微机原理补充

1 CPU模块

1.1 CPU寄存器

2812芯片大体上分为：中央处理单元（CPU），存储器和外设。其中的CPU寄存器如下（大致）：

```
CPU寄存器
    |- 读写改集成的累加器模块 ALU (算数逻辑模块)
            |- 加法器 ACC(32) = AH(16) + AL(16)
    |- 32位乘法器模块
            |- 乘数寄存器(暂存) : XT
            |- 乘法寄存器(结果) : P(32) = PH(16) + PL(16)
    |- 地址寄存器算数单元 ARAU(32) : 地址寄存器算术单元
            |- 辅助寄存器 AR(8个×32位) : XARn (n=0~7) = 高16位 + ARn (低16位)
            |- 数据页指针 DP(16) : 直接寻址(22,4M) = DP(16) + 偏移量(6)
    |- 堆栈指针 SP(16) : 偏移量小于64，只能访问[0,0xffffH]低地址空间单元，复位0x0400
    |- 程序计数器 PC : 总是包含到达D2阶段指令的地址
    |- 指令计数器 IC : 装入下一条指令的地址，保持到下个D2阶段
    |- 状态寄存器 ST0/ST1
            |- 溢出位 V (Overflow): 1为有溢出，用于判断有符号运算是否出错
                                    溢出一旦置位不会被下一次运算清除，会一直保留
                                    比较运算CMP不会溢出（溢出的要求是“存入寄存器”）
            |- 进位/借位位 C (Carry): 用于判断无符号数高低，对有符号数而言不重要
                    |- 0 : 无进位/有借位
                    |- 1 : 有进位/无借位
            |- 负标志位 N (Negative): 1为有负数产生,用于有符号数判断大小
                    CMP比较时，看计算结果的真实值（不怕溢出）
                    SUB减法时，看计算机操作、存储的结果，默认读最高位
            |- 零标志位 Z (Zero): 1为有0产生
            |- 符号扩展模式位 SXM : 0-无扩展，1-符号扩展
            |- (ST1) - 辅助寄存器指针 ARP
    |- 其他 (如中断控制寄存器IFR,IER等)
```

1.2 总线

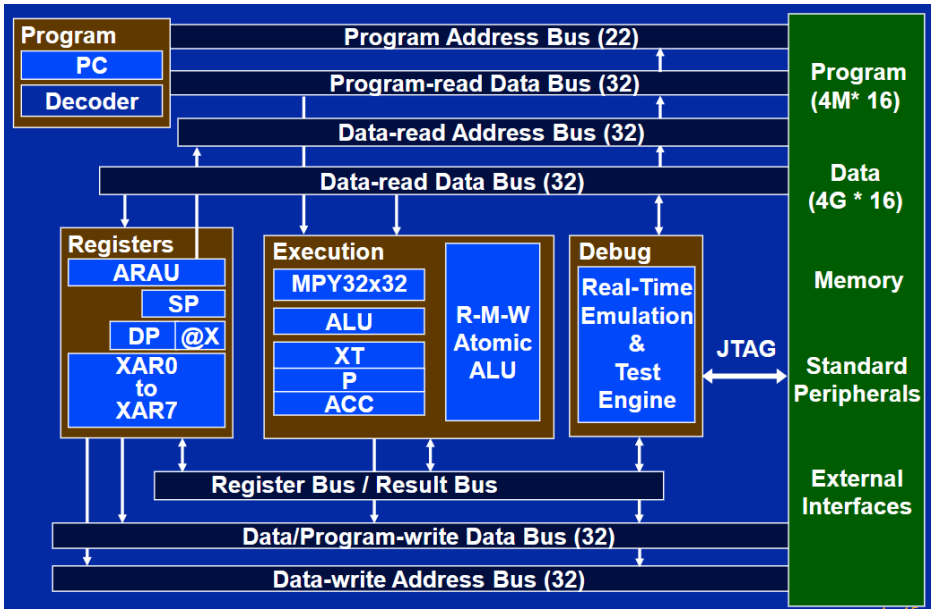
2812的存储器空间被分为了两大块：程序空间（P）和数据空间（D）。

访问任一空间，都需要两种总线配合：地址总线（A）和数据总线（D），前者终于传送存储单元的地址，后者用于传送存储单元的具体内容。

- 下面对英文缩写做一点简单的说明：

```
P : Program 程序(空间)
A : Address 地址(总线)
D : Data    数据(空间或总线)
```

