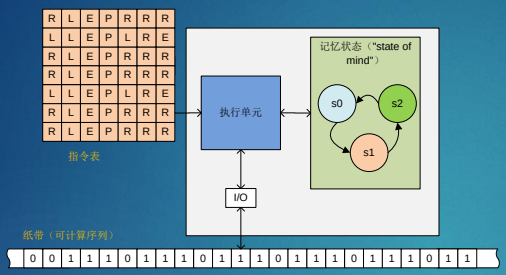


在调试器下理解计算机系统

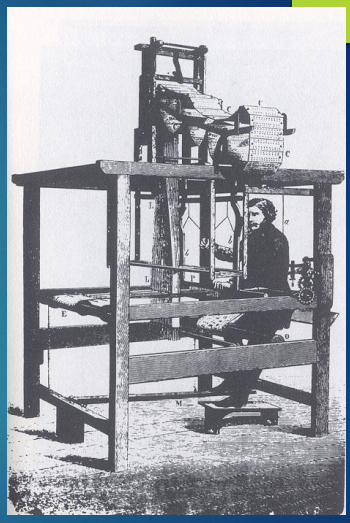
万法归一：图灵机和指令引擎

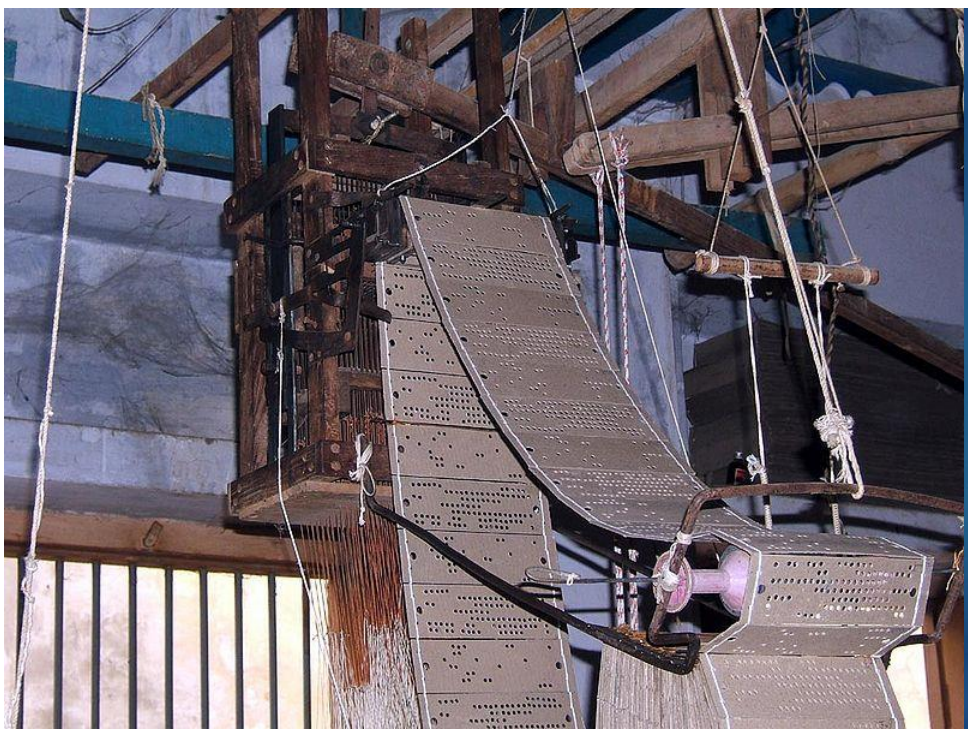
张银奎 2020.8.22 / 2021/4/10

人类追求自动计算的梦想从很久以前就开始了……

1801年 -雅卡尔织布机

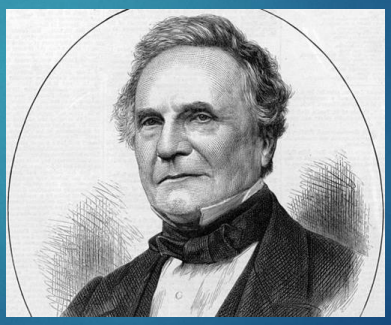
- 法国人约瑟夫·玛丽·雅卡尔(7 July 1752 – 7 August 1834)发明雅卡尔织布机(Jacquard loom)
- 根据穿孔卡片自动编织花样
- 刷新了人类对机器能力的理解
- 1805年，拿破仑参观雅卡尔织布机，将专利颁发给里昂市，授予雅卡尔终生津贴

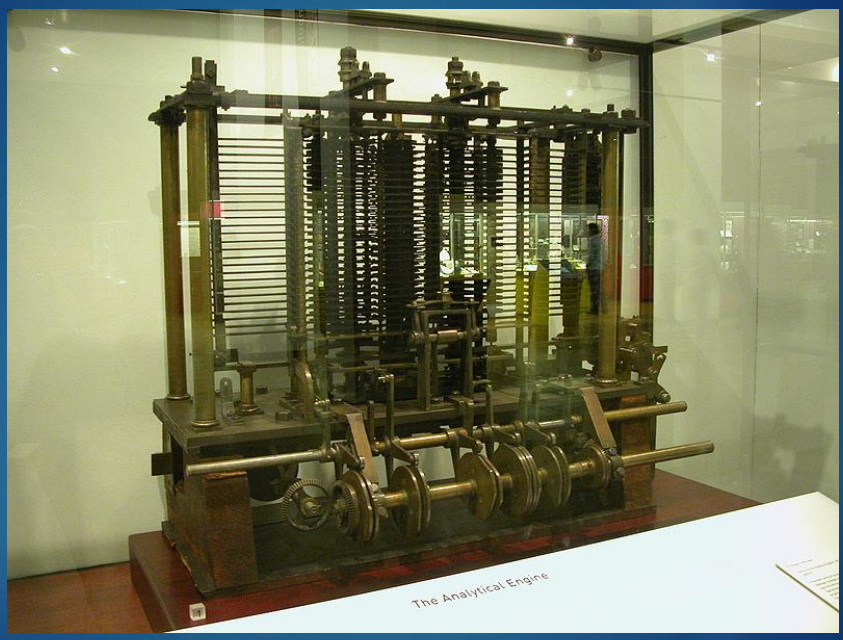




1836 -可编程计算机

- 英国数学家查尔斯·巴贝奇(1791-1871)将雅卡尔绣像挂在自己的客厅里
- 1836年，第一台可编程计算机诞生了，取名为分析引擎(Analytical Engine)
- 三种卡片：运算卡片，变量卡片，常量卡片

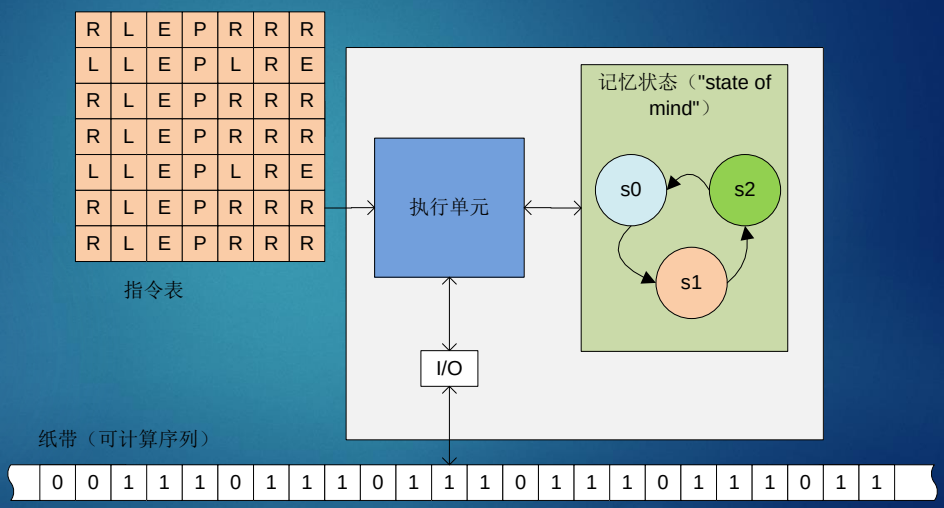


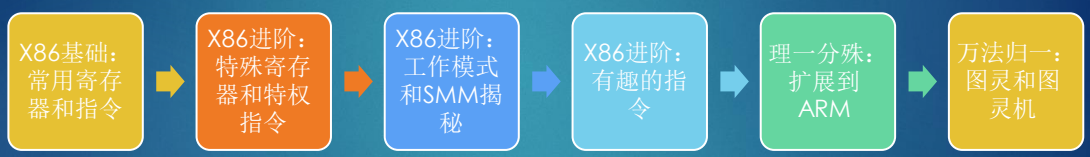


伦敦科技博物馆中的分析引擎模型（部分）

1936年：图灵机

- 一条长长的打孔纸带穿过一个神奇的机器
- 从指令表读取指令，决定如何操作纸带上的数据
- 可编程的万能机器





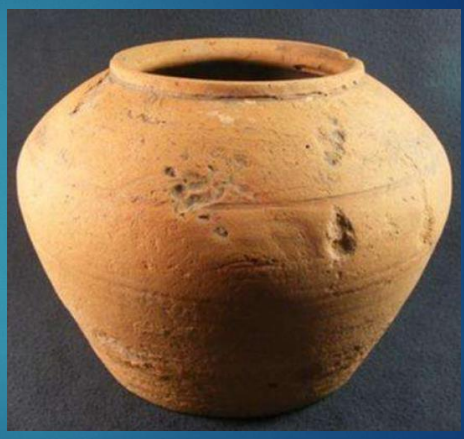
寄存器：托付暂放之器

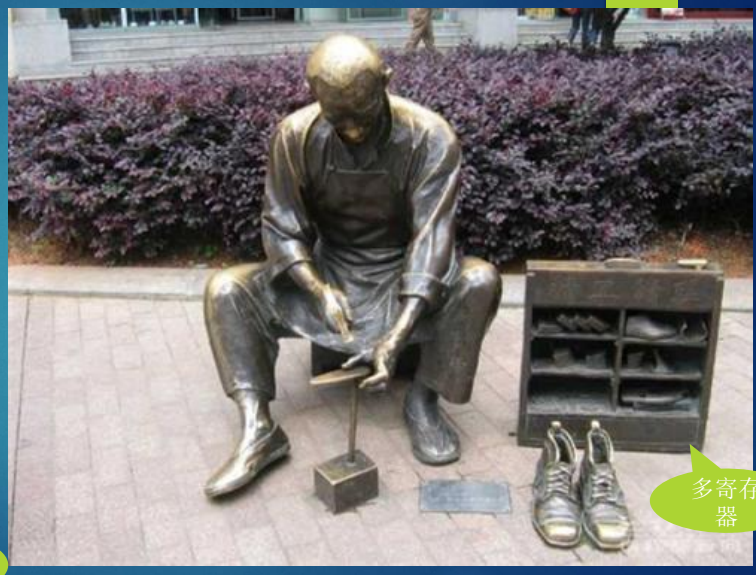
器：工具

常常与用途绑定

君子不器

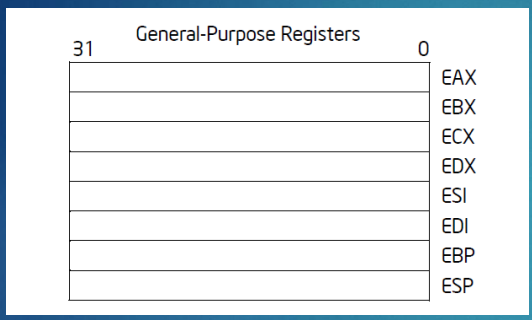
《论语·为政》





- **寄存器**是CPU执行指令时使用的临时空间，也是暴漏给软件的数据接口
- 它的设计关乎CPU的执行效率和上层软件编程
- 理解CPU的第一步便是理解它的寄存器

8个通用寄存器



- EAX函数返回值
- ECX循环次数
- ESI，EDI串操作的源和目标
- ESP栈顶
- EBP栈帧基地址
- 长模式下都为64位，名字为RAX, ...

