**023A-CPU指令与寄存器结构详解**

1. **寄存器概述**
2. 023A中总计存在15个可被访问的寄存器:R1 R2 R3 R4 R5 R6 R7 FLAG PC TPC IPC SP DS TLB SYS。全部为32位寄存器。
3. 其中R1-R7为通用算数寄存器，用作缓存数据和运算时作为参数寄存器、得数寄存器。

FLAG为算数标志位寄存器，在运算后保存附加数据(溢出值等)。

1. PC、TPC、IPC都是用于保存指令地址的寄存器，但三者分工不同，PC寄存器指向待进入流水线执行的指令地址、TPC寄存器在跳转时保存返回地址、IPC寄存器在中断时保存中断返回地址。
2. SP、DS为保存数据内存地址的寄存器。SP保存当前的栈顶地址，DS保存读取内存块的段地址。
3. TLB保存的是如果CPU要启用虚拟内存，虚拟内存的TLB表内存地址。

SYS为CPU系统模式标志位寄存器。[0]为是否启用中断，[1]是否启用保护模式,[2]是否启用虚拟内存

1. 寄存器的访问权限:

R1 R2 R3 R4 R5 R6 R7 FLAG TPC IPC SP DS 任何时候都能读写

TLB在未启用虚拟内存前可以读写，启用后只读

PC SYS使用只读

1. **段地址、偏移地址**

程序读写数据、跳转指令执行时必须要提供跳转地址/读写地址。而这个地址一般是通过2种方法给出:

1. 在内存中某片固定区域存储常量地址指针，在跳转/读写时就先从该内存区域读取指针值入寄存器，通过寄存器来做跳转/读写地址。这种方法的好处是可以访问32位机的4GB内存空间中任意地址。缺点是需要先加载常量指针，非常浪费性能。
2. 直接通过立即数作为地址，方便快捷。但却有个问题，就是在32位RISC处理器中，所有指令都要已4字节做对齐。而这4字节中必然有一部分空间是用作表示是什么指令，那么留个立即数的空间必然是不满32位的，所以如果用立即数来做地址时表示范围是有限的。

而段地址、偏移地址的作用就是解决立即数做地址时表示范围有限的问题。在通过立即数表示地址时，高n位寄存器由固定的某个寄存器给出，低32-n位由立即数表示。这里面由固定寄存器给出的地址就叫段地址，立即数给出的地址就叫偏移地址。

1. 在任意指令跳转指令中，立即数可占据32位指令的[26:0]位，即总计0-2^27-1之间。在通过立即数指令跳转时，地址的[31:27]由PC寄存器的[31:27]给出，[26:0]就由立即数给出。所以每个段128MB，4GB空间被分成了32个段。最终读写的地址=PC[31:27]\*128MB+立即数。此时立即数就是段的偏移地址，最多偏移128MB。
2. 在内存读写指令中，立即数占据32位指令的[15:0]，即总计0-2^16-1之间，在通过立即数读写地址时，地址的[31:16]由DS寄存器的[15:0]给出，[15:0]就由立即数给出。所以每个段64KB，4GB空间被分为了65536个段。最终读写的地址=DS[15:0]\*64KB+立即数。
3. **内存读写指令**

汇编指令语法:

读内存 RD: 数据长度, 读到的寄存器, 读的地址;

写内存 WE: 数据长度, 写数据的寄存器, 写的地址;

出栈 POP: 数据长度, 读到的寄存器;

入栈 PUSH: 数据长度, 写数据的寄存器;

数据长度类型: BYTE 字节(8bit) WORD 半字(16bit) DWORD 字(32bit)

寄存器选择: 写内存/入栈时选择任意可读寄存器即可

读内存/出栈时选择任意可写寄存器即可

读写地址: 选择一个可读的寄存器，或者写立即数。

立即数写法: 数值(进制)

示例1: 将R1数据写入 DS[15:0]\*64KB+78的位置

WE: DWORD,R1,78d;

示例2: 将DS[15:0]\*64KB+0x45位置的short值读入R3

RD: WORD,R3,45h;

示例3: 将R4数据写入R3指向的内存地址

WE: DWORD,R4,R3;

示例4: 将R2数据压栈

PUSH:DWORD,R2;