R/W ： Read/Write 读取/写入 省略表示两者均可
B ： Base 总线



- 下面就是2812CPU对存储空间内数据的调用的过程

```
存储空间          程序空间P                数据空间D
                     |                  |-----|-----|
                     |                  |           |
地址总线A          PAB(22)           DWAB(32)     DRAB(32)
                     |-------|          |            |
                     |       |------|   |            |
数据总线D          PRDB(32)          D/PWDB(32)   DRDB(32)
```

可以看到：

1. 一共有6跟总线，三根数据总线，3根地址总线
2. 地址总线A少一根的原因是程序空间P的读写共用一根总线（因此程序的读写不能同时进行）
3. 数据总线D少一根的原因是程序、数据空间共用一根总线进行数据的写入

1.3 指令流水线

2812的指令流水线大体上遵循一下几点规则：

1. 一条指令最多分8步完成
2. 每一步都要一个时间间隔完成
3. 同一时间最多可能有8条指令在执行
4. 步与步之间可能会插入间隔（如当此步需要用到上一步的结果，就会等待）

流水线的8步分别为：

```
    获取指令地址 --> 获取指令内容 --> 对指令进行解码 --> 解析操作数地址
--> 锁定操作数地址 --> 获取操作数 --> CPU执行'real work' --> 将结果存入到内存
```

2 2812指令系统——寻址

2.1 寻址的基本操作

2812采用增强型哈佛总线结构，能够并行的访问地址和数据存储空间，其寻址的范围为`[0,0x3fffff]`（即2^{22}=4M）。介绍寻址之前，大致先了解一下`MOV`指令，大意就是：`MOV A B`近似于把B放到A里面，这也是最重要的汇编语言之一。

寻址方式大致上的分类有以下3种：

寻址方式	含义	写法
立即寻址	直接对应数字本身	#
直接寻址	某一个单元格内对应的数字	@
间接寻址	指向某个单元的指针对应的单元中的数字	*

下面介绍一些常用的寻址方法及其写法（下文中`loc16`表示任一16位的地址/数据存储单元）：

1. **立即数寻址**（把数据送到寄存器）

   (a) 数据不能太大，一般为16bit，只有XAR可以接受22bit的数据
   (b) ACC不能直接接受更多位的立即数

```
MOVL XARn, #22bit
MOVW DP, #16bit
MOVW DP, #16bit >> 6
MOV SP, #16bit
MOV ACC, #16bit
```

2. **偏移量直接寻址**（DP的使用，好用的）

```
MOVW DP, #0x3F9000 >> 6    // 把3F9000“赋值”给DP
MOV @4, #16bit             // 偏移量=4，把某个数16bit“赋值”给3F9004单元
```

因为DP寻址方式是用16位DP“并上”6位偏移量来实现的，因此为了避免繁杂的计算，在给DP输入时要把`0x70D4`右移6位，直接移到偏移量的位置上，就可以把“并”换成简单的加法。

3. **堆栈间接寻址**（SP的使用）

```
MOV SP, #0x70DF          // 把70DF“赋值”给SP
MOV *-SP[5], #16bit      // 偏移量=-5，*表示C语言的指针，把某个数16bit“赋值”给70DA单元
```

SP就是只能用负的偏移量，也就是`-SP[n]`；

其中`n=#6bit`，即 $n \leq 2^6 - 1 = 63$

4. **寄存器间接寻址**（AR/XAR的使用）

```
MOV XAR6, #0x70D0        // 把70D0“赋值”给XAR6
MOV *+XAR6[4], #16bit    // 偏移量=4，把某个数16bit“赋值”给70D4单元
```

AR/XAR就是只能用正的偏移量，也就是`+XARn[m]`；

其中`m=#3bit`，即 $m \leq 2^3 - 1 = 7$

上述方法2/3/4的第二行都属于`MOV loc16 #16bit`的形式

5. **寄存器直接寻址**（说实话我看不太懂）

```
MOVL ACC, @XAR2        // MOV AX, loc16
MOVL @6, ACC           // MOV loc16, AX
MOVL T, @AL            // MOV loc16, AX
```

ACC(32)=AH(16)+AL(16)，这里用AX指代AH或AL中的任意一个，是汇编里面比较特殊的一个量

AL的目的之一就是作为中转站，来代替被禁止的 `MOV loc16 loc16`

6. **空间立即寻址**

```
MOV *(0:16bit),loc16     // 这个是通式
MOV *(0:0x70D4), @AX     // 这个是例子
```