ntdll!memcpy

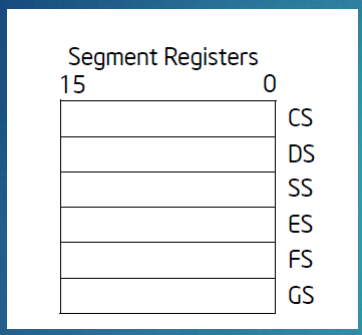

```
76f62345 8b750c          mov     esi,dword ptr [ebp+0Ch]
76f62348 8b4d10          mov     ecx,dword ptr [ebp+10h]
76f6234b 8b7d08          mov     edi,dword ptr [ebp+8]
76f62373 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

- 为性能考虑按DWORD操作，函数中要处理“零头”字节

Memset:
76f6df85 f3ab rep stos dword ptr es:[edi]

6个段寄存器

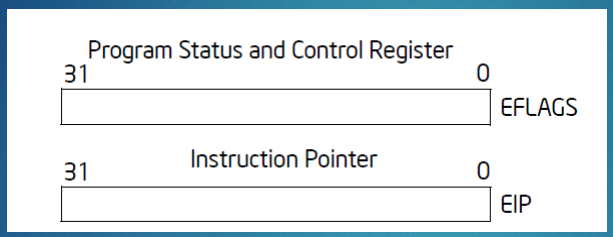
15



- 所有工作模式下都为16位长

1个标志寄存器+1个IP

16



64位下为RIP

EFlags

64位时，标志寄存器也是32位，考虑软件兼容问题，后来功能使用新的寄存器了

22条常用x86指令 (1/2)

指令	机器码	说明
INT 3	0xCC	软件断点
NOP	0x90	空操作
PUSH	* 0x50/51 …(通用寄存器)	压入栈
POP	* 0x58/59…(通用寄存器)	从栈中弹出
ADD	* 0x80/81/83	加法
SUB	* 0x80/81/83	减法
IDIV	0xF6/0xF7	整数除法
RET	* 0xC3	函数返回
CALL	* 0xE8XXXXXXXX	调用函数
INC/DEC	* 0xFF	递增/递减
MOV	* 0x88/89/8A/8B/8C/8E	赋值

* 部分情况的机器码

22条常用x86指令 (2/2)

指令	机器码	说明
JMP	* 0xE9/EA/EB	绝对跳转
JZ/JNZ	0x74/75	条件跳转
JB/JNB	0x72/73	条件跳转
JA/JBE	0x77/76	条件跳转
JL/JGE	0x7C/7D	条件跳转
JG/JLE	0x7F/7E	条件跳转
TEST	* 0x85XX	逻辑比较
CMP	* 0x38/39/3A/3B	数学比较
XOR	* 0x30/31/32/33	异或
LEA	* 0x8DxxXXXXXXXXXX	取有效地址
MOVS	* 0xA4/A5	串赋值

* 部分情况的机器码

强大的x86指令

```
1020f789 c78491c400000000000000 mov dword ptr [ecx+edx*4+0C4h],0
```

```
for(i=0;i<ARRAY_LENGTH;i++)
{
    pArray[i]->m_nField = 0;
}
```

- 11字节机器码
- 循环处理一个数组，ECX指向数组的基地址，EDX做循环变量，索引数组的元素
- http://advdbg.org/blogs/advdbg_system/articles/6213.aspx

细看一个函数


```
1  LocalVar!FuncC:
2  00401080 55              push    ebp ; 压入EBP寄存器的当前值
3  00401081 8bec            mov     ebp,esp ; ESP寄存器的值（栈顶）赋给EBP
4  00401083 83ec08          sub     esp,0x8 ; 为变量szTemp分配空间
5  00401086 57              push    edi ; 保存EDI寄存器的以前值
6  00401087 6a04            push    0x4 ; 准备调用strncpy，压入参数4
7  00401089 8b4508          mov     eax,[ebp+0x8] ; 将szPara赋给EAX
8  0040108c 50              push    eax ; 压入szPara
9  0040108d 8d4df8          lea     ecx,[ebp-0x8] ; 将szTemp的有效地址放入ECX寄存器
10 00401090 51              push    ecx ; 压入szTemp
11 00401091 e88a000000      call    LocalVar!strncpy (00401120) ; 调用函数strncpy
12 00401096 83c40c          add     esp,0xc ; 调整栈指针，释放6，8，10行压入的参数
13 00401099 8d7df8          lea     edi,[ebp-0x8] ; 将szTemp放入EDI
14 0040109c 83c9ff          or      ecx,0xffffffff ; 将ECX寄存器设为-1
15 0040109f 33c0            xor     eax,eax ; 将EAX置为0
16 004010a1 f2ae            repne   scasb ; 在EDI开始的字符串中寻找0（AL），即求长度
17 004010a3 f7d1            not     ecx ; 对ECX取反
18 004010a5 83c1ff          add     ecx,0xffffffff ; 对ECX减1
19 004010a8 51              push    ecx ; 压入ECX，即strlen(szTemp)
20 004010a9 8d55f8          lea     edx,[ebp-0x8] ; 将szTemp的有效地址放入EDX寄存器
21 004010ac 52              push    edx; 压入EDX，即szTemp
22 004010ad 6848804000      push    0x408048 ; 压入字符串常量，即"%s;Len=%d.\n"
23 004010b2 e829000000      call    LocalVar!printf (004010e0) ; 调用printf函数
24 004010b7 83c40c          add     esp,0xc ; 释放19、21、22行压入的参数
25 004010ba 5f              pop     edi ; 弹出第5行压入的EDX寄存器，恢复其以前值
26 004010bb 8be5            mov     esp,ebp ; 将EBP寄存器的值赋给ESP
27 004010bd 5d              pop     ebp ; 恢复EBP寄存器的以前值
28 004010be c3              ret ; 返回
29 004010bf cc              int     3 ; 补位用的断点指令
```

P600

命令U

- WinDBG的反汇编命令
- U 虚拟地址
  - 反汇编指定地址开始的8条指令 (安腾下9条)
- U 虚拟地址 Ln
  - 反汇编指定地址开始的n条指令
- U 虚拟地址 L-n
  - 反汇编从指定地址-n处开始到指定地址的指令
  - 容易出错，难以指向指令起点

命令ub

- 反方向反汇编
- 从参数指定地址开始反向求解
- 典型应用：栈回溯时，观察子函数被调用经过

```
0:000> kv
ChildEBP RetAddr  Args to Child
00128f54 7e4191be 7e4191f1 00128f98 00000000 ntdll!KiFastSystemCallRet (FPO: [0,0,0])
00128f74 5d0c8a6a 00128f98 00000000 00000000 USER32!NtUserGetMessage+0xc
00128fd4 5d0c8c55 00000000 0101b728 00000001 COMCTL32!_RealPropertySheet+0x2a7 (FPO: [1,15,4])
00128fec 5d0c8c70 0012ec14 00000000 0012fa98 COMCTL32!_PropertySheet+0x138 (FPO: [2,0,4])
00128ffc 0100b996 0012ec14 00000000 00000000 COMCTL32!PropertySheetW+0xf (FPO: [1,0,0])
0012fa98 01013a07 0101ae88 00000007 00252258 gflags!ShowGflagsUI+0x596 (FPO: [1,6820,0])
0012ff7c 01015d07 00000000 00392bf0 003956b0 gflags!wmain+0x12f7 (FPO: [2,310,0])
0012ffc0 7c817077 0154f6f2 0154f736 7ffde000 gflags!_initterm_e+0x163 (FPO: [Non-Fpo])
0012fff0 00000000 01015e38 00000000 78746341 kernel32!BaseProcessStart+0x23 (FPO: [Non-Fpo])
0:000> ub 0100b996
gflags!ShowGflagsUI+0x563:
0100b963 898594f1ffff    mov     dword ptr [ebp-0E6Ch],eax
0100b969 c78598f1ffff00000000 mov dword ptr [ebp-0E68h],0
0100b973 8d8db0f1ffff    lea     ecx,[ebp-0E50h]
0100b979 898d9cf1ffff    mov     dword ptr [ebp-0E64h],ecx
0100b97f c785a0f1ffffd0750001 mov dword ptr [ebp-0E60h],offset gflags!PropSheetPageProc (010075d0)
0100b989 8d957cf1ffff    lea     edx,[ebp-0E84h]
0100b98f 52              push    edx
0100b990 ff1530100001    call    dword ptr [gflags!_imp__PropertySheetW (01001030)]
```

识别函数

- 最重要的软件“单元”
- 识别函数起点
  - 函数序言(Prolog)
  - CALL指令
  - 调试符号
- 识别函数结束点
  - 函数结语(Epilog)
  - RET指令
  - 调试符号

函数序言

函数体

函数结语

函数序言例析1


```
0:000> u 00401010
AcsVio!main [C:\DbgLabs\AcsVio\AcsVio.cpp @ 8]:
00401010 55              push    ebp
00401011 8bec            mov     ebp,esp
00401013 83ec48          sub     esp,48h
```

- 保存父函数栈帧基地址
- 将本函数的栈帧基地址记录到EBP
- SUB ESP分配栈空间给局部变量
- 最基本的三个动作

函数结语例析1


```
AcsVio!main+0x7f [C:\DbgLabs\AcsVio\AcsVio.cpp @ 32]:
   32 0040108f 33c0            xor     eax,eax
   33 00401091 5f              pop     edi
   33 00401092 5e              pop     esi
   33 00401093 5b              pop     ebx
   33 00401094 83c448          add     esp,48h
   33 00401097 3bec            cmp     ebp,esp
   33 00401099 e8c2000000      call    AcsVio!_chkesp (00401160)
   33 0040109e 8be5            mov     esp,ebp
   33 004010a0 5d              pop     ebp
   33 004010a1 c3              ret
```

- 最后6句
- ADD ESP释放栈空间局部变量
- 检查栈指针
- 恢复栈指针
- 恢复父函数栈帧基地址
- 返回

函数序言例析2

```
0:000> uf gflags!ShowGflagsUI
gflags!ShowGflagsUI:
0100b400 8bff            mov     edi,edi
0100b402 55              push    ebp
0100b403 8bec            mov     ebp,esp
0100b405 b8906a0000      mov     eax,6A90h
0100b40a e871aa0000      call    gflags!_alloca_probe (01015e80)
0100b40f a114ab0101      mov     eax,dword ptr [gflags!__security_cookie (0101ab14)]
0100b414 33c5            xor     eax,ebp
0100b416 8945fc          mov     dword ptr [ebp-4],eax
0100b419 6804010000      push    104h
0100b41e 6a00            push    0
0100b420 8d85b0f1ffff    lea     eax,[ebp-0E50h]
0100b426 50              push    eax
0100b427 e83caa0000      call    gflags!memset (01015e68)
```

- 分配了0x6A90字节的栈空间
  - 因为超过一个页长度，所以要调用函数来逐页分配
- 安全Cookie用于检测栈上的缓冲区溢出

函数结语例析2

```
gflags!ShowGflagsUI+0x59b:
0100b99b ff1598100001    call    dword ptr [gflags!_imp__GetLastError (01001098)]
0100b9a1 eb02            jmp     gflags!ShowGflagsUI+0x5a5 (0100b9a5)

gflags!ShowGflagsUI+0x5a3:
0100b9a3 33c0            xor     eax,eax

gflags!ShowGflagsUI+0x5a5:
0100b9a5 8b4dfc          mov     ecx,dword ptr [ebp-4]
0100b9a8 33cd            xor     ecx,ebp
0100b9aa e898a40000      call    gflags!__security_check_cookie (01015e47)
0100b9af 8be5            mov     esp,ebp
0100b9b1 5d              pop     ebp
0100b9b2 c20400          ret     4
```

准备返回值

- 从栈上读出Cookie，检查完好性
- 恢复栈指针
- 恢复父函数栈帧基地址
- 返回并弹出参数，暗示有1个参数

函数序言例析3


```
0:000> u kernel32!ReadFile
kernel32!ReadFile:
7c801812 6a20            push    20h
7c801814 68489c807c      push    offset kernel32!`string'+0xc (7c809c48)
7c801819 e8b80c0000      call    kernel32!_SEH_prolog (7c8024d6)
```

- 调用_SEH_prolog函数
  - 登记异常处理器
  - 建立栈帧基地址
  - 分配栈空间
- Windows内核函数和系统DLL普遍适用此类序言

_SEH_prolog

- 支持SEH的函数序言函数