1. **跳转指令**
2. 跳转指令类型概述:

无条件立即跳转 JMP:跳转地址;

带返回无条件立即跳转 JT:跳转地址;

TPC带返回间接跳转 RT;

IPC带返回间接跳转 IT;

带返回有条件立即跳转（一般配合布尔运算/比较运算使用）

Flag[0] JE:跳转地址; （比较运算A==B跳转,布尔运算为true跳转）

!Flag[0] JNE:跳转地址; （比较运算A！=B跳转,布尔运算为false跳转）

!Flag[1] && !Flag[0] JA:跳转地址; （比较运算A>B跳转）

!Flag[1] JNA:跳转地址;（比较运算A>=B跳转）

Flag[1] JB:跳转地址; （比较运算A<B跳转）

Flag[1] ||Flag[0] JNB:跳转地址;（比较运算A<=B跳转）

1. 间接跳转与立即跳转区别:

立即跳转:(JMP/JT/JE/JNE/JA/JNA/JB/JNB)

跳转地址由立即数给出。

特点:只可向当前程序所在的程序段内任意地址跳转，可跳地址为 PC[31:27]\*128MB+立即数。但是其跳转速度快，可以立即进行跳转而无需刷新流水线(消耗1个时钟周期)。

间接跳转:(RT/IT)

其跳转地址由固定的寄存器给出。RT由TPC寄存器给出地址，IT有IPC寄存器给出地址。

特点:可向4GB内存空间中任意位置跳转。但跳转前需要先将地址存入指定的跳转寄存器中，并且跳转时需要冲刷CPU流水线。相比立即跳转会有一定的性能消耗(消耗4个时钟周期)。

1. 带返回跳转与不带返回跳转

带返回功能指的是在跳转时，会自动将跳转前的PC寄存器数据看做是返回地址保存到指定位置。如果在跳转到的程序中想要返回到跳转前的地方继续执行的话，就只需要将保存返回地址的寄存器中的值作为跳转地址进行跳转即可。

根据是否带返回功能，跳转指令可分为3类:

第一类. JMP无返回指令

第二类.IT使用IPC寄存器保存返回地址的指令

第三类.JT/RT/JE/JNE/JA/JNA/JB/JNB使用TPC寄存器保存返回地址的指令。

1. 跳转指令使用示例:
2. 函数内的分支跳转:

一个C语言的函数几乎不可能超过128MB的大小，同时编译器在编译过程中也会避免函数指令块横跨多个段。所以函数内的跳转可以全部使用立即数跳转。

常量跳转指令:JMP/JNE

示例1:[if语句实现]

C语言:

If(条件){

//执行内容A

}

//执行内容B

汇编语言:

#条件判断，判断结果存入Flag寄存器

JNE:标记点地址; #如果判断!=true，就直接执行内容B

#执行内容A

标记点>

#执行内容B

示例2:(if else语句实现)

C语言:

If(条件A){

//执行内容A

}else if(条件B){

//执行内容B

}else{

//执行内容C

}

//执行内容D

汇编语言:

#判断条件A是否成立，判断结果存入Flag寄存器

JNE:标记点1地址;

#执行内容A

JMP:标记点3地址;

标记点1>

#判断条件B是否成立，判断结果存入Flag寄存器

JNE:标记点2地址;

#执行内容B

JMP:标记点3地址;

标记点2>

#执行内容C

标记点3>

#执行内容D

示例3:(while语句实现)

C语言:

while(条件){

执行内容A

}

执行内容B

汇编语言:

标记点1>

#判断条件是否成立，判断结果存入Flag寄存器

JNE:标记点2地址;

#执行内容A

JMP:标记点1;

标记点2>

#执行内容B

1. 函数间的调用跳转:

对于一些复杂项目来说，所有函数编译出的指令块很可能会超过128MB的段大小。而受限于立即数跳转的范围，所以函数间跳转一般是通过函数指针表配合间接跳转完成的。

函数指针表:

对于C语言来说，被编译出的函数的基地址指针都会被统一存储在一片常量内存中。这个常量块就被称为函数指针表。而函数间的调用就是先从函数指针表中取出要跳转的函数指针值，然后根据值进行跳转。

