023A内核的寄存器大全:

普通数值寄存器: R1 R2 R3 R4 R5 R6 R7(CS) R8(DS) (可以存放任意32及以下位数的数据)

(实践发现有点不够用,后续准备扩展为24个)

R8寄存器兼用做数据段寄存器(内存已64KB为一数据段)

R7寄存器兼用做指令段寄存器(内存已128MB为一指令段)

特殊功能寄存器: SP 栈顶寄存器

PC 程序指针寄存器

TPC 普通函数调用链接寄存器

IPC 中断处理函数链接寄存器

FLAG 标志位寄存器

SYS 处理器运行模式寄存器

TLB 虚拟内存TLB表起始地址寄存器

023A汇编编译器支持的指令大全:

(所有运算指令全部都是以32位值进行的运算)

数值运算指令: ADD整型加法 UADD 无符号整型加法 FADD 浮点加法

SUB 整型减法 USUB 无符号整型减法 FSUB 浮点减法

MUL 整型乘法 UMUL 无符号整型乘法 FMUL 浮点乘法

DIV 整型除法/取余 UDIV 无符号整型除法/取余 FDIV 浮点除法

参数2个或3个, 已ADD为例: ADD a,b,c; 等于 a=b+c;(c可为0-4095的立即数)

ADD a,b; 等于 a+=b;(b可为0-4095的立即数)

ADD/UADD/SUB/USUB/MUL/UMUL计算后溢出的32位数据存放在FLAG寄存器,

DIV/UDIV计算后，除法结果存在a, 余数存放在FLAG寄存器

布尔运算/位运算指令: AND 位与/布尔与 XOR位异或 SOL 位循环左移

OR 位或/布尔或 SAL 位左移 SOR 位循环右移

NOT 位非/布尔非 SAR 位右移

AND/OR/XOR参数2个或3个: 例如AND a,b,c; 等于 a=b&c;

(c可为0-4095的立即数)

AND a,b; 等于 a&=b;

(b可为0-4095的立即数)

NOT参数只能为2个: 例如NOT a,b; 等于 a=~b;(b可为0-4095的立即数)

其中,AND/OR/NOT是同时有位运算和布尔运算的功能,位运算结果存在选定的a

寄存器,布尔运算结果存在FLAG寄存器。

SAL/SAR/SOL/SOR参数可为2个或3个:例如 SAL a,b,c; 等于a = b<<c;

(c可为0-32的立即数)

SAL a,b; 等于 a<<=c;

(b可为0-32的立即数)

移位运算结果存储在选定的a寄存器,移位溢出值存放在FLAG寄存器

位操作指令: SET 设置一个32位数据的[m:n]位的值

(基本不用，后续准备删除该指令)

GET 获取一个32位数据的[m:n]位的值

(基本不用，后续准备删除该指令)

指令参数3个: 例如: set 15-0,a,b; 等于verilog语法: a[15:0] = b;

也就是意思为将a寄存器中0-15位的数值替换为b的低15位

(b可为0-63的立即数)

其中,第一个参数的填写语法是 操作的高位-操作的低位(不能够 是一寄存器的数据)

比较运算指令: EC 整型== UEC 无符号整型== FEC 浮点==

NEC 整型!= UNEC无符号整型!= FNEC浮点!=

LC 整型< ULC 无符号整型< FLC 浮点<

LEC 整型<= ULEC无符号整型<= FLEC浮点<=

MC 整型> UMC 无符号整型> FMC 浮点>

MEC 整型>= UMEC无符号整型>= FMEC浮点>=

指令参数2个,示例: EC a,b;等于 FLAG = (a==b)

b可为0-65535的立即数,运算结果存放在FLAG寄存器

(后续准备修改指令集，添加支持将运算结果直接存放 到其它通用数据寄存器的 功能)

数据格式转换指令: UITF 无符号整型转浮点型

ITF 整型转浮点型

FTI 浮点型转整型

TPN 有符号整型正负号转换(基本不用，后续准备删除该指令)

BTD 8位有符号整型转32位有符号整型

WTD 16位有符号整型转32位有符号整型

指令参数2个，示例: ITF a,b;等于 a = (float)b;

b可为0-65535的立即数

数值初始化指令: INIT 初始化一个32位数(int32 uint32 float)