```
0:000> uf kernel32!_SEH_prolog
kernel32!_SEH_prolog:
7c8024d6 68d89a837c      push    offset kernel32!_except_handler3 (7c839ad8)
7c8024db 64a100000000    mov     eax,dword ptr fs:[00000000h]
7c8024e1 50              push    eax
7c8024e2 8b442410        mov     eax,dword ptr [esp+10h]
7c8024e6 896c2410        mov     dword ptr [esp+10h],ebp
7c8024ea 8d6c2410        lea     ebp,[esp+10h]
7c8024ee 2be0            sub     esp,eax
7c8024f0 53              push    ebx
7c8024f1 56              push    esi
7c8024f2 57              push    edi
7c8024f3 8b45f8          mov     eax,dword ptr [ebp-8]
7c8024f6 8965e8          mov     dword ptr [ebp-18h],esp
7c8024f9 50              push    eax
7c8024fa 8b45fc          mov     eax,dword ptr [ebp-4]
7c8024fd c745fcffffffff  mov     dword ptr [ebp-4],0FFFFFFFFh
7c802504 8945f8          mov     dword ptr [ebp-8],eax
7c802507 8d45f0          lea     eax,[ebp-10h]
7c80250a 64a300000000    mov     dword ptr fs:[00000000h],eax
7c802510 c3              ret
```

函数结语例析3


```
kernel32!ReadFile+0xb9:
7c801894 33c0            xor     eax,eax
7c801896 40              inc     eax

kernel32!ReadFile+0x1c5:
7c801897 e8750c0000      call    kernel32!_SEH_epilog (7c802511)
7c80189c c21400          ret     14h
```

- 调用_SEH_epilog函数
  - 注销异常处理器
  - 恢复栈指针
  - 恢复EBP寄存器
- 释放参数并返回
  - 14h暗示有0x14/4=5个参数

```
0:000> uf kernel32!_SEH_epilog
kernel32!_SEH_epilog:
7c802511 8b4df0          mov     ecx,dword ptr [ebp-10h]
7c802514 64890d00000000  mov     dword ptr fs:[0],ecx
7c80251b 59              pop     ecx
7c80251c 5f              pop     edi
7c80251d 5e              pop     esi
7c80251e 5b              pop     ebx
7c80251f c9              leave
7c802520 51              push    ecx
7c802521 c3              ret
```

C++

```
BOOL WINAPI ReadFile(
  __in        HANDLE hFile,
  __out       LPVOID lpBuffer,
  __in        DWORD nNumberOfBytesToRead,
  __out_opt   LPDWORD lpNumberOfBytesRead,
  __inout_opt LPOVERLAPPED lpOverlapped
);
```

传递this指针

- 调用C++类的方法时，需要将this指针传递给方法
- 因调用协议和编译器而不同
  - 默认和FASTCALL协议使用寄存器传递
    - VC++使用ECX
    - Borland C++使用EAX
  - C协议和STDCALL协议通过栈传递
  - Pascal协议作为最后一个参数传递
- 可以据此判断C++方法，定位对象地址

索引栈帧和识别参数

- 帧指针——EBP (P602)
  - 浮动栈上的稳定参照物
  - *EBP = 父函数的EBP
  - *(EBP+4) = 函数返回地址
  - *(EBP+8) = 栈上的第一个参数
  - *(EBP+C) = 栈上的第二个参数
  - …
- ESP ——FPO (P604)
  - 优化措施
  - 不利于调试

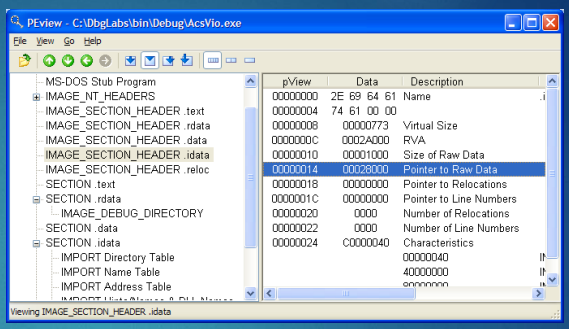
识别函数返回值

- 适合EAX(<=4字节)的使用EAX
- 8字节的用EDX:EAX
- 长度大于8字节的使用隐含参数用引用传递
- 使用EAX的情况最多

```
0:000> u kernel32!GetLastError
kernel32!GetLastError:
7c830771 64a118000000    mov     eax,dword ptr fs:[00000018h]
7c830777 8b4034          mov     eax,dword ptr [eax+34h]
7c83077a c3              ret
```

识别变量

- 局部变量
  - EBP-xxx
  - ESP+xxx
- 全局变量
  - 位于模块数据区



mov eax,dword ptr [AcsVio!_crtheap (00428114)]

```
msvcrt!_heap_alloc+0xc1:
77c2c3aa 46              inc     esi
77c2c3ab 833d1c24c67701  cmp     dword ptr [msvcrt!__active_heap (77c6241c)],1
77c2c3b2 7406            je      msvcrt!_heap_alloc+0xd1 (77c2c3ba)
77c2c3b4 83c60f          add     esi,0Fh
77c2c3b7 83e6f0          and     esi,0FFFFFFF0h
77c2c3ba 56              push    esi
77c2c3bb 6a00            push    0
77c2c3bd ff351824c677    push    dword ptr [msvcrt!_crtheap (77c62418)]
```

识别字符串

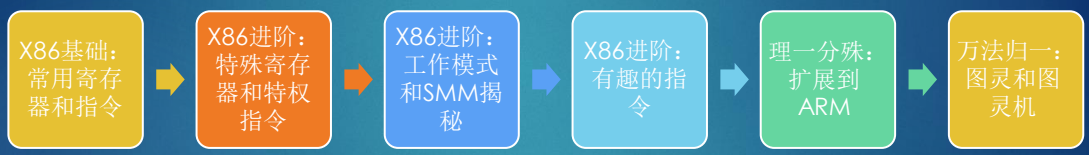
- 根据已知函数原型的识别字符串类型的参数
- 根据地址识别字符串常量
- 使用PEView或者!dh命令寻找.rdata节的起始地址
- 使用db命令浏览
- 搜索字符串s –sa <起始地址> L10000

```
SECTION HEADER #2
   .rdata name
    143C virtual size
   22000 virtual address
    2000 size of raw data
   22000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         (no align specified)
         Read Only
```

```
0:000> db 00422000
00422000  00 00 00 00 b0 93 54 49-00 00 00 00 02 00 00 00  ......TI........
00422010  30 00 00 00 00 00 00 00-00 a0 02 00 52 65 61 64  0...........Read
00422020  46 69 6c 65 00 00 00 00-4b 65 72 6e 65 6c 33 32  File....Kernel32
00422030  2e 64 6c 6c 00 00 00 00-48 69 2c 20 65 76 65 72  .dll....Hi, ever
00422040  79 20 6f 6e 65 2e 20 0a-54 68 69 73 20 73 69 6d  y one. .This sim
00422050  70 6c 65 20 75 74 69 6c-69 74 79 20 77 69 6c 6c  ple utility will
00422060  20 73 69 6d 75 6c 61 74-65 20 61 6e 20 61 63 63   simulate an acc
00422070  65 73 73 20 76 69 6f 6c-61 74 69 6f 6e 2e 0a 2d  ess violation..-
```

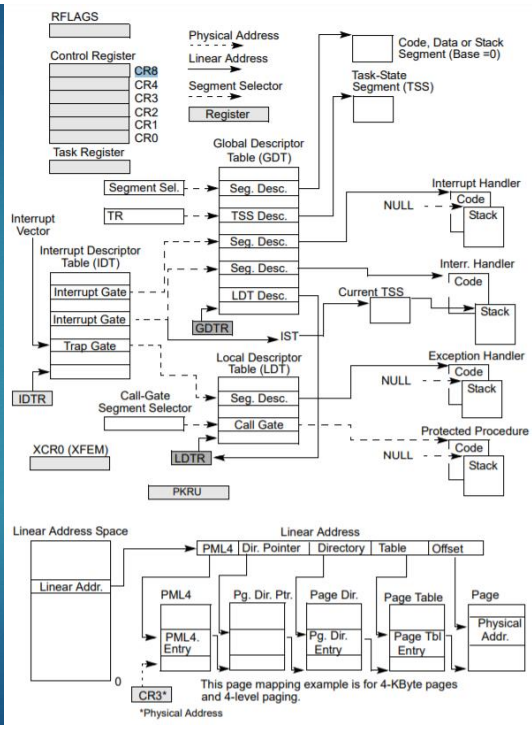
识别流程语句

- If then else
- Switch case break
- Do while
- For
- 发挥想象力！
- 实践中积累经验

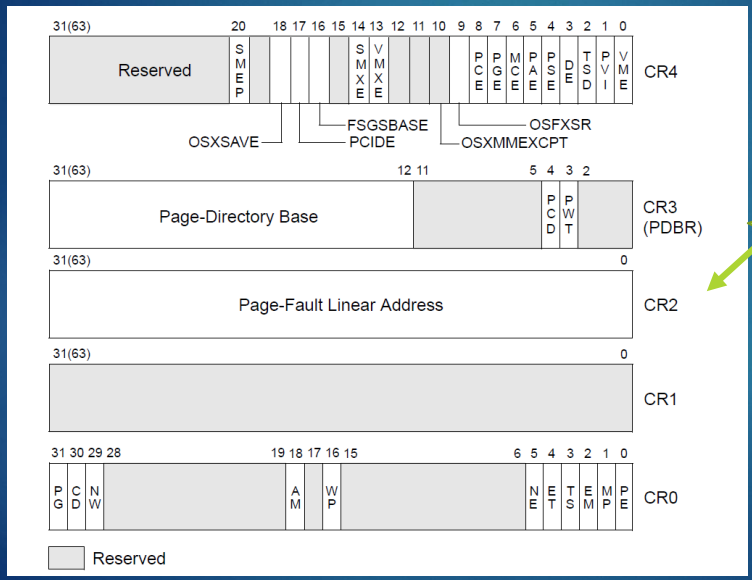


系统寄存器和数据结构

- 一般供操作系统内核使用
- 学习操作系统原理之要津



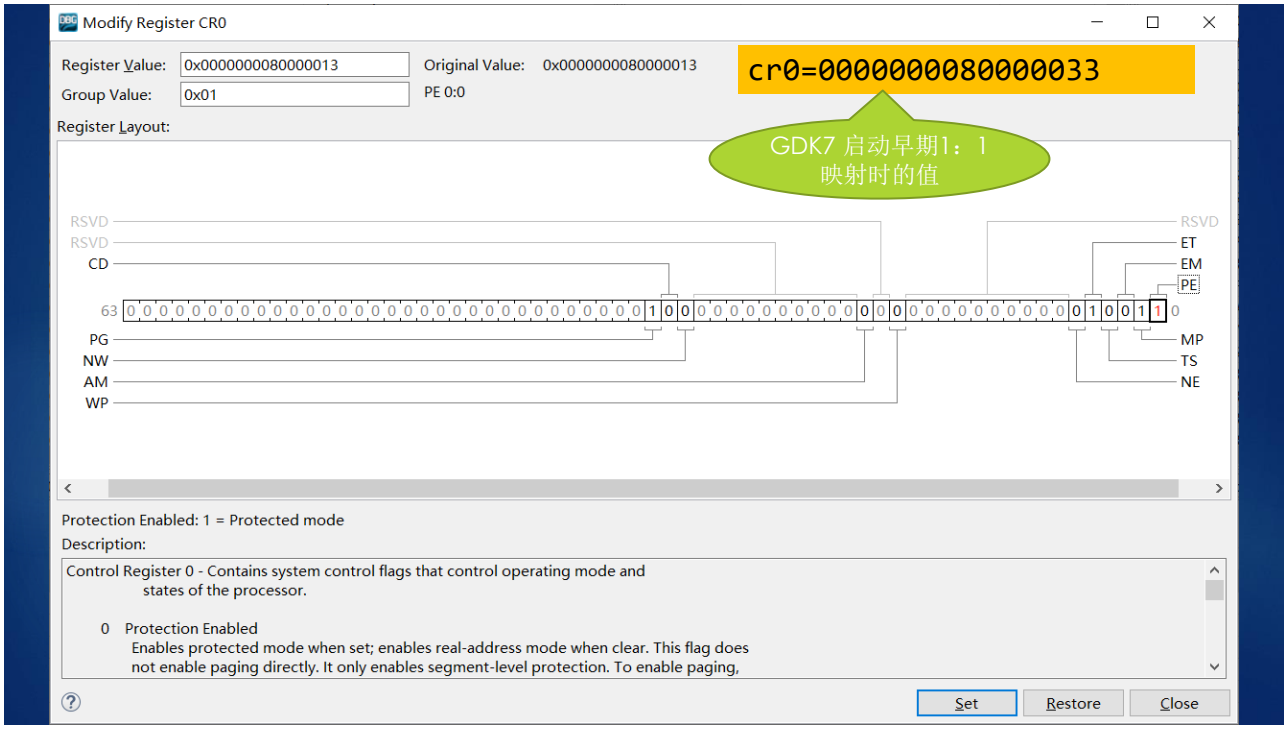
控制寄存器(Control Registers)



内存管理之机要，内存一讲详细介绍

控制全局之意

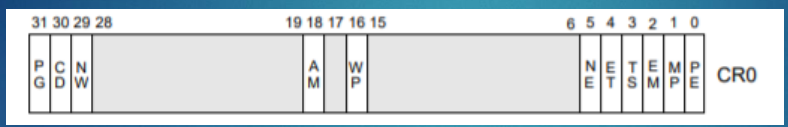
p2858



格物：W10转储中的观察值

