网页版：嵌入式知识点与笔记

微处理器程序开发基础

程序开发过程与工具

处理器计算 a=b+c 的过程？

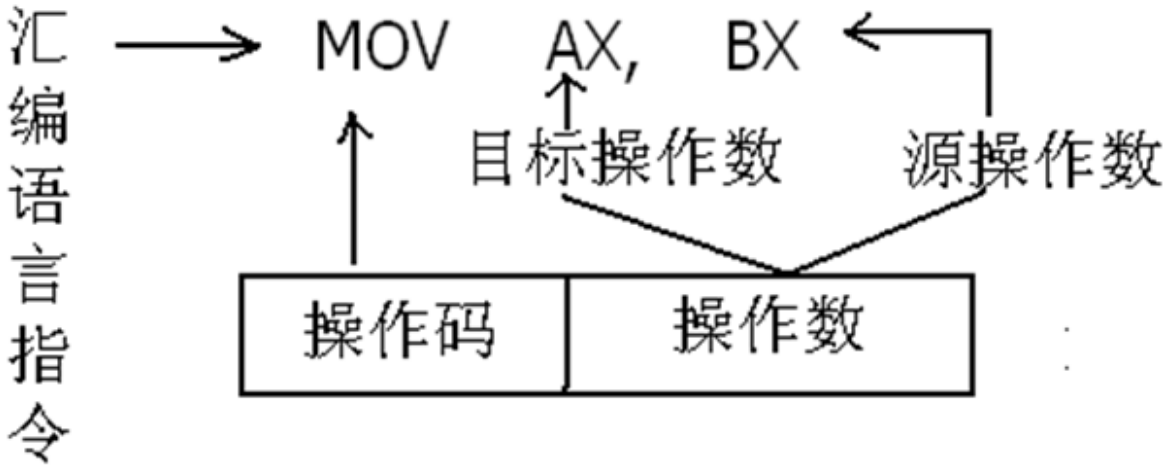
可执行代码是什么形式？如何产生的？

指令

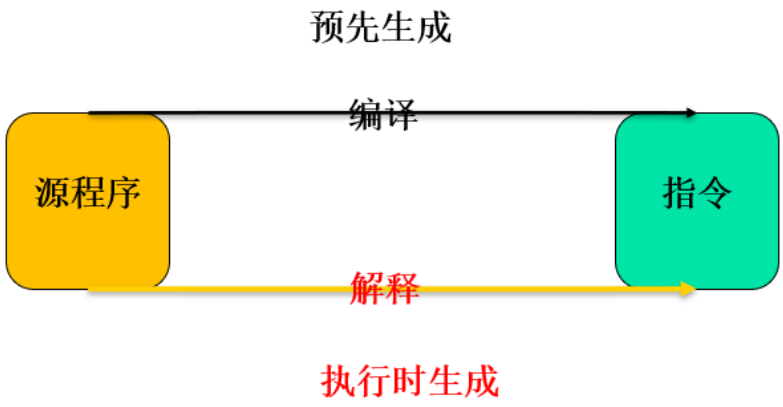
形式

机器语言：由CPU理解和执行的二进制代码

机器指令的格式（助记符）



执行代码生成



开发过程

PC程序开发

编辑　编译　汇编　连接　调试

嵌入式程序开发

编辑　编译　汇编　连接　下载　调试

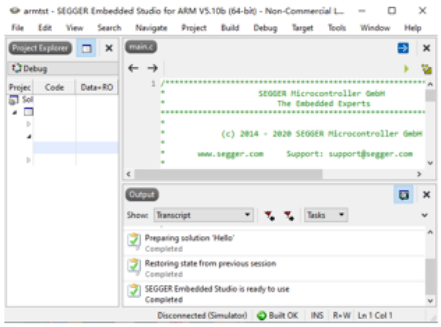
如何下载程序?

开发工具

集成开发环境

- Keil MDK（www.arm.com）

- Arm Development Studio（DS-5） （www.arm.com）
- Segger for Arm （www.segger.com）



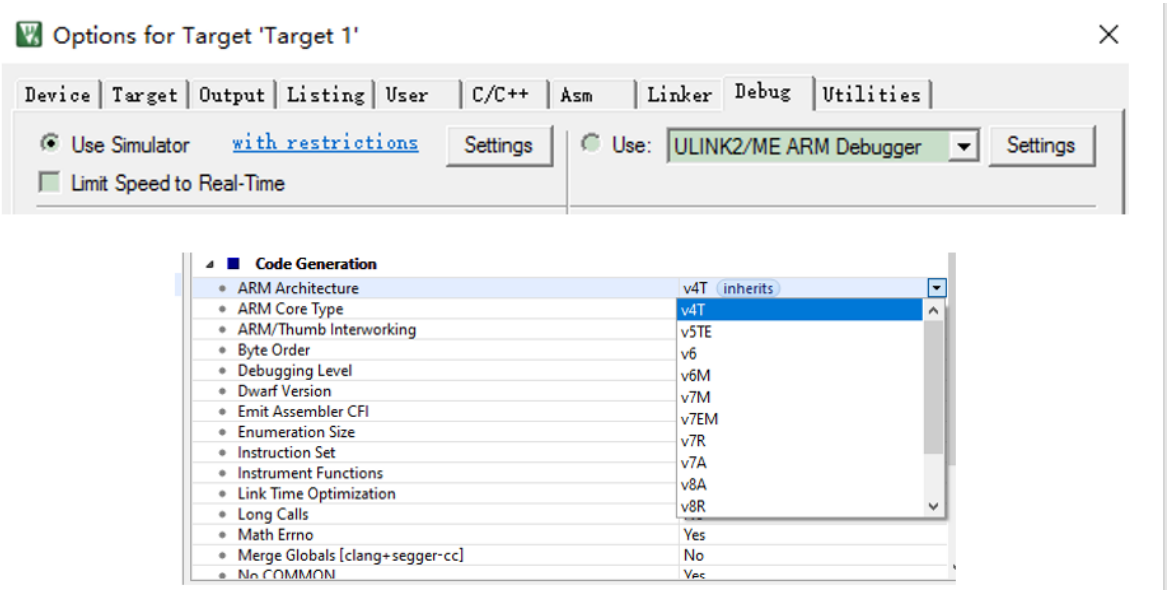
调试环境

- 物理处理器和开发板(Emulator)
  - Ulink

  - Jlink

没有开发板可以调试程序吗?

- 模拟处理器(Simulator)



目标处理器如何加载程序?

- **目标地址**
  - Linker-分配
  - Debugger-指定
- **加载方式**
  - Debugger-下载
  - 实际系统-ROM
- **启动方式**
  - 初始化
    - Debugger-程序/Set
    - 实际系统-程序
  - 程序入口
    - Debugger-跳转/Set PC
    - 实际系统 –跳转

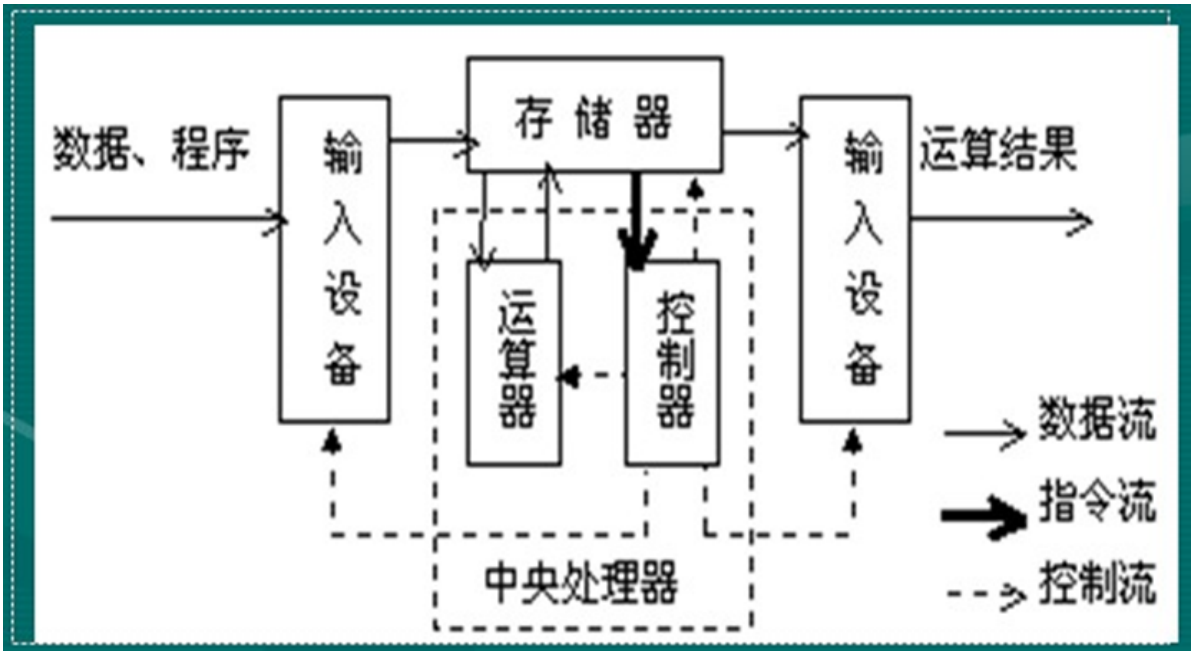
还熟悉哪些处理器程序开发环境?

处理器与程序

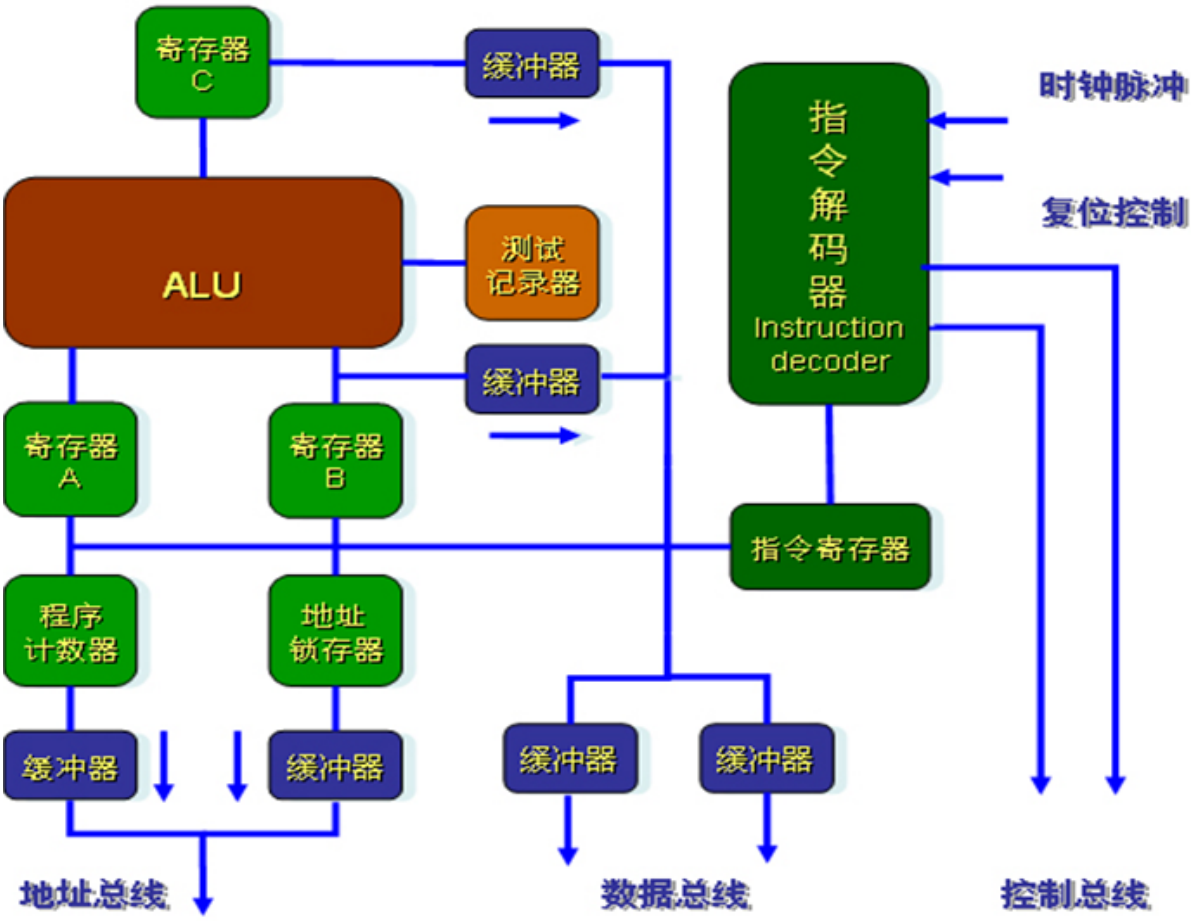
问题

处理器如何执行程序?

计算机系统



CPU内核



CPU性能

- CPI-执行一条指令所需的平均时钟周期

CPI= 一个程序的CPU时钟周期/该程序的指令数
CPU时间=IC（指令数）X CPI /时钟频率

- **CPI**：由处理器组成和指令系统决定
- 指令数：由指令系统和编译器决定

- MIPS-每秒百万条指令

MIPS=指令数/（指令的执行时间X 10^6）
=时钟频率/（CPI X 10^6 ）
执行时间=指令数/（MIPS X 10^6 ）

- **MIPS**依赖于指令集
- 同一台机器的**MIPS**可能因程序而异
- **MIPS**可能不能反映处理器的性能

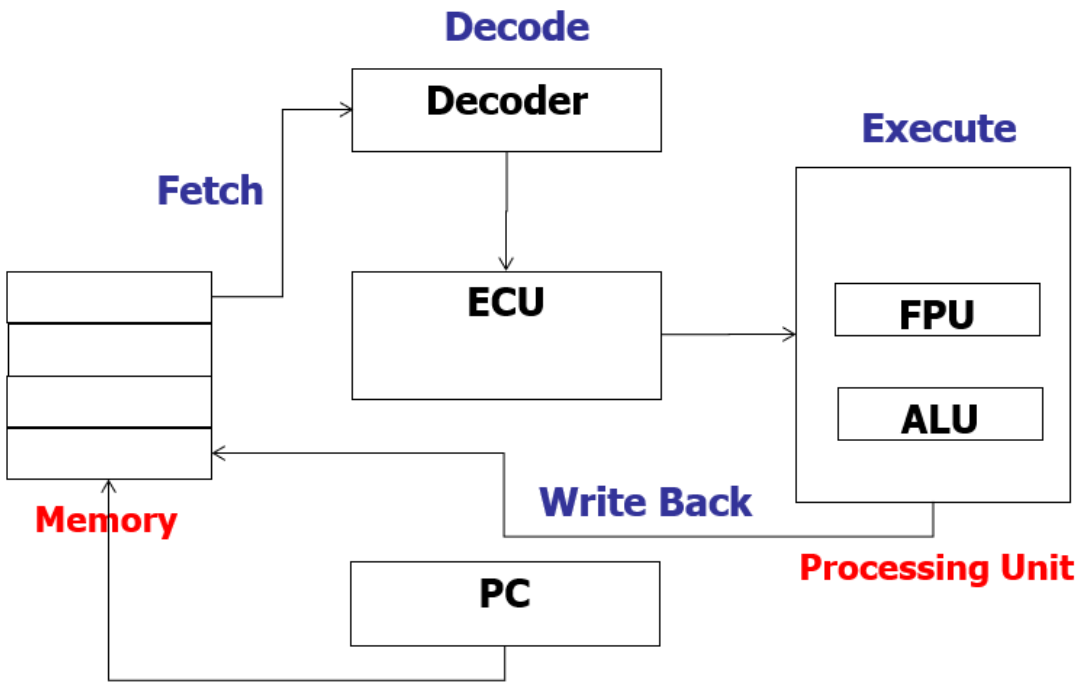
- MFLOPS-每秒百万次浮点运算次数

MFLOPS=浮点运算次数/（浮点运算的时间X 10^6）

- **MFLOPS**不能反映处理器的实际性能
- 通常用来比较浮点运算器的性能
- **Instruction**数 和 **Operation**数可能不一致

执行时间

指令执行过程



取指（IF）->译码 （ID） ->执行（IE） ->写回（WB）

指令执行时间

$$t_i = t_{if} + t_{id} + t_{ie} + t_{iw}$$

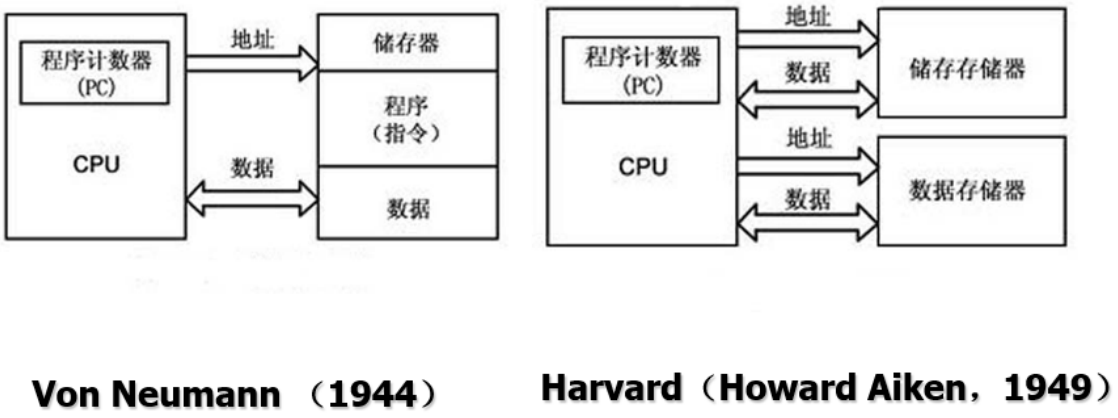
程序执行时间

$$T_p = \sum_{i=1}^{n} t_i$$

影响处理器程序运行速度的因素有哪些?

结构与性能

总线结构



Von Neumann （1944） **Harvard（Howard Aiken，1949）**

哪种结构的处理器速度更快？ 为什么?

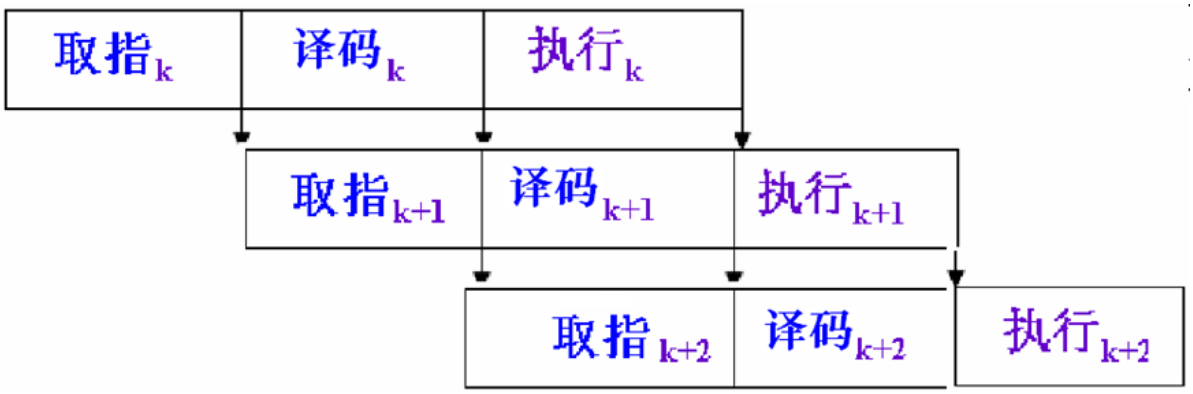
处理器字长（ALU 及通用数据寄存器）

- **8 bits**：**8008, 89C51**
- **16 bits**：**8086, Ti 54X**
- **32 bits**：**80486, Ti C3X, C6X，ARM V3-6**
- **64 bits**：**Itanium，PowerPC 970, ARM V8**
- **128 bits?**

8位处理器可以通过程序进行32位数值运算吗?

如何用C语言编程实现?

流水线



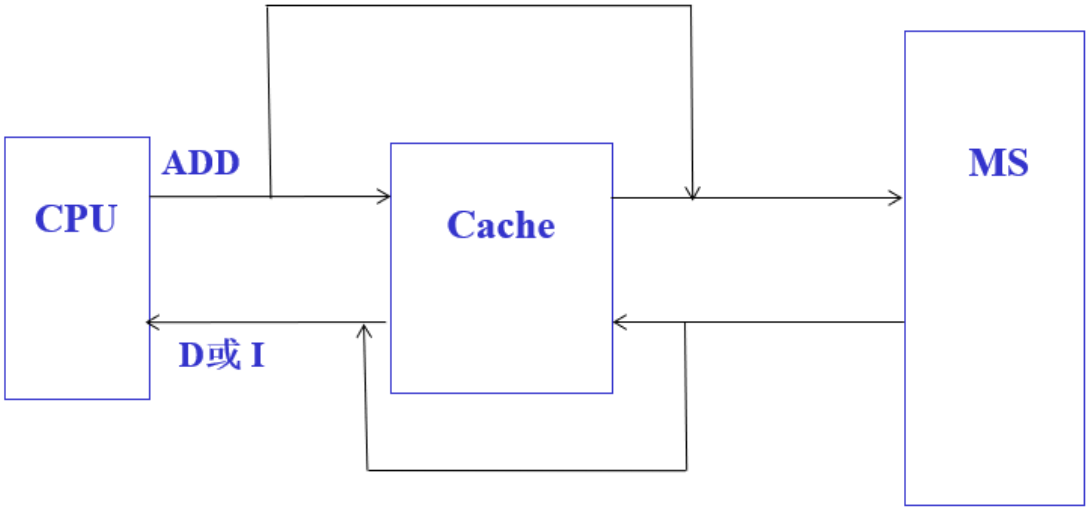
- 执行时间（n条指令）

$$T = t_{取1} + \max\{t_{译1},\ t_{取2}\} + \sum_{i=2}^{n-1}[\max\{t_{译i}, t_{取i+1},\ t_{执i-1}\}] + \max\{t_{取n},\ t_{执n-1}\} + t_{执n}$$

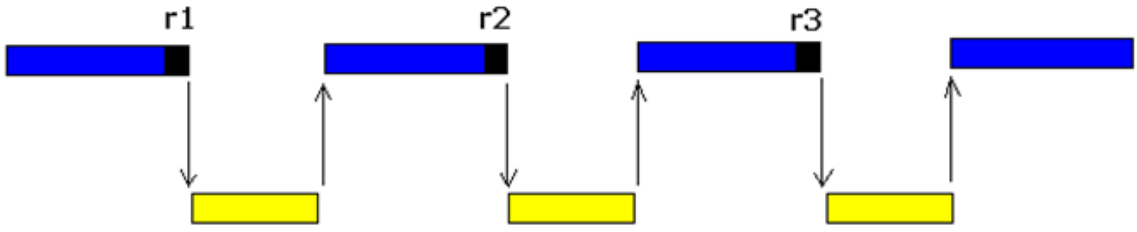
影响流水线加速效果的因素?

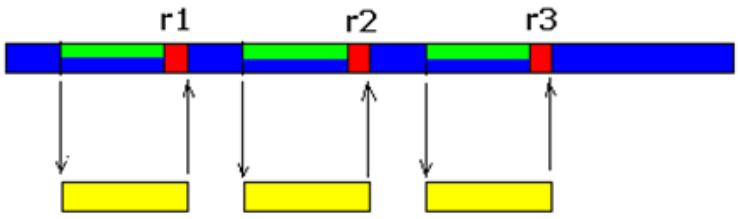
能否通过程序验证处理器是否采用了流水线?

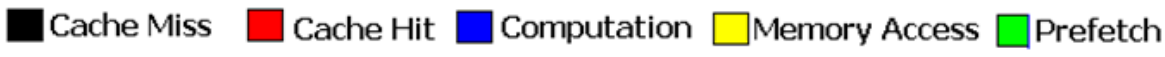
Cache



程序可以直接访问Cache中指定的数据单元?



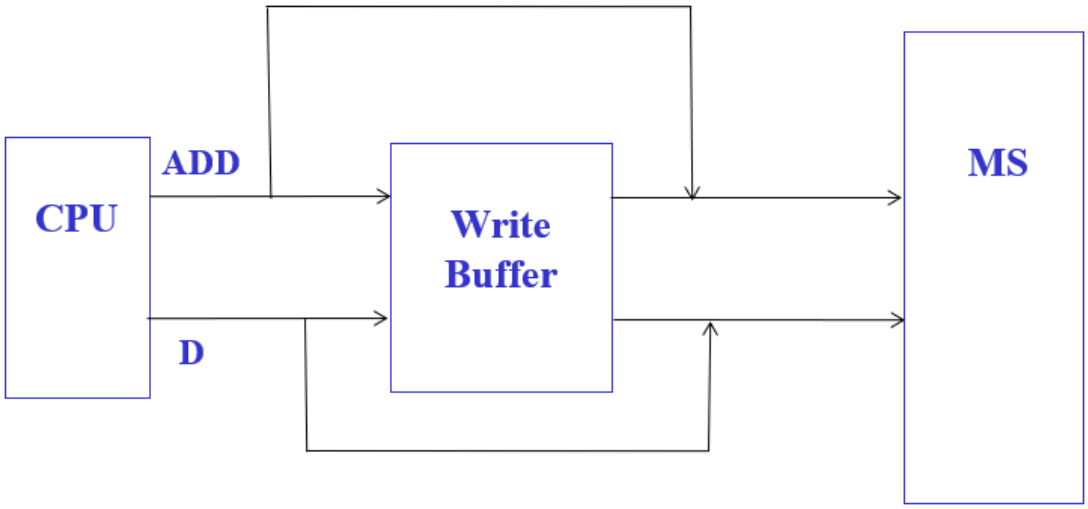




Cache 的容量对大小对加速性能的影响?