INIT64 初始化一个64位数(int64 uint64 double)

INIT指令参数2个: INIT a,立即数; 立即数可为任意32位立即数

INIT64指令参数3个: INIT64 a,b,立即数;立即数可为任意64位立即数

a寄存器存储立即数高32位数据,

b寄存器存储立即数低32位数据

数值转移指令: MOV 用于寄存器间数据的交换,

也可用于初始化数据,但只能初始化16位数值

指令参数2个: MOV a,b; b可为0-65535的立即数

内存访问指令: RD 从内存任意区域位置读取一个8-32位数值到寄存器

WE 将一个寄存器中低8-32位的数值存入内存任意位置

指令参数3个, 已RD为例: RD 读写宽度,a,b;

功能: a = \*b;

读写宽度: DWORD 4字节

WORD 2字节

BYTE 1字节

(b可为0-65535的立即数,如果是立即数，那么实际读写地址等于

R8(DS)数据段寄存器的值 \* 64KB + 立即数)

栈访问指令: POP从栈顶弹出一个8-32位数值到寄存器

PUSH 将寄存器中低8-32位的数值压入栈顶

指令都为2个参数，已POP为例: POP 读写宽度,a;

其中push指令的参数a可以是0-65535的立即数

读写宽度: DWORD 4字节

WORD 2字节

BYTE 1字节

栈内存快捷读写指令: (虽然内存访问指令也能读写栈内存中容易位置的数据,但速度较慢)

(故,023A指令集中添加了专门的栈内存读写指令,速度基本是普通内存访 问的3倍)

SRD 从栈顶向下2MB范围内取一个8-32位的数据到寄存器

SWE 将寄存器低8-32位数据存入栈顶向下2MB范围内的区域

指令支持3个参数，已SRD为例: SRD 读写宽度,a,b;

读写宽度: DWORD 4字节

WORD 2字节

BYTE 1字节

读写的实际地址等于 SP+b;

b只可为0-2097151之间的立即数，正好能偏移等于2MB

短跳转指令: JMP 跳转至当前指令内存段的任意位置(023A内存寻址范围为4GB，每128MB 为一个指令内存分段。一个函数编译出的二进制指令严禁横跨2个指令内 存分段。)该指令一般用于函数中流程块的跳转。

JNE 条件跳转至当前指令内存段的任意位置，该指令一般用于函数中流程块的 跳转。当FLAG寄存器数值=0就跳转，否则不跳转。

指令参数都为1个，已JMP为例: JMP a; a为0-134217728之间的立即数

跳转到的实际内存地址 = PC[31:27]\*128MB + a;

也就是说，该指令支持跳转到当前PC寄存器所在的指令段128MB中的任意位置。

长跳转指令:

RT 长跳转至4GB寻址空间中任意位置，并保存跳转前地址到链接寄存器TPC。 一般用于普通函数的调用与返回。

IT 长跳转至4GB寻址空间中任意位置，并保存跳转前地址到中断链接寄存 器IPC。一般用于普通中断处理函数的返回。

(基本不用，后续准备删除该指令)

EIT 在IT指令基础上，启用中断响应。一般用于系统内核中断处理函数的返 回。

VIT 在VIT指令基础上，启用虚拟内存功能。一般用于驱动层请求内核功能的 处理函数的返回上

PIT 在VIT指令基础上，启用CPU保护模式。一般用于应用层请求内核功能的 处理函数的返回上

所有指令均无参数，例如使用RT指令。直接写作RT;即可

实际的工作原理就是将链接寄存器与PC寄存器的值交换一下。

已RT为例，如果要跳转至一个函数可以编写以下指令:

INIT TPC,[要跳转到的函数地址]; #此时,将链接寄存器指向了要跳转位置

RT; #调用RT。程序跳入指定地址，并保存了(返回地址+4)到链接寄存器

当在函数中药返回时，再调用一次RT即可。

如果是中断处理函数，在进入中断处理函数前，CPU硬件就自动初始化了IPC的值

为中断退出后程序继续执行的位置

软中断请求指令: INT 发出一个16-255中断号的中断请求

参数1个，示例: INT num; num可为16-255的立即数

CPU运行模式控制指令:

EI 启用中断响应 DI 禁止中断响应

EV 启用虚拟内存 DV 禁用虚拟内存

EP 启用保护模式(保护模式开启后只有中断响应才能退出)

RST 重启CPU以及所有外设

NOP 流水线暂停一个时钟周期

无参数。

023A汇编语法详解:

注释语法: 注释符‘#’能够注释一行文本。用法和python注释符一样

一．定义一个基本的函数的基本语法:

汇编语言支持定义2种函数:

1.仅在当前汇编源码文件中可以被访问到的函数(相当于c/c++中加了static关键字的函数)

CODE 函数名{

#函数中的指令

};

2.全局共享的函数，在其他汇编源文件中声明该函数就能使用(一般情况下都是定义此种函数,特别是程序入口函数必须得是全局共享的，否则无法在链接脚本中声明入口函数存放的内存基地址)

CODE-EXPORT 函数名{

#函数中的指令

};

3.声明外部导入的函数

如果要使用其它汇编源文件中的全局共享函数，就必须要在当前汇编源文件中声明

IMPORT 函数名

二．汇编函数中定义程序功能指令的语法

单条指令的基本语法格式:

指令类型 参数1,参数2,参数3,...;

示例:

ADD R3,R1,R2; #功能: R3 = R1+R2;

立即数的定义语法:

数值F 浮点型型立即数

数值H 16进制整型立即数

数值O 8进制整型立即数

数值B 2进制整型立即数

数值D 10进制整型立即数

示例:

ADD R3,R1,32H; #功能: R3 = R1 + 0x32;

INIT R1,323.32F; #功能: R1 = 323.32f;

1. 汇编函数中标记点的定义与使用

标记点的定义语法:

标记点名称>

使用当前函数块中标记点的语法:

[this.标记点名称]

使用其它函数块中标记点的语法:

[定义标记点的函数名称.标记点名称]

函数开头默认存在一个无名标记点,使用该无名标记点:

[函数名]

标记点本质上就是个立即数，它在链接过程中会由编译器自动转换为定义该标记点的内存地址的立即数值。

使用示例1:（使用标记点语法调用外部函数）

CODE add{

ADD R3 ,R1,R2;

RT;

};

CODE-EXPORT main{

MOV R1,78D;

MOV R2,54D;

#此时就是使用了add函数的无名标记点,指向add起始地址

INIT TPC,[add];

RT;

};

使用示例2:(使用标记点语法实现本函数中循环跳转执行)

CODE-EXPORT while\_Fun{

restart>

#函数功能

#使用了当前函数中定义的标记点

JMP [this.restart];

};

在使用标记点时，进行标记点指向的偏移。以及指定标记点转换为立即数时的位数。

语法： [标记定义函数块.标记名 +/- 偏移立即数](高位-低位);

示例:

CODE test{

mark1>

MOV R1,[this.mark2 + 10D](31-16);

MOV R2,[this.mark1 - 3D];

mark2>

};

其中MOV R1,[this.mark2 + 10D](31-16);表示，指向的是

this.mark2 + 10字节的位置。并去指向位置转换出的内存地址立即数的高

16位存入R1。

MOV R2,[this.mark1 - 3D];没有定义是去内存地址立即数的哪几位，那么默认 是去低N位。（N = 当前参数上支持的立即数最高位数）

1. 定义一个静态变量

定义静态变量块的基本语法:

1. 定义一个非初始化的静态变量块

DATA-NOINIT 变量块名{

#DATA-NOINIT变量块只支持ARR初始化伪指令

ARR 每个单元的字节数,变量块中每个单元的总单元数;

};

示例: (定义一个static int noinitVar[10])

DATA-NOINIT noinitVar{ARR 4D,10D;};

1. 定义一个需要初始化的静态变量块

DATA 变量块名{

#定义变量中初始化数值的伪指令

};

静态变量块内初始化数据的伪指令:

基础数值型的初始化

1. 无符号整型: UBYTE USHORT UINT ULONG
2. 有符号整型: BYTE,SHORT,INT,LONG
3. 浮点型: FLOAT,DOUBLE

