第九章　AVR C 语言的应用

★ 更详细资料参阅光盘文件<< AVR C 语言的应用>>

9.1　AVR – 支持 C 和高级语言编程的结构

高级语言

- **提高了 MCU 的重要性–上市的时间**

–简化维护工作

–轻便

–学习时间

–可重用性

–库

- **潜在的缺点**

–代码大小

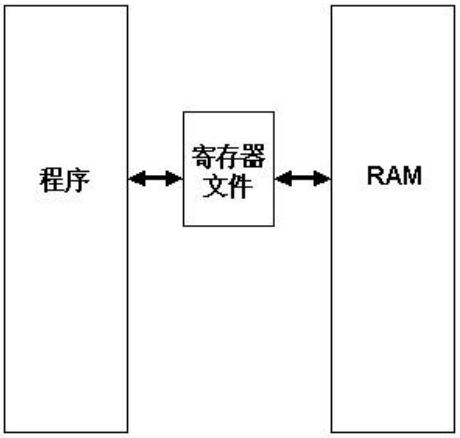
–执行速度

为什么 AVR **适宜用高级语言编程**?

因为它是为高级语言而设计的!

IAR 对 AVR 结构和指令集的影响

- **在结构/指令集确定之前，编译器的开发就开始了**
- **潜在的瓶颈得到确认并消除**
- IAR **的反馈在硬件设计上得到了反映**
- **几次循环反复**
- **修改后的结果从代码当中可看出来**



Memory system

- 32 **通用寄存器文件**

–数量多

–直接 与 ALU 连接

–可保存变量，指针和之间结果

- **线性程序存储空间**

–1KBytes - 8MBytes

–无需页寻址

–常数区(SPM 可修改)

- **线性数据存储空间**

–16 MBytes

–无需页寻址

类似于 C 的寻址模式

C **源代码**

```
unsigned char *var1, *var2;
*var1++ = *--var2;
```

产生的代码

```
LD  R16,-X
```

ST Z+,R16

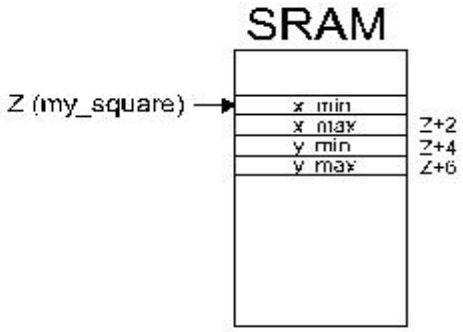
带偏移量的间接寻址

•有效访问数组和结构

•Auto (local variables)**放置于软件堆栈之中**

–为适应重入的要求，高级语言都基于堆栈结构

```
struct square
{
    int x_min;
    int x_max;
    int y_min;
    int y_max;
} my_square;
```

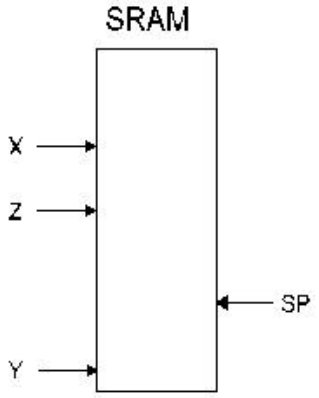


四种指针

16 和 32 位支持

- **加法指令**
  - **–加和减**
  - **–寄存器之间**
  - **–寄存器和立即数之间**
  - –Zero **标志的传播**

```
SUB     R16,R24
SUBI    R16,1
SBC     R17,R25      SBCI    R17,0
```