- 以下是两种被**禁止**的寻址写法：
  - `MOV loc16 loc16`，代表为：`MOV DP XAR`，但是 `ACC(AH/AL)` 除外
  - `MOC *(0:0x16bit), #16bit`，代表为：`MOV *(0:0x70D4), 0xFF00`
- 一般对于 `MOV A B` 而言，若A/B表示寄存器（DP，XAR等）时，A中不加@，B中加@（大部分情况下）

2.2 寻址例程

```
// Gpfmux=0xF0FF (Gpfmux地址为0x70D4)
/* Code 1 */
MOVW DP,#0x01C3    // 01C3 = 0000 0001 1100 0011
MOV @20, #0xF0FF   // 01C3 & 20 = 00 0000 0111 0000 1101 0100
/* Code 2 */
MOV AL,#0xF0FF
MOV *(0:0x70D4),@AL  // 利用中间变量AL进行空间立即寻址
/* Code 3 */
MOVL XAR2,#0x70D0
MOV *+XAR2[4],#0xF0FF
```

3 汇编语言

3.1 常用汇编语言

1. **传送指令**

```
MOV / MOVL (32bit) / MOVW (专用于DP) / MOVU (高位0扩展，常用于ACC)
```

2. **简单指令集**（AX=AH或AL）

```
/* 加法 */ ADD AX, loc16  // AX = AX + loc16
/* 减法 */ SUB AX, loc16  // AX = AX - loc16
/* 比较 */ CMP AX, loc16  // 根据比较结果置位，但不会改动数值
                          // 如置零位ZF，符号位SF
/* 与 */  AND AX, loc16  // AX = AX & loc16
/* 或 */  OR AX, loc16   // AX = AX | loc16
/* 取反 */ NEG AX, loc16  // AX = -AX
```

上面的指令后面多一个B表示短指令，如 `ANDB AX #8bit`

3. **移位指令**（可以代替以2为倍数的乘除法）

```
LSL AX, #16bit  // 逻辑左移(unsigned)
LSR AX, #16bit  // 逻辑右移(unsigned)
ASR AX, #16bit  // 算数右移(signed)，保留符号位
                // 例: LSR AL, #4 表示 AL = AL / 16
```

4. **重复执行**（常用于除法，要背除法的代码段）

```
RPT #N-1    // 重复下一行N次
|| ....     // 只能写简单的代码，比如加、减等
```

5. **乘法**（写法很多，只介绍其中一种）

```
MPYB P,T,#8bit   // P (signed 32) = T (signed 16) * 8bit (unsigned 8)
            // P (32) = PH (16) + PL (16)
```

6. **条件减法**（常用于除法，要背除法的代码段）

```
SUBCU ACC, loc16   // ACC = ACC 条件减 loc16，具体原理不想写了
```

7. **条件指令**（好多啊，这咋记得住啊）

```
B CODE, LOGIC    // B代表条件指令，SB短跳8bit，LB长跳22bit
                 // CODE是代码块的名称，可以理解为C或Py的函数
                 // LOGIC表示条件，满足跳转，不满足不跳
```

条件列表：

中文名	条件名	翻译	具体判断标准
不等于	NEQ	Not Equal To	Z = 0
等于	EQ	Equal To	Z = 1
大于	GT	Greater Then	Z = 0 AND N = 0
大于等于	GEQ	Greater Then Or Equal To	N = 0
小于	LT	Less Then	N = 1
小于等于	LEQ	Less Then Or Equal To	Z = 1 OR N = 1
高于	HI	Higher	C = 1 AND Z = 0
高于等于	HIS,C	Higher Or Same, Carry Set	C = 1
低于	LO,NC	Lower, Carry Clear	C = 0
低于等于	LOS	Lower Or Same	C = 0 OR Z = 1
未溢出	NOV	No Overflow	V = 0
溢出	OV	Overflow	V = 1
我不到啊	NTC	Test Bit Not Set	TC = 0
我不到啊	TC	Test Bit Set	TC = 1

8. **自增**（i++，这个不用背）

```
INC loc16  // 如 INC @2
```

3.2 汇编代码块

1. **除法**（需要记）

```
/* 无符号数除法 */
/*
  Num ÷ Den = Quot ... Rem
  商: Quot = Num / Den
  余数: Rem = Num % Den
*/
MOVU ACC, @Num    // AH = 0, AL = Num,
RPT #15          // 16bit, Repeat 15 times
||SUBCU ACC, @Den  // 条件减法
MOV @Rem, AH     // 余数存在高位AH, 移到Rem里
MOV @Quot, AL    // 商存在低位AL, 移到Quot里
```