能否通过程序验证处理器中Cache是否工作?

Write buffer



Write buffer 与 Cache的区别?

指令执行过程中如何访问的数据?

寻址

定义：执行单元获取数据（地址）的方式

编址单元宽度

- 字节（8bit，0x86，ARM）
- 半字 （C54）
- 字 （C30）
- 双字

地址空间（独立编址）

- **三个地址空间**
  - 通用寄存器
  - 主存储器
  - I/O设备
- **两个地址空间**
  - 通用寄存器
  - 主存储器与IO设备
- **一个地址空间**
  - 所有存储设备统一编址

如果有多个地址空间，在程序中如何区分?

方式

- **立即数寻址**

ADD R4，#5； reg（R4） 〈- reg（R4）+5

- **寄存器寻址**

ADD R4，R3； reg（R4）〈- reg（R4）+reg（R3）

- **直接寻址（绝对）**

ADD R1，（1001）； reg（R1）〈- reg（R1）+ Mem（1001）

注：1001是内存的地址，取出该地址对应的值

- **堆栈寻址**

PUSH R1

POP R1

注：堆栈操作的对象与内存有关

隐含了什么地址?

- **间接寻址**
  - 寄存器间接寻址 ADD R4，（R1）；reg（R4）〈- reg（R4）+ Mem（reg（R1））；

    注：R1寄存器的数值对应为内存中的地址，取内存中该地址的数据（R1）
  - 存储器间接寻址 ADD R1，@(R2)； reg（R1）〈- reg（R1）+ Mem[Mem[reg（R2）]]

    注：R2寄存器数值对应为内存中的地址，改地址对应的数值所对应的地址，取内存中该地址的数据@（R2）
- **自动递增寻址**

ADD R1，（R2）++；Reg(R1)〈- reg（R4）+ Mem（reg(R2)）; R2+1

- **自动递减寻址**

ADD R1，- -（R2）； R2-1 ; Reg(R1)〈- reg（R1）+ Mem（reg(R2)）

注：自动递增递减会改变所操作寄存器的值

- **偏移量寻址**

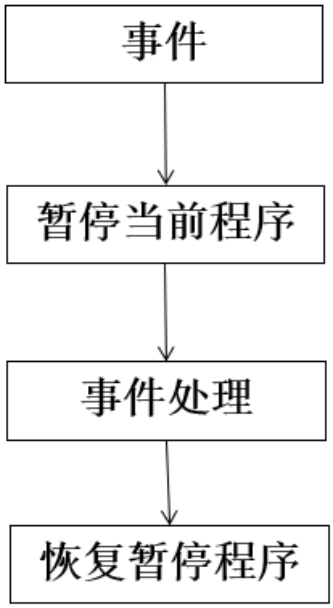
ADD R4，±100（R1）； reg（R4）〈- reg（R4）+ Mem（reg(R1)±100）

注：偏移量寻址不会改变所操作寄存器的值

处理器如何处理异常（外部）事件?

事件响应

事件响应过程



检测事件？进入事件处理程序？

中断（interrupt）

定义：**为导致程序正常执行流程发生改变的事件（不包括程序的分支情况）**

来源

- **外中断**：由于CPU 外部的原因而改变程序执行流程的过程，属于异步事件，又称为硬件中断
- **内中断**：
  - 自陷（软中断，trap）：通过处理器所拥有的软件指令、可预期地使处理器正在执行的程序的执行流程发生变化，以执行特定的程序
  - 异常：异常为CPU 自动产生的自陷（除0，非法指令，内存保护错误）

可屏蔽性

- **可屏蔽中断**
  - 能够被屏蔽掉的中断称为可屏蔽中断。
  - 一般需要先通过CPU外部的中断控制器，再与CPU相应的引脚相连。
- **不可屏蔽中断**
  - 不能够被屏蔽掉的中断称为可屏蔽中断
  - 例：复位（Reset）

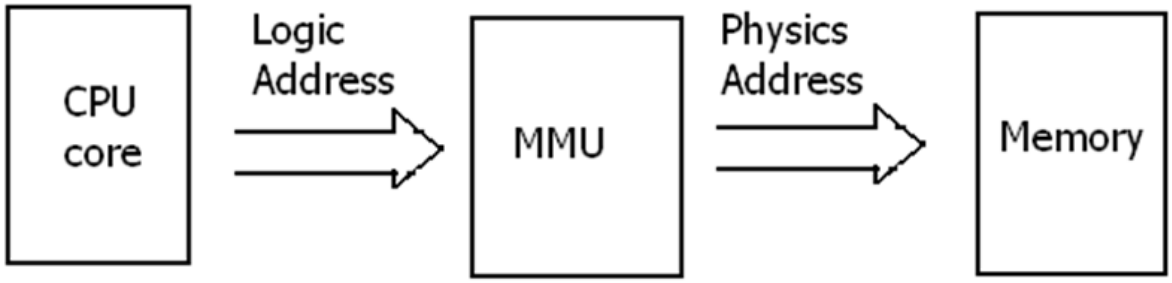
中断向量表

- **中断向量与中断服务程序入口的对照表**

支持多任务（多程序）的操作系统，如何为不同的任务分配存储空间?

虚拟地址 VS 物理地址?

内存映射



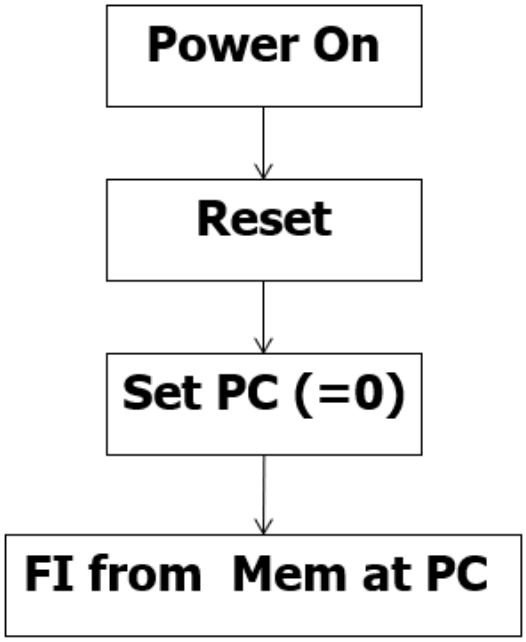
- **地址转换**
- **空间保护**

程序中如何实现地址转换?

上电后处理器是如何启动boot 程序?

系统启动

处理器启动



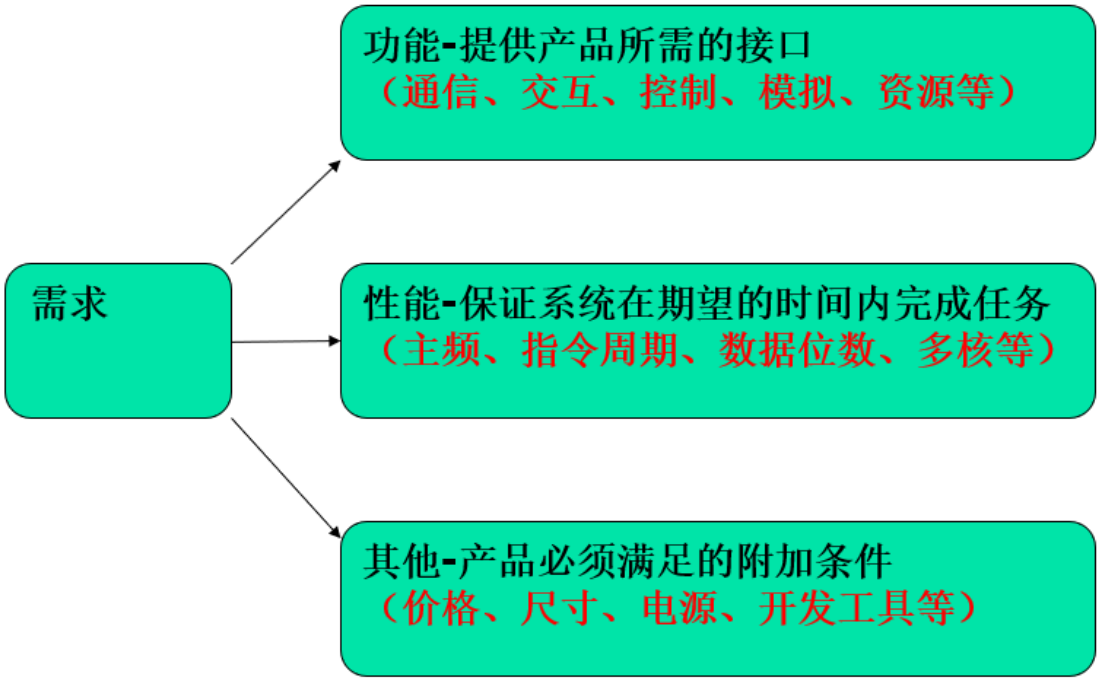
条件?

- Memory（ROM） 可执行程序（指令）
- PC指到程序入口

处理器系统

你知道哪些CPU？具体型号?

CPU要满足系统需求



系统需求

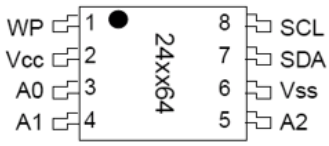
资源

- 片内 (MCU, DSP, SoC)
  - RAM，ROM，Programmable ROM
    - DATA
    - INSTRUCTION
- 片外可扩展 (DSP, MPU)
  - MEMORY
    - DATA
    - INSTRUCTION

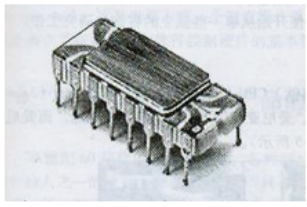
接口

- IO 控制（GPIO）
- 通信（UART， Ether NET ，USB）
- 外设 （A/D，D/A，LCD，KEYBOARD，Video）

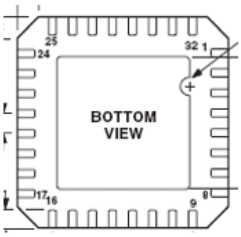
封装（尺寸, 工艺）



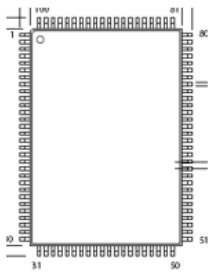
SOP



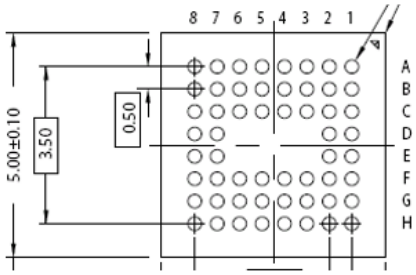
DIP



CSP



QFP



BGA

- 电源要求
  - 电压
    - 种类
    - 正常工作范围
    - 极限范围
    - 上电顺序要求
  - 功率
    - 正常工作电流
    - 省电模式电流
- 工作环境
  - 温度
    - 0-70 ° C/-40 – 80 ° C
  - 湿度

嵌入式系统需求

- 功率低

  设想一下，普通手机电池能够维持P4（75W）工作多长时间?

  移动领域 X86 败于 ARM
- 稳定性好

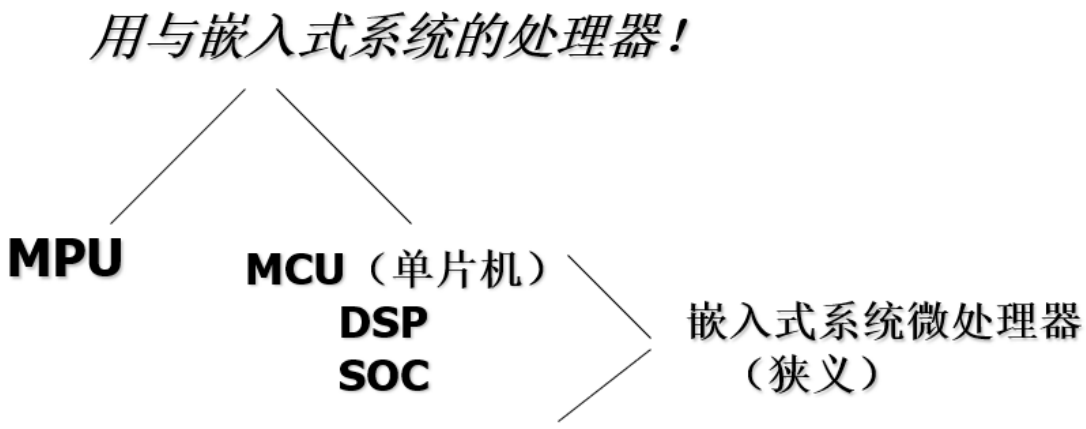
  卫星上的系统出错后如何复位?
- 体积小

  PC中的处理器有多大?
- 成本低

  最便宜的处理器价格?

嵌入式处理器

嵌入式处理器？-满足嵌入式系统需求



AI芯片属于哪一类?

嵌入式系统微处理器的发展

- 位宽
  - 4位 TMS1000, COPS 计算器、遥控器、玩具...
  - ↓
  - 8位 8048, Z8 马达控制、电话录音...
  - ↓
  - 16位 8096,68200 手机、mp3 ...
  - ↓
  - 32 位 ARM, MIPS32 Core 掌上电脑、PDA...
  - ↓
  - 64位 MIPS 64 Core ， ARM 多媒体...
- 复杂度
  - CPU -> MCU (单片机)
    (8086) (8096)
  - MCU - > SoC
    (c51) (Xscale)
  - Single Core - > Multi Core
    (ARM V5) (ARM V8)
- 发行方式
  - CPU 器件 传统处理器
  - IP Core ARM, MIPS, NIOS
  - 开源内核 RISC-V

What is next to ARM?

处理器发展

微处理器结构上的第一

First Computer ENIAC (University of Pennsylvania,46)

First general-purpose,single-chip microprocessor:Intel 4004,1971

First 8-bit architecture:Intel 8008,1972

First 16-bit architecture:Intel 8080,1974

First 32-bit architecture:Motorola 68000,1979

First RISC microprocessor: MIPS R2000, 1985

First microprocessor to provide integrated support for instruction & data cache: MIPS R2000, 1985

First pipelined microprocessor (sustains 1 instruction/clock): MIPS R2000, 1985

First 64-bit architecture: MIPS R4000, 1991

First multiple issue（superscalar) microprocessor: Intel i860, 1991

First CMP processor: IBM Power 4,2001

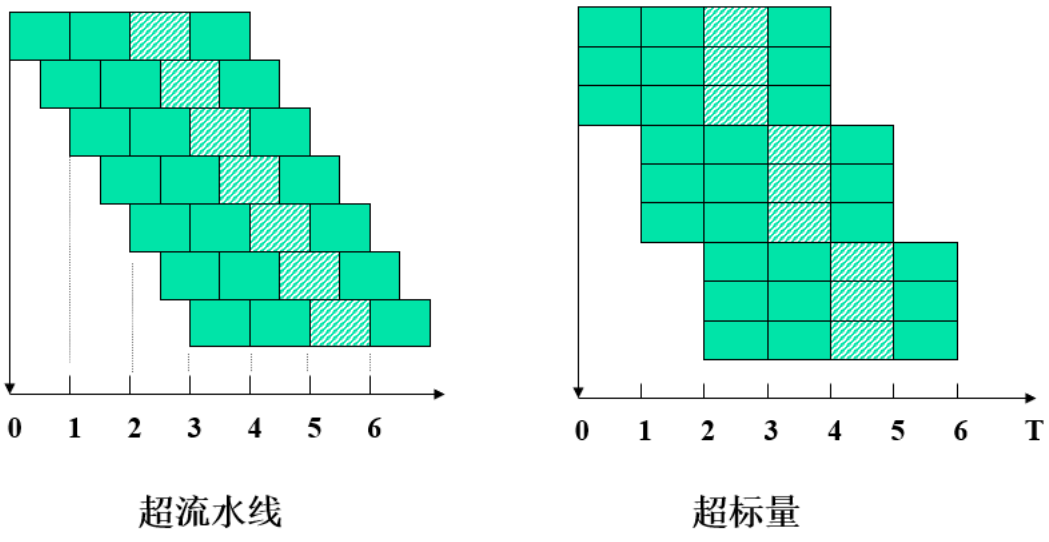
First SMT processor: Intel Pentium IV Xeon,2003

提高运行速度

- 提高主频
- 提高存储器速度
- 改进处理器结构-并行化/存储管理
    - 数据
    - 步骤->pipe line
    - 指令->Superscale
    - 内核->Multicore
    - 访存->Cache &Write buffer
- 降低能耗

并行化

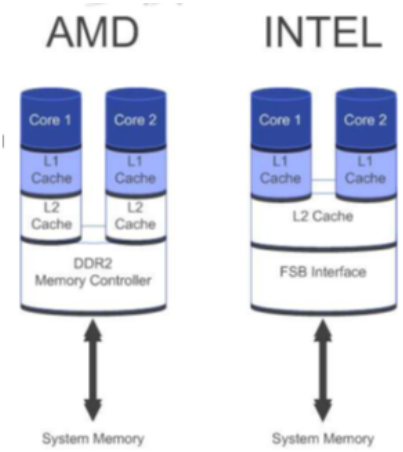
超流水线和超标量



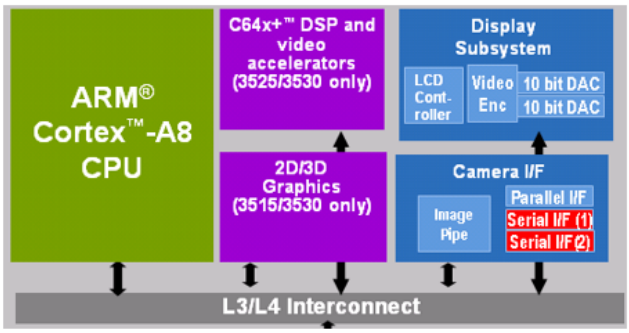
超流水线　　超标量

矢量计算?

多核



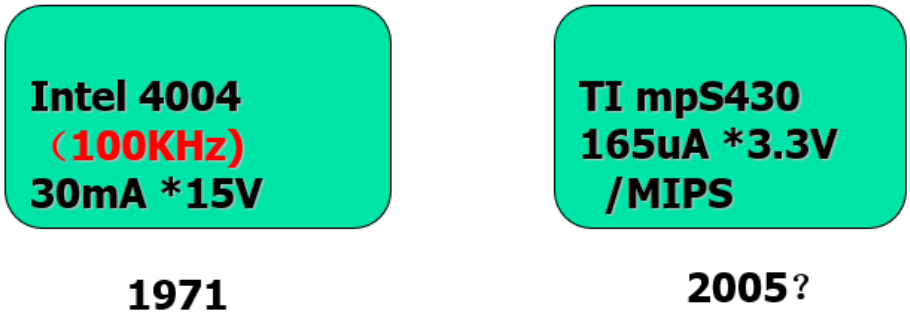
同构



异构

- **GPU, NPU, TPU**

比较两款处理器功耗



1971　　2005？

Intel处理器败走移动领域的主要因素?

小结

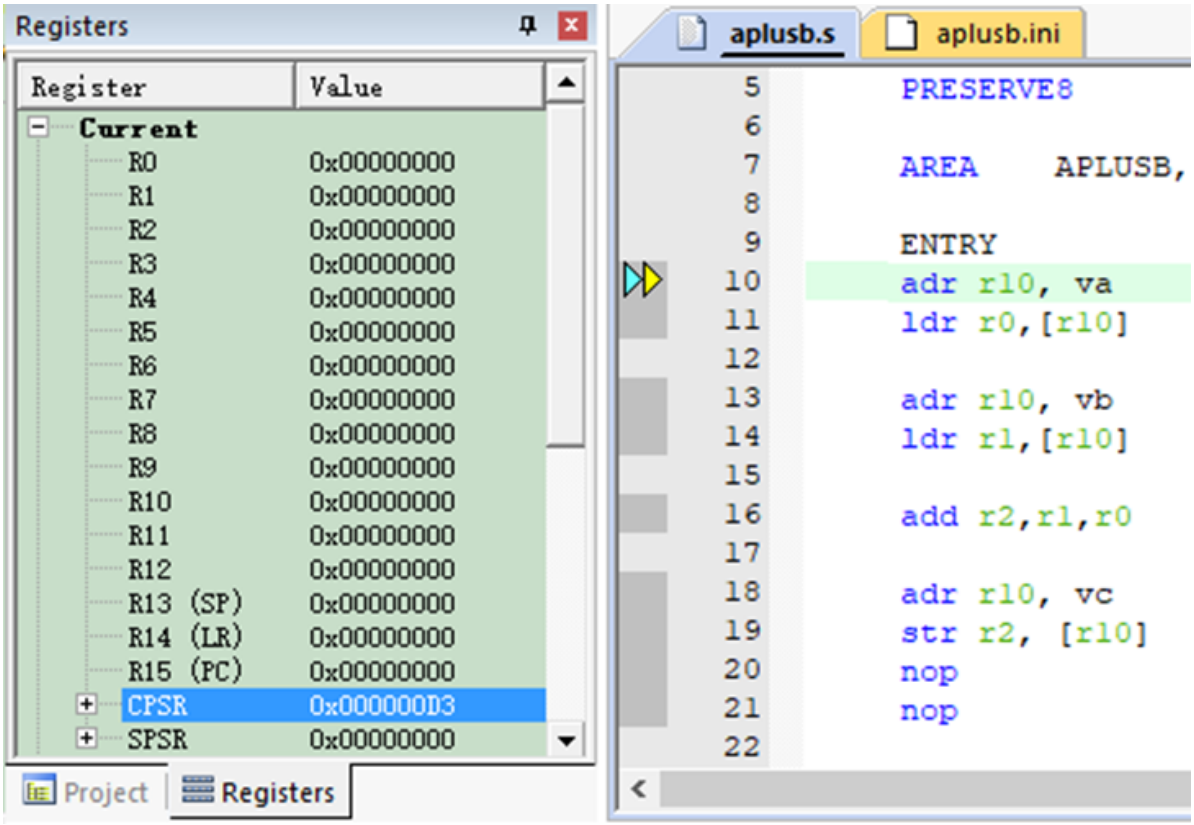
- **处理器程序开发过程。**
- **处理器结构提对程序执行速度影响。**
- **嵌入式处理器特点。**

ARM 处理器

例程：c=a+b

```
1          PRESERVE8

2          AREA    APLUSB, CODE, READONLY

3          ENTRY
4          adr r10, va              ;获取va地址
5          ldr r0,[r10]             ;读取（va）数值

6          adr r10, vb              ;获取vb地址
7          ldr r1,[r10]             ;读取（vb）数值

8          add r2,r1,r0              ;vc=va+vb

9          adr r10, vc              ;获取vc地址
10         str r2, [r10]            ;保存（vc）数值
11         nop
12         nop

13 va      DCD    0x123456
14 vb      DCD    0x89abcd
15 vc      DCD    0x000000

16         END
```

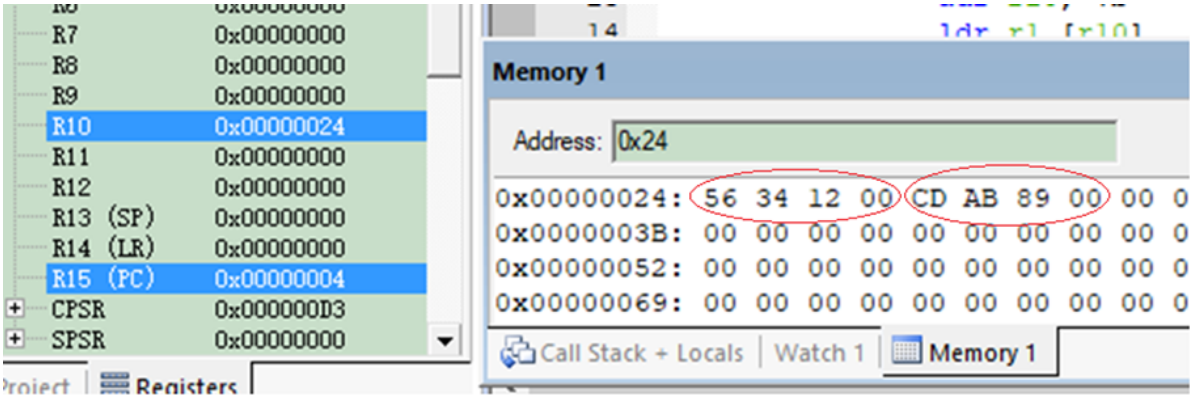
观察寄存器



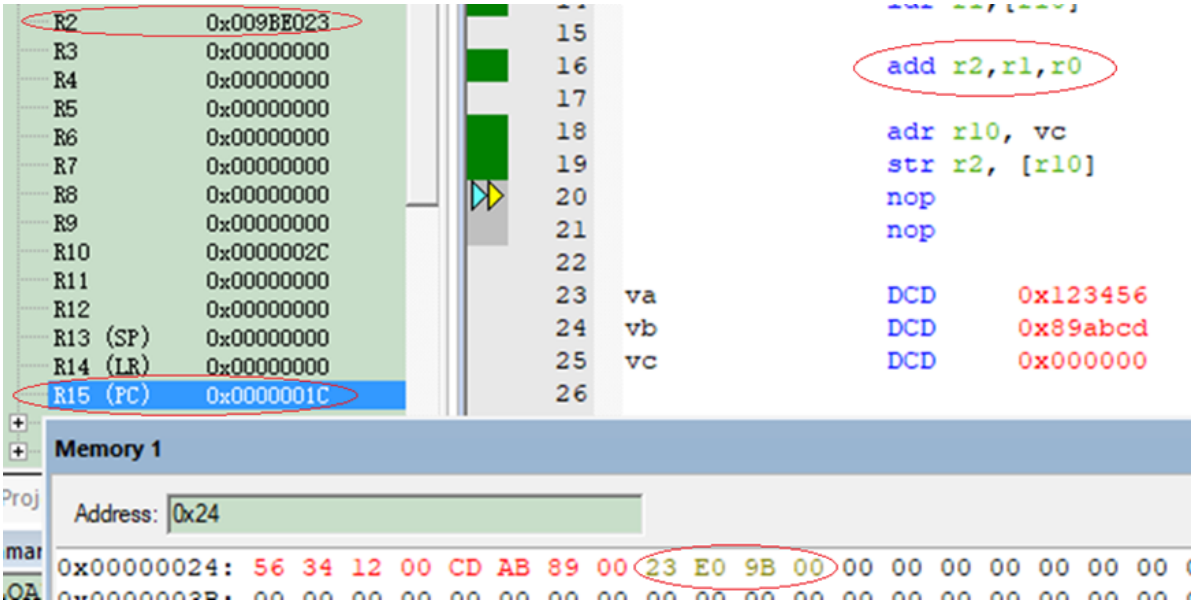
va, vb, vc 地址?