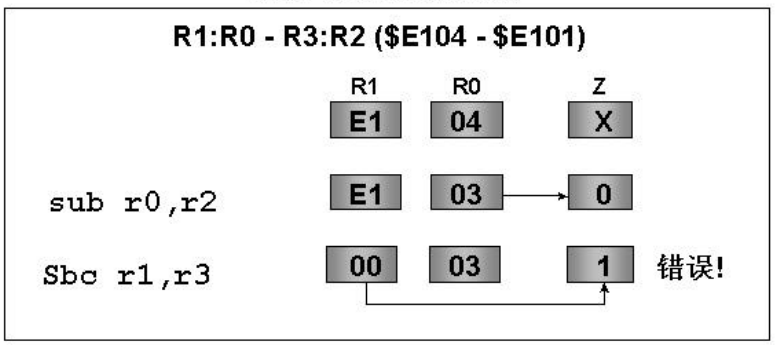


- 从内存拷贝到内存
- 最小化指针的反复加载
- 高功能（HIGH FUNCTIONALITY）

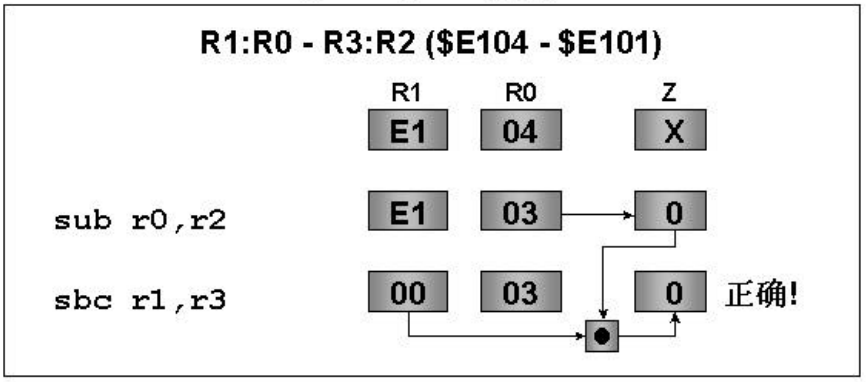
所有的跳转都基于最后结果

两个 16 位数相减

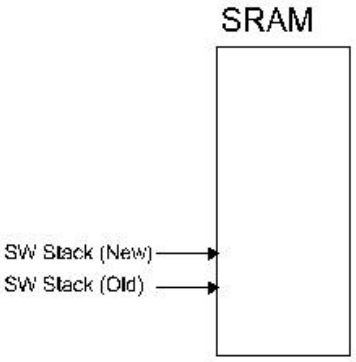
没有Zero标志的传播



有Zero标志的传播



16 位指令



- 小数值的加减
- 适合于索引变量和指针操作
- 软件堆栈

Non-destructive comparison

```
CP      R16,R24
CPC     R17,R25
CPC     R18,R26
CPC     R19,R27
```

- **带进位比较**
- Zero **传播**
- **无需保存结果**
- **可使用所有形式的跳转**

Switch 支持

- Switches **在** CASE **语句中经常遇到**
- Straight forward approach **效率低**
- **间接跳转适合于紧凑的** switch **结构**
- switch **由通用库管理**

摘 要

- AVR **结构从一开始就是针对高级语言设计的**

- Atmel 与 IAR 在结构和指令调整上的合作
- 从而编译器可以产生高效的代码

Efficient C-coding for AVR

减少代码的提示和诀窍汇编(Assembly) 与 C 比较

汇编:

- 可以完全控制资源
- 在小应用当中可以产生紧凑的、高速的代码
- 在大的应用当中代码效率低
- 可读性差（Cryptic code）
- 不好维护
- 不易移植（Non-portable）

C:• 对资源的控制有限

- 在小应中产生的代码量大，执行速度慢
- 在大的应用当中代码效率高
- 结构化的代码
- 容易维护
- 容易移植

访问 I/O

- **读 I/O:** temp = PIND;

```
IN      R16,LOW(16)
```

- **写 I/O:** TCCR0 = 0x4F;

```
LDI     R16,79
OUT     LOW(51),R16
```

- **I/O 的位设置与清除**
- **地址小于 0x1F 的 I/O:**

```
PORTB |= (1<<PIND2);
  SBI     LOW(24),LOW(2)
ADCSR &= ~(1<<ADEN);
  CBI     LOW(6),LOW(7)
```

- **地址高于 0x1F 的 I/O:**TCCR0 &= ~(0x80);

```
IN      R16,LOW(51)
ANDI    R16,LOW(127)
OUT     LOW(51),R16
```

测试 I/O 的单个位

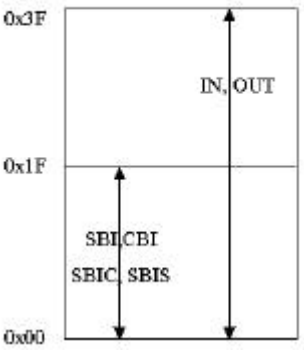
- **等待地址低于 0x1F 的单个位的清除**

```
while(PIND & (1<<PIND6));
  SBIC    LOW(16),LOW(6)
  RJMP    ?0002
```

- **等待地址高于 0x1F 的单个位的设置**

```
while(!(TIFR & (1<<TOV0)));
```

AVR I/O 内存映像



```
IN      R16,LOW(56)
SBRS    R16,LOW(0)
RJMP    ?0004
```

16 位变量• 总是使用最小的数据类型

- **8 位计数器:**

```
char count8 = 5;
do{
}while(--count8);
LDI     R16,5
DEC     R16
BRNE    ?0004
```

- Total 6 bytes

- **16 位计数器:**

```
int count16 = 5;
do{
}while(--count16);
LDI     R24,LOW(5)
LDI     R25,0
SBIW    R24,LWRD(1)
BRNE    ?0004
```

Total 8 Bytes

全局和局部变量

- **全局变量**
  - **在** startup **初始化**
  - **存储于** SRAM
  - **必须加载到寄存器文件中**
- **局部变量**
  - **在函数初期初始化**
  - **存储于寄存器当中直至函数结束**

全局变量和局部变量

- **局部变量**

```
void main(void)
{
   char local;
   local=local - 34;
}
   SUBI    R17,LOW(34)
```

- Total 2 bytes

- **全局变量**

```
char global;
void main(void)
{
```

```
    global=global - 45;
}
LDS   R16,LWRD(global)
SUBI R16,LOW(45)
STS   LWRD(global),R16Total 10 Bytes
```

高效地使用全局变量

- **将全局变量收集到一个结构中:**

```
typedef struct {
    int   t_count;
   char       sec;          // global seconds
   char       min;          // global minutes
}t;
t time;
Void main(void)
{
    t *temp = &time;
    temp->sec++; temp->min++; temp->t_count++;
}
```

带参数的函数调用

- **使用参数将数据传递到函数中去**

```
char add(char number1, char number2)
{
return number1+number2;
}
```

函数间参数的传递通过 R16-R23 **来实现**

循　环

- **死循环** for(; ;)

```
{
}
```

- **循环** char counter = 100;

```
do{
} while(--counter) ;
```

预减变量（Pre-decrement）**代码效率最高**

优化代码的选项

- **代码大小优化编译**
- **使用局部变量• 使用允许的最小数据类型**
- **将全局变量收集到结构中去• 死循环使用** for(;;)
- **使用预减的** do{ } while;

C　AVR 的程序设计

内　容

- **安装必须的工具**