**高级汇编编译器使用教程**

**汇编编译器程序使用方法:**

1. 打开cmd或pworshell，cd到编译器所在目录。敲击命令: ./汇编编译器程序名.exe 编译 模式 编译文件
2. 汇编编译器支持的编译模式:

将汇编语言文件编译为二进制程序数据: ./汇编编译器程序名.exe -c “汇编文件路径” “生 成后的.bin文件路径”

将二进制程序数据打印为能够复制入tb测试的缓存器中的数据文本: ./汇编编译器程序 名.exe -t “要转换为tb测试文本的.bin文件路径”

**高级汇编编译器语法:**

高级汇编语言相比于简易编译器，是增加了内存块定义、标记定义/解析、自动插入空指令解决数据冒险问题的功能。其汇编指令的数量与语法是和简易汇编编译器完全一样的，可直接去查看简易汇编编译器的使用教程(唯一不同的是高级汇编编译器不支持NULL指令，因为该指令由编译器自动插入)

1. 注释:

单行注释: #注释文本

1. 定义内存块:

内存块的功能:本质上内存块就是内存中一片数据的集合。在高级汇编编译器中可以定 义一片内存块，这内存块在编译完成后就会实实在在的成为一片内存区域。内存块整体 上可分为数据内存块与指令内存块。

数据内存块:其内定义的指令会已数据汇编语法进行解析。一般用于描述一片常量数据。

指令内存块:其内定义的指令会已指令汇编语法进行解析。也就是描述要执行的程序

指令内存块与数据内存块存储位置的区别：

在高级汇编编译器中是可以指定当前cpu结构是冯诺依曼架构还是哈佛架构的。如果是 冯诺依曼架构指令内存与数据内存块在编译完成后是混合存储的。而如果指定的哈佛架 构，编译完成后会将指令内存块与数据内存块分别整合起来。

定义语法:

(内存块类型 内存块编译后在内存中的基地址 内存块名){

#内存块内数据/指令的描述文本

};

内存块类型可写: DATA[数据内存块] CODE[指令内存块]

基地址可写: 1.手动确定一个静态地址: 数值(进制) 进制可为

H 十六进制 D 十进制 O八进制 B 二进制

2.AUTO由编译器在综合了所有内存块后自动分配基址

内存块名:一个由英文、数字、下划线组成的名字即可。

[注:大小写敏感，内存块间名称不得重复，首字符必须为英文]

示例:

#定义一个主指令内存块,作为程序入口。地址固定为0x00010000

(CODE 10000H Main){

mov:r8,[Const.A]<31-16>;

rd:dword,sp,[Const.Stack\_base]<15-0>;

rd:dword,r1,[Const.A]<15-0>;

rd:dword,r2,[Const.B]<15-0>;

mov:r3,10d;

cmp:r1,r2;

mov:tpc,[this.TestIF\_end]<15-0>;

jc:flag,[this.TestIF\_begin]<15-0>;

rt;

TestIF\_begin>

mov:r3,78d;

mov:r2,94d;

add:r3,r2;

TestIF\_end>

mov:r4,r3;

end>

jmp:[this.end]<26-0>;

}

#定义一个常量内存块。地址由编译器自动分配

(DATA AUTO Const){

A>int:3213d;

B>int:32132d;

Stack\_base>uint:[StackBase];#栈底地址

}

2.标签语法:

因为内存块的地址往往是由编译器自动分配的，当程序要读写某个数据内存块的数据、调用某个程序内存块的程序时，不可能去由程序员来推算该内存块的基地址，然后再自行判断要偏移多少位。所以，这些工作由编译器自动完成。当程序员要操作某个地方的数据、跳转到某个地方执行程序时，只需要在该处前方上打上标签。然后再使用解引用标签的语法即可。编译器会自动从解引用语法中转换出实际操作的内存地址。

定义标签:

标签名>标签指向的地方

示例1:(DATA AUTO Const){

1. int:3213d;
2. int:32132d;

}

中就是定义了两个标签，A/B两个标签分别指向了两个int型常量值

示例2:(CODE AUTO FUN1){

Enter>

mov:r3,78d;

mov:r2,94d;

add:r3,r2;

mov:r4,r3;