小端/大端?

查看内存数值?



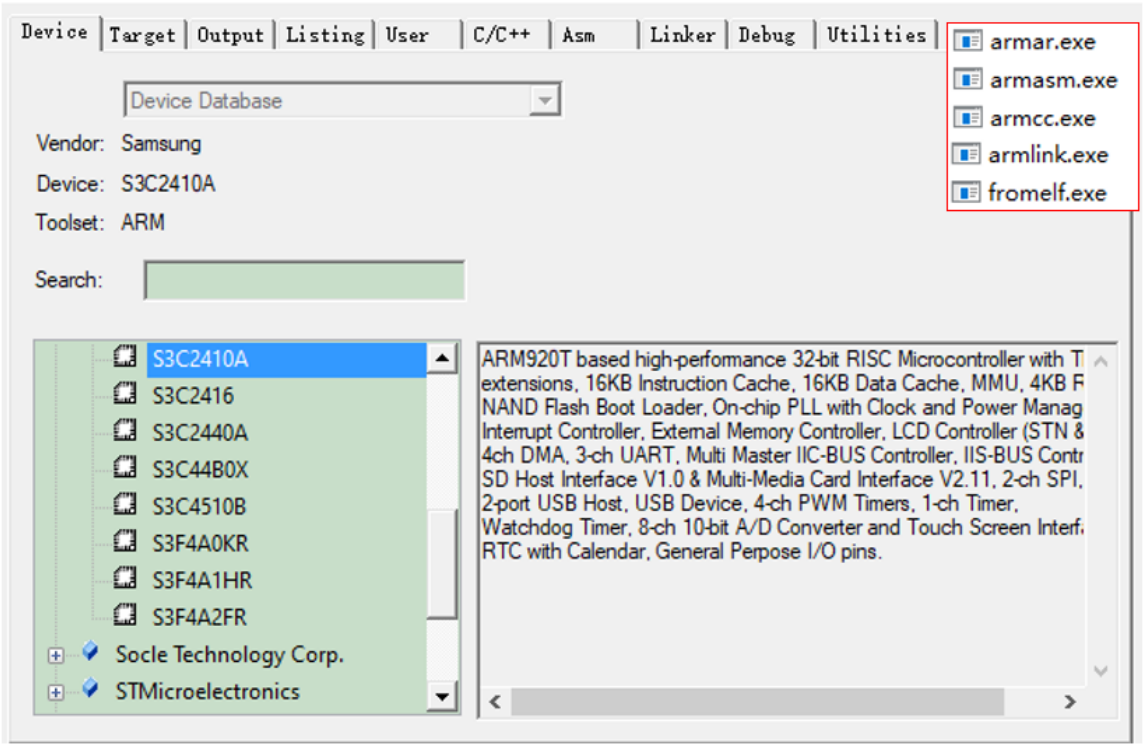
C=?

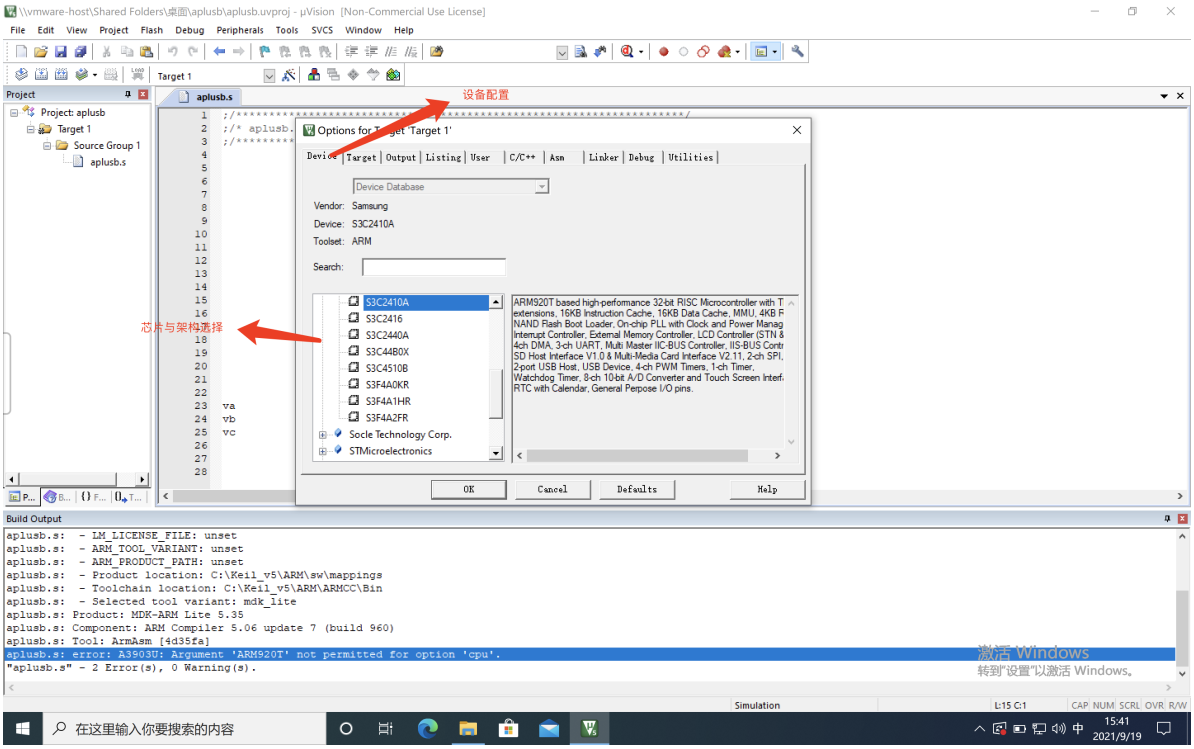


R15?

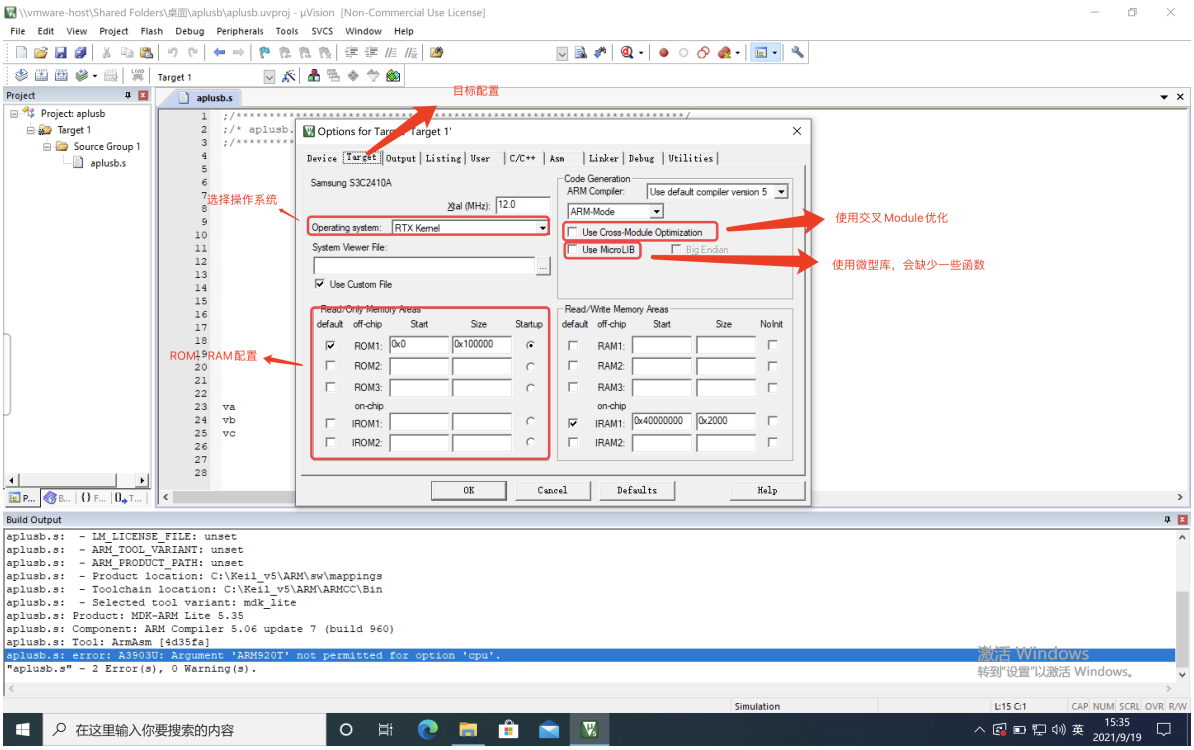
Option

Device

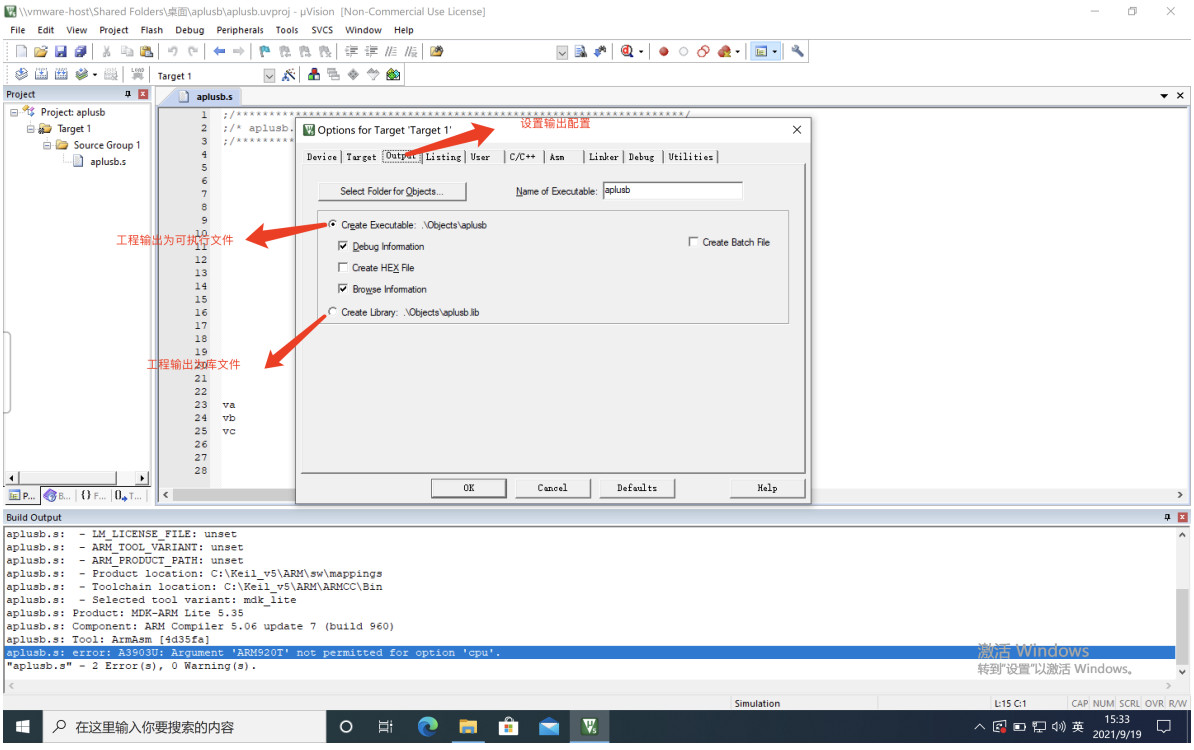

Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
armar.exe
armasm.exe
armcc.exe
armlink.exe
fromelf.exe
Device Database
Vendor: Samsung
Device: S3C2410A
Toolset: ARM
Search:
S3C2410A
S3C2416
S3C2440A
S3C44B0X
S3C4510B
S3F4A0KR
S3F4A1HR
S3F4A2FR
Socle Technology Corp.
STMicroelectronics
ARM920T based high-performance 32-bit RISC Microcontroller with T
extensions, 16KB Instruction Cache, 16KB Data Cache, MMU, 4KB F
NAND Flash Boot Loader, On-chip PLL with Clock and Power Manag
Interrupt Controller, External Memory Controller, LCD Controller (STN &
4ch DMA, 3-ch UART, Multi Master IIC-BUS Controller, IIS-BUS Contr
SD Host Interface V1.0 & Multi-Media Card Interface V2.11, 2-ch SPI,
2-port USB Host, USB Device, 4-ch PWM Timers, 1-ch Timer,
Watchdog Timer, 8-ch 10-bit A/D Converter and Touch Screen Interf
RTC with Calendar, General Perpose I/O pins.


设备配置
芯片与架构选择
Build Output
aplusb.s: error: A3903U: Argument 'ARM920T' not permitted for option 'cpu'.
"aplusb.s" - 2 Error(s), 0 Warning(s).

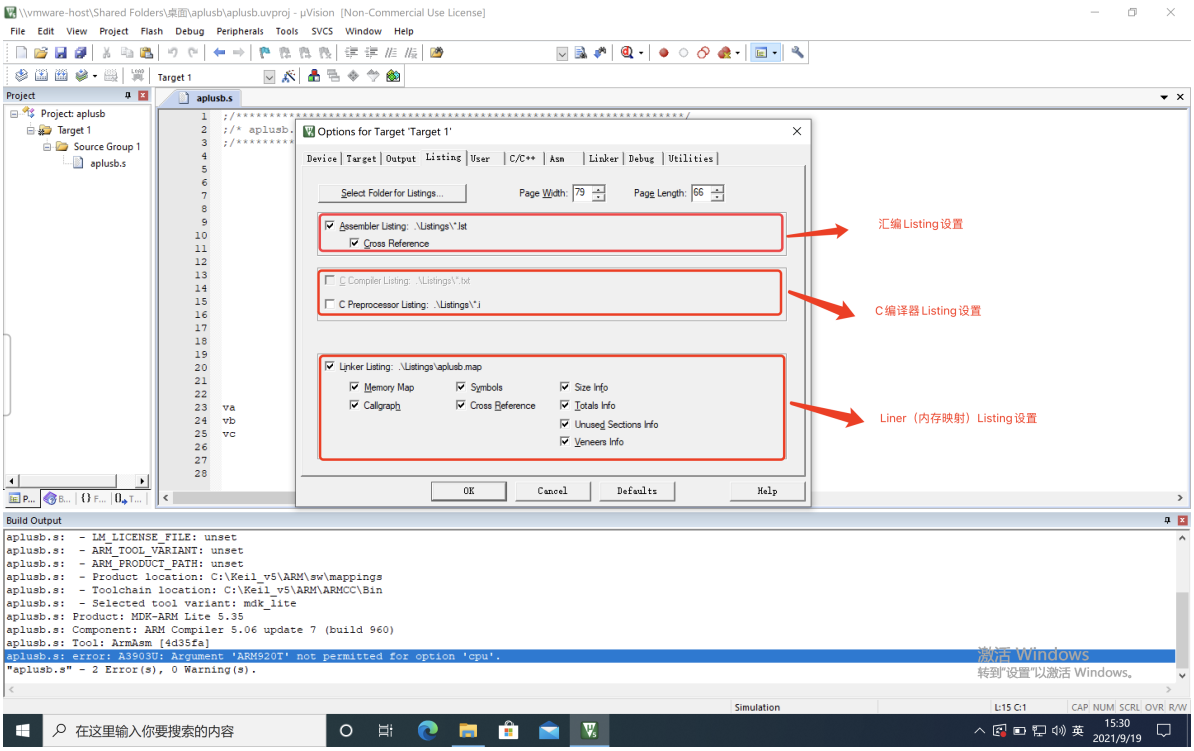
Target



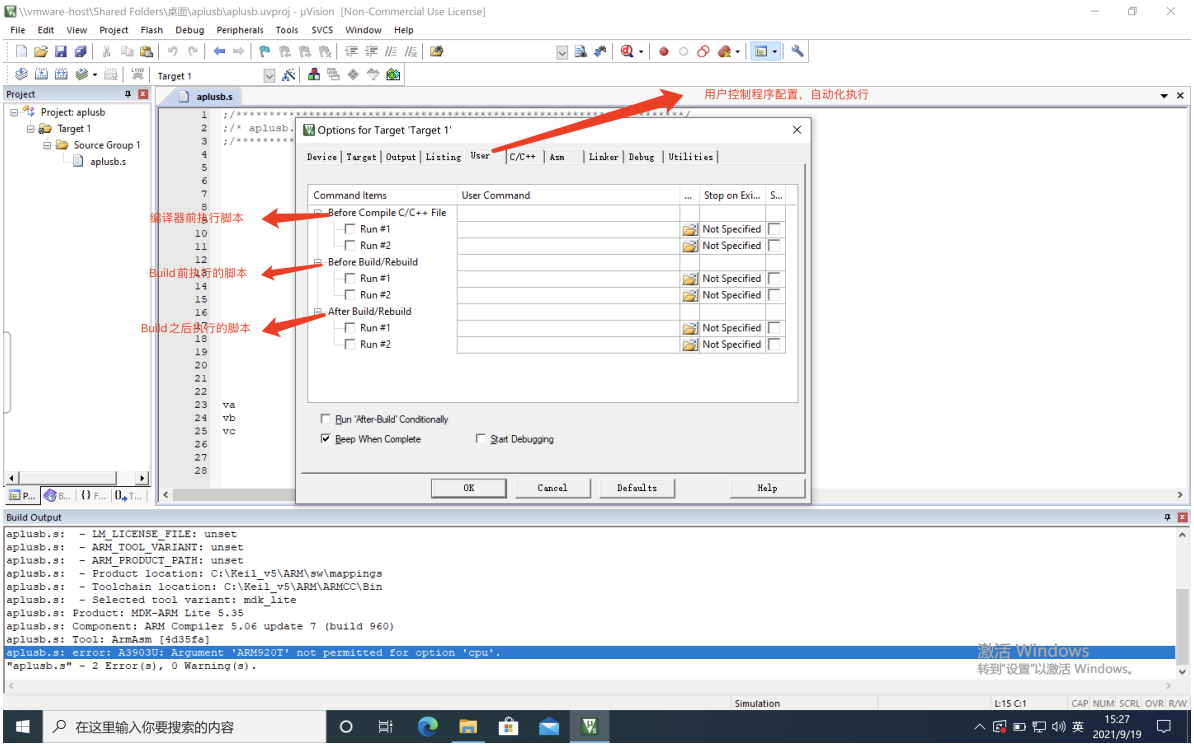
Output



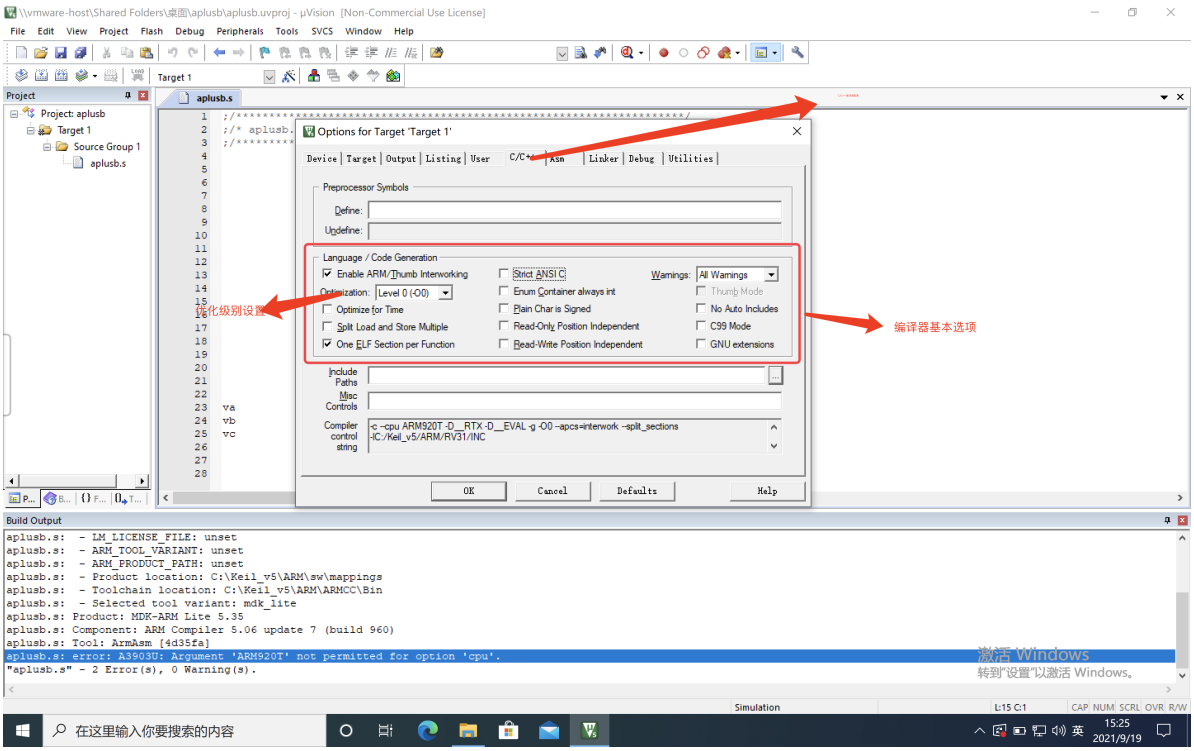
Listing


Options for Target 'Target 1'
Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities
Select Folder for Listings...
Page Width: 79
Page Length: 66
Assembler Listing: .\Listings*.lst
Cross Reference
C Compiler Listing: .\Listings*.txt
C Preprocessor Listing: .\Listings*.i
Linker Listing: .\Listings\aplusb.map
Memory Map
Symbols
Size Info
Callgraph
Cross Reference
Totals Info
Unused Sections Info
Veneers Info
OK
Cancel
Defaults
Help
汇编 Listing 设置
C编译器 Listing 设置
Liner（内存映射）Listing 设置
Build Output
aplusb.s: - LM_LICENSE_FILE: unset
aplusb.s: - ARM_TOOL_VARIANT: unset
aplusb.s: - ARM_PRODUCT_PATH: unset
aplusb.s: - Product location: C:\Keil_v5\ARM\sw\mappings
aplusb.s: - Toolchain location: C:\Keil_v5\ARM\ARMCC\Bin
aplusb.s: - Selected tool variant: mdk_lite
aplusb.s: Product: MDK-ARM Lite 5.35
aplusb.s: Component: ARM Compiler 5.06 update 7 (build 960)
aplusb.s: Tool: ArmAsm [4d35fa]
aplusb.s: error: A3903U: Argument 'ARM920T' not permitted for option 'cpu'.
"aplusb.s" - 2 Error(s), 0 Warning(s).
激活 Windows
转到"设置"以激活 Windows。
在这里输入你要搜索的内容
15:30
2021/9/19

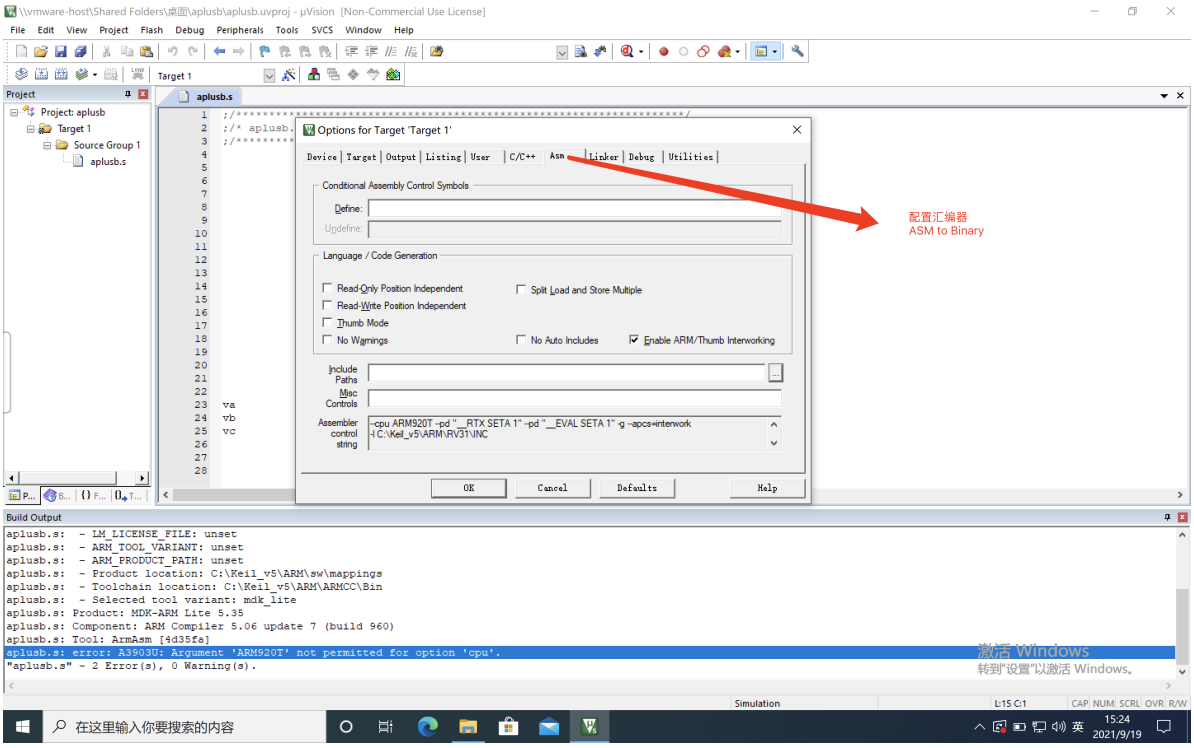
User


用户控制程序配置，自动化执行
Options for Target 'Target 1'
Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities
Command Items
User Command
Stop on Exi...
Before Compile C/C++ File
Run #1
Run #2
Before Build/Rebuild
After Build/Rebuild
Not Specified
Run 'After-Build' Conditionally
Beep When Complete
Start Debugging
OK
Cancel
Defaults
Help
编译器前执行脚本
Build前执行的脚本
Build之后执行的脚本
Build Output
aplusb.s: - LM_LICENSE_FILE: unset
aplusb.s: - ARM_TOOL_VARIANT: unset
aplusb.s: - ARM_PRODUCT_PATH: unset
aplusb.s: - Product location: C:\Keil_v5\ARM\sw\mappings
aplusb.s: - Toolchain location: C:\Keil_v5\ARM\ARMCC\Bin
aplusb.s: - Selected tool variant: mdk_lite
aplusb.s: Product: MDK-ARM Lite 5.35
aplusb.s: Component: ARM Compiler 5.06 update 7 (build 960)
aplusb.s: Tool: ArmAsm [4d35fa]
aplusb.s: error: A3903U: Argument 'ARM920T' not permitted for option 'cpu'.
"aplusb.s" - 2 Error(s), 0 Warning(s).
激活 Windows
转到"设置"以激活 Windows。
在这里输入你要搜索的内容
15:27
2021/9/19