常用跳转指令:RT

示例:

C语言:

void FUN1(){

//函数体内容

}

void main(){

FUN1();

}

汇编语言:

/////////main函数指令//////////

MOV:DS,函数指针表的段地址;

RD:DWORD,TPC,FUN1的函数指针在指针表中的偏移地址;

RT; #通过TPC间接跳转来挑战入FUN1函数

//////////FUN1函数指令///////////

PUSH:DWORD,TPC; #保存函数返回地址入栈

#FUN1函数体指令

POP:DWORD,TPC; #取出函数返回地址

RT; #退出函数

1. 中断处理函数

中断处理函数和普通函数没太大区别。只是中断处理函数在出现中断信号后由CPU自动完成调用。唯一是区别是普通函数调用正常来说都是用RT来完成调用，中断函数则由CPU自动使用IT指令完成调用。所以中断函数的返回也要用IT来完成。

(如此设计的原因:普通函数由RT来调用/返回，返回地址是由TPC寄存器保存。如果说中断函数也是通过TPC保存返回地址的话就可能发生破坏原函数调用过程中TPC寄存器数据的情况，所以中断调用另使用IPC作为返回地址寄存器)

常量指令:IT

C语言:

/\*因为中断函数的退出方式与普通函数不同，所以C语言对于中断函数是要特别声明的。声明方法和当前C语言编译器针对的cpu架构有关系。因为针对023A cpu的c语言编译器还未开发，所以就暂时用ARM的声明方法举例\*/  
 void \_\_attribute\_\_((interrupt(“IRQ”))) FUN1(){

//中断处理内容

}

汇编语言:

//////////FUN1中断处理函数指令///////////

PUSH:DWORD,IPC; #必须在开启中断响应前保存保存IPC

#开启中断指令[在中断触发后CPU会自动关闭中断响应]

#FUN1函数体指令

POP:DWORD,IPC; #取出函数返回地址

IT; #退出中断处理函数

1. **转移/格式转换指令**

1.转移指令: MOV:寄存器A，寄存器B;

功能:将B的数据写入A。B也可选择为0-65535的UINT32型立即数

示例: MOV:R1,R2; #R2数据写R1

MOV:R1,78d; #十进制数78写入R1

MOV:R4,10010B; #二进制数10010写入R4

2.格式转换指令

(1)功能: 将一个数据写入另一个寄存器中时进行数据格式的转换

023A支持3种数据格式:

32位无符号整形数据

32位有符合整形数据

32位IEEE754标准的单精度浮点数据

(2)格式转换指令概述:

32位无符号整数转32位浮点数: UITF:A,B;

32位有符号整数转32位浮点数: ITF:A,B;

32位浮点数转32位有符号整数: FTI:A,B;

32位有符号整数符号正负颠倒(乘-1): TPN:A,B;

8位有符号整数转32位有符号整数: BTD:A,B;

16位有符号整数转32位有符号整数: WTD:A,B;

(将B中数据格式转换后存入A)

(3)功能

该类指令是用于解决各种数据类型的互相运算和强转。

示例1:

int a = 54;

float b = a;

此时就需要先将a的数据用ITF指令强转为浮点格式后才能存入FLOAT变量中

示例2:

int a = 32;

short b = -17;

int c = a+b;

此时如果直接将a+b存入c的话会发生运算结果出错。

原因:rd:word,r1,[b变量所在地址];将b取入寄存器的话，其高16位为0，不会自动补1。如果直接将其和32位有符号整形a进行运算的话必然出错。所以需要先用WTD指令将b的数据强转为32位有符号整形后才能运算。

1. **比较运算指令**

功能:一般配合有条件跳转语句使用

(1)指令概述:

32位无符号整形比较指令: UCMP:A,B;

32位有符号整形比较指令: CMP:A,B;

32位浮点比较指令: FCMP:A,B;

(整数比较时，B可为0-65535的UINT32型数据)

1. 比较结果:

FLAG[0]为保存A是否等于B

FLAG[1]为保存A是否小于B

1. 按位运算/布尔运算指令

用处:一般配合有条件跳转语句使用

1. 汇编指令格式

运算指令: C, A, B; #将A与B的位运算结果存入C,布尔运算结果存入flag[0]

