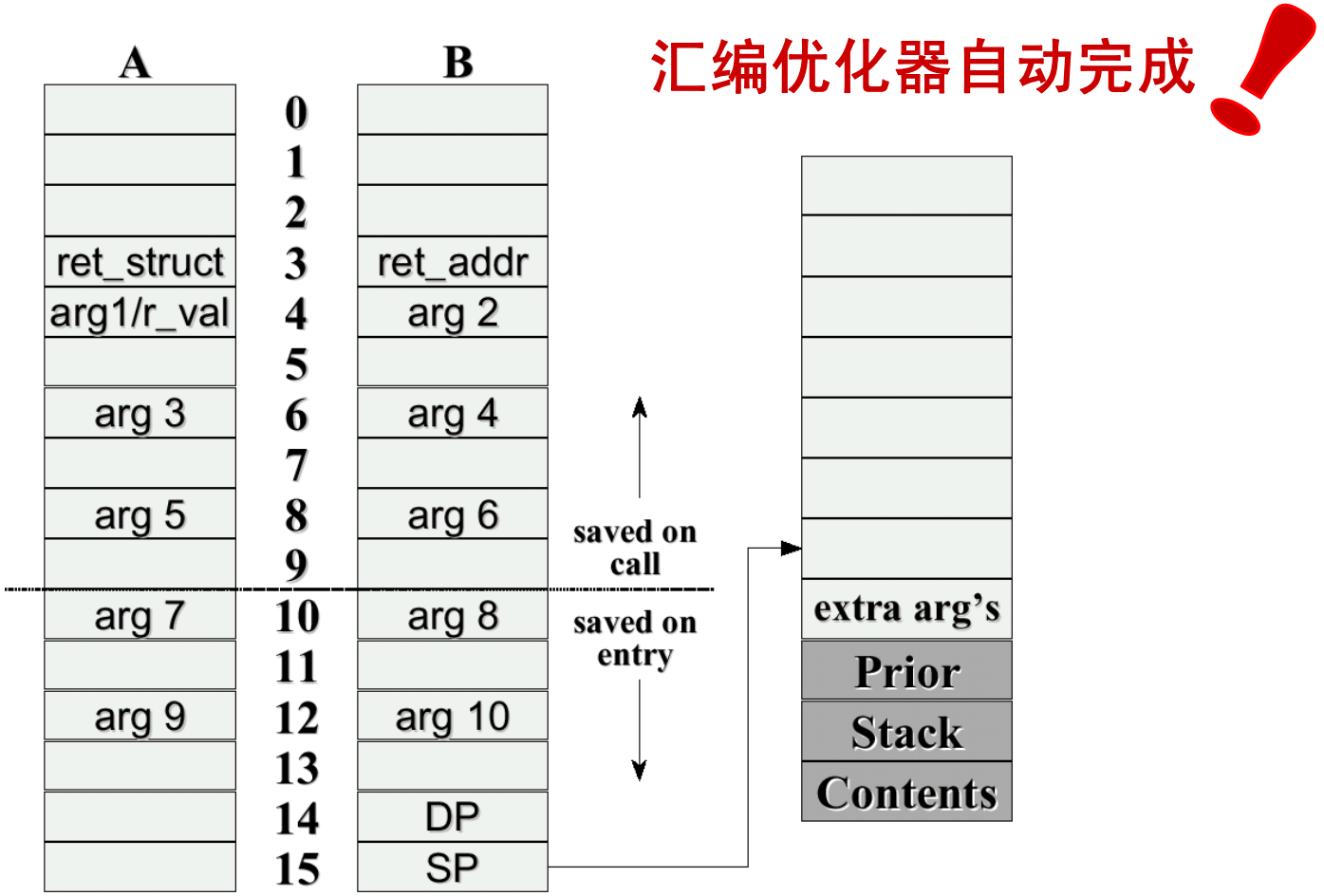
```
.def     _DP                              ; 声明函数名，建立汇编环境
_DP:.cproc  param0, param1                ; 函数参数
    .reserve reserve_reg                  ; 需要保护的寄存器
    .reg     local_val0, local_val1, ...  ; 函数内的寄存器变量
    MV       param0,local_val0            ; 建立函数参数与
    MV       param1,local_val1            ; 寄存器变量名的关系
             .                            ; 算法
             .
             .
```

出口代码

```
    .return      return_value            ; 函数返回，return_value为
                                         ; 物理寄存器或寄存器变量名
    .endproc
```

线性汇编

线性汇编的寄存器保护问题



线性汇编

实验1

- 实验目的：

  学习和掌握利用intrinsics进行字长优化；

- 实验内容

  改写C语言点积函数，用强制类型转换法实现字优化。

实验2

- 实验目的：

  掌握线性汇编语言和C语言混合编程的方法；

- 实验内容

  用线性汇编改写C的点积函数。

TI DSP培训以及技术服务简介

上海交大BME-美国德州仪器联合DSP实验室成立于2007年，是国内最权威的TI技术服务于培训机构。实验室有TI（C6000，C2000，C5000，达芬奇，多核DSP）全系列开发平台，提供DSP ，MSP430等技术培训与技术服务，项目合作等。培训内容有

1）CCS开发环境精解与实例；

2）DSP/SYS BIOS 实例；

3）C6000/C5000/C2000全系列DSP架构以及汇编，C语言，混合编程等；

4）HPI，EMIF，EDMA，Timer等外设；

5）C6416、DM642， C6678多核EVM开发平台实例；

6）Bootloader 原理以及实例等。

常年开班，三人以上集体报名8折优惠，学生5折。

联系电话：13651621236（牛老师），

邮件报名：jhniu@sjtu.edu.cn ，niujinhai@yahoo.com.cn

颁发TI授权的培训证书

TI DSP培训以及技术服务简介

上海交大BME-美国德州仪器联合DSP实验室成立于2007年，是国内最权威的TI技术服务于培训机构。实验室有TI（C6000，C2000，C5000，达芬奇，多核DSP）全系列开发平台，提供DSP ，MSP430等技术培训与技术服务，项目合作等。培训内容有

1）CCS开发环境精解与实例；

2）DSP/SYS BIOS 实例；

3）C6000/C5000/C2000全系列DSP架构以及汇编，C语言，混合编程等；

4）HPI，EMIF，EDMA，Timer等外设；

5）C6416、DM642， C6678多核EVM开发平台实例；

6）Bootloader 原理以及实例等。

常年开班，三人以上集体报名8折优惠，学生5折。

联系电话：13651621236（牛老师），

邮件报名：jhniu@sjtu.edu.cn ，niujinhai@yahoo.com.cn

颁发TI授权的培训证书

DSP实验室介绍

- 美国德州仪器（TI）—上海交通大学（SJTU）联合DSP实验室成立于2007年10月，位于上海交大闵行校区，致力于TI DSP技术的推广，以及相关数字信号处理算法的研究与开发，为客户提供优质的产品与服务，涉及的技术领域有，无线通信，音频/视频信号处理，医学信号/图像处理，数字马达控制等。实验室研发与培训教师主要由上海交通大学青年教师承担，同时聘请了多位有企业工作背景的DSP技术专家为实验室的顾问。

DSP实验室介绍


DSP联合实验室成立暨揭牌仪
Texas Instruments - Shanghai Jiao Tong University
DSPS Laboratory
上海交通大学－美国德州仪器
数字信号处理方案实验室

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

DSP实验室介绍

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

End

Thanks