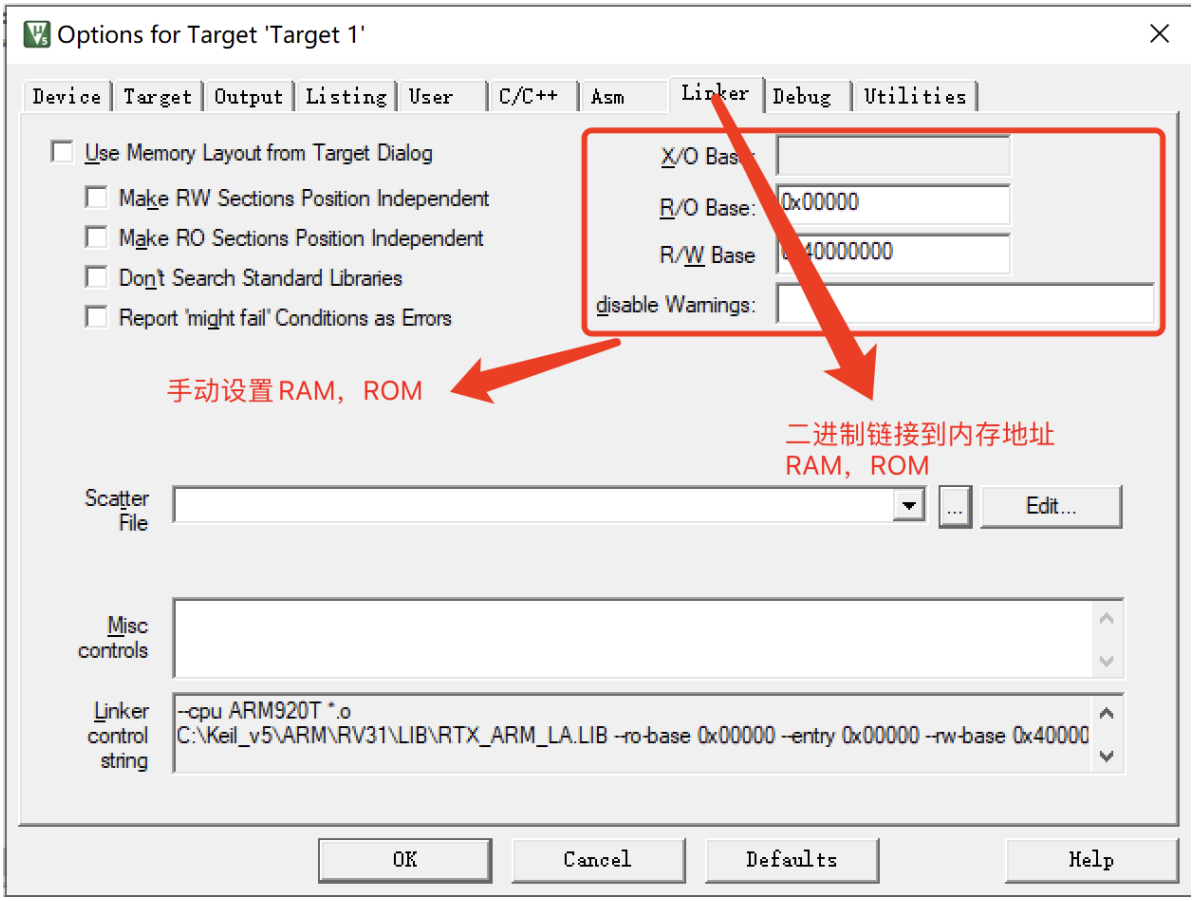
C/C++


Options for Target 'Target 1'
优化级别设置
编译器基本选项
Build Output
aplusb.s: error: A3903U: Argument 'ARM920T' not permitted for option 'cpu'.
"aplusb.s" - 2 Error(s), 0 Warning(s).

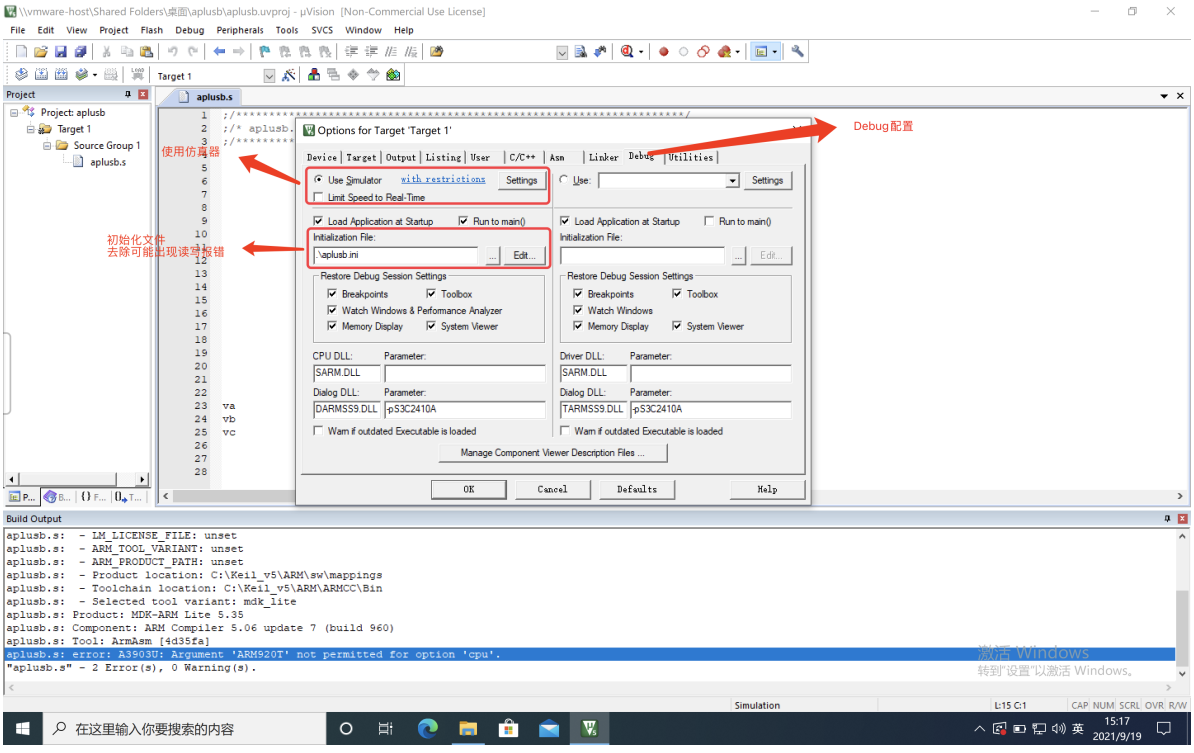
Asm


Options for Target 'Target 1'
配置汇编器
ASM to Binary
Build Output
aplusb.s: error: A3903U: Argument 'ARM920T' not permitted for option 'cpu'.
"aplusb.s" - 2 Error(s), 0 Warning(s).

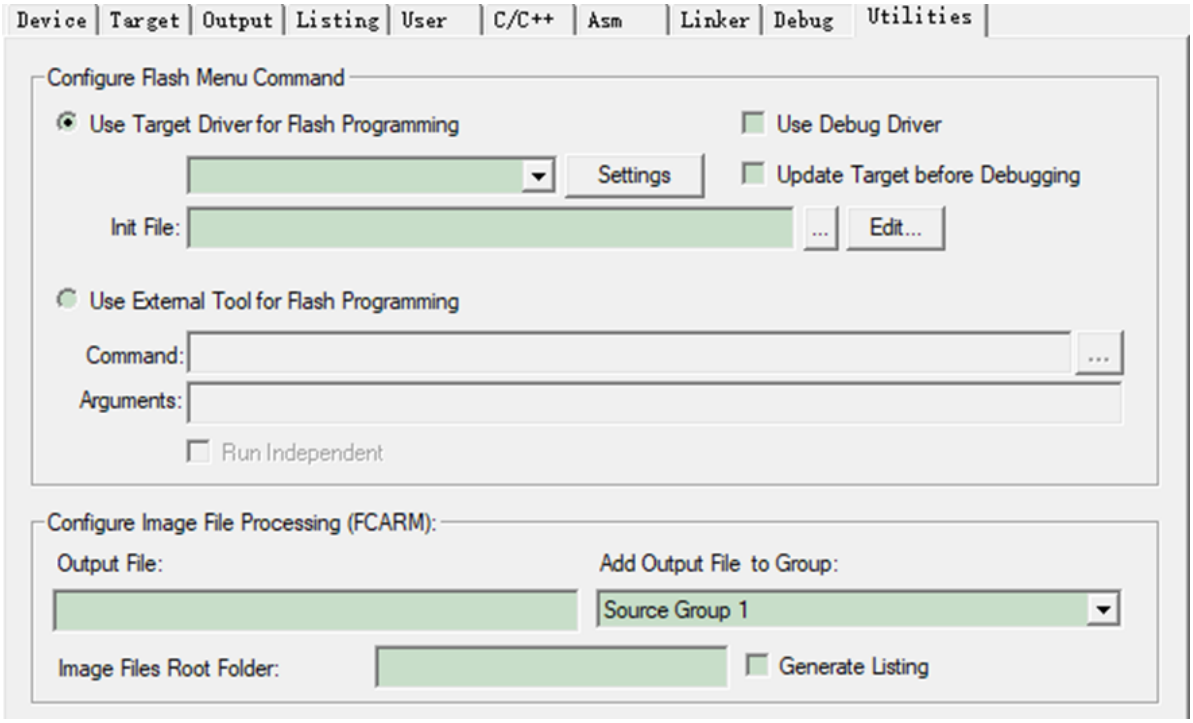
Linker



Debug



Utilities



aplusb.ini

去掉该文件重新调试，什么现象?

```
//  Object           : Generic Macro File for KEIL
MAP 0x00000000, 0x010000 exec write read
MAP 0x30000000, 0x30010000 write read
MAP 0x53000000, 0x53001000 write read
MAP 0x48000000, 0x48001000 write read
MAP 0x4C000000, 0x4C001000 write read
MAP 0x50000000, 0x50001000 write read
//
_WDWORD(0x4c000000,0x00ffffff)
_WDWORD(0x4c000004,0x00000000)
_WDWORD(0x4c000008,0x0005C080)
_WDWORD(0x4c00000C,0x00028080)
_WDWORD(0x4c000010,0x00000004)
_WDWORD(0x4c000014,0x0007FFF0)
//
PC = 0;
```

aplusb.lst (armasm)

```
 1 00000000
;/*****************************************************/
 2 00000000       ;/* aplusb.S: c=a+b  on Samsung S3C410A        */
 3 00000000
;/*****************************************************/
 4 00000000
 5 00000000            PRESERVE8
 6 00000000
 7 00000000            AREA         APLUSB, CODE, READONLY
 8 00000000
 9 00000000            ENTRY
10 00000000 E28FA01C       adr          r10, va   ; 获取va的地址
11 00000004 E59A0000       ldr          r0,[r10]  ; 读取va数据
          ......
21 00000020 E1A00000       nop
23 00000024 00123456
              va     DCD          0x123456
          ......
27 00000030             END
```

aplusb.map

Memory Map

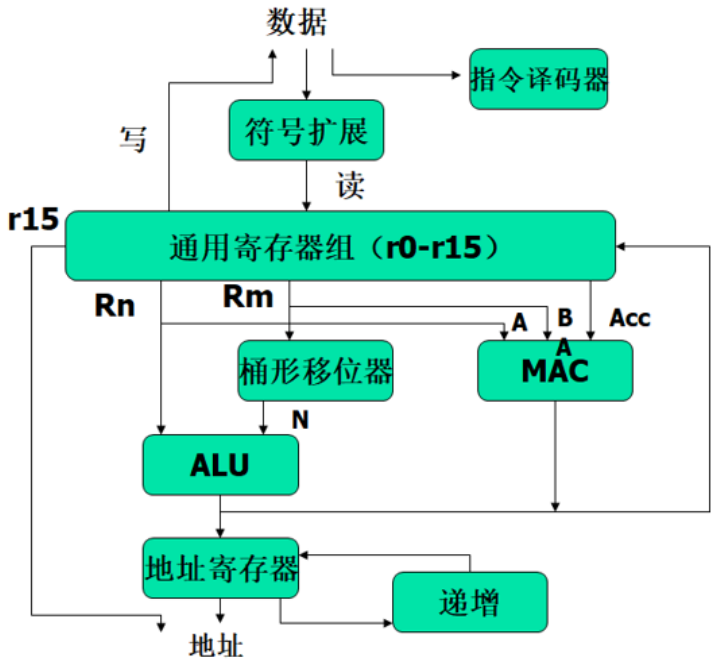
```
=============================================================
Memory Map of the image
  Image Entry point : 0x00000000
  Load Region LR_1 (Base: 0x00000000, Size: 0x00000030, Max: 0xffffffff, ABSOLUTE)
    Execution Region ER_RO (Base: 0x00000000, Size: 0x00000030, Max: 0xffffffff, ABSOLUTE)

    Base Addr        Size        Type    Attr      Idx    E Section Name        Object

    0x00000000  0x00000030  Code    RO          1     * APLUSB              aplusb.o


    Execution Region ER_RW (Base: 0x40000000, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)

    **** No section assigned to this execution region ****


    Execution Region ER_ZI (Base: 0x40000000, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)

    **** No section assigned to this execution region ****

=============================================================
```

Size

```
==============================================================
Image component sizes
  Code (inc. data)   RO Data    RW Data    ZI Data      Debug      Object Name
      48         12         0          0          0        236       aplusb.o
  ----------------------------------------------------------------------
      48         12         0          0          0        236       Object Totals
       0          0         0          0          0          0       (incl. Generated)
       0          0         0          0          0          0       (incl. Padding)
  ----------------------------------------------------------------------
       0          0         0          0          0          0       Library Totals
       0          0         0          0          0          0       (incl. Padding)
  ----------------------------------------------------------------------
  Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      48         12         0          0          0        236       Grand Totals
      48         12         0          0          0        236       ELF Image Totals
      48         12         0          0          0          0       ROM Totals
==============================================================
  Total RO  Size (Code + RO Data)                 48 (   0.05kB)
  Total RW  Size (RW Data + ZI Data)               0 (   0.00kB)
  Total ROM Size (Code + RO Data + RW Data)       48 (   0.05kB)
==============================================================
```

ARM9内核

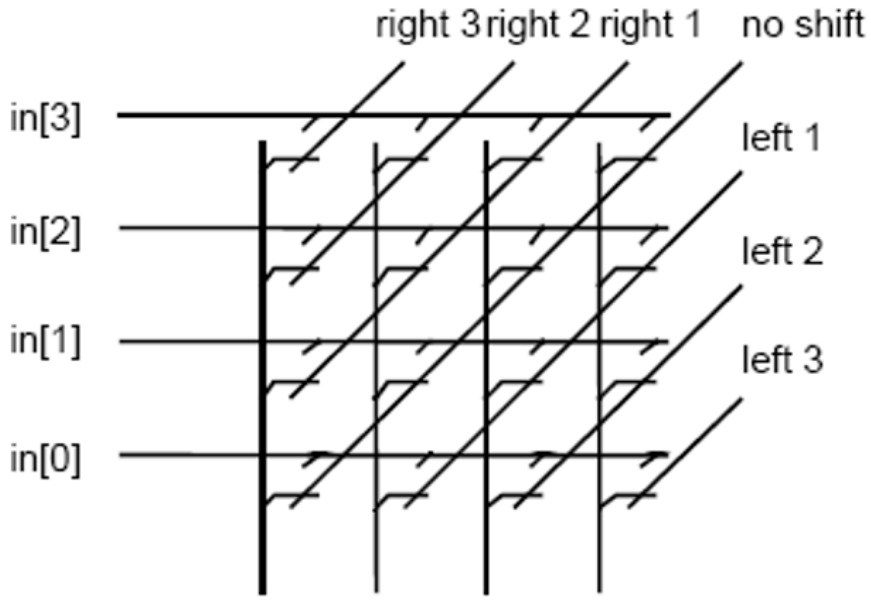
内核结构



C=a+b 执行过程？

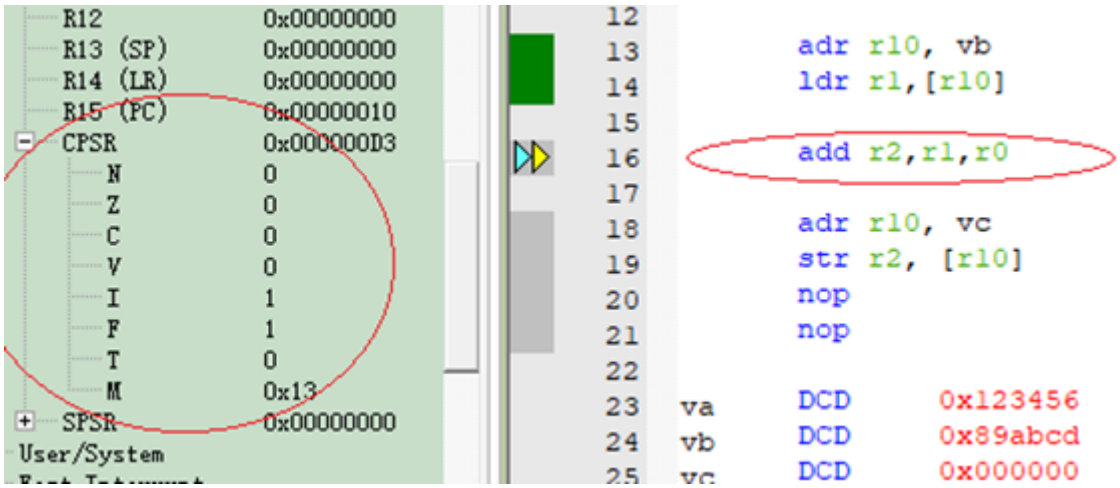
Load-Store结构？

桶形移位器

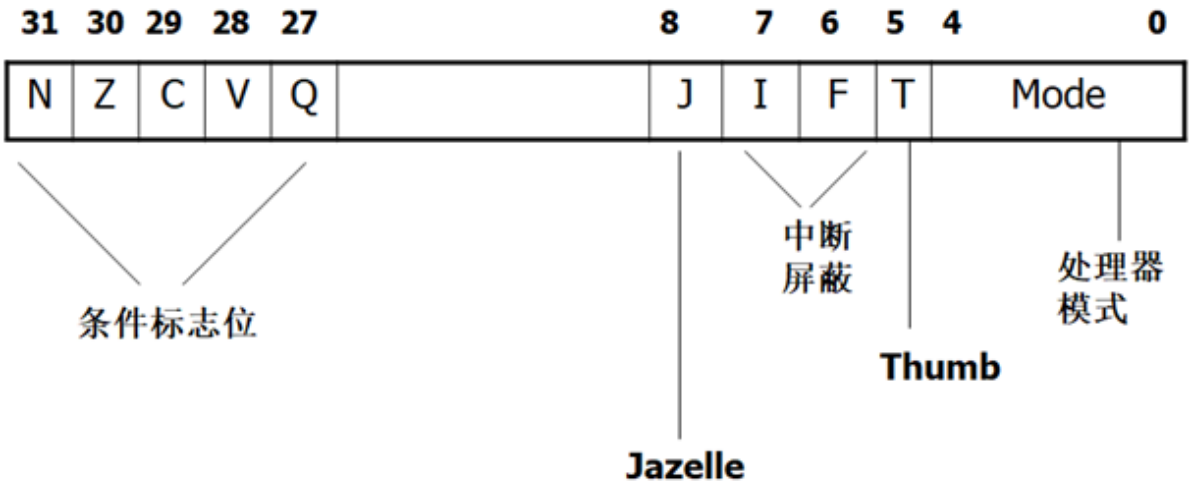


多个位同时移动？

调试程序，观察加运算前后处理器状态



CPSR



- 条件标志

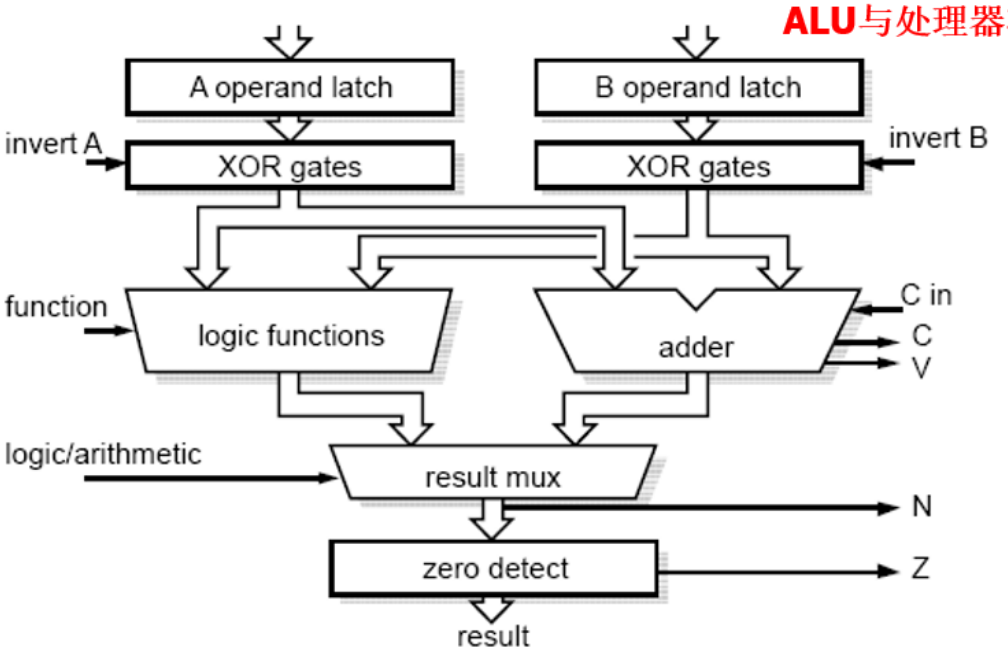
标志位	名称	置位条件
Q	饱和	增强的**DSP**指令发生溢出或饱和
V	溢出	结果导致一个有符号数溢出
C	进位	结果导致一个无符号数进位
Z	零	结果是**0**
N	负数	结果的第**31**位为**1**

- 状态标志

标志位	名称	状态说明
T	**Thumb**	处理器执行**16**位**Thumb**指令
J	**Jazelle**	处理器执行**8**位**Java**代码

ALU

ALU与处理器状态?