```
Struct
{
    char Flag;
    char padding[3];
    int Age;
}
```

0: kd> r cr0
cr0=0000000080050031

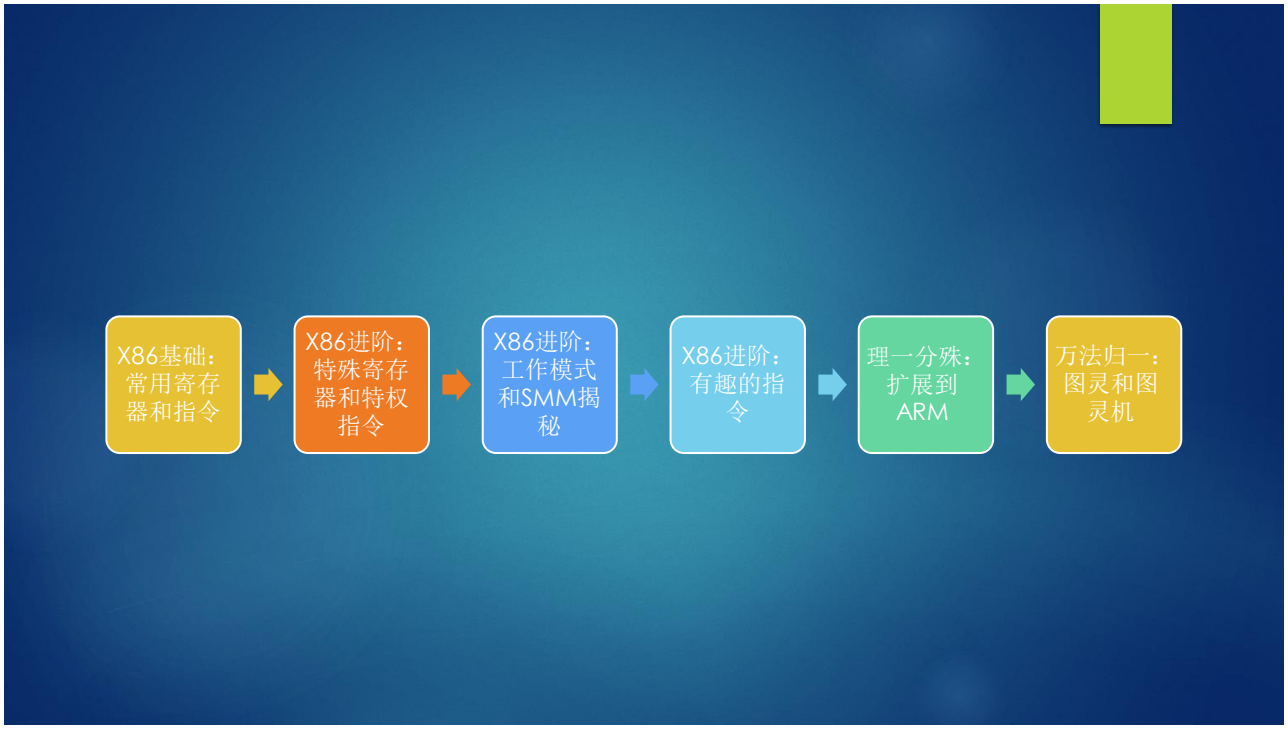


10000000 00000101 00000000 00110001

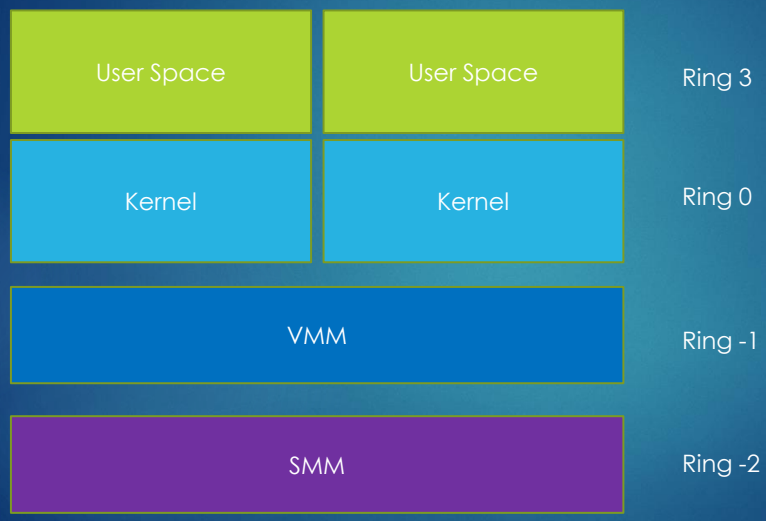
CR0.AM Alignment Mask

CR0.WP Write Protect

桌面阶段，已经启用写保护和内存对齐检查（帮助发现应用程序的性能问题）



多层架构



十万米高空看X86架构

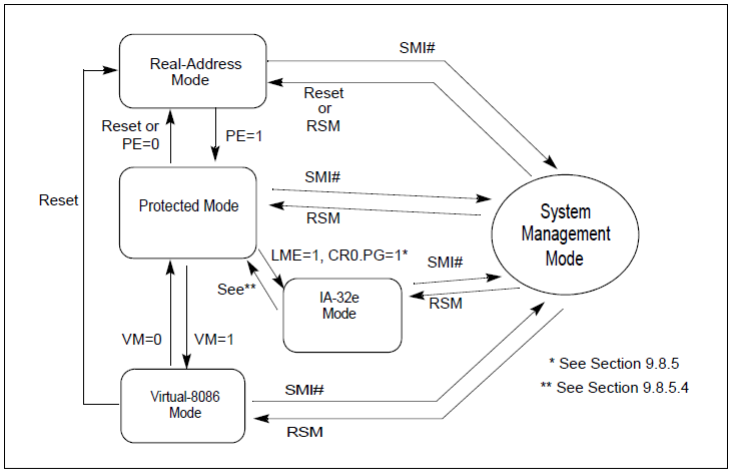
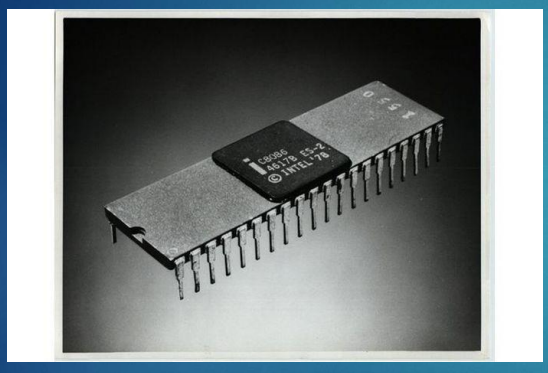


Figure 2-3. Transitions Among the Processor's Operating Modes

从实模式开始



June 8, 1978

从约定好的起跑地点出发

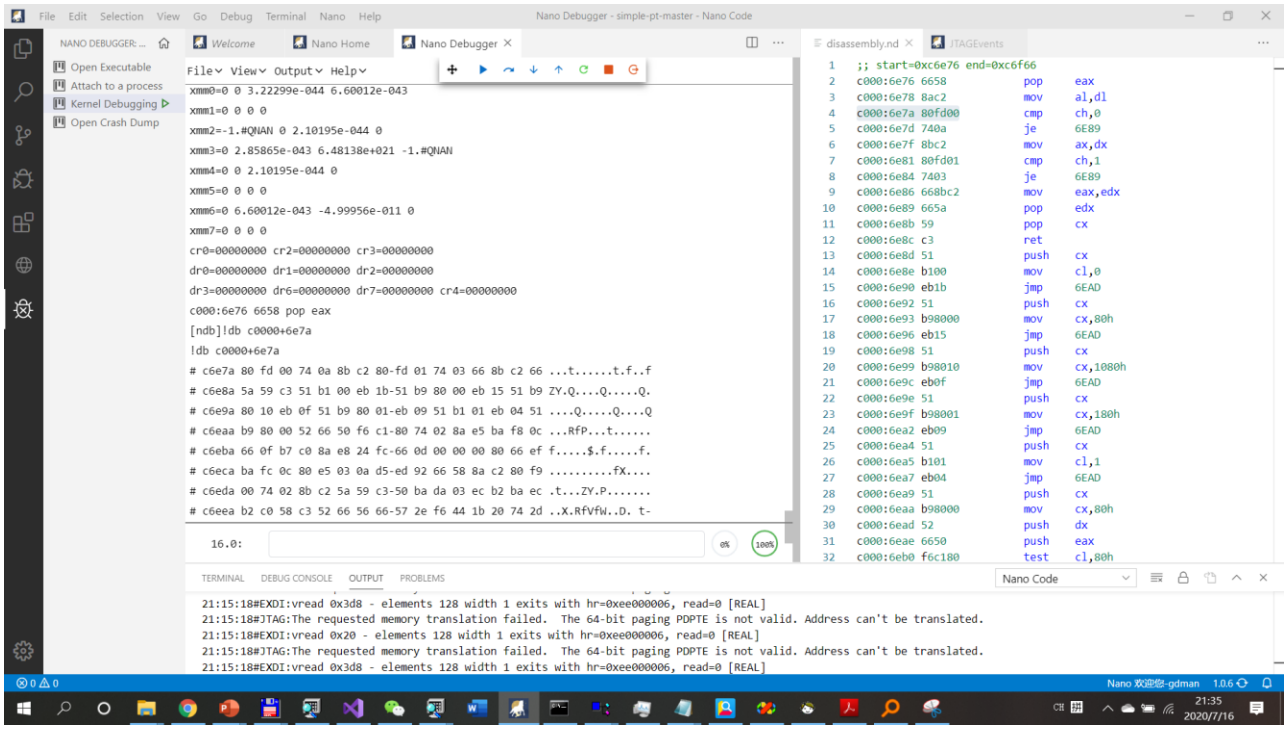
0xF000:0xFFF0

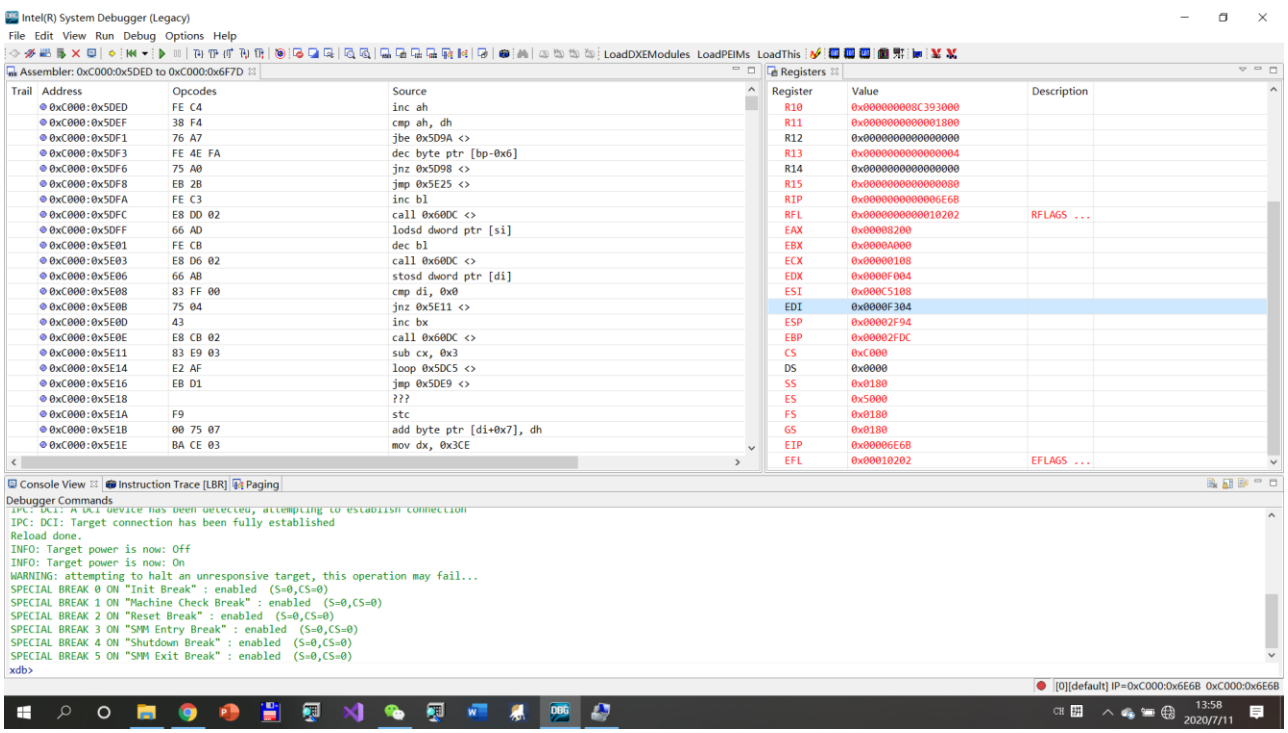
Name	Id	State	Address	Location
IA				
	0	Init	0xF000:0xFFF0	
	1	Wait for SIPI loop	0xF000:0x0000	
	2	Wait for SIPI loop	0xF000:0x0000	
	3	Shutdown	0x0038:0x000000008D7CB172	

黑暗中崛起

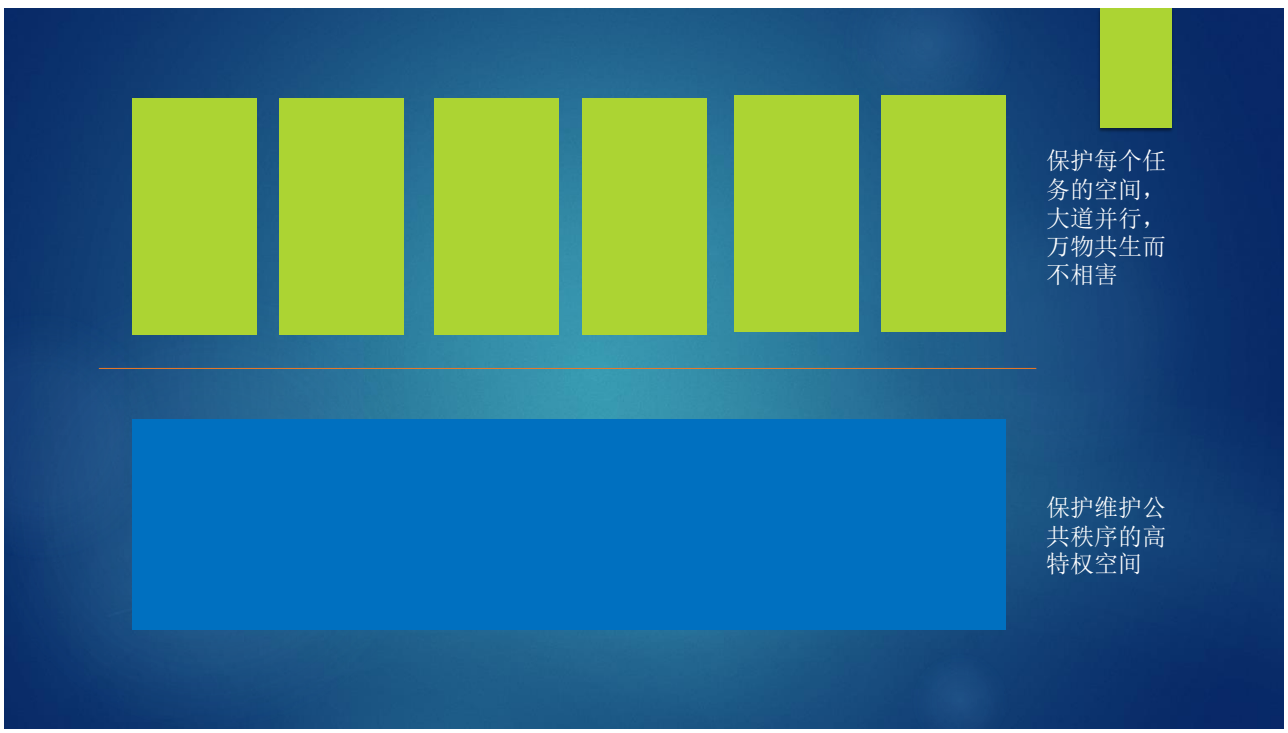
此时还没有栈可用，不可以call，只能jmp