}

中也定义了一个标签Enter，指向了FUN1指令内存块的头部

假如在另外某个内存块中解引用了FUN1.Enter，然后已此标签指向的地址进行跳跃的话， 就会跳入FUN1内存块头部后继续执行。

标签定义的名称规范:

一个由英文、数字、下划线组成的名字即可。

[注:大小写敏感，同一个内存块中的标签名称不得重复，首字符必须为英文]

标签名的识别对于编译器来说其实是有一个前缀的，即 内存块名.标签名

标签解引用语法:

[内存块名.标签名+偏移地址]<高位-低位>

偏移地址:即已 内存块名.标签名 解引用出的地址值再+-一个常量。

<高位-低位>:[内存块名.标签名+偏移地址]解引用出的是32位值，将整个值裁切，只保 留下<高位-低位>之间的位。

比如:

[内存块名.标签名+偏移地址]解引用出为 0x00a10000

[内存块名.标签名+偏移地址]<31-15>后结果就是0x00a1

使用示例1:

(CODE AUTO FUN){

Enter>

#FUN程序内容

}

(DATA AUTO Const){

#将FUN的首地址存入常量区

uint:[FUN.Enter];

}

使用示例2:

(CODE AUTO FUN){

#FUN程序内容

end>

jmp:[this.end]<26-0>;

}

如果解引用中使用的是[this.标签名]。表示该标签名是该程序内存块中的。

jmp:[this.end]<26-0>;指令就是循环跳转到FUN.end处，而FUN.end标签指向的就是 jmp:[this.end]<26-0>;所以该程序就是个while(true);死循环

内存块基址自动分配的依赖信息

基址自动分配的策略:

对于高级汇编编译器来说，需要提供cpu的内存架构、物理内存寻址范围、cpu指令集 架构、入口地址等信息（宏定义中进行描述）。

首先，会先给基址手动确定的内存块以及入口程序内存块分配内存区块。然后在从剩下 的内存中寻找符合条件的连续内存区块分配给AUTO基址的内存块。且在分配过程中， 会优先从低地址端开始分配，并尽可能避免一个内存块横跨多个内存段(如果跨越了会 报警告)。

[内存段:023Acpu将4GB的总寻址空间分成了65536个小份，每个小份为64kb,即一个 段]

(内存段的作用:在有条件跳转、内存读写时往往会通过立即数来确定要跳/读写的内存地 址。但是立即数只有16位，只能表示0-65535。为了解决该问题，在读写内存时立即 数仅仅作为地址的低16位，高16位由R8寄存器的[15:0]给出。同样的，有jc跳转时 立即数仅仅作为跳转地址的低16位，高16位由pc[31:16]确定。那么实际上就可以将 跳转/读写时的高16位地址看做是段地址，而低16立即数给出的地址则是段的偏移地 址。

假如当前执行的CODE内存块是仅仅存在于一个段中，那么jc指令就能往该CODE块的 任意标记处跳转。但是如果CODE块横跨了多个段，要跳转时jc指令就不得使用立即数 作为跳转地址了，否则可能出现段偏移被该了，但段地址却没被该的情况。

同样的，如果要读取的一个DATA块只存在于一个段中，那么只需要在读写其数据前先 将R8改为DATA块段地址，然后就能通过立即数做段偏移地址来访问DATA块中的任意 一个数据了。)

宏定义语法:

@宏类型 参数1 参数2 参数3.... [宏定义必须换行写]

宏类型大全:

@Import "源文件路径" #导入一个外部源文件

@Define 原文本 替换文本 #将代码中的所有原文本该为替换文本

@MemoryArch 内存架构 #填FN或HD，FN为冯诺依曼，HD为哈佛架构

@InstArch 指令集架构 #当前只支持023A，填023A即可

@Inlet 入口地址 入口内存块 #入口地址填写格式 数值(进制) 示例:10000H

#入口内存块填入口CODE块的名字(如果指定了入口CODE块，该块地址必须为auto)