运算指令: A,B; #将A与B的位运算结果存入A,布尔运算结果存入flag[0]

1. 指令类型(布尔运算时不为0的全部看做true)

与运算: AND:C,A,B; 功能C=A&B, Flag[0]=A&&B

或运算: OR:C,A,B; 功能C=A|B, Flag[0]=A||B

非运算: NOT:C,A; 功能C=~A,Flag[0]=!C

异或运算: XOR:C,A,B; 功能C=A^B,Flag[0]=A^B

1. **四则运算指令**

[FLAG寄存器保存运算溢出值，整数除法时保存A%B]

无符号整数四则运算

加法: UADD:C,A,B; #C=A+B

减法: USUB:C,A,B; #C=A-B

乘法: UMUL:C,A,B; #C=A\*B

除法: UDIV:C,A,B; #C=A/B

有符号整数四则运算

加法: ADD:C,A,B;

减法: SUB:C,A,B;

乘法: MUL:C,A,B;

除法: DIV:C,A,B;

浮点四则运算

加法: FADD:C,A,B;

减法: FSUB:C,A,B;

乘法: FMUL:C,A,B;

除法: FDIV:C,A,B;

(整数运算中B可为0-4095的立即数)

1. **移位运算指令**

左移运算: SAL:C,A,B;

右移运算: SAR:C,A,B;

循环左移运算: SOL:C,A,B;

循环右移运算: SOR:C,A,B;

(B可为0-32的立即数)

1. **软中断指令**

指令: INT:中断号;

中断号可为16-255的立即数。(0-15为保留中断号，不允许软中断指令触发)

1. **系统控制指令**

(1)指令类型:

暂停1个时钟周期: NULL;

开启中断: EI;

关闭中断: DI;

开启虚拟内存: EV;

关闭虚拟内存: DV;

开启保护模式: EP;

重启CPU: RST;

1. 保护模式功能:

在进入保护模式后，CPU不允许使用指令DI、DV、RST指令。

即不允许关闭中断、关闭虚拟内存、重启CPU。但可以允许在进入保护模式后再进入虚拟内存模式、开启中断响应。

一旦进入保护模式后，唯一退出保护模式的办法只有通过中断响应。

保护模式一般在运行多进程操作系统中使用，在保护模式下运行应用层程序，通过外部tick定时器产生的中断退出保护模式进入操作系统时间片切换程序中。

1. 虚拟内存功能:

虚拟内存可以在保护模式下启用，也可以不在保护模式下启用。在启用保护模式时，需先将保护模式启用后程序执行的首地址存入IPC寄存器中。EV指令会自动开启保护模式后跳转到IPC所指定的位置执行保护模式下的应用程序，并将TLB寄存器作为虚拟内存的TLB表地址。同样的，DV指令关闭虚拟内存时也会已IPC寄存器的值作为退出虚拟内存后的执行地址。

1. 中断功能:

023A支持0-255的中断号，其中0-15中断号为系统保留，用于在CPU运行错误时的中断，不允许通过外部/软中断触发。剩下的240个中断号由外部中断/软中断自行分配。

一旦被触发中断后，CPU会强制退出保护模式、虚拟内存模式、关闭中断触发。

其中外部中断是通过CPU内核对外提供的硬件通道触发，一般用于响应外设的信号以及运行操作系统时接收TICK定时器的切换任务片信号。

软中断一般用于在多进程操作系统中，用于应用层程序调用操作系统系统对应用层提供的API接口时陷入操作系统内核使用。

(常见系统保留中断号对应的事件:

8: 无指令的执行权限

3: 错误指令类型

5: 除法运算除数为0

6: 左移运算位数>32

7: 右移运算位数>32

)

(6)023A-CPU的内存结构

023A最高支持4GB的内存空间，外设与内存共有寻址空间。

其中0-1023的内存区域固定为中断函数向量表。在中断触发时会根据中断号从向量表中取出中断处理函数的地址。可以将中断触发后的函数调用理解为是此2条指令的组合:

RD:DWORD,IPC,中断号\*4;

IT;

其中从65536地址处固定为CPU的主程序入口地址。在CPU上电启动后会自动从该地址处开始加载指令并执行。