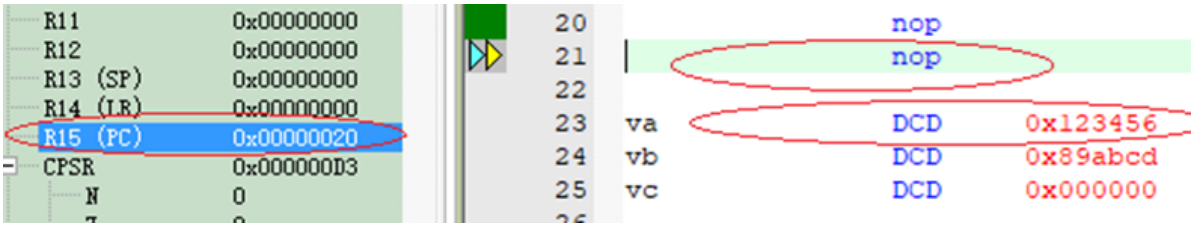
特点

RISC:	ARM:
•指令集 •每条指令执行一个周期 •指令长度固定 •流水线 •寄存器 •大量的通用寄存器 •Load-Store结构	•指令集 •一些特定指令的周期数可变 •内置桶形移位寄存器（32位） 产生复杂的指令 •支持Thumb 指令 •条件执行 •增强指令（DSP） •流水线 •寄存器（32位） •Load-Store结构

异常与模式

异常与模式

调试程序，在nop处单步执行，观察pc的变化



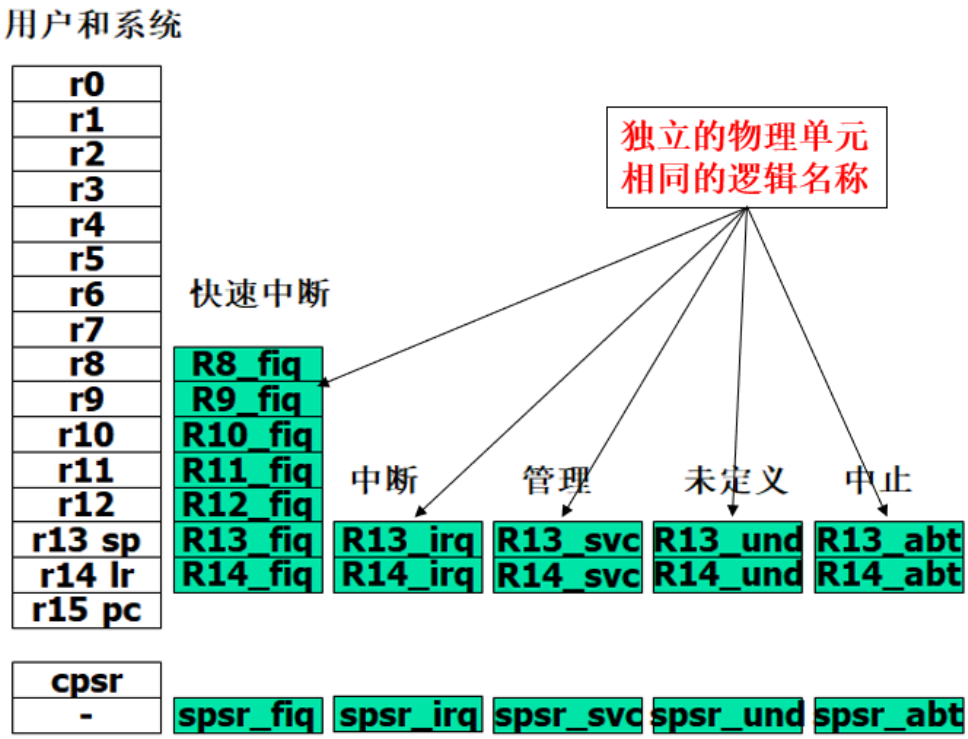
异常向量

异常类型	处理器模式	异常向量地址	优先级
复位	管理	0×00000000	1（最高）
未定义指令	未定义	0×00000004	6
软件中断（SWI）	管理	0×00000008	6
预取中止（取指令存储器中止）	中止	0×0000000C	5
数据中止（数据访问存储器中止）	中止	0×00000010	2
中断 IRQ	中断 IRQ	0×00000018	4
快中断 FIQ	快中断 FIQ	0×0000001C	3

异常模式

模式	特点	模式位
中止（abt）	存储器访问失败	10111
快速中断（fiq)	快速中断源	10001
中断（irq)	普通中断源	10010
管理（svc)	复位后进入，操作系统内核	10011
系统（sys）	完全cpsr访问模式	11111
未定义（und）	不支持的指令	11011
用户（usr）	运行应用程序	10000

寄存器



处理器异常处理

- 处理过程

```
R14_<exception_mode>=return link
SPSR_<exception_mode>=CPSR
CPSR[4:0]=exception mode number
CPSR[5]=0           /* 在ARM状态执行*/
If<exception_mode>=Reset or FIQ then
CPSR[6]=1           /*禁止快速中断*/
CPSR[7]=1           /*禁止正常中断*/
PC=exception vector address
```

处理器处理过程不需要用程序实现吗!

如何恢复到异常发生前的状态?

- 复位(reset)

```
R14_svc=UNPREDICTABLE Value
SPSR_svc= UNPREDICTABLE Value
CPSR[4:0]=0b10011  /*进入管理模式*/
CPSR[5]=0
CPSR[6]=1
CPSR[7]=1
PC=0x000000
```

- 中断(irq) (fiq?)

```
R14_irq =address of next instruction to be executed+4
SPSR_irq = CPSR
CPSR[4:0]=0b10010  /*进入IRQ模式*/
CPSR[5]=0
CPSR[7]=1
PC=0x000018
```

```
返回：SUBS PC, R14,#4
```

- 数据终止(abort)

```
R14_abt=address of aborted instruction+8
SPSR_abt= CPSR
CPSR[4:0]=0b10111  /*进入abt模式*/
CPSR[5]=0
CPSR[7]=1
PC=0x000010
```

```
返回：SUBS PC, R14, #8
```

- 指令终止(undefined)

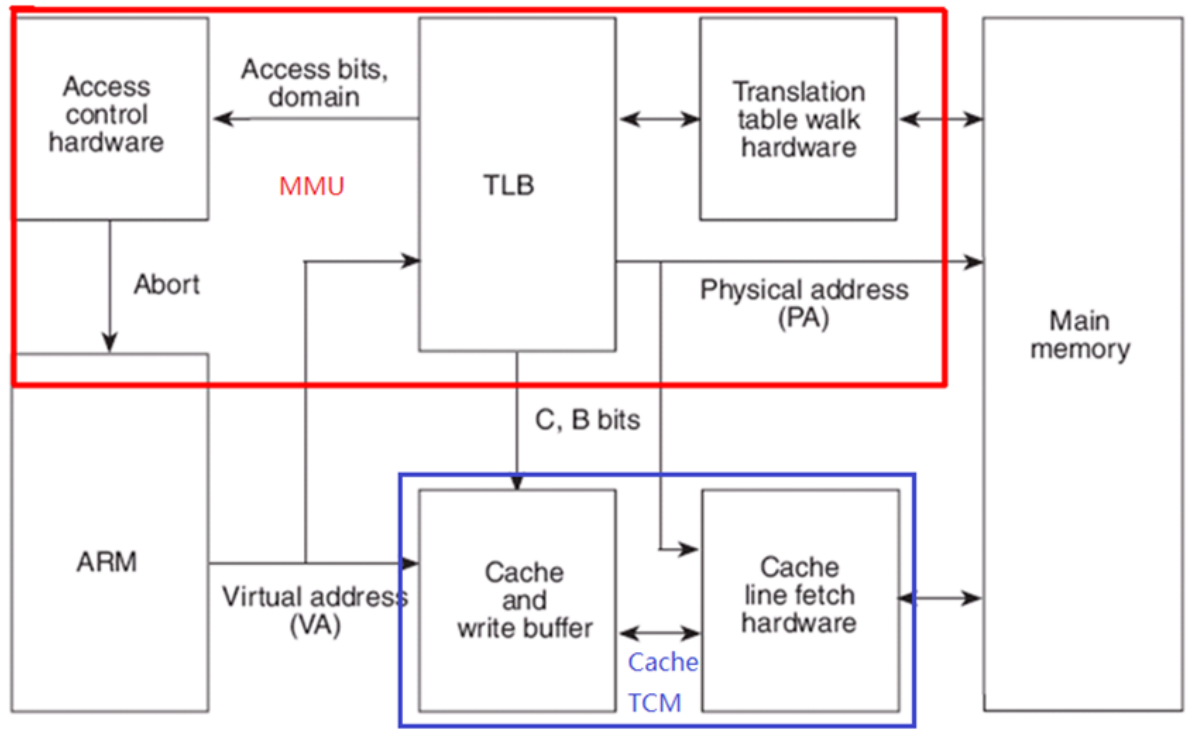
```
R14_abt=address of aborted instruction+4
SPSR_abt= CPSR
CPSR[4:0]=0b10111  /*进入abt模式*/
CPSR[5]=0
CPSR[7]=1
PC=0x00000c
```

```
返回：SUBS PC, R14,#4
```

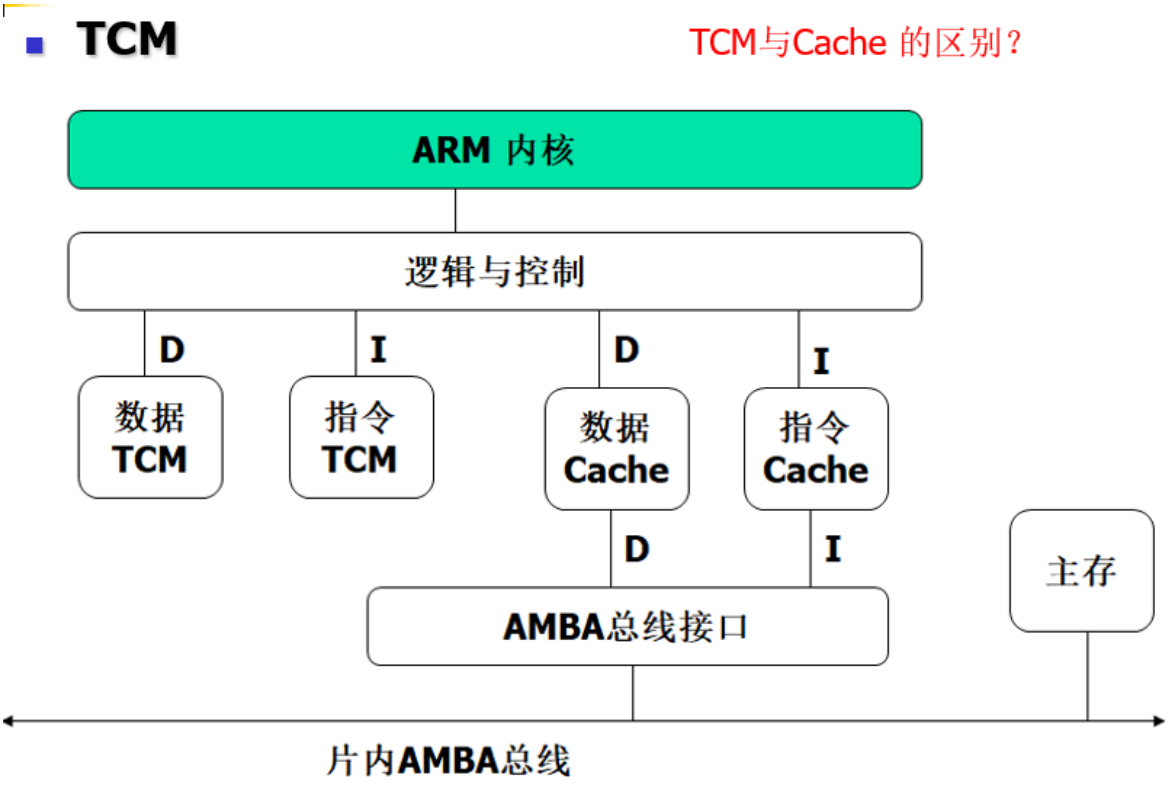
存储系统

结构

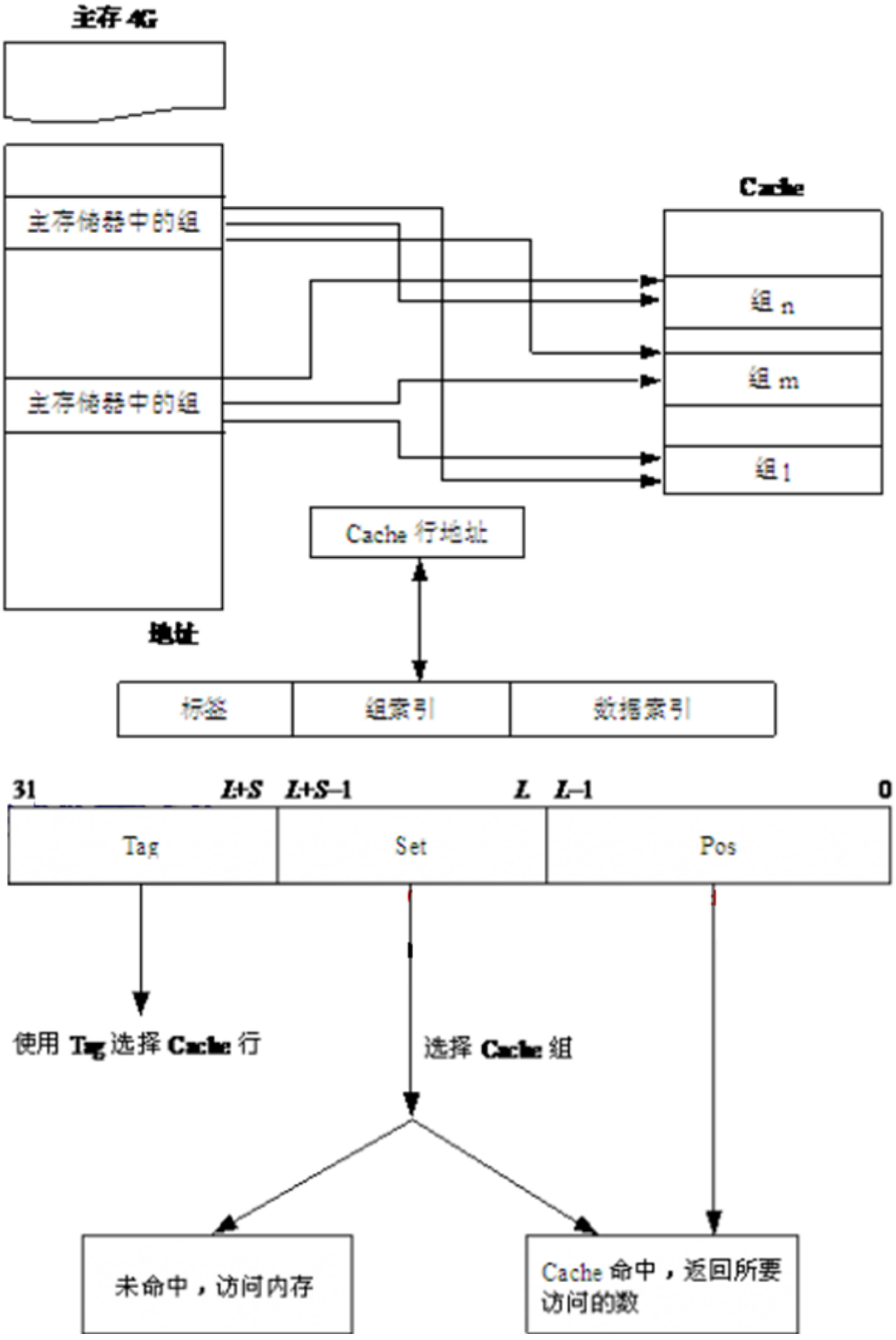
按字节编址!



TCM与Cache 的区别?



Cache

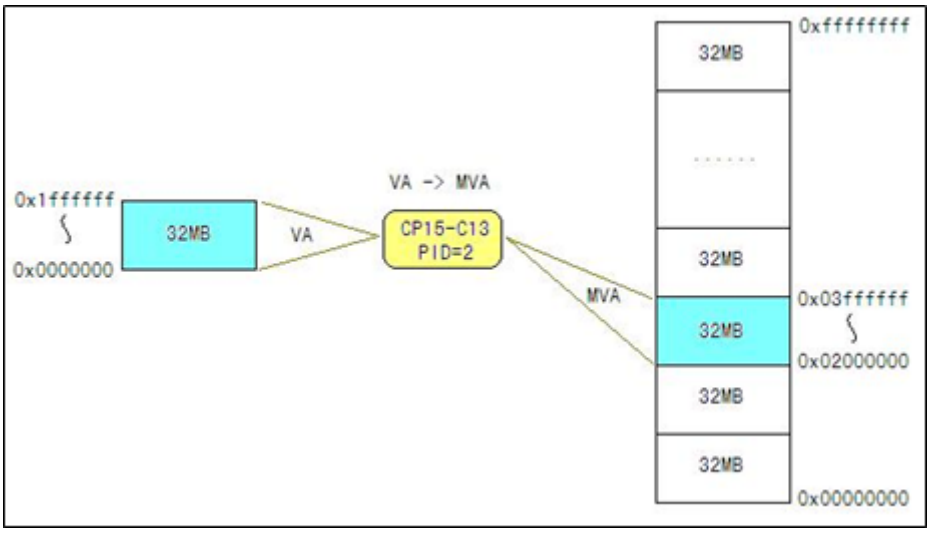


存储管理

ARM9的三类地址:

- 虚拟地址（VA），是程序中的逻辑地址，0x00000000~0xFFFFFFFF。
- 改进的虚拟地址（MVA），由于多个进程执行，逻辑地址会重合。所以，跟据进程号将逻辑地址分布到整个内存中。MVA = (PID << 25) | VA。PID占7位，所以最多只能有 128 个进程，每个进程只能分到 32MB 的逻辑地址空间。

- 物理地址（PA），MVA通过MMU转换后的地址。



协处理器CP15

- 内存管理单元(Memory Manage Unit，MMU)
  - 控制虚拟地址（VA）映射到物理地址（PA）
  - 控制内存的访问权限
  - 控制可缓存性和缓冲性
- 内存保护单元(Protection Unit，PU)
  - 无MMU的简单内存保护
- 使能Cache and Write Buffer
- 快速上下文(进程之间)切换扩展(Fast Context Switch Extension，FCSE)

寄存器编号	基本作用	在 MMU 中的作用	在 PU 中的作用
0	ID 编码（只读）	ID 编码和 cache 类型	
1	控制位（可读写）	各种控制位	
2	存储保护和控制	地址转换表基地址	Cachability 的控制位
3	存储保护和控制	域访问控制位	Bufferablity 控制位
4	存储保护和控制	保留	保 留
5	存储保护和控制	内存失效状态	访问权限控制位
6	存储保护和控制	内存失效地址	保护区域控制
7	高速缓存和写缓存	高速缓存和写缓存控制	
8	存储保护和控制	TLB 控制	保 留
9	高速缓存和写缓存	高速缓存锁定	
10	存储保护和控制	TLB 锁定	保 留
11	保留		
12	保留		
13	进程标识符	进程标识符	
14	保留		
15	因不同设计而异	因不同设计而异	因不同设计而异

C2 （CP15）

- 别名：Translation table base (TTB) register
- 用来保存页表的基地址，即一级映射描述符表的基地址。

31	13	0
一级映射描述符表的基地址（物理地址）		

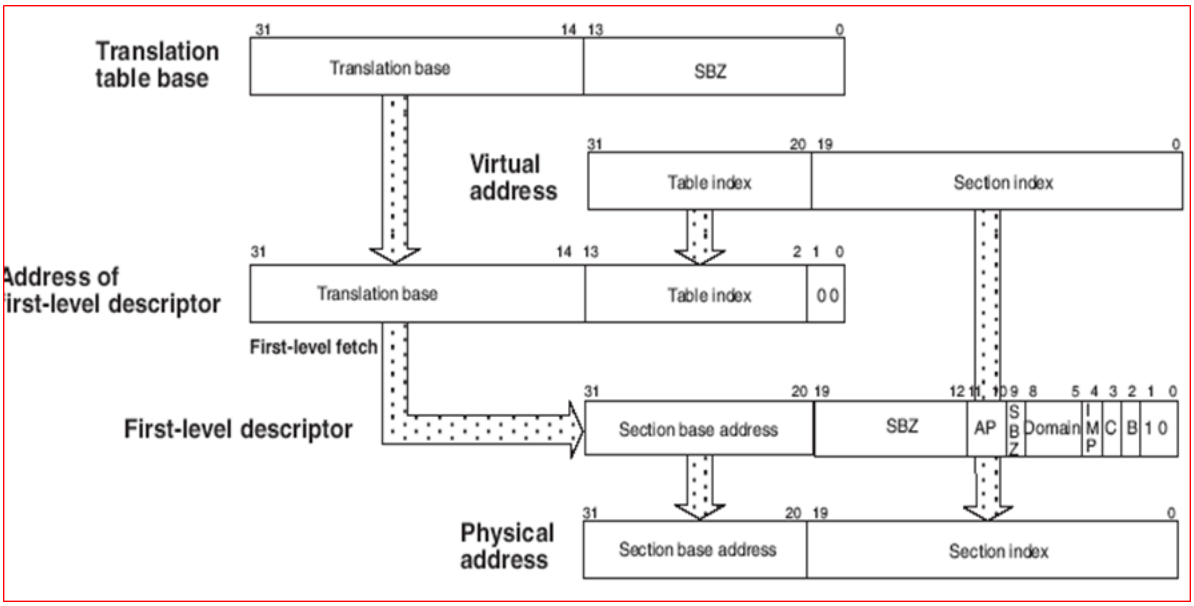
C12（CP15）

- CP15寄存器C12用来设置异常向量基地址

31	4	0
异常向量基地址	Reserve	

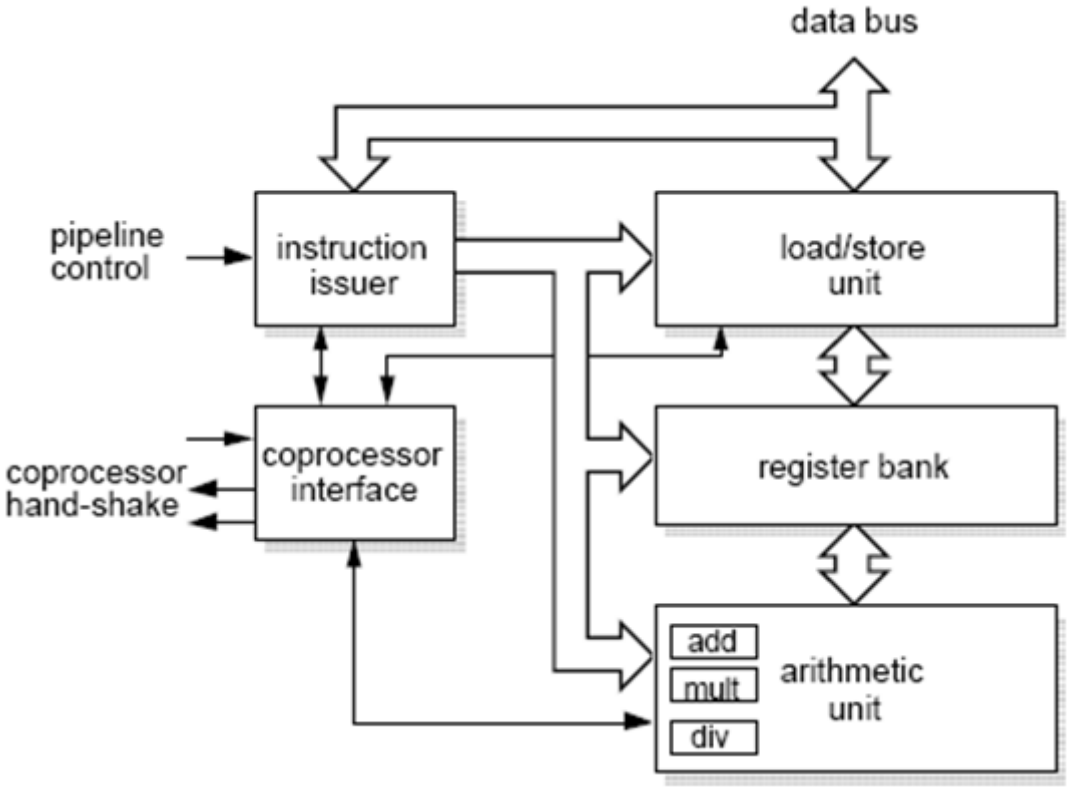
ARM11和cortex-a 可以任意修改异常向量基地址。ARM9只可以在0地址或0xffff0000中

段（section, 1MB）地址转换



内核扩展协处理器

- 协处理器接口，允许扩展16个协处理器
- 协处理器扩展（FFA10）



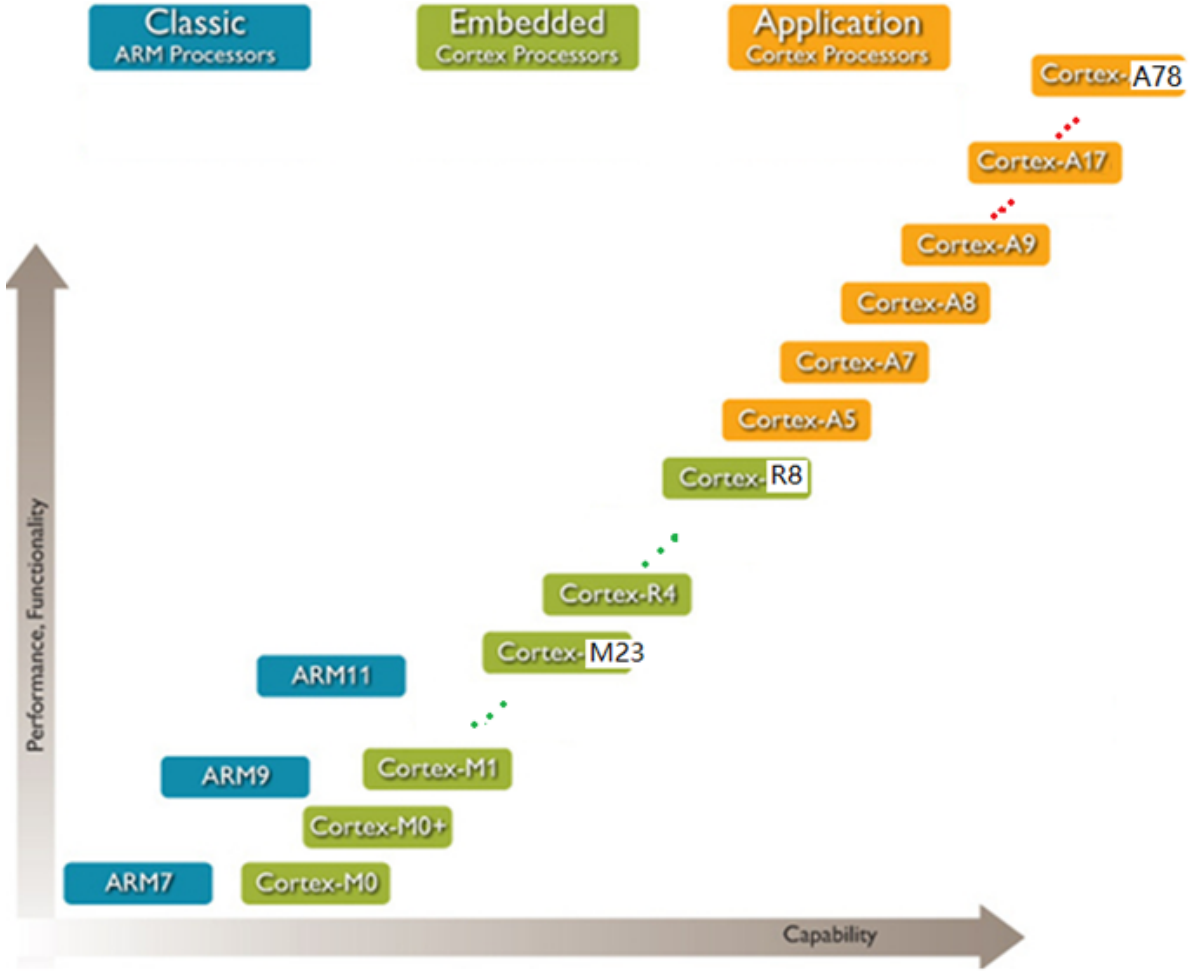
处理器如何访问协处理器?