```
u f0000+fff0
00000000`000ffff0 ea5be000f03034 jmp 3430:F000E05B
00000000`000ffff7 2f das
00000000`000ffff8 30342f xor byte ptr [edi+ebp],dh
00000000`000ffffb 3136 xor dword ptr [esi],esi
00000000`000ffffd 00fc add ah,bh
00000000`000fffff 005a5a add byte ptr [edx+5Ah],bl
00000000`00100002 5a pop edx
00000000`00100003 5a pop edx
```





```
u 0038:00000000`8d7dd3d6 L30
0038:00000000`8d7dd3d6 4883c408 add rsp,8
0038:00000000`8d7dd3da 488bf4 mov rsi,rsp
0038:00000000`8d7dd3dd 0fae0e fxrstor [rsi]
0038:00000000`8d7dd3e0 4881c400020000 add rsp,200h
0038:00000000`8d7dd3e7 4883c430 add rsp,30h
0038:00000000`8d7dd3eb 58 pop rax
0038:00000000`8d7dd3ec 0f22c0 mov cr0,rax
0038:00000000`8d7dd3ef 4883c408 add rsp,8
0038:00000000`8d7dd3f3 58 pop rax
0038:00000000`8d7dd3f4 0f22d0 mov cr2,rax
0038:00000000`8d7dd3f7 58 pop rax
0038:00000000`8d7dd3f8 0f22d8 mov cr3,rax
0038:00000000`8d7dd3fb 58 pop rax
0038:00000000`8d7dd3fc 0f22e0 mov cr4,rax
0038:00000000`8d7dd3ff 58 pop rax
0038:00000000`8d7dd400 440f22c0 mov cr8,rax
0038:00000000`8d7dd404 8f4528 pop qword ptr [rbp+28h]
0038:00000000`8d7dd407 4883c430 add rsp,30h
0038:00000000`8d7dd40b 8f4518 pop qword ptr [rbp+18h]
0038:00000000`8d7dd40e 58 pop rax
0038:00000000`8d7dd40f 58 pop rax
0038:00000000`8d7dd410 58 pop rax
0038:00000000`8d7dd411 488ec0 mov es,ax
0038:00000000`8d7dd414 58 pop rax
```

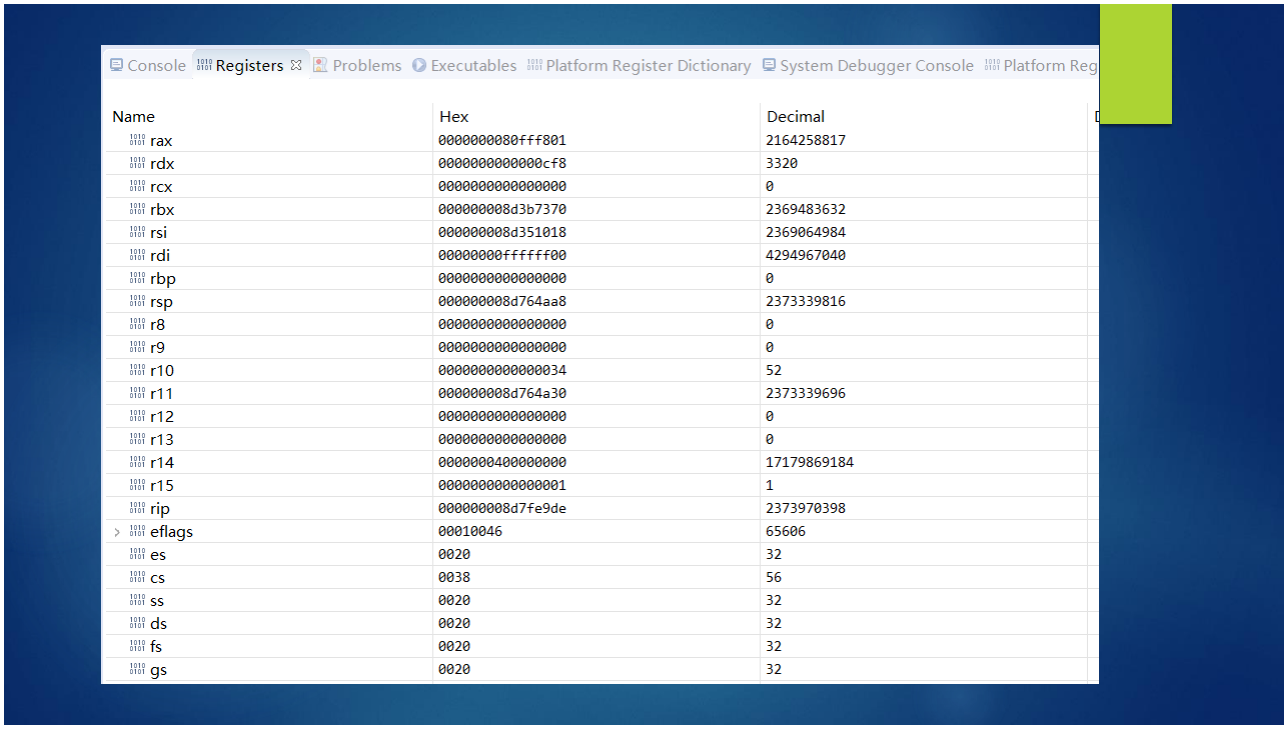


1:1映射