@CodeMemInfo 基址 长度 #定义一个物理指令内存寻址范围

#[一个程序可定义多个不连续的指令内存寻址范围](哈佛架构该宏才生效)

@DataMemInfo 基址 长度 #定义一个物理数据内存寻址范围

#[一个程序可定义多个不连续的数据内存寻址范围]

#(冯诺依曼架构下，数据内存块/指令内存块都使用的数据内存空间)

**高级汇编编译器示例程序:**

将全局静态变量int Array[4]中的数据从小到大排序：（使用冒泡排序算法）

[编译完成后代码内存总计消耗500Byte,数据内存消耗28Byte]

@MemoryArch HD #设置为哈佛架构

@DataMemInfo 0d ffffh #设置数据内存寻址范围

@CodeMemInfo 0d ffffh #设置指令内存寻址范围

@InstArch 023A #设置使用的指令集为023A

@Inlet 00010000h Main #设置入口地址和对应的CODE内存块

@StackInfo fffh 64d #设置栈底地址、栈长度。

#主程序段，cpu启动后立即执行的程序

(CODE AUTO Main){

enter>

mov:r8,[Const.FUN1]<31-16>;#将r8段寄存器指向常量内存块

rd:dword,sp,[Const.Stack]<15-0>;#初始化栈地址

rd:dword,r7,[Const.Array]<15-0>;#将全局变量的地址取到r7

mov:r3,0d;

while1\_if>#外循环，已r3为index。

mov:r4,r3;

cmp:r4,3d;

mov:tpc,[this.while1\_end]<15-0>;

jc:flag,[this.while1\_begin]<15-0>;

rt;

while1\_begin>

mov:r5,r7;

mov:r4,r3;

mul:r4,4d;

add:r5,r4;

rd:dword,r1,r5;

push:dword,r5;#将x1的地址入栈保存

mov:r4,r3;

add:r4,1d;

while2\_if>#子循环，已r4作为index

mov:r5,r4;

cmp:r5,4d;

mov:tpc,[this.while2\_end]<15-0>;

jc:flag,[this.while2\_begin]<15-0>;

rt;

while2\_begin>

mov:r6,r7;

mov:r5,r4;

mul:r5,4d;

add:r6,r5;

rd:dword,r2,r6;

#调用FUN1函数

rd:dword,tpc,[Const.FUN1]<15-0>;

rt;

#结果写回内存

rd:dword,r5,sp;#读取x1地址

we:dword,r1,r5;

we:dword,r2,r6;

add:r4,1d;

jmp:[this.while2\_if]<26-0>;

while2\_end>

add:r3,1d;

jmp:[this.while1\_if]<26-0>;

while1\_end>

#将Static.Array的数据依次取到r1-r4寄存器。观察是否已经排序成功了

rd:dword,r1,r7;

add:r7,4d;

rd:dword,r2,r7;

add:r7,4d;

rd:dword,r3,r7;

add:r7,4d;

rd:dword,r4,r7;

end>#程序结束后进入死循环，防止跑飞了

jmp:[this.end]<26-0>;

}

#FUN1函数功能:如果r1大于r2的值，就将r1与r2的值进行交换

(CODE AUTO FUN1){

enter>

#保存函数中会修改的寄存器入栈

push:dword,tpc;

push:dword,r3;

push:dword,flag;

#如果r1>r2,FLAG就为1

mov:r3,r2;

cmp:r3,r1;

#如果FLAG为1就执行if\_begin，否则跳到if\_end

mov:tpc,[this.if\_end]<15-0>;

jc:flag,[this.if\_begin]<15-0>;

rt;

if\_begin>

mov:r3,r1;

mov:r1,r2;

mov:r2,r3;

if\_end>

#退出函数

pop:dword,flag;

pop:dword,r3;

pop:dword,tpc;

rt;

}

#常量区

(DATA AUTO Const){

FUN1>uint:[FUN1.enter];

Stack>uint:[StackBase];

Array>uint:[Static.Array];

};

#全局静态变量区

(DATA AUTO Static){

Array>int:432d,653d,78d,512d;

}