ARM发展历程

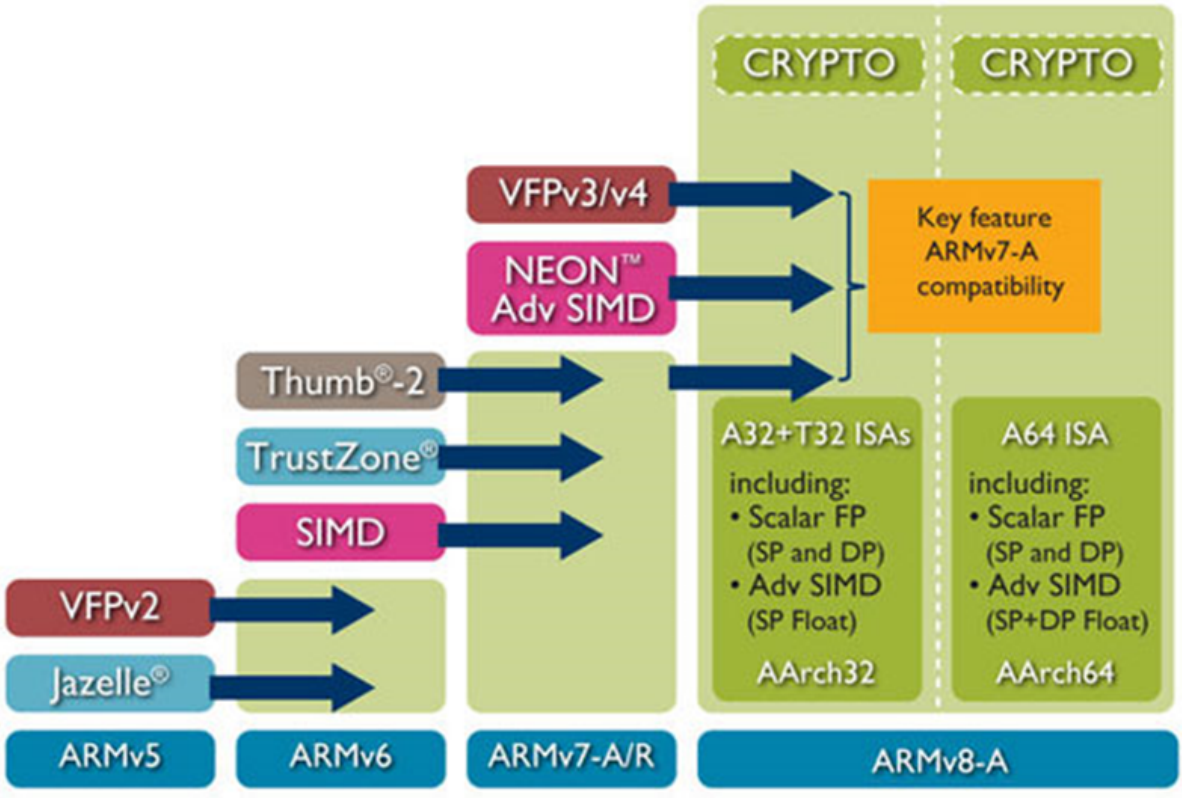
ARM- Advanced RISC Machine

- ARM的版本
  - V1, V2, V3,
  - V4 (ARM 7，8，9),
  - V5 (ARM 10),
  - V6 (ARM 11)
  - V7(ARM-Cortex A8,9,15,17)
  - V8(Cortex-A/R/M/, 64/32bit)
  - V9
- 扩展字母含义
  - T:内含16位压缩指令集 Thumb
  - D:支持片内Debug调试
  - M:采用增强型乘法器（Multiplier)
  - I: 内含嵌入式ICE宏单元
  - E: 具有DSP功能
  - S:可综合的软核Software
  - J: Jazeller,允许直接执行Java字节码

产品线


Classic
ARM Processors
Embedded
Cortex Processors
Application
Cortex Processors
Cortex-A78
Cortex-A17
Cortex-A9
Cortex-A8
Cortex-A7
Cortex-A5
Cortex-R8
Cortex-R4
Cortex-M23
ARM11
Cortex-M1
ARM9
Cortex-M0+
ARM7
Cortex-M0
Performance, Functionality
Capability

架构


CRYPTO
CRYPTO
VFPv3/v4
NEON™
Adv SIMD
Key feature
ARMv7-A
compatibility
Thumb®-2
TrustZone®
SIMD
VFPv2
Jazelle®
A32+T32 ISAs
including:
• Scalar FP
(SP and DP)
• Adv SIMD
(SP Float)
AArch32
A64 ISA
including:
• Scalar FP
(SP and DP)
• Adv SIMD
(SP+DP Float)
AArch64
ARMv5
ARMv6
ARMv7-A/R
ARMv8-A

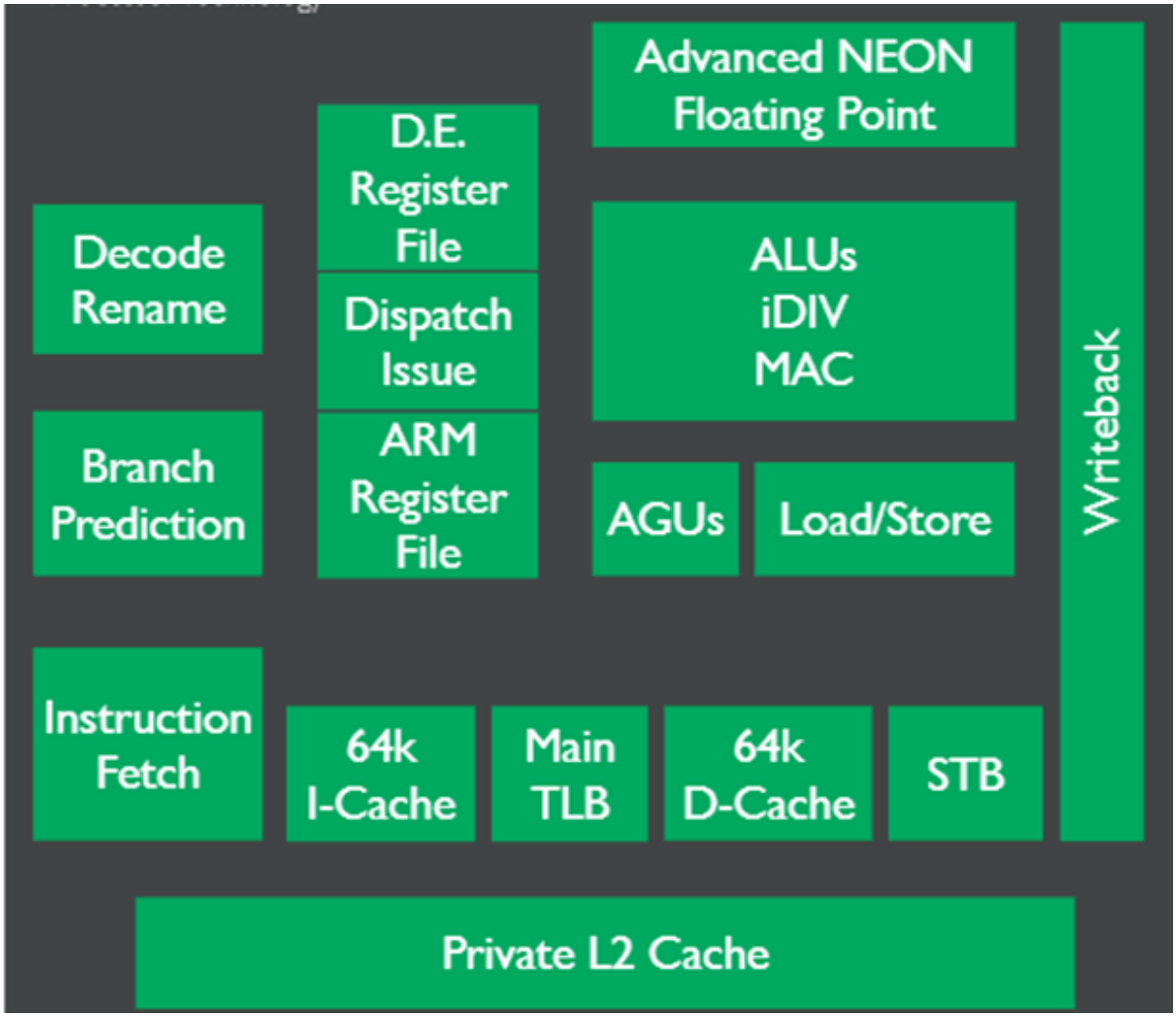
- 指令

处理器	流水线	指令集	存储管理	协处理器	多核
ARM7	3	ARM/T	N	N	N
ARM9	5	ARM/T	Y	Y	N
ARM10	6	ARM/T	Y	Y	N
ARM11	8	Thumb-2	Y	Y	N
Cortex-A8	NEON	V7-A	Y	Y	N
Cortex-A9,1X	NEON	V7-A	Y	Y	Y
Cortex-A5X	NEON	V8-A	Y	Y	Y

- 总线

版本	处理器	发布时间	总线(bit)	总线结构
V1，V2	ARM1	未商业授权		
V3	.....	未商业授权	32	
V4	ARM7/9	1996	32	7-Von Neumann 9-Harvard
V5	ARM10	2000	32	Harvard
V6	ARM11	2002	32	Harvard
V7	Ax, 1X	2004	32	Harvard
V8	A53/57	2013	64	Harvard

Cortex-A75-Core（Snapdragon 845）

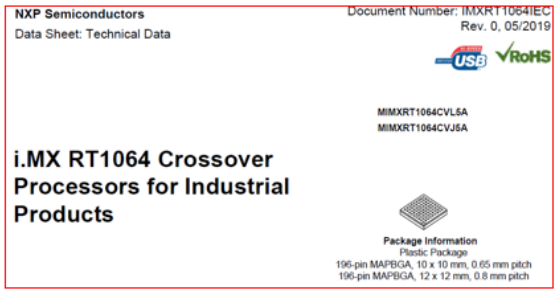


S3C2410A

硬件工程师在设计包含处理器的电路板时需要掌握处理器的哪些信息？

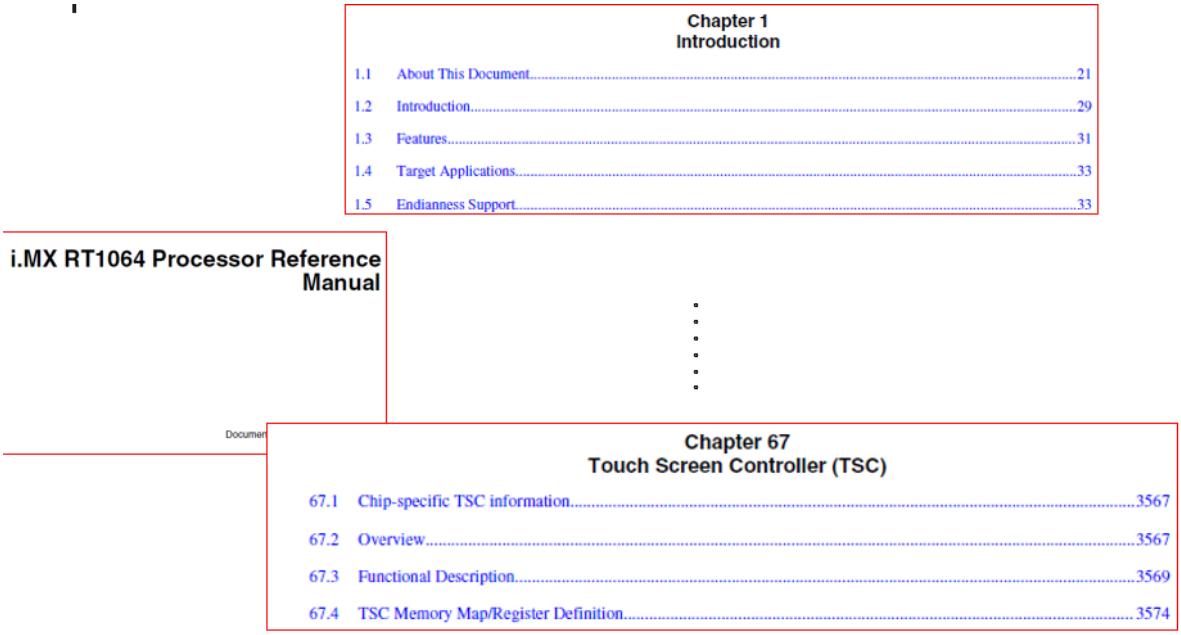
数据手册（datasheet）

datasheet


NXP Semiconductors
Data Sheet: Technical Data
Document Number: IMXRT1064IEC
Rev. 0, 05/2019
MIMXRT1064CVL6A
MIMXRT1064CVJ6A
i.MX RT1064 Crossover Processors for Industrial Products
Package Information
Plastic Package
196-pin MAPBGA, 10 x 10 mm, 0.65 mm pitch
196-pin MAPBGA, 12 x 12 mm, 0.8 mm pitch

软件工程师在为处理器开发程序时需要掌握处理器的哪些信息?

使用手册（manual）


i.MX RT1064 Processor Reference Manual

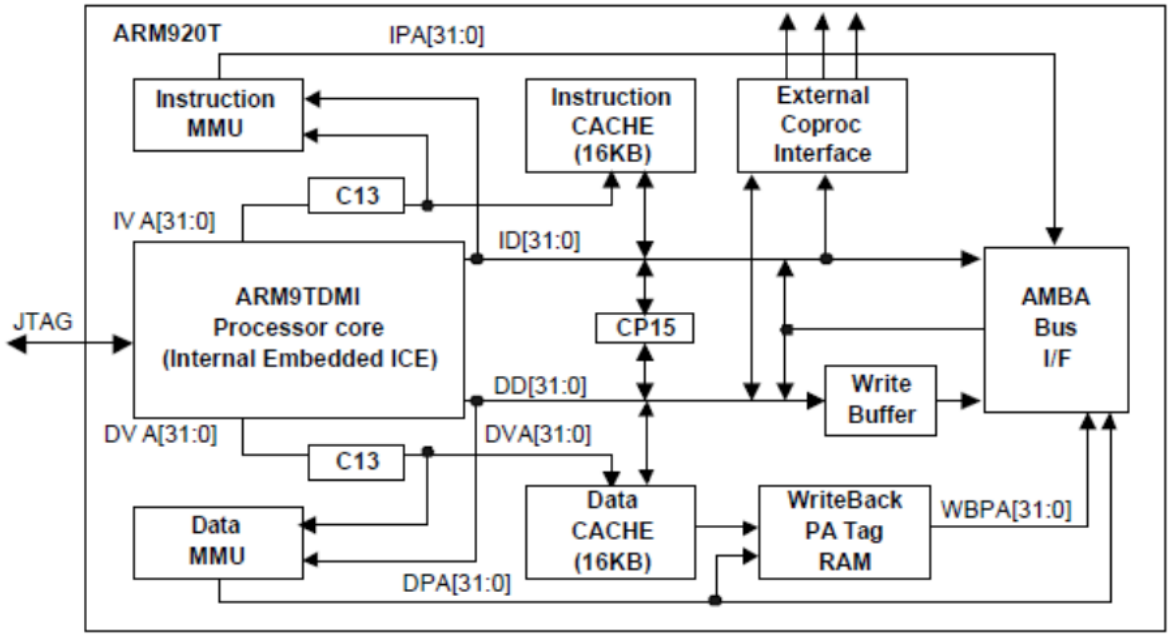
S3C2410

USER'S MANUAL

S3C2410A – 200MHz & 266MHz

32-Bit RISC Microprocessor

Revision 1.0



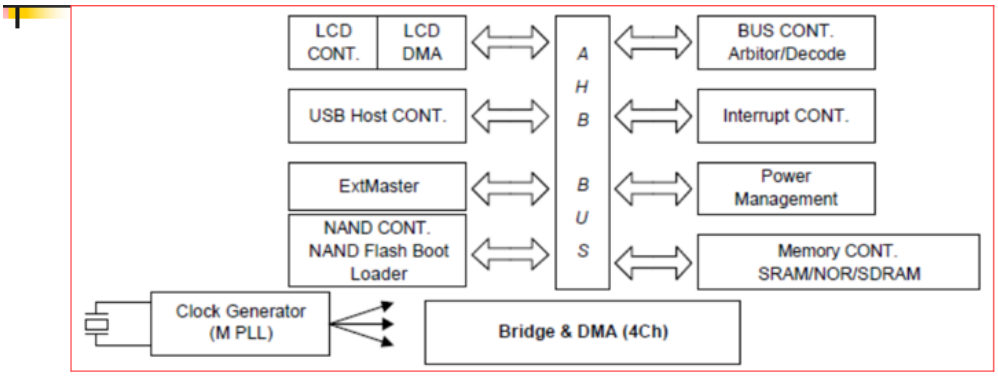
Architecture

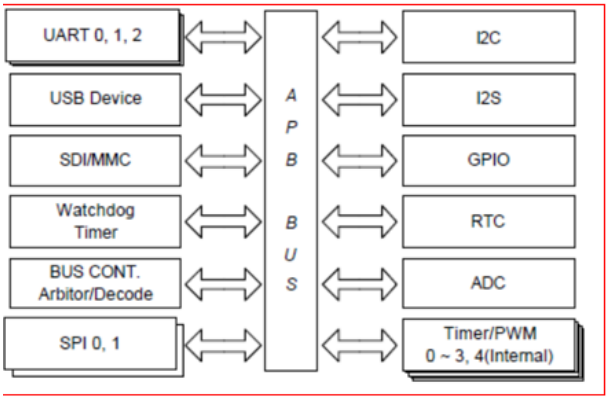
- Integrated system for hand-held devices and general embedded applications
- 16/32-Bit RISC architecture and powerful instruction set with ARM920T CPU core
- Enhanced ARM architecture MMU to support WinCE, EPOC 32 and Linux
- Instruction cache, data cache, write buffer and Physical address TAG RAM to reduce the effect of main memory bandwidth and latency on performance
- ARM920T CPU core supports the ARM debug architecture.
- Internal Advanced Microcontroller Bus Architecture (AMBA) (AMBA2.0, AHB/APB)

System Manager

- Little/Big Endian support
- Address space: 128M bytes for each bank (total 1G bytes)
- Supports programmable 8/16/32-bit data bus width for each bank
- Fixed bank start address from bank 0 to bank 6
- Programmable bank start address and bank size for bank 7

- Eight memory banks:
  - Six memory banks for ROM, SRAM, and others.
  - Two memory banks for ROM/SRAM/ Synchronous DRAM
- Fully Programmable access cycles for all memory banks
- Supports external wait signals to expend the bus cycle
- Supports self-refresh mode in SDRAM for power-down
- Supports various types of ROM for booting (NOR/NAND Flash, EEPROM, and others)





SPI Interface

- Compatible with 2-ch Serial Peripheral Interface Protocol version 2.11
- 2x8 bits Shift register for Tx/Rx
- DMA-based or interrupt-based operation

Operating Voltage Range

- Core: 1.8V for 200MHz (S3C2410A-20)
  2.0V for 266MHz (S3C2410A-26)
- Memory & IO: 3.3V

Peripheral Registers

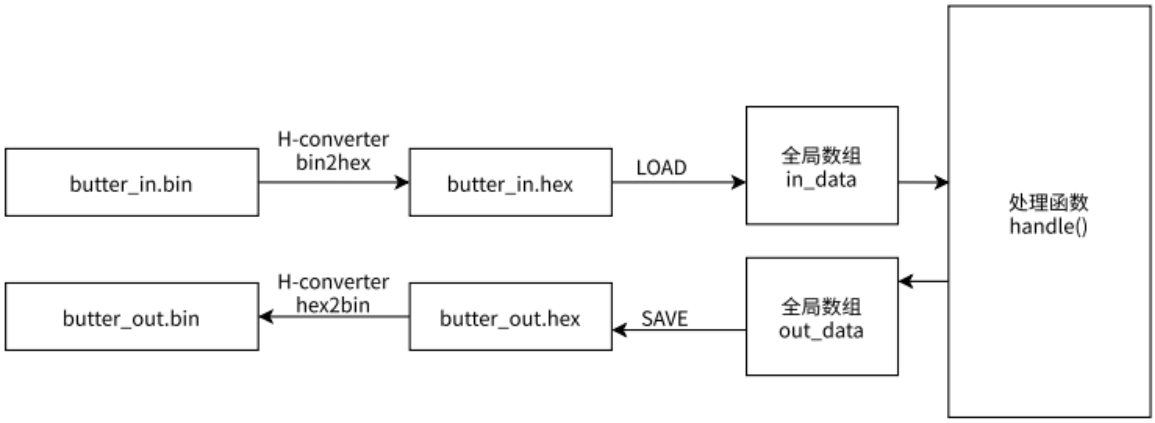
Register	Address	R/W	Description
ULCON0	0x50000000	R/W	UART channel 0 line control register
ULCON1	0x50004000	R/W	UART channel 1 line control register
ULCON2	0x50008000	R/W	UART channel 2 line control register

Register	Address	R/W	Description
ADCCON	0x58000000	R/W	ADC control register

Register	Address	R/W	Description
RTCCON	0x57000040(L) 0x57000043(B)	R/W (by byte)	RTC control register

ARM汇编程序

1. Keil作为半主机模式如何导入导出数据?



2. 如何测试处理器性能?

3. Keil工程中startup.s/S3c2410.s的作用?

ALIGN= n， 2^n次方对齐！

```
Stack_Size      EQU     (UND_Stack_Size + SVC_Stack_Size +ABT_Stack_Size +
\FIQ_Stack_Size + IRQ_Stack_Size + USR_Stack_Size)
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
Stack_Top       EQU     Stack_Mem + Stack_Size
```

```
Heap_Size       EQU     0x00000000
                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
Heap_Mem        SPACE   Heap_Size
```

```
                AREA    RESET, CODE, READONLY
                ARM
Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                ......
                IMPORT  __main
                LDR     R0, =__main
                BX      R0
                ......
                END
```

- Heap_size：动态分配空间
- Heap_mem：需指定位置与大小
- 全局空间是link时自动分配的空间

汇编程序格式

ARM汇编

Keiln(ARM ASM)

```
AREA    ARMex, CODE, READONLY
        ENTRY
start
        MOV       r0, #10
        MOV       r1, #3
        ADD       r0, r0, r1

        END
```

Segger (gnu asm)

```
.weak _start
            .global __start
            .section .init, “ax”
            .type _start, function
            .code 32
            .balign 4
_start:
__start:
        //
        // Setup Stacks
        //for asm testing
        mov r0, #0x1000
```

```
        mov r1, #0x300
        mov r2, #0x400
        add r1,r1,r2
        strb r1,[r0]
       //testing end
    ……
```

- ax （allocation execute）
- .weak _start 有外部调用则调用外部，没有调用内部
- .code 32 表示32位
- .balign 4 对齐
  - 有b，表示四个字节对齐；
  - 没有b，表示2的多少次方对齐。
- _start:缺省入口
- __start:系统启动定义的

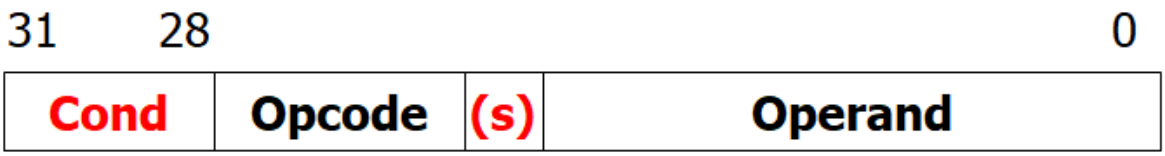
指令级程序

格式

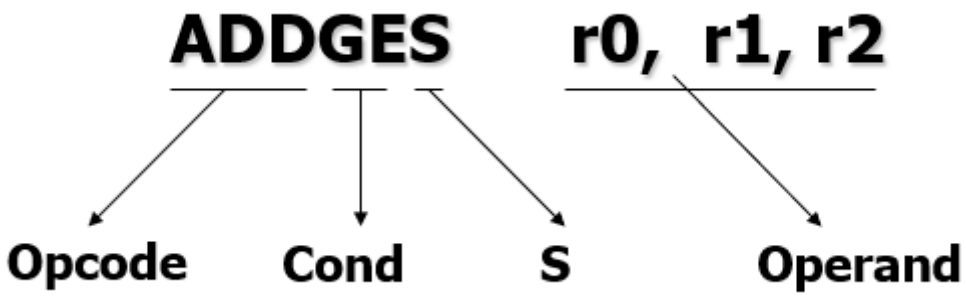
ASM Directive

- AREA
  - Mark the start of a section
  - Names the section and sets its attributes.
  - The attributes are placed after the name, separated by commas
- ENTRY
- Marks the first instruction to be executed
- Initialization code and exception handlers also contain entry points.
- END
  - Instructs the assembler to stop processing this source file
  - On a line by itself.