```
u
0038:00000000`8d7dd3ef 4883c408 add rsp,8
0038:00000000`8d7dd3f3 58 pop rax
0038:00000000`8d7dd3f4 0f22d0 mov cr2,rax
0038:00000000`8d7dd3f7 58 pop rax
0038:00000000`8d7dd3f8 0f22d8 mov cr3,rax
0038:00000000`8d7dd3fb 58 pop rax
0038:00000000`8d7dd3fc 0f22e0 mov cr4,rax
0038:00000000`8d7dd3ff 58 pop rax
[ndb]!db 8d7dd3ef
!db 8d7dd3ef
#8d7dd3ef 48 83 c4 08 58 0f 22 d0-58 0f 22 d8 58 0f 22 e0 H...X."."X."."
#8d7dd3ff 58 44 0f 22 c0 8f 45 28-48 83 c4 30 8f 45 18 58 XD.".."E(H..0.E.X
#8d7dd40f 58 58 48 8e c0 58 48 8e-d8 8f 45 20 8f 45 38 5f XXH..XH...E .E8_
#8d7dd41f 5e 48 83 c4 08 8f 45 30-5b 5a 59 58 41 58 41 59 ^H....E0[ZYXAXAY
#8d7dd42f 41 5a 41 5b 41 5c 41 5d-41 5e 41 5f 48 8b e5 5d AZA[A\A]A^A_H..]
#8d7dd43f 48 83 c4 10 48 83 7c 24-e0 00 74 14 48 83 7c 24 H...H.|$..t.H.|$
#8d7dd44f d8 01 74 04 ff 64 24 e0-48 83 ec 08 ff 64 24 e8 ..t..d$.H....d$.
#8d7dd45f 48 83 3d 09 69 ff ff 00-74 18 50 48 8b c4 48 8b H.=.i...t.PH..H.
r cr0
cr0=0000000080000033
```

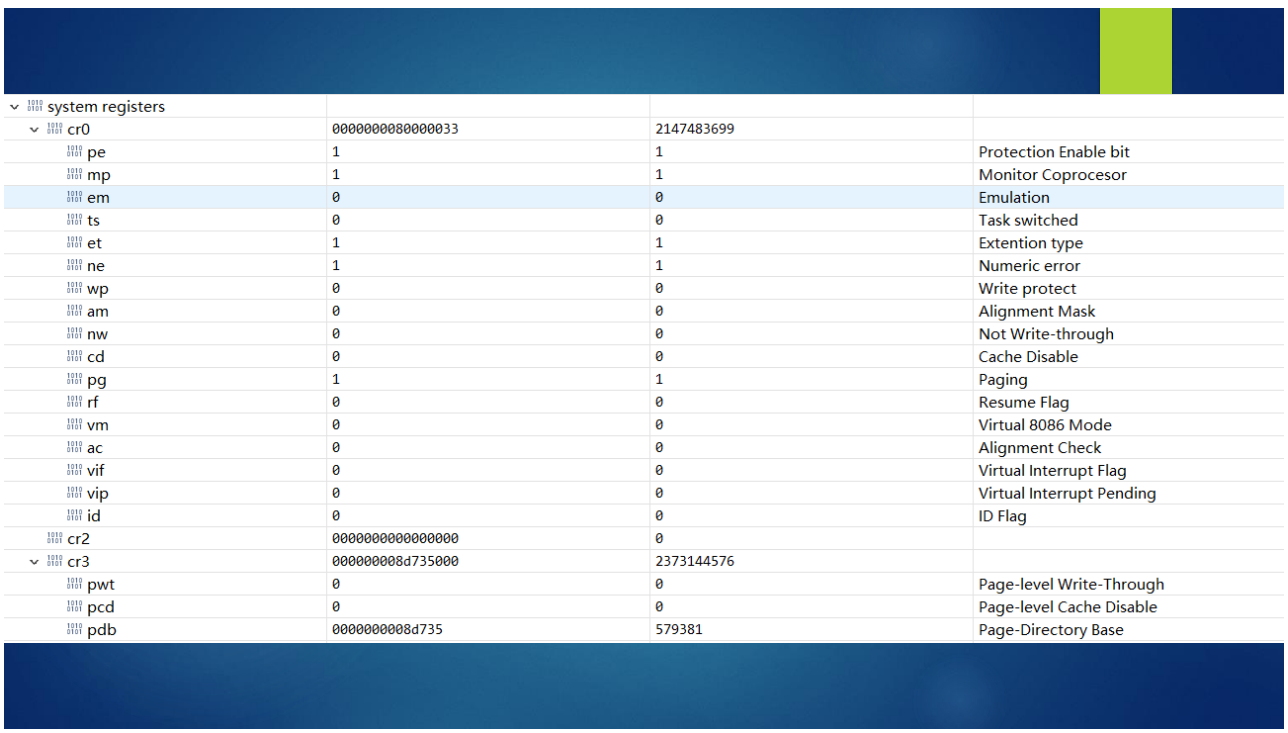
Console | Registers | Problems | Executables | Platform Register Dictionary | System Debugger Console | Platform Reg

Name	Hex	Decimal
rax	0000000080fff801	2164258817
rdx	00000000000000cf8	3320
rcx	0000000000000000	0
rbx	000000008d3b7370	2369483632
rsi	000000008d351018	2369064984
rdi	00000000ffffff00	4294967040
rbp	0000000000000000	0
rsp	000000008d764aa8	2373339816
r8	0000000000000000	0
r9	0000000000000000	0
r10	0000000000000034	52
r11	000000008d764a30	2373339696
r12	0000000000000000	0
r13	0000000000000000	0
r14	0000000400000000	17179869184
r15	0000000000000001	1
rip	000000008d7fe9de	2373970398
eflags	00010046	65606
es	0020	32
cs	0038	56
ss	0020	32
ds	0020	32
fs	0020	32
gs	0020	32



Console | Registers | Problems | Executables | Platform Register Dictionary | System Debugger Console | Platform Reg

Name	Hex	Decimal
rax	0000000080fff801	2164258817
rdx	0000000000000cf8	3320
rcx	0000000000000000	0
rbx	000000008d3b7370	2369483632
rsi	000000008d351018	2369064984
rdi	00000000ffffff00	4294967040
rbp	0000000000000000	0
rsp	000000008d764aa8	2373339816
r8	0000000000000000	0
r9	0000000000000000	0
r10	0000000000000034	52
r11	000000008d764a30	2373339696
r12	0000000000000000	0
r13	0000000000000000	0
r14	0000000400000000	17179869184
r15	0000000000000001	1
rip	000000008d7fe9de	2373970398
eflags	00010046	65606
es	0020	32
cs	0038	56
ss	0020	32
ds	0020	32
fs	0020	32
gs	0020	32



system registers			
cr0	0000000080000033	2147483699	
pe	1	1	Protection Enable bit
mp	1	1	Monitor Coprocesor
em	0	0	Emulation
ts	0	0	Task switched
et	1	1	Extention type
ne	1	1	Numeric error
wp	0	0	Write protect
am	0	0	Alignment Mask
nw	0	0	Not Write-through
cd	0	0	Cache Disable
pg	1	1	Paging
rf	0	0	Resume Flag
vm	0	0	Virtual 8086 Mode
ac	0	0	Alignment Check
vif	0	0	Virtual Interrupt Flag
vip	0	0	Virtual Interrupt Pending
id	0	0	ID Flag
cr2	0000000000000000	0	
cr3	000000008d735000	2373144576	
pwt	0	0	Page-level Write-Through
pcd	0	0	Page-level Cache Disable
pdb	0000000008d735	579381	Page-Directory Base

cr4	0000000000000668	1640	
vme	0	0	Virtual-8086 Mode Extensions
pvi	0	0	Protected-Mode Virtual Interrupts
tsd	0	0	Time Stamp Disable
de	1	1	Debugging Extensions
pse	0	0	Page Size Extensions
pae	1	1	Physical Address Extension
mce	1	1	Machine-Check Enable
pge	0	0	Page Global Enable
pce	0	0	Performance-Monitoring Counter Enable
osfxsr	1	1	OS Support for FXSAVE and FXRSTOR inst...
osxmmexcpt	1	1	OS Support for Unmasked SIMD Floating-...
vmxe	0	0	VMX-Enable Bit
smxe	0	0	SMX-Enable Bit
fsgsbase	0	0	FSGSBASE-Enable Bit
pcide	0	0	PCID-Enable Bit
osxsave	0	0	XSAVE and Processor Extended States-En...
smep	0	0	SMEP-Enable Bit
smap	0	0	SMAP-Enable Bit
pke	0	0	Protection-Key-Enable Bit

cr8	0000000000000000	0	
tpl	0	0	Task Priority Level
efer	0000000000000d00	3328	
SYSCALL Enable	0	0	SYSCALL Enable
IA-32e Mode Enable	1	1	IA-32e Mode Enable
IA-32e Mode Active	1	1	IA-32e Mode Active
Execute Disable Bit Enable	1	1	Execute Disable Bit Enable
mxcsr	00001f80	8064	
ie	0	0	Invalid Operation Flag
de	0	0	Denormal Flag
ze	0	0	Divide-by-Zero Flag
oe	0	0	Overflow Flag
ue	0	0	Underflow Flag
pe	0	0	Precision Flag
daz	0	0	Denormals Are Zeros
im	1	1	Invalid Operation Mask
dm	1	1	Denormal Operation Mask
zm	1	1	Divide-by-Zero Mask
om	1	1	Overflow Mask
um	1	1	Underflow Mask
pm	1	1	Precision Mask
rc	0	0	Rounding Control
fz	0	0	Flush to Zero

tssbas	8d734050	2373140560	
tsslim	00000067	103	
tssar	008b	139	
Type	b	11	Type
S	0	0	descriptor type flag
DPL	0	0	descriptor privilege level field
P	1	1	segment-present flag
AVL	0	0	Available for use by system software
L	0	0	64-bit code segment flag
D/B	0	0	default operation size/default stack point...
G	0	0	granularity flag
csar	a09b	41115	
Type	b	11	Type
S	1	1	descriptor type flag
DPL	0	0	descriptor privilege level field
P	1	1	segment-present flag
AVL	0	0	Available for use by system software
L	1	1	64-bit code segment flag
D/B	0	0	default operation size/default stack point...
G	1	1	granularity flag

MSR

- Model-specific registers
- 很多已经成为架构的一部分， 稳定下来
- SDM 卷4

读CPU的体温

```
else if (strcmp(szArgs[0],"!hot")==0)
    {
        hr = pNdRemote->ReadMSR(IA32_THERM_STATUS, val);
        if (hr == S_OK)
        {
            int nDegree = ((val & 0x3f0000) >> 16);
            m_pNdBoss->CmdOut(DEBUG_OUTPUT_ERROR, "CPU's temperature is %d degree in Celsius.\n"
                "It is quite %s.(%I64x)\n",
                nDegree, nDegree<80?"cool":"hot", val);
        }
        else
            m_pNdBoss->CmdOut(DEBUG_OUTPUT_ERROR, "Failed to read MSR %x. Are you using Intel CPU?\n",
                IA32_THERM_STATUS);
    }
```

Nano Debugger(www.nanocode.cn)中!hot命令的源代码

IA32_THERM_STATUS

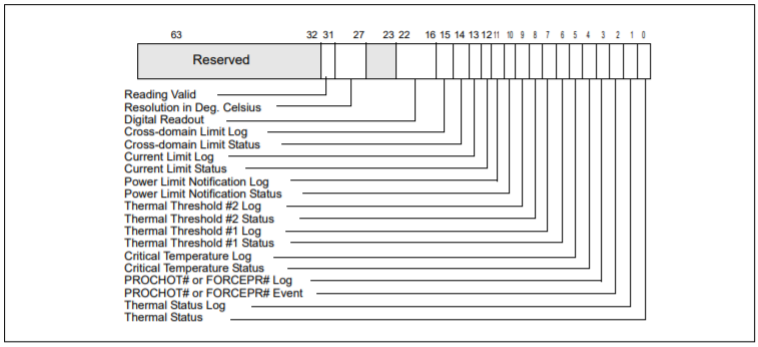
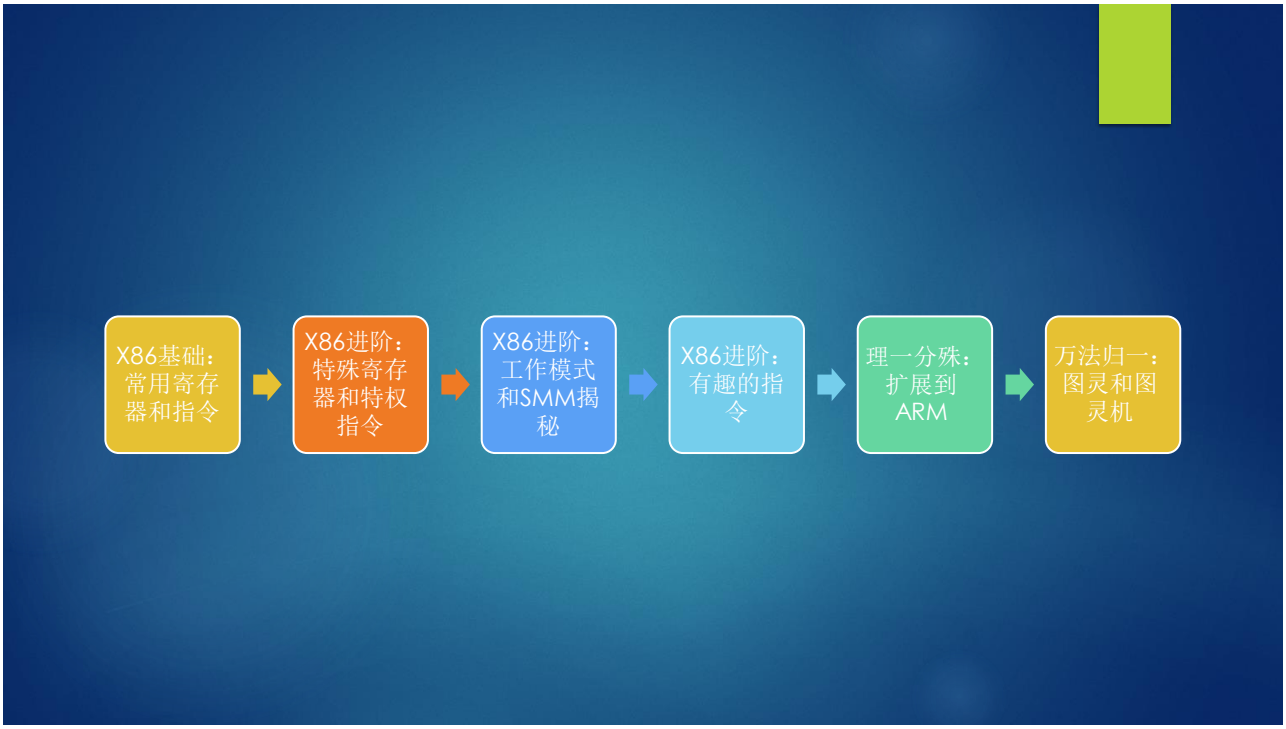


Figure 14-10. IA32_THERM_STATUS Register With HWP Feedback



断点指令

- Int 3
- 机器码0xCC
- CPU执行即报断点异常，转去执行调试逻辑

4.1.5 特殊用途

因为 INT 3 指令的特殊性，所以它有一些特别的用途。让我们从一个有趣的现象说起。当用 VC6 进行调试时，我们常常会观察到一块刚分配的内存或字符串数组里面被填满了“CC”。如果是在中文环境下，因为 0xCCCC 恰好是汉字“烫”字的简码，所以会观察到很多“烫烫烫……”（见图 4-3），而 0xCC 又正好是 INT 3 指令的机器码，这是偶然的么？当然不是。因为这是编译器故意这样做的。为了辅助调试，编译器在编译调试版本时会用 0xCC 来填充刚刚分配的缓冲区。这样，如果因为缓冲区或堆栈溢出时程序指针意外指向了这些区域，那么便会因为遇到 INT 3 指令而马上中断到调试器。

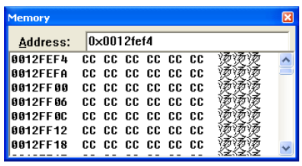


图 4-3 填充了 INT 3 指令的缓冲区

事实上，除了以上用法，编译器还用 INT 3 指令来填充函数或代码段末尾的空闲区域，即用它来做内存对齐。这也可以解释为什么有时我们没有手工插入任何对 INT 3 的调用，但还会遇到图 4-1 所示的对话框。

花指令

- X86指令集中，指令不等长，1到十几字节不等
- 反汇编时，需要推断指令的起始位置
  - 通常是根据已知的指令起点来推测
- 添加冗余字节，故意欺骗反汇编器