示例: static uint64\_t arr = {21,43,654,32};

DATA arr{

ULONG 21D,43D,654D,32D;

};

字符串的初始化(转为utf8编码，不自带\0终止符)

4.STR “字符串内的文本1”,”字符串内的文本1”,.....;

示例: static char txt[] = “hello world”;

DATA txt{

STR “hello world”;

UBYTE 0D; #末尾添加个终止符

};

或:

DATA txt{

STR “hello world\0”;

};

5.空白区域，全部填充0xCC

ARR 填充的单元字节数,共多少个单元;

静态变量的导出至外部与从其他源文件导入

静态变量与函数一样，如果要允许导出至外部成为全局变量块，添加-EXPORT关键字即可。

示例: int a;

DATA-NOINIT-EXPORT globalVar1{

ARR 4D,1D;

};

从外部导入一个全局变量块:

IMPORT 全局变量块名;

静态/全局变量也支持标记语法，与函数中的标记语法完全一致

示例:定义2个全局变量，并在函数中读取至寄存器，然后加法运算:

CODE add{

UADD R3,R1,R2;

RT;

};

DATA-EXPORT varGroup{

Var1> INT 3232D;

Var2> UBYTE A1H;

AddPtr> UINT [add];#uint型数据,其中存储了add函数的起始地址

};

CODE-EXPORT main{

#读取varGroup.Var1数据到R1

INIT R8,[varGroup.Var1];

RD DWORD,R1,R8;

#读取varGroup.Var2数据到R2

INIT R8,[varGroup.Var2];

RD BYTE,R2,R8;

#调用加法函数,varGroup.Var1+varGroup.Var2的结果存入R3

INIT R8,[varGroup.AddPtr];

RD DWORD,TPC,R8;#读取AddPtr中存储的add函数地址到tpc寄存器

RT;#跳转入add函数

};

1. 链接脚本编写语法

链接脚本的功能:

1. 指定一个全局函数、变量块的存放内存地址
2. 指定检测整个程序中是否定义了必要的全局函数、变量块
3. 指定函数、变量块生成的二进制数据分配的内存空间范围
4. 指定函数、变量块生成的二进制数据分配的内存空间范围:

2种语法:

SPACE 起始地址 终止地址

SPACE 起始地址 L字节数

(起始地址、字节数的填写语法和汇编中的立即数相同,但是不能为浮点数)

可以定义多个SPACE来制定多段范围。最终允许分配函数、数据块的范围就 是这些SPACE定义的范围的并集

示例:

SPACE 0D L1024D

SPACE 2000D 3000H

最终函数、数据块可以分配在 (0,1023)∪(2000,3000)的区间内。

1. 指定检测是否定义了必要的全局函数、变量块。

语法:

CHECK CODE 函数名

CHECK DATA 数据名

示例:

(检测是否定义了程序入口main函数和中断向量表数据块interrputTable)

CHECK CODE main

CHECK DATA interrputTable

1. 指定全局函数、变量块的存储基地址

(如果不指定的话，所有的函数、变量块的基地址都是随机分配的，只是确保不会超出SPACE 语句定义的范围。对于部分特殊功能的函数、数据块就必须要固定其分配的地址。此时就要使 用该语句了)

FIXED 函数/数据块名 基地址

示例:

(CPU重启后PC寄存器默认等于1024。要让main在CPU上电后立即执行)

FIXED main 1024D

链接脚本综合示例:

(需求,当前CPU接了一块ROM在0-65535内存地址上。

定义了一个4096Byte的非初始化全局变量stackPlace用于作为堆栈空间,栈底在65536。

定义了一个1024Byte存放中断向量表的全局变量interrputTable，CPU中断向量表起始地址在0x00000000处。

定义了一个main函数作为入口函数，CPU上电起始地址为0x00000400)

####################脚本代码##########################

SPACE 0D 65535D

CHECK DATA stackPlace

CHECK DATA interrputTable

CHECK CODE main

FIXED main 400H

FIXED interrputTable 0D

FIXED stackPlace 64512D

#######################################################