2. 代码块的调用

```
/* 代码块的调用—类似于C与Py的函数 */
TODO  // PRE CODE
LC NAME  // 调用, LC = CALL
TODO  // POST CODE

NAME:
  TODO  // FUNCTION
```

3. 考试的代码块要求

```
/* 考试的代码块要求 */
/*
  1. 有标号, 或者"函数名"
  2. 以LRET结束
*/
/* 示例 */
TEST:
  TODO  // Code Here
  LRET
```

3.3 汇编实例

例1

```
/*
  将9000H单元中的16进制数(<99)转为8421BCD码存入9002H单元中
  如: 51H = 81 -> 81H
  实现方法: 51H/10 = 8...1 -> 8*16+1 = 81
*/
  MOVW DP, #0x9000 >> 6
  MOVU ACC, @0
  LC HEX2BCD
  MOV @2, AL

HEX2BCD:
  MOV T, #10
  RPT #15
  ||SUBCU ACC, @T  // 除法: 高位AH=余数, 低位AL=商
  LSL AL, #4    // AL = AL(3..0) * 16
  ADD AL, @AH    // AL = AL + AH
  LRET
```

例2

```
/*
  将9004H单元中的8421BCD码转为16进制数存入9006H单元
  如: 56H -> 56 = 38H
  实现方法: 56H -> 5*10+6 = 56 = 38H
```

```
*/
BCD2HEX:
  MOVW DP, #0x9000 >> 6
  MOVU ACC, @4  // ACC = 0 0 | 5 6
  MOV AH, @AL  // ACC = 5 6 | 5 6
  ASR AH, #4  // AH = 0 5
  AND AL, #0x0F  // AL = 0 6
  MOV T, @AH
  MPYB P,T,#10  // P = AH * 10
  MOV AH, @PL  // AH = 50
  ADD AH, @AL  // AH = 50 + 6
  MOV @6, AH
  LRET
```

例3

```
/*
  程序阅读题，求下列代码执行后，V、C、N、Z 的值
  并说明最终结果和存放位置
*/
MOV SP, #0x420
MOV *-SP[10], #0x10
MOV AL, #0x12
SUB *-SP[10], AL
```

- 分析：
  - `*-SP[10]` 给单元 `*(0x416)` 赋值 `0x10`
  - 减法SUB得到 `0x10 - 0x12 = -0x2 = 0xFFFE`，因此答案为：
  1. V保持不变，原来是1就是1，原来是0还是0
  2. 零位Z=0（因为结果不是0）
  3. 负位N=1，因为结果是负数（最高位是1就是负数）
  4. 进位/借位位C=0，减法C=0表示有借位
  5. 最终结果为 `0xFEFF`，存放在 `0x416` 的地址中

例4

```
/*
  *(Uint *)0x3F9008 = (*((Uint *)0x3F9002)) / 5 + 6
*/
AAAA:
  MOVL XAR4, #0x3F9002
  MOVU ACC, *+XAR4[0]
  MOVW T,#5
  RPT #15
  ||SUBCU ACC, @T
  MOV AH, #0x6
  ADD AH, @AL    // AL 和 @AL 应该是一样的
  MOV *+XAR4[6], AH
  LRET
```

例5

```
/*
  *(Uint *)0x3F9008 = (*((Uint *)0x3F9002)) * 15 + 6
*/
AAAA:
  MOVW DP, #0x3F9002 >> 6
  MOV AL, @2
  MOV T, @AL
  MPYB P, T, #15
  MOV AL, #6
  ADD AL, @PL
  MOV @8, AL
  LRET
```

例6（这个太难了考试不会考的）

```
/*
  数组求和，求给定数组前10个元素之和，C代码如下：
*/
int k = 0,i;
int m[10];
for (i=0;i<10;i++) k += m[i];
/* 下面是汇编语言写法 */
/* int k = 0,i; */
  MOVW DP, #0x3F9000 >> 6  // 设i为@0, k为@1
  MOV @1, #0  // k = 0
/* int m[10] */
  MOV XAR4, #0x3F9040  // 设为数组初始地址
/* for(i=0;i<10;i++) k += m[i]; */
  MOV @0, #0  // i = 0
  MOV AL, @0
L1:
  MOV ACC, @0
  ADDL @XAR4, ACC  // XAR4 = &m[i]
  MOV AL, *+XAR4[0]  // AL = m[i]
  ADD @1, AL  // k = k + m[i]

  INC @0  // i++
  MOV AL, @0  // AL = i
  CMPB AL, #10
  SB L1, LT  // LT = Less Than
  LRET
```