```
0047effd 0000              add     byte ptr [eax],al
0047efff 00558b            add     byte ptr [ebp-75h],dl
0047f002 ec                in      al,dx
0047f003 6aff              push    0FFFFFFFFh
0047f005 681d321305        push    513321Dh
0047f00a 6888888808        push    8888888h
0047f00f 64a100000000      mov     eax,dword ptr fs:[00000000h]
0047f015 50                push    eax
```

```
0:000> u 0047f000
image00400000+0x7f000:
0047f000 55              push    ebp
0047f001 8bec            mov     ebp,esp
0047f003 6aff            push    0FFFFFFFFh
0047f005 681d321305      push    513321Dh
0047f00a 6888888808      push    8888888h
0047f00f 64a100000000    mov     eax,dword pt
```

倒车指令

- Call指令是调用子函数的一般方法
- 黑客可能用ret指令来调用
  - push 参数
  - push 参数
  - push 子函数地址
  - ret

反跟踪示例

```
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
    PEXCEPTION_REGISTRATION_RECORD Next;
    PEXCEPTION_DISPOSITION Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

67

① xor eax,eax
② push dword ptr fs:[eax]
③ mov dword ptr fs:[eax],esp
④ pushfd
⑤ or byte ptr [esp+1],1
⑥ popfd
⑦ nop
⑧ nop
⑨ ret

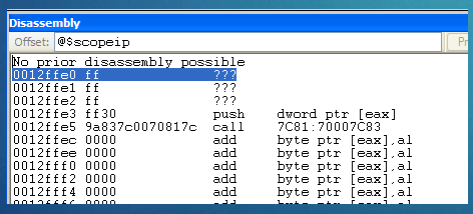
```
0:000> dd esp
0012ffbc  0012ffe0 0047f02f 7c816ff7 0122f6f2
0012ffcc  0122f760 7ffdf000 e4e149f8 0012ffc8
0012ffdc  884f7020 ffffffff 7c839a30 7c817000
0012ffec  00000000 00000000 00000000 0047f000
0012fffc  00000000 78746341 00000020 00000001
0013000c  00002498 000000c4 00000000 00000020
0013001c  00000000 00000014 00000001 00000006
0013002c  00000034 00000114 00000001 00000000
```

```
0:000> !exchain
0012ffbc: image00400000+7f02f (0047f02f)
0012ffe0: kernel32!_except_handler3+0 (7c839a30)
  CRT scope  0, filter: kernel32!BaseProcessStart+29
                func:   kernel32!BaseProcessStart+3a
```

原理

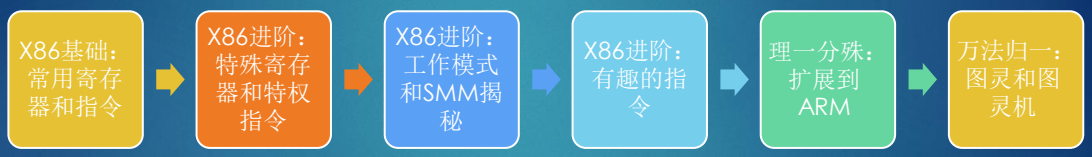
68

- 在没有调试器的情况下，单步异常会被分发给动态注册的异常处理“函数”，而这个“函数” 就是当前函数返回地址处的代码，因此可以正常返回到父函数
- 在有调试器的情况下，调试器会处理单步异常，导致函数返回到栈顶指向的EXCEPTION_REGISTRATION_RECORD结构
- 化解方法：gn



滑板指令

- Nop
- 空操作
- 只是递增程序指针
- 机器码0x90



ARM

2.9.1 ARM 的多重含义

可能是因为 ARM 公司的人太喜欢 A、R、M 这 3 个字母了，他们总是一有机会就使用这 3 个字母，不断赋予其更多含义。

ARM 缩写的最初含义是 Acorn RISC Machine，代表英国 Acron 计算机公司的 RISC 芯片项目。该项目于 1983 年开始，于 1985 年 4 月在 VLSI（总部在硅谷的半导体公司）流片并通过测试，于 1986 年开始应用于个人电脑、PDA 等领域。

1990 年，苹果公司、VLSI 准备和 Acorn 一起合作研发 ARM CPU，大家一致认为应该成立一家新的公司，于是在 1990 年 11 月成立了名为 Advanced RISC Machines Ltd.的公司。于是，ARM 缩写的含义改变为 Advanced RISC Machines。1998 年，这家公司改名为 ARM Holdings，即今天使用的名字。

A、R、M 3 个字母在 ARM 架构中的另一种重要含义是代表 ARM 架构的 A、R、M 三大系列（Profile）。

	User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

Banked Register

- 具有多份
- 比如R8-R12，SP，LR

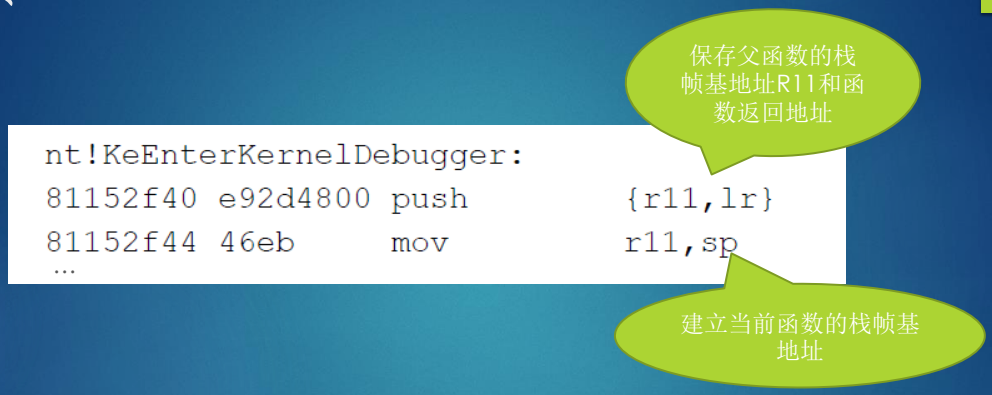
LR – Link Register

返回值放入r0，相当于x86的RAX

```
nt!KeGetCurrentStackPointer:
81034e68 4668      mov         r0,sp
81034e6a 4770      bx          lr
```

- 在调用子函数时，ARM 处理器会自动将子函数的返回地址放到这个寄存器中。如果子函数是所谓的叶子函数（不再调用子函数），那么就可以不必额外保存返回地址

续



- 如果是非叶子函数，那么通常在函数开头将LR 的值保存到栈上

联系

ARM	x86	典型用法
R0	EAX	函数返回值
R1-R5	EBX, ECX, EDX, ESI, EDI	General Purpose
R6-R10	–	
R11 (FP)	EBP	栈帧基地址
R12	–	Intra Procedural Call
R13 (SP)	ESP	栈指针
R14 (LR)	–	Link Register
R15 (PC)	EIP	程序指针
CPSR	EFLAGS	标志寄存器

指令格式

谓词执行

- Predicated Execution

- 消除分支的一种技术
- RISC中兴盛
- X86在引入了CMOV

示例

ADD R0, R1, R2	R0 = R1+R2	
MOVLE R0, #5	当LE成立时，R0 = 5	
MOV R0, R1, LSL #1	R = R1<<1	

常用指令

Instruction	Description	Instruction	Description
MOV	Move data	EOR	Bitwise XOR
MVN	Move and negate	LDR	Load
ADD	Addition	STR	Store
SUB	Subtraction	LDM	Load Multiple
MUL	Multiplication	STM	Store Multiple
LSL	Logical Shift Left	PUSH	Push on Stack
LSR	Logical Shift Right	POP	Pop off Stack
ASR	Arithmetic Shift Right	B	Branch
ROR	Rotate Right	BL	Branch with Link
CMP	Compare	BX	Branch and eXchange
AND	Bitwise AND	BLX	Branch with Link and eXchange
ORR	Bitwise OR	SWI/SVC	System Call

* https://azeria-labs.com/memory-instructions-load-and-store-part-4/

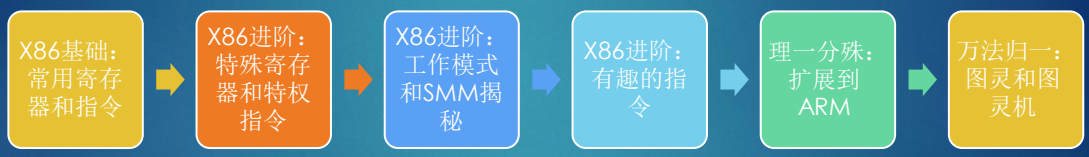
读写内存

STR　Ra, [Rb, 偏移]
把Ra的值写到Rb+偏移处

LDR　Ra, [Rc,偏移]
把Rc+偏移处的内容读到Ra

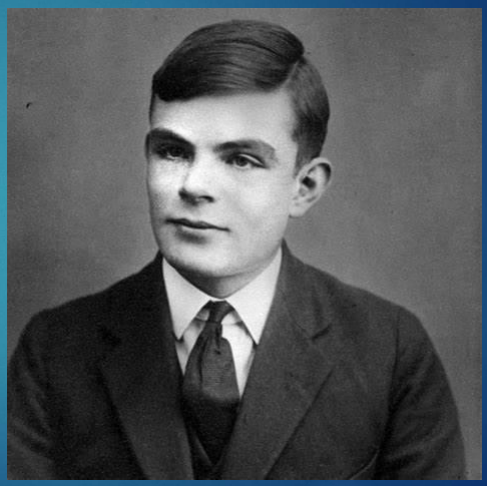
读写内存块（多个单元）

- LDM (load multiple) and STM (store multiple)
  - ldm r0, {r4,r5} 把r0指向的内存块读到R4和R5
  - stm r1, {r4,r5} 把r4和r5写到r1指向的内存块
- LDM后面的后缀
  - -IA (increase after), -IB (increase before), -DA (decrease after), -DB (decrease before)



图灵

- 1912年6月23日，艾伦·图灵在英国伦敦出生
- 1936年发表著名论文，同年到美国留学
- 1954年6月7日，自杀去世



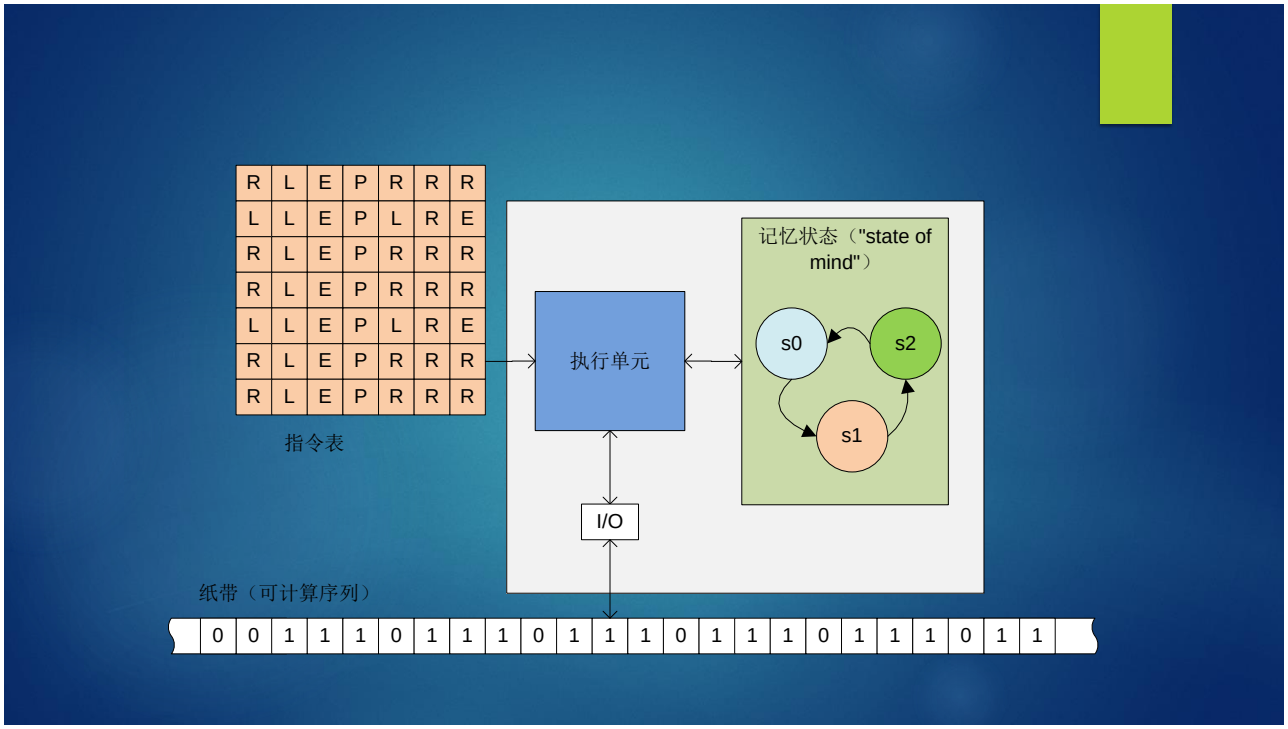
ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