指令格式



助记符



- GE：Cond，条件判断
- S：状态位

语句格式

- Lines in an ARM assembly language module is:

{symbol} {instruction|directive|pseudo-instruction} {;commen}

- Examples

lable1 ADD r1，r2，r3 ;”+

```
AREA     ARMex, CODE, READONLY

        ENTRY     ; Mark first instruction to execute

start
        MOV       r0, #10        ; Set up parameters
        MOV       r1, #3
        ADD       r0, r0, r1      ; r0 = r0 + r1

        END                       ; Mark end of file
```

- 标签最左边对齐
- 语句最左边需要空格

用汇编语言实现把下列C程序功能?

```
int main(void)
{   int i, j;
    unsigned char Inimage[480][640];
    unsigned char Tmpimg[120][160];
    for(i=0;i<120;i++)
      for(j=0;j<160;j++)
                Tmpimg[i][j]=Inimage[i*4+2][j*4+2];
    memcpy(Inimage,Tmpimg,160*120);
    return 1;
}
```

汇编指令

#	别名	用途
R0	-	通用寄存器
R1	-	通用寄存器
R2	-	通用寄存器
R3	-	通用寄存器
R4	-	通用寄存器
R5	-	通用寄存器
R6	-	通用寄存器
R7	-	一般放系统调用号
R8	-	通用寄存器
R9	-	通用寄存器
R10	-	通用寄存器
R11	FP	栈帧指针
R12	IP	内部程序调用
R13	SP	栈指针
R14	LR	链接寄存器(一般存放函数返回地址)
R15	PC	程序计数寄存器
CPSR	-	当前程序状态寄存器

- R0-R15和r0-r15
- a1-a4(参数,结果或者临时寄存器,与r0-r3同意)
- v1-v8(变量寄存器,与r4-r11同意)
- sb和SB(静态基址寄存器,与r9同意)
- sl和SL(堆栈限制寄存器,与r10同意)
- fp和FP(帧指针,与r11同意)
- ip和IP(过程调用中间临时寄存器,与r12同意)
- sp和SP(堆栈指针,与r13同意)
- lr和LR(连接寄存器,与r14同意)
- pc和PC(程序计数器,与r15同意)
- cpsr和CPSR(程序状态寄存器)
- spsr和SPSR(程序状态寄存器)
- f0-f7和F0-F7(FPA寄存器)
- s0-s31和S0-S31(VFP单精度寄存器)
- d0-d15和D0-D15(VFP双精度寄存器)
- p0-p15(协处理器0-15)
- c0-c15(协处理器寄存器0-15)

变量与数据

数据类型？ Byte，Halfword，Word，Doubleword，QuadWord (Signed or unsigned？) – Only unsigned (except for MUL)

变量位置？ Memory (Stack，static), Register

变量赋值？ Address (immediate，memory，Register)

常量定义？ EQU

指令和寄存器大小写有区别?

MOV 立即数寻址可以装载任意数据?

- MOV 是ALU单元的操作，改变状态
- MVN 按位取反再装载

Loading immediate

- MOV
  - 8-bit ,0x0 to 0xFF (0-255).
  - Rotate by any even number.
  - MVN load “NOT” Value
  - Thumb： only 0x00-0xff

> 理解8位或者偶数移位（**Rotate by any even number**）
>
> 例如260的二进制是100000100 可以由1000001循环右移30位或左移2位，位数位偶数，可以作为立即数
>
> 在比如258二进制是100000010 可以由10000001循环右移31位或左移1位，位数是奇数，不可以作为立即数

实际理解为为了将12位的数字映射到32位上，用8位作为基：0~255；4位作为rotate，循环右移2*rotate

下列指令是否正确?

```
MOV  rn ,#1025 错
MOV  rn ,#4096 对
MVN  rn ,#0x111 错
MVN  rn ,#0xffffffff 对
```

装载任意大小数值

- LDR Rd, =const

伪指令?

这里两条指令的执行过程与立即数寻址过程的查差别?

- LDR
  - LDR -> Appropriate instruction (by assember)
  - Constant is in the range of MOV or MVN

    LDR -> MOV (MVN)

    MOV or MVN” depend on what?
  - Constant is over the range of MOV or MVN
    - Places the value in a literal pool
    - LDR instruction with a program-relative address that reads the constant from the literal pool.

```
LDR r0, =0x23
LDR r0, =0xffffff       ;MVN r0，#ff000000
LDR r0, =0x5555     ;?
;LDR   rn, [pc, #offset to literal pool]
```

LDR对于不符合的立即数如何转换?

利用pc和literal pool读取

- How to put the “literal pool” in memory?
  - Marking with “LTORG”
    - After unconditional branch instructions
    - after the return instruction
- By assembler
  - After the end of “AREA”
- The offset from the pc to the constant
  - ARM state
    - Less than 4KB
    - Either direction
  - Thumb state
    - Less than 1KB
    - Forward

为什么 “LTORG” 要放在跳转后或返回后？不会当作指令取进去.

下列程序中 LDR 指令能否正确转换?

```
AREA   Loadcon, CODE, READONLY
       ENTRY

       LDR      r0, =42          ; => MOV R0, #42
       LDR      r1, =0x55555555 ; => LDR R1, [PC, #offset to Literal Pool 1]
       LDR      r2, =0xFFFFFFFF           ; => MVN R2, #0
        LTORG                            ; Literal Pool 1 contains
                                               ; literal 0x55555555
       LDR      r3, =0x55555555  ; => LDR R3, [PC, #offset
                                             ; Literal Pool 1
       LDR      r4, =0x666
       END
                  ; Literal Pool 2 contains 0x666
```

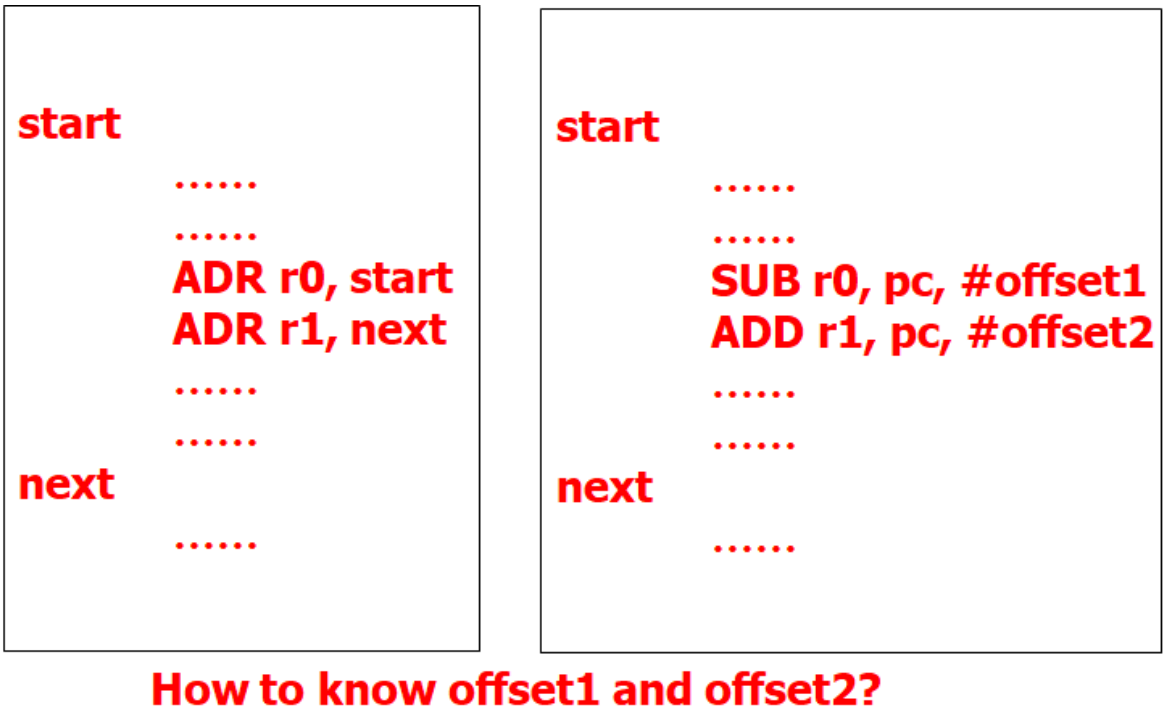
寄存器常数装载方式

指令	功能	说明
MOV(MVN)	装载立即数	8bit
LDR	装载常数(伪)	32bit位常量
ADR（L)	装载地址(伪)	32bit 相对地址

```
MOV{<cond>}{S}   Rd, #const
LDR Rd, =const
ADR Rd, label
```

- Load address
  - ADR and ADRL
    - Generate an address, within a certain range
      - program-relative expression
        - Label with an optional offset
        - Be relative to the current pc.
      - Register-relative expression
        - Label with an optional offset
        - Be relative to an address held in a specified general-purpose register.
- **ADR**
  - Converts ADR rn,label into a Single ADD or SUB instruction that loads the address, if it is in range
  - The offset range of ADR
    - ±255 bytes for an offset to a non word-aligned address.
    - ±1 020 bytes (255 words) for an offset to a word-aligned address.
  - An error message if the address cannot be reached in a single instruction

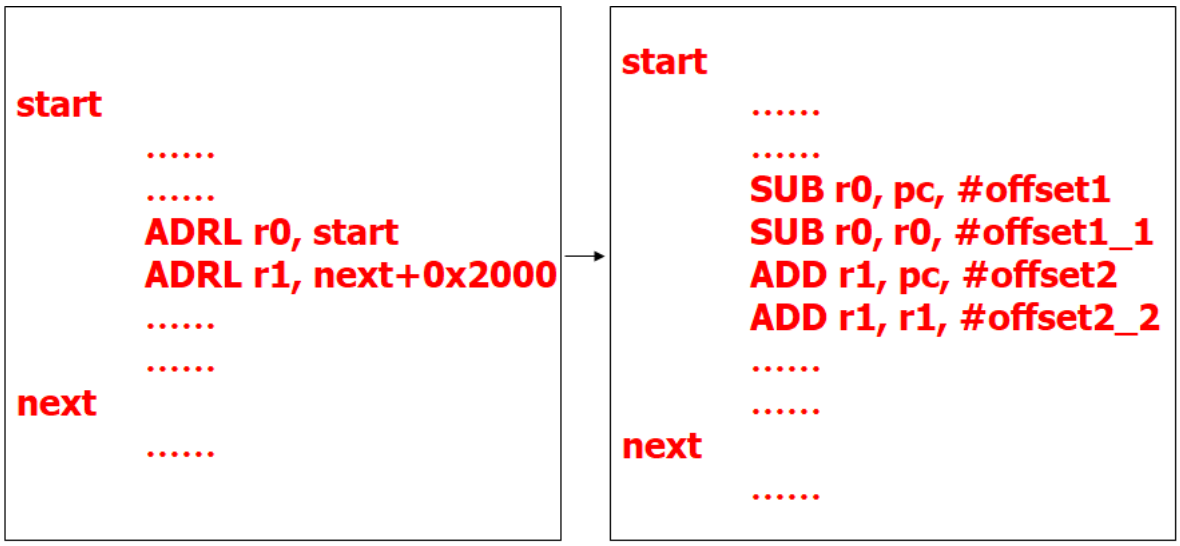
分析下列两条ADR指令的转换结果



- **ADRL**
  - Converts an ADRL rn,label into two data-processing instructions that load the address, if it is in range
  - The range of an ADRL
    - ± 64KB for a non word-aligned address
    - ± 256KB for a word-aligned address.
  - An error message if the address cannot be constructed in two instructions.

- There is no ADRL pseudo-instruction for Thumb.

分析下列两条ADRL指令的转换结果



“ADR or ADRL” depend on what?

运算

循环中的递加：r5=r5+1，r6=r6+1

```
add r5, r5, #1
add r6, r6, #1
```

语法

```
add{cond}{s} Rd, Rn, Operand2
```

第二操作数（Operand2）

- #<immediate>
- <Rm>
- <Rm>, ASR(LSR, ROR, ASL, LSL）#<shifter_imm>
- <Rm>, ASR(LSR, ROR, ASL, LSL） <Rs>
- <Rm>, RRX

<Shifter_imm> L: 0-31; R: 1-32
RRX: C->bit[31], (Rm>>1)→ Rm

运算符的操作数一定是寄存器或者立即数

- ASR 算数右移
- LSL 逻辑左移
- LSR 逻辑右移
- ROR 循环右移
- RRX 带扩展的循环右移
  - <Shifter_imm> **L: 0-31; R: 1-32。**
  - RRX 只能移动一位

分析下指令

```
ADD     r0, r1, #10
ADD   r11,r12,r3, ASR #5
ADD   r11,r12,r3,LSL R4
ADD     r5,r4, r3, RRX
```

算术指令

- 语法

  Op{cond}{s} Rd, Rn, Operand2

- 标志位

  N, Z, C,V

- 指令

指令	操作
ADC	Rd=Rn + Operand2+C
ADD	Rd=Rn + Operand2
RSB	Rd=Operand2-Rn
RSC	Rd=Operand2-Rn-!C
SBC	Rd=Rn-Operand2-!C
SUB	Rd=Rn-Operand2

当看溢出时，要把数据看成是有符号数，就是最高位的数字代表是符号，0正1负。如果次高位计算改变符号位则溢出。

当看进位时是将数据看成无符号数，全部是数据没有符号位，如果运算超出数的范围向前进位了，则进位标志置位。

下列指令哪些时错误的?

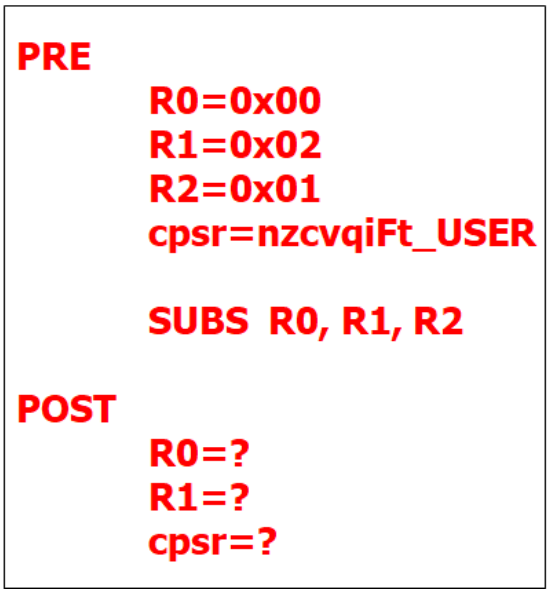
```
ADC         r1, r1, #5  对
SUB         r11,r12,r3,ASR #5   对
RSB         r5,r3,r4, LSR #32       对
ADD         r3,r7,#1023     错
SUB         r11,r12,r3,LSL #32  错 最多31
SBC         r5,r4, r4, RRX #3       错 最多1
```

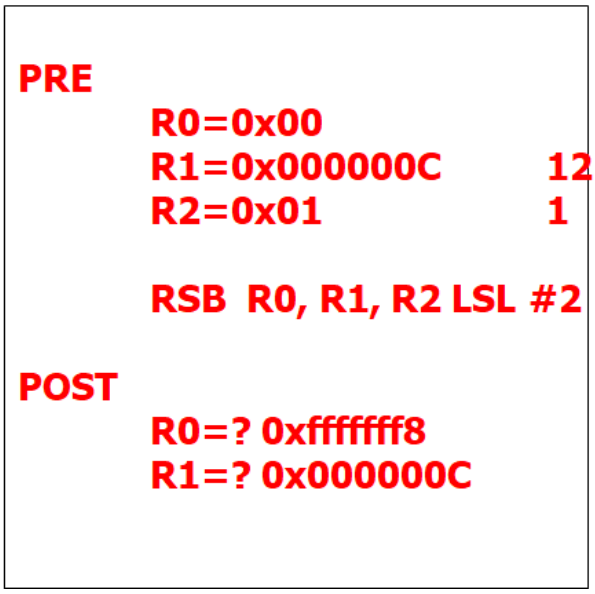
给出指令的执行结果

C？

加法：有进位 C=1 (结果超过范围，最高位)

减法：无借位 C=1 (结果大于0)





- cpsr=nzcvqiFt_USER；指定当前个标志位状态，并且工作在用户模式

乘法指令

- MUL和MLA (32bit *32bit -> least 32bit)
  - 语法

    MUL{***cond***}{*S*} *Rd, Rm, Rs*

    MLA{***cond***}{*S*} *Rd, Rm, Rs, Rn*
  - 执行

    MUL: Rs*Rm ->Rd (least)

    MLA: Rs*Rm+Rn -> Rd (least)
  - 标志位

    N, Z

d!=m

> 32位乘32位低32位！

分析下列指令

```
MUL     r10,r2,r5
MLA     r10,r2,r1,r5
MULS    r0,r2,r2

MUL     r15,r0,r3       ;错
MLA     r1,r1,r3, r6    ;错
```

- U（S）MULL, U（S）MLAL(32bit *32bit -> 64bit)
  - 语法

    Op{cond}{S} RdLo, RdHi, Rm, Rs
  - 执行

    U（S）MULL: Rs*Rm ->RdLo, RdHi

U（S）MLAL: Rs*Rm+(RdLo,RdHi) -> RdLo , RdHi

- 标志位

  N, Z

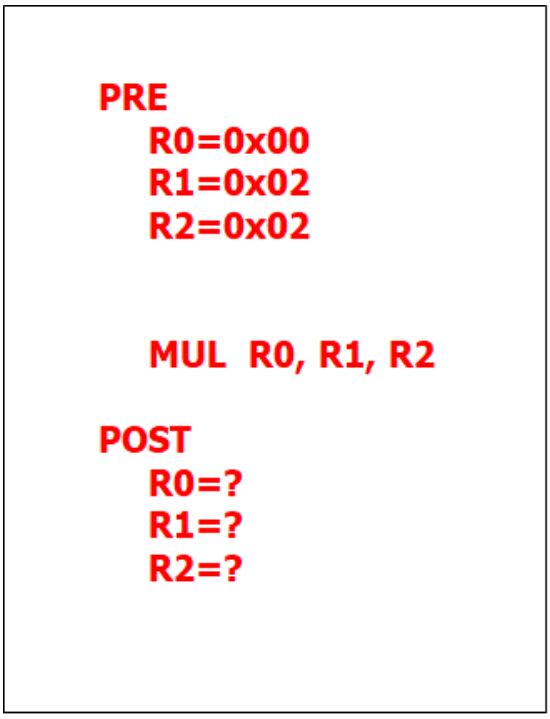
- 例:

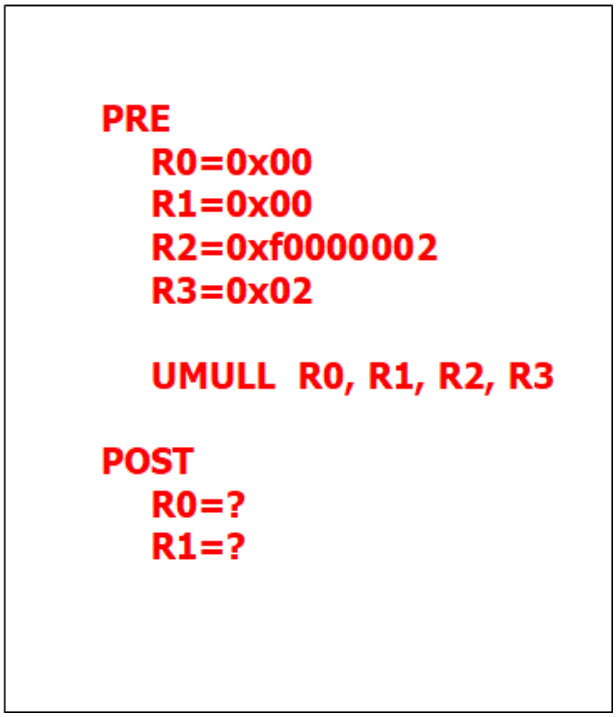
```
UMULL       r0,r4,r5,r6
UMLALS     r4,r5,r3,r8
UMULL       r1,r15,r10,r2
```

> 目的寄存器需要用两个32位来存储一个64位

UMLAL 与 MLA 操作的差别?

给出下列指令的执行结果





逻辑指令

- **语法**

  Op{cond}{s} Rd, Rn, Operand2

- 指令

指令	操作
AND	Rd=Rn & Operand2; shift_operand
ORR	Rd=Rn | Operand2
EOR	Rd=Rn ^ Operand2
BIC	Rd=Rn & (~Operand2)

- 例

PRE R1=0b1111 R2=0b0101	BIC R0, R1,R2	POST R0=? R1=? R2=?

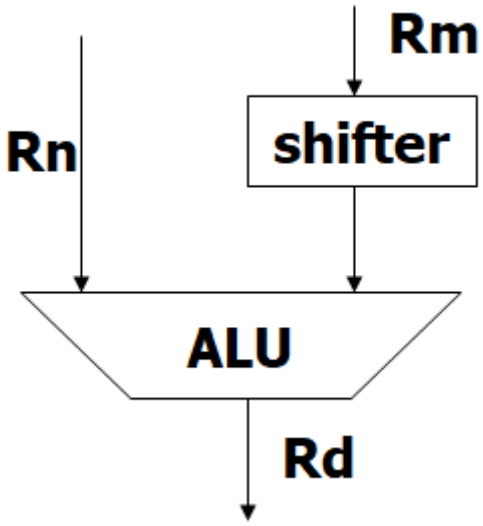
MOV

寄存器间数据复制

```
PRE
    R0=0x80
    R1=0x50
    MOV  R0, R1

POST
    R0=? 0x50
    R1=? 0x50
```

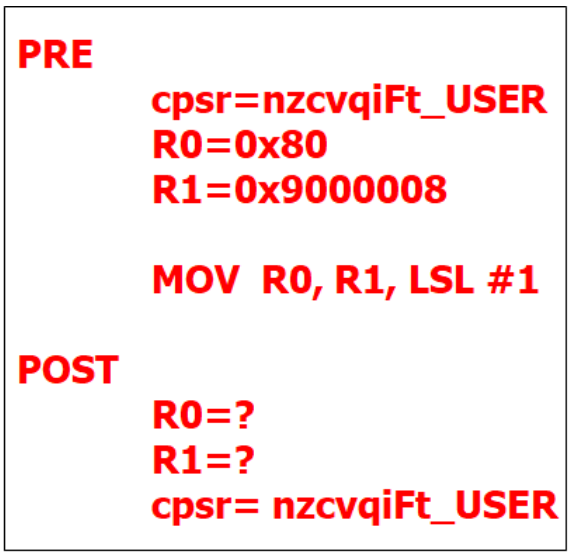
使用桶形移位寄存器

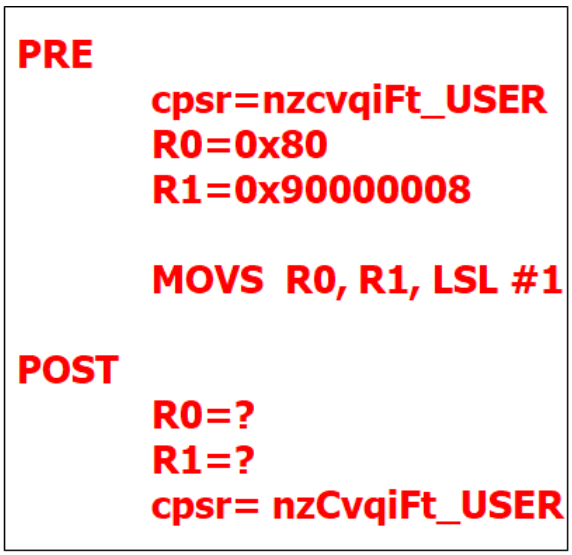
```
PRE
    R0=0x80
    R1=0x50

    MOV  R0, R1, LSL #4

POST
    R0=? 0x500
    R1=? 0x50
```



移位操作	结果	Y**值范围**
x LSL y	x << y	#0-31 or Rs
x LSR y	(unsigned) x >> y	#1-32 or Rs
x ASR y	(signed) x >> y	#1-32 or Rs
x ROR y	((unsigned)x >>y)|(x<<(32-y))	#1-32 or Rx
x RRX	(c flag <<31)|((unsigned)x>>1)	none

移位对C标志的影响





MVNS r0, #1　N=1

> 状态寄存器小写0，大写1

比较

```
比较判断    r6< Imageheight? ;  r5<Imagewidth ?
    cmp r6, #Imageheight
    cmp r5, #Imagewidth
```

- 语法

  CMP{cond} Rn, Operand2

- 第二操作数（Operand2）

```
#<immediate>
<Rm>
```

> ALU_out[31]->N; alu_out=0, Z=1 else Z=0;
>
> Not borrow from “Rn-Operand2 “ : C=1 else C=0;
>
> Overflow from “ Rn-Operand2”: V=1 else V=0;

> 自动改变状态寄存器

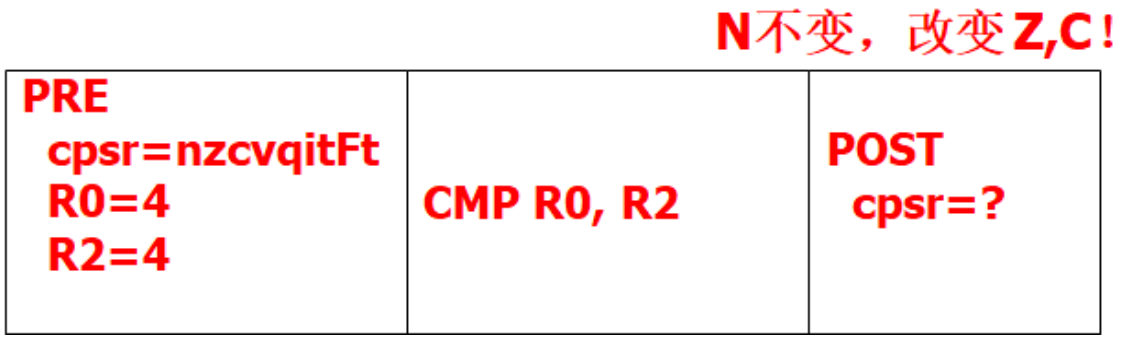
- 语法

Op{cond} Rn, Operand2

- 指令

指令	操作
CMN	flag of Rn + Operand2;<Rm>
CMP	flag of Rn - Operand2; <Rm>, <imm_8>
TEQ	flag of Rn ^ Operand2;<shift_operand>
TST	flag of Rn & Operand2;<Rm>

- 例



> 不改变通用寄存器数值，自动改变状态寄存器状态

跳转

```
跳转      for (;;)
行内循环                cmp    r5, #Imagewidth
         bmi      Linecircle
行间循环
                   cmp    r6, #Imageheight
                            bmi      Framecircle
```

- 语法

```
B{<cond>} label
```

- 如果Imagewidth 或 Imageheight 大于0xff，如何处理?
- label？ 有什么要求?
- 支持哪些 cond？ 如何表达?
- 语法

```
B{<cond>} label
BL{<cond>} label
BX{<cond>} Rm
BLX{<cond>} label|Rm
```

- 指令

指令	功能
B(跳转)	pc=label
BL(带返回跳转)	pc=label, lr=BL 后面第一条指令地址
BX(切换状态跳转)	pc=Rm & 0xfffffffe, T=Rm & 1
BLX	pc= label, T=1 pc=Rm & 0xfffffffe, T=Rm & 1 lr=BLX 后面的第一条指令地址

- **B**

```
B   forward
backward
    ADD     r1, r2, #4
    ADD r0, r6, #2
    ADD     r3, r7, #5
forward
    SUB r1, r2, #4
    ADD r1, r2, #4
    SUB r1, r2, #4
    ADD r4, r6, r7
    B   backward
```

B 可以在4G空间任意范围内跳转吗?