4 连接命令文件CMD

- 连接文件的内容：把软件安排到硬件中去，用于控制程序文件中代码和数据输出段在存储器区域中的定位

- **MEMORY**

  1. 划分程序页、数据页，一页可分为若干段
  2. 与存储器映射有关
  3. 调试程序放在RAM中（也可用FLASH，没学），有RESET（复位向量）项

```
MEMORY{
  PAGE 0:  // 程序空间
    PRAMH0 : origin = 0x3F8000 length = 0x001000
         //程序段不要写到保留段中去
         //定下长度后，程序段就不能超过这个长度，否则会报错
    RESET : origin = 0x3FFFC0, length = 0x000002
         //复位向量，固定，从这个地址中取出32位地址为程序开头
  PAGE 1:  // 数据空间
    SPI_A : origin=0x007740, length=0x000010
}
```

- **SECTION**

  1. 把程序中的段定位到硬件的段
  2. 程序段 .reset .text .cinit
  3. 数据段 .bss .ebss .stack
  4. 数据段根据要求增加
  5. 要求掌握安排变量到固定地址中，如GPIOF

- **.text**：初始化段、所有可以执行的代码和常量、存储类型：ROM或RAM（FLASH）、Page0
  - 常量例如define PI=3.14（也可以用立即数赋值，但不常用）
- **.cinit**：初始化段、全局变量和静态变量的C初始化记录、存储类型：ROM或RAM（FLASH）、Page0
- **.stack**：非初始化段、为系统堆栈保留的空间，主要用于和函数传递变量或位局部变量分配空间、存储类型：ROM或RAM（FLASH）、Page1
- **.bss**：非初始化段、为全局变量和局部变量保留的空间、存储类型：ROM或RAM（FLASH）、Page1，在程序上电时.cinit空间中的数据复制出来并存 储在.bss空间中。分配范围被限制在**低64K 16位**数据区 位
- **.ebss**：为使用大寄存器模式时的全局变量和静态变量预留的空间。分配范围为**4M 22位**数据区 位 （分配范围不同于寻址方式相关）
- .cinit、.bss、.ebss：三者只与C相关，用汇编时不需要

```
SECTIONS{
  SciaRegsFile  : > SCI_A,   PAGE = 1

  .text         : > PRAMH0,  PAGE = 0
  .reset        : > RESET,   PAGE = 0, TYPE = DSECT /* not used, */
}
```

4.1 一个简单的例子

```
MEMORY{
  PAGE 1:
    GPFMUX  : origin = 0x0070D4, length = 0x000001
    GPFDIR  : origin = 0x0070D5, length = 0x000001
    GPFDAT  : origin = 0x0070D6, length = 0x000001
}
SECTION{
  GpiofMuxRegs  : > GPFMUX,  PAGE = 1
  GpiofDirRegs  : > GPFDIR,  PAGE = 1
  GpiofDataRges : > GPFDAT,  PAGE = 1
}
#pragma DATA_SECTION(Gpfmux,"GpiofMuxRegs")
#pragma DATA_SECTION(Gpfdir,"GpiofDirRegs")
#pragma DATA_SECTION(Gpfdat,"GpiofDataRegs")
```

```
volatile int Gpfmux,Gpfdir,Gpfdat
// 注意 GPFMUX -> GpiofMuxRegs -> Gpfmux 的关系
//       空间          段           变量
```

4.2 流程

```
/* .c */
#pragma DATA_SECTION (<variable>,"<section>");
// 定义一个变量和一个段，这个变量会被分配到这个段中去
volatile struct <struct_name> <variable>;
// 声名这个变量的变量类型
/* .cmd */
MEMORY{
    PAGE 1:
      <space_name> : origin = 0x0000, length = 0x0400
      // 定义一个数据空间
}
SECTION{
    <section>  : > <space_name>, PAGE = 1
    // 把一个段放到这个数据空间里
}
```

示例:

```
/* .c */
#progma DATA_SECTION (AdcRegs,"AdcRegsFile");
volatile struct ADC_REGS AdcRegs;  // struct ADC_REGS 是一个被定义好的变量类型
/* .cmd */
MEMORY{
    PAGE 1:
      ADC : origin = 0x007100, length = 0x000020
}
SECTION{
    AdcRegsFile : >ADC, PAGE = 1
                // load = ADC, PAGE = 1
}
```