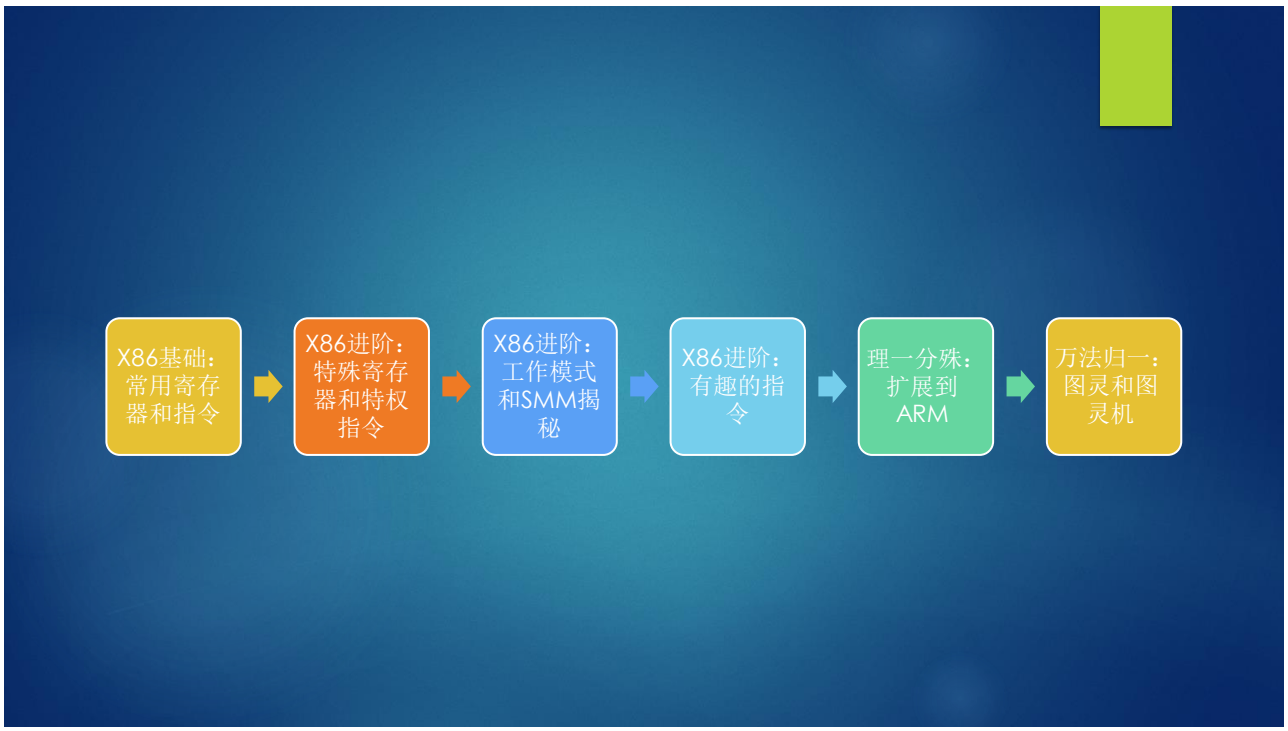
By A. M. TURING.

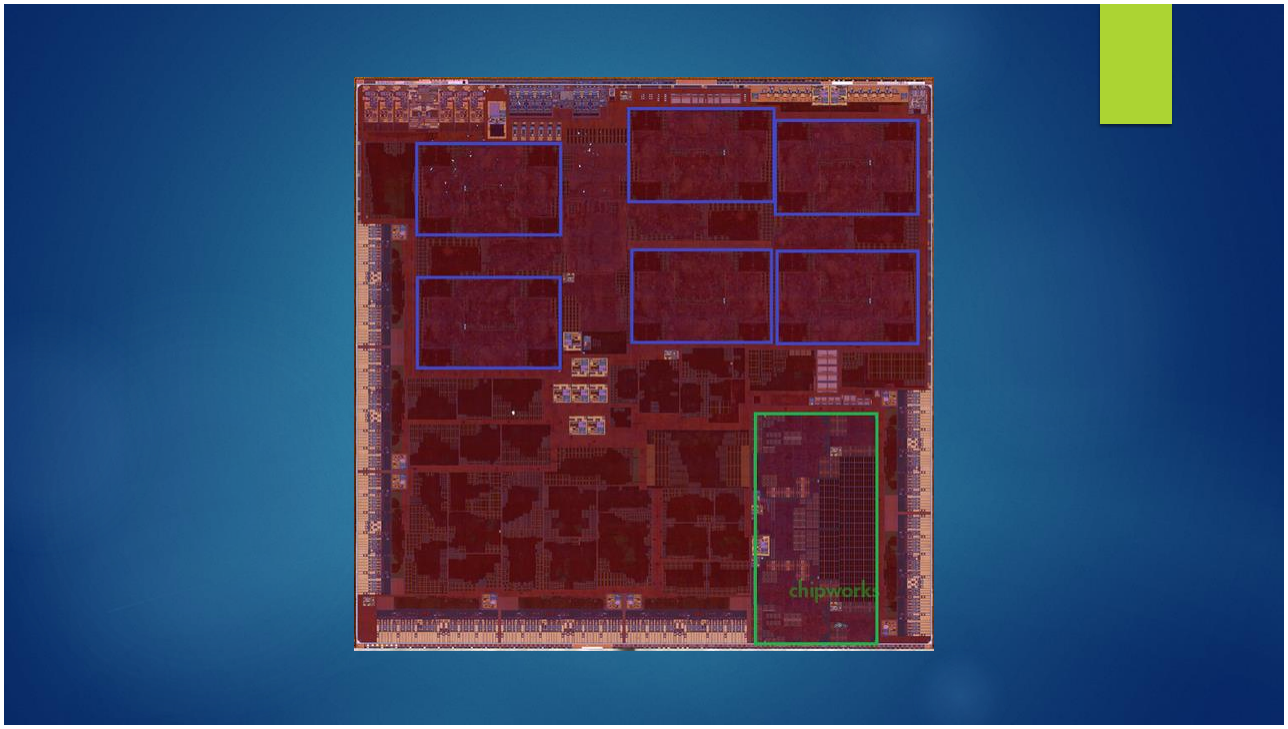
[Received 28 May, 1936.—Read 12 November, 1936.]

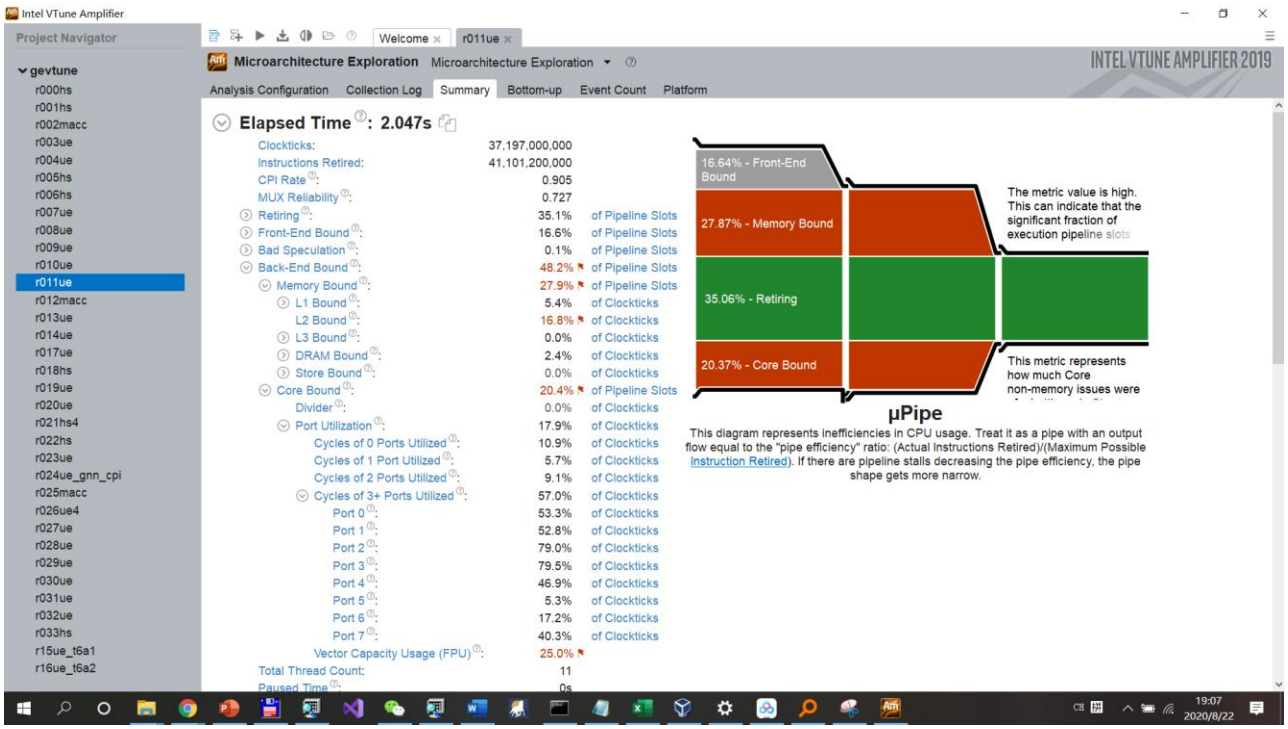
6. *The universal computing machine.*

It is possible to invent a single machine which can be used to compute any computable sequence. If this machine $\mathfrak{U}$ is supplied with a tape on the beginning of which is written the S.D of some computing machine $\mathfrak{M}$,









本讲调试命令归纳

- WinDBG反汇编
  - u，ub，uf
- GDB反汇编
  - disassemble
  - set disassembly-flavor intel
- 观察寄存器
  - r
  - info reg

课后阅读

- IA-32软件开发者手册 (SDM)
- （http://www.intel.com/products/processor/manuals/index.htm）
  - 卷1 基础
  - 卷2A 指令集 **(指令格式、指令详解A-M)**
  - 卷2B 指令集 **(指令详解N-Z)**
  - 卷3A 系统编程指南
  - 卷3B 系统编程指南（调试、VT）
  - 优化手册
  - APIC手册
- 《软件调试》第2篇
- “深入理解英特尔架构”

92

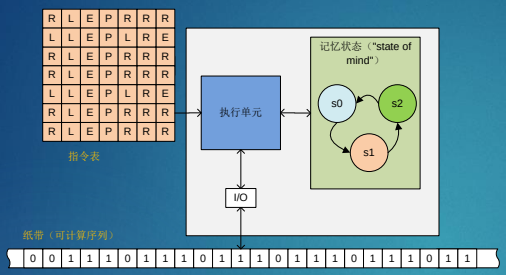
* 王宇拍摄

课后作业

- 按照试验指导完成试验1
- 阅读学习资料（books）目录中的SDM，浏览章节结构，以便以后遇到问题时结合问题阅读

切问而近思

欢迎关注格友公众号



计算机系统演义之一

图灵机和指令引擎

张银奎/格蠹科技