±32MB

- **BX**

```
CODE32
header
    MOV r0, #0
    ADR     r0, start + 1
        BX      r0

    CODE16
start
            ADD r1, r2
    SUB r2, #0x55
            CMP r0, r1
       ……
```

- 如何添加指令，从CODE16段跳到header
- Thumb指令特点:
  - 使用ARM的r0-r7 8个通用寄存器
  - Thumb指令没有条件执行
  - 指令自动更新标志，不需加（s）
  - 仅有LDMIA（STMIA）
    - 只有IA一种形式
    - 必须加“！”
  - 在数据运算指令中，不支持第二操作数移位

> Thumb指令是16位的Arm指令集，可以理解为Arm的阉割版
>
> https://blog.csdn.net/qq_20880415/article/details/101037010
>
> 指令操作16位
>
> 操作依然是32位

- **BL**

```
BL  subroutine
    CMP     r1, #5
    MOV     r1, #0
    ADD     r1, r2, #4
    ……
subroutine
    ADD r1, r2, #4
    SUB r1, r2, #4
    MOV pc, lr
```

函数的返回方式?

> Lr是跳转前下一条指令的地址

- 条件类型

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS/HS	C set	Higher or same (unsigned >=)
CC/LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

如果结果等于1，哪个状态为置1？

LE和LS有什么不同？(<=)

HI和GT有什么不同？(<=)

- 比较下列指令

```
ADD     r0, r1, r2    ; r0 = r1 + r2, don't update flags
ADDS    r0, r1, r2    ; r0 = r1 + r2, and update flags
ADDCSS  r0, r1, r2    ; If C flag set then r0 = r1 + r2,
                                  ;    and update flags
CMP     r0, r1        ; update flags based on r0-r1.
```

- **下列程序中哪些指令不执行**

```
MOV r0, #1
MOV r1, #2
CMP r0, r1  ;N=1
SUBGT r0,r0,r1  ;不执行
SUBLT r1,r1,r0  ;r1=1
CMP r0,r1       ;z=1
SUBGT r0,r0,r1  ;不执行
SUBLT r1,r1,r0  ;不执行
```

- **不同跳转方法的对比**
  - 实现方法

```
int gcd(int a, int b)
{
    while (a != b) do
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUB     r0, r0, r1
        B       gcd
less
        SUB     r1, r1, r0
        B       gcd
end
```

```
gcd
        CMP     r0, r1
        SUBGT   r0, r0, r1
        SUBLT   r1, r1, r0
        BNE     gcd
```

algorithm **condition branch** **execute conditional**

  - 执行时间分析（condition branch）

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	CMP r0, r1	1
1	2	BEQ end	1 (not executed)
1	2	BLT less	3
1	2	SUB r1, r1, r0	1
1	2	B gcd	3
1	1	CMP r0, r1	1
1	1	BEQ end	3
			Total = 13

不执行的指令也占时间?

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	CMP r0, r1	1
1	2	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1
1	1	BNE gcd	3
1	1	CMP r0,r1	1
1	1	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1 (not executed)
1	1	BNE gcd	1 (not executed)
			Total = 10

小结

```
//c/c++
for(i=0;i<120;i++)
{
    for(j=0;j<160;j++){……}
}
```

```
Imageheight equ 120
Imagewidth  equ 160

        area    ImScale , code , readonly
        entry
        mov r6, #0
Framcircle
    mov r5, #0
Linecircle
    ……
        add r5, r5,#1
        cmp r5, #Imagewidth
      bmi  Linecircle
        add r6, r6, #1
        cmp r6, #Imageheight
        bmi Framecircle
        end
```

ARM汇编

把下列C程序改成汇编程序：

```c
int main(void)
{   int i, j;
    unsigned char Inimage[480][640];
    unsigned char Tmpimg[120][160];
    for(i=0;i<120;i++)
      for(j=0;j<160;j++)
                Tmpimg[i][j]=Inimage[i*4+2][j*4+2];
    memcpy(Inimage,Tmpimg,160*120);
    return 1;
}
```

存储访问

内存分配

- 常数
  - 无符号整数:0 to $2^{32}-1$
  - 整数:-2^{31} to $2^{31}-1$.
  - 赋值符号：EQU

```
//c/c++
Tmpimg[120][160];
Inimage[480][640];

//分配
Tmpimg  space       160*120 ;单位字节
Inimage space       640*480

//语法
[label]     SPACE       expression
可用 %代替 SPACE
```

“160×120”和“640×480”中的“×”运算在程序运行时完成吗?

内存区域分配

- DCX{U-对齐选项}
  - 种类
    - DCB-(1byte)
    - DCD and DCDU-(4 bytes)
    - DCFD and DCFDU-(double-precision float point)
    - DCFS and DCFSU-(single-precision float point)
    - DCI-(like DCD and DCDU)
    - DCQ and DCQU-(8 bytes)
    - DCW and DCWU-(2 bytes)
  - 用法

    {label} DCX{U} *expression*, *{expression,.....}*
  - Example

```
data        DCW     -225,2*number
              DCWU    number+4    ;number must be   defined
    DCQ     -225, 2_101
DCFD    1E308, -4E-100
DCFDU   10000, -.1, 3.1E26
```

```
加“U”不需要对齐
不同命令对其格式不一样
DCD-4Bytes
DCW-2Bytes
```

分析内存中数据存储状况

```
    DCB     1，2，3
    DCDU 0x11223344
    DCW     0x2233，0x4455
    DCWU 0x677
    DCD 0x06070809
```

- Endian 模式对存储状况的影响？大端小端
- SPACE与 DCX使用上的区别？ DCX分配空间与写入值
- 程序中如何找到分配区域的起始地址？自定义标签

> 练习，画出内存中存储位置

数据结构

```
//Declare
typedef struct Point
{
        float x;
    float y;
    float z;
} Point;

//Allocate space
Point origin;
```

```
//Declare
PointBase   RN      r11
                MAP     0,PointBase     ;r11地址作为初始地址
Point_x     FIELD   4
Point_y     FIELD   4
Point_z     FIELD   4

//Allocate space
origin      SPACE   12
```

内存访问

```
//c/c++
    Tmpimg[i][j]; Inimage[i*4+2][j*4+2];

//i->r5; j->r6; Tmpimg-> r8; Inimage->r9;
    ldr r8, =Tmping
    ldr r9, =Inimage
//Tmpimg[i][j]->r10;
    mov r0, #Imagewidth
    mul r10, r5, r0   ;i*Imagewidth
    add r10, r10, r6   ;i*Imagewidth+j
//Inimage[i*4+2][j*4+2]-> r11
    mov r1, r5, asl #2 ; i*4
    add r1, r1, #2  ;i*4+2
    mul r11, r1, r0 ;(i*4+2)*Imagewidth
    mov r1, r6, asl #2 ;j*4
    add r1, r1, #2    ;j*4+2
    add r11, r11, r1; (i*4+2)*Imagewidth+4*j+2
```

```
//c/c++
    Tmpimg[i][j]=Inimage[i*4+2][j*4+2];

//实现
    ldrb    r0, [r11]    读取方式
    strb    r0, [r10]    存储数据

//语法
LDR {<cond>}{B} Rd, addressing1
STR{<cond>}{B}  Rd, addressing1
```

读（Memory->Register）

指令	功能
LDR	Rd<-mem32[add]
LDRB	Rd<-mem8[add]
LDRH	Rd<-mem16[add]
LDRSB	Rd<-SignExtend(mem8[add])
LDRSH	Rd<-SignExtend(mem16[add])

- 指令形式

  ```
  LDR {<cond>}{B} Rd, address1
  LDR {<cond>}SB|H|SH Rd, address2
  ```

- address
  - 基于寄存器的寻址
  - 地址对齐要求?

LDR 可能改变状态寄存器状态位?

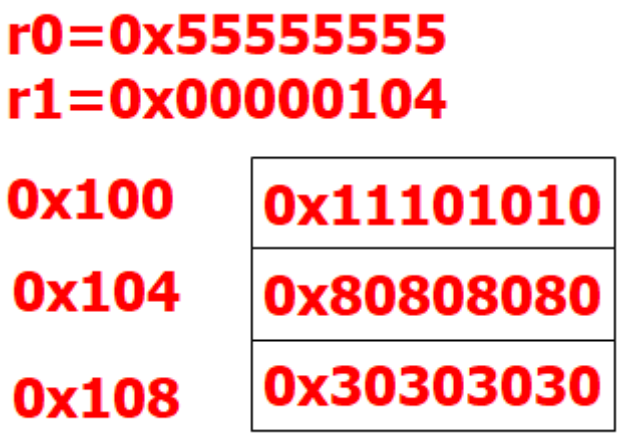
- Address1
  - [Rn]
  - 前偏移，Rn不变
    - [Rn, # +/- offset_12]
    - [Rn, +/-Rm]
    - [Rn,+/-Rm, shift_imm]
  - 前偏移，Rn更新
    - [Rn,#+/-offset_12]!
    - [Rn,+/-Rm]!
    - [Rn,+/-Rm, shift_imm]!
  - 后偏移，Rn更新
    - [Rn], # +/- offset_12
    - [Rn], +/-Rm
    - [Rn],+/-Rm, shift_imm
- **带“！”表示需要更新Rn内寄存器的值**

解释下列指令

```
ldr r0, [r1, #0x04]!
ldr r0, [r1, r2]!               ;R1+r2->r0，更新r1
ldr r2,[r1, -r0, LSR #0x04]!    ;R0<<4 +r1->r2，更新r1
ldr r0, [r1], #0x04             ;r1->r0，r0->r0+4
ldr r0,[r1],r2
rdr r0,[r1], -r0, LSL #0x04
ldr r0, [r1, #0x04]
ldr r0, [r1, r2]
ldr r2,[r1, -r0, LSR #0x04]
```

比较五条指令的执行结果

```
LDR r0, [r1, #4]    r0=0x3030303,r1=0x104
LDR r0, [r1, #4]!   R0=0x3030303,r1=0x108
LDR r0, [r1], #4    r0=0x80808080,r1=0x108
LDRB r0, [r1]       r0=0x00000080
LDRSB r0, [r1]      r0=0xFFFFFF80
```



- address2(针对LDR(S)B)
  - [Rn, #+/-offset_8]
  - [Rn,+/-Rm]
  - [Rn, #+/-offset_8]!
  - [Rn,+/-Rm]!
  - [Rn], #+/-offset_8
  - [Rn],+/-Rm

下列指令，哪些是正确的?

```
LDR r1, r2, r3           错，r2加[]
LDR r1,[r2, #0x111]      对，符合12位
LDR r1,[r2], +#0x11111   错，超出12位
LDRSB r1,[r2], -#0x111   错，SB只支持8位
LDRH [r1],r2             对
LDRSH r1,[r2,r3]         对
LDRB r1,[r2,-r3]         对
```

写（Register->Memory）

指令	功能
STR	Rd->mem32[add]
STRB	Rd->mem8[add]
STRH	Rd->mem16[add]

```
STR{<cond>}{B} rd, addressing1
STR{<cond>}{H} rd, addressing2
```

> **ldr vs str？**
>
> **为什么没有strsb和strsh?**

双字访问

语法

```
LDR|STR{cond}D Rd, [Rn]
LDR|STR{cond}D Rd, [Rn, Offset]{!}
LDR|STR{cond}D Rd, label
LDR|STR{cond}D Rd, [Rn], Offset
```

例:

```
LDRD    r6,[r11]         ;隐含r7,r7 = r11+4
STRD    r4,[r9,#24]
STRD    r0,abcd
LDRD    r1,[r6]         ; Rd 必须为偶数
STRD    r14,[r9,#36]    ; Rd 不能是R14.读可以写不可以
STRD    r2,[r3],r6      ; Rn 不能是 Rd 或 R(d+1).
```

> D表示读两个32位，两个字，第一个寄存器是偶数，一个字32位，半字16位

> 注：32位CPU的话，一个字是32位，正常16位CPU一个字16位
>
> 两个数字同时读取到寄存器中嘛？为什么？
>
> 一般一条总线或同一区间时，分先后

交换数据

指令

指令	Operation
SWP	tmp=mem32[Rn] Mem32[Rn]=Rm Rd=tmp
SWPB	tmp=mem8[Rn] Mem8[Rn]=Rm Rd=tmp

格式

```
SWP {B} {<cond>} Rd, Rm, [Rn]
```

分析下列指令的执行结果

```
mem32[0x8000]=0x12345678
R0=0x00000000
R1=0x22223333
R2=0x8000

SWP r0, r1, [r2]

mem32[0x8000]=? 0x22223333
R0=?            0x12345678
R1=?            0x22223333
R2=?            0x8000
```

结构访问

```
//C
origin.x = 0;
origin.y = 2;
origin.z = 3;

//asm
        LDR     PointBase,=origin
        MOV    r0,#0
        STR     r0,Point_x
        MOV    r0,#2
        STR     r0,Point_y
        MOV    r0,#3
        STR     r0,Point_z
```

如何实现数据在内存中的移动?

批量读写

```
//c/c++
    memcpy(Inimage,Tmpimg,160*120);

//asm
    ldr     r10,    =160*120
memcpy
    ldmia   r8! ,   {r0-r7} ;r8!地址寄存器
    stmia   r9! ,   {r0-r7}
    subs    r10,r10, #8*4
    bgt memcpy
```

多存储器数据传送

指令

指令	功能
LDM	{Rd}*N <- mem32[start add+4*N]
STM	{Rd}*N -> mem32[start add +4*N]

格式

- LDM|STM{cond}address-mode Rn{!},reg-list{^}
    - Cond: condition code.
    - address-mode: the mode of change address.
    - Rn: base register for the load operation.
    - ^ : includes lr and spsr ； 注意，cpsr切换异常模式

地址变化方式

符号	变化方式	数据起始地址	数据结束地址	Rn!
IA	后增加	Rn	Rn+4N-4	Rn+4N
IB	先增加	Rn+4	Rn+4N	Rn+4N
DA	后减少	Rn-4N+4	Rn	Rn-4N
DB	先减少	Rn-4N	Rn-4	Rn-4N

解释下列指令

```
LDMIA r1, {r0, r2-r7}   ;共7个
STMIA r1!, {r2,r3}
LDMDA r3, {r4-r7}
STMDA r3!, {r4-r7}
LDMIB r1, {r2, r3}
STMIB r1!, {r2,r3,r7, r10}
LDMDB r3, {r4-r7}
STMDB r3!, {r4-r7}
```

> ！表示寄存器读写完最后地址写入

这两条指令的执行结果相同吗?

```
LDMIA r0, {r1, r2, r3, r4}
LDMIA r0, {r4, r3, r1, r2}
```

运行下列程序，看结果

```
    area datarw , code , readonly, align=3
    entry
    ldr r0,=dmem
    LDMIA r0, {r1, r2, r3, r4}
    nop
    LDMIA r0, {r4, r3, r1, r2}
stop
        b stop
dmem    dcd 0x1234,0x2345,0x3456,0x5678
    end
```

内存中的地址顺序与寄存器序号对应，与列表中的次序无关!

根据图中内存和寄存器的初始值，分析下列指令的执行结果

```
R0=0x80020
R1=0x1111
R2=0x2222
R3=0x3333
```

Mem add	data
0x80030	0x009
0x8002c	0x008
0x80028	0x007
0x80024	0x006
0x80020	0x005
0x8001c	0x004
0x80018	0x003
0x80014	0x002
0x8000c	0x001
0x80008	0x000

```
LDMIA r0!,{r1-r3} r0=0x8002c,r1=0x005,r2=0x006
LDMDA r0, {r3,r2}
LDMIB r0!,{r1-r3} r0=0x80030,r1=0x006,r2=0x007
LDMDB r0, {r3,r2}
STMIA r0!,{r1-r3}
STMDA r0, {r3,r2}
STMIB r0!,{r1-r3}
STMDB r0, {r3,r2}
```

堆栈操作

- 堆栈的物理位置?
- sp?
- 属性： 基址，指针和限制

符号	功能	pop	=LDM	push	=STM
FA	Full In	LDMFA	LDMDA	STMFA	STMIB
FD	Full de	LDMFD	LDMIA	STMFD	STMDB
EA	Empty In	LDMEA	LDMDB	STMEA	STMIA
ED	Empty de	LDMED	LDMIB	STMED	STMDA

```
R1=0x005
R3=0x004
SP=0x80014

STMFD sp!,{r1, r3}
```

pre

地址	数据
0x80018	0x01
0x80014	0x02
0x80010	empty
0x8000c	empty

sp →

post

地址	数据
0x80018	0x01
0x80014	0x02
0x80010	0x004
0x8000c	0x005

sp →

设定sp时如何确定初始值?

R13,必须有效内存空间

标签命名

标签ARM 汇编程序中自定义符号（label, variables, constant等）有限制吗?

规则

- Can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names.
- Do not use numeric characters for the first character of symbol names, except in local labels (see Local labels).
- Symbol names are case-sensitive.
- All characters in the symbol name are significant.
- Symbol names must be unique within their scope.
- Symbols must not use built-in variable names or predefined symbol names

保留的（predefined）字

- register names
- r0-r15 and R0-R15
- a1-a4 (argument, result, or scratch registers, synonyms for r0 to r3)
- v1-v8 (variable registers, r4 to r11)
- sb and SB (static base, r9)
- sl and SL (stack limit, r10)
- fp and FP (frame pointer, r11)

- ip and IP (intra-procedure-call scratch register, r12)
- sp and SP (stack pointer, r13)
- lr and LR (link register, r14)
- pc and PC (program counter, r15).
- program status register names
  - cpsr and CPSR (current program status register)
  - spsr and SPSR (saved program status register).
- floating-point register names
  - f0-f7 and F0-F7 (FPA registers)
  - s0-s31 and S0-S31 (VFP single-precision registers)
  - d0-d15 and D0-D15 (VFP double-precision registers).
- coprocessor names
  - p0-p15 (coprocessors 0-15)
  - c0-c15 (coprocessor registers 0-15).

下列自定义符号哪些是正确的?

```
SPSR          错
Cpsr          对
MOV
Add
Codesize
moV
||ASSERT||    对    ?
8_123d        错
102srd        错
```

预计算

指令“ subs r10, r10, #84”中，“ 84” 在什么时候计算?

汇编器中计算，非处理器计算

双目计算

- 算术运算：+，-，*，/，MOD

  MOV r0, #(5*4)

  LDR R0, =start+3*n

- 移位运算：ROL, ROR, SHL, SHR

  A:ROL:B

  (3: ROR:4)

- 逻辑运算：AND, OR, EOR

  A: AND:B

  (0xcc55:OR:0x55cc)

- 关系运算：=，<, >, >=, <=, <>, /=

  A<>B

  (7<>7)

单目运算

运算符	用法	说明
?	? A	A所在行的代码长度
BASE	:BASE:A	（寄存器或程序相对表示）A的基地址寄存器
INDEX	:INDEX:A	（寄存器相对表示）A的偏移地址
+/-	+A, -A	正负号
NOT	:NOT:A	按位取反
LNOT	:LNOT:A	逻辑取反
DEF	:DEF:A	A是否定义，{TRUE/FALSE}
SB_OFFSET_19_12	:SB_OFFSET_19_12:label	label-sb的bits[19:12]
SB_OFFSET_11_0	:SB_OFFSET_11_0: label	label-sb的bits[11:0]

确定运行时寄存器中的数据（汇编后的结果）

Example1

```
    LDR r0, =? Mydata       ;10*4=40;
    (LDR r0, =mydata)?      ;没有？则读取的是label地址
    LDR r1, =? Mydata1      ;1*4=4

mydata    DCD 1,2,3,4,5,6,7,8,9,0
Mydata1   DCB ‘a’, ‘b’, ‘c’, ‘d’
```

Example 2

```
datastruc       SPACE   280
                MAP     0,r8 ; (MAP datastruc) ?
       consta               FIELD   4
       constb           FIELD   4
         x                      FIELD   8
         y              FIELD   8
      string                FIELD   256
              ……
                           LDR r3,=:BASE:y
                           LDR r4,=:INDEX:y
```

逻辑变量

- LAND（处理器指令？）
- LOR
- LEOR

字符串变量

双目运算运算符

运算符	用法	说明
LEFT	A:LEFT:B	返回字符串A中左起的B个字符
RIGHT	A:RIGHT:B	返回字符串A中右起的B个字符
CC	A:CC:B	A和B字符串相连接，A在左边

单目运算运算符

运算符	用法	说明
LEN	:LEN:A	返回字符串A的长度
CHR	:CHR:A	反回字符A的ASCII码值，A是单个字符
STR	:STR:A	返回数值或逻辑A的字符串

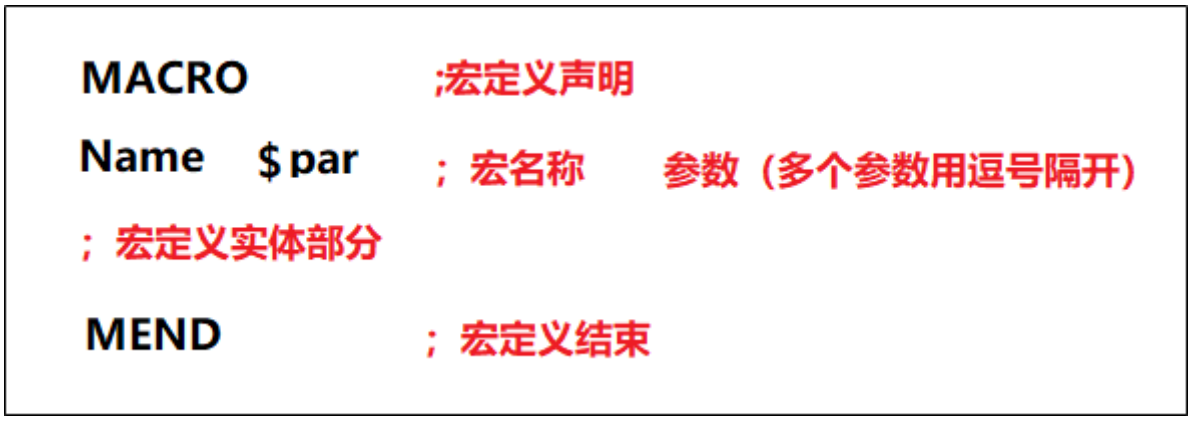
Example

- ： LEN：”I am an excellent student!”
- : CHR: ‘A’
- What is the returned result?

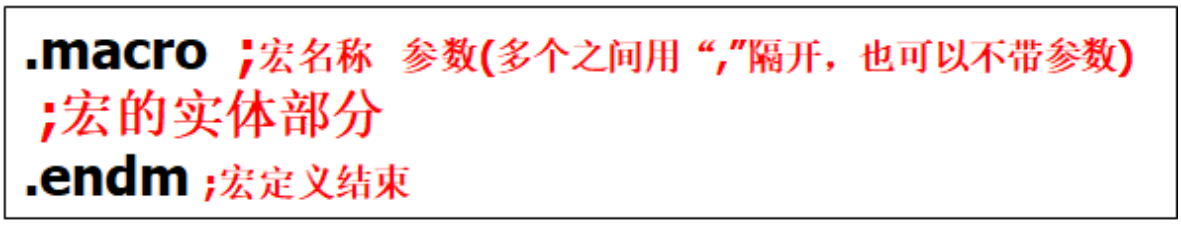
宏定义

封装汇编指令模块，作为用户定义的功能单元

ARM格式



GNU格式



```
                MACRO
$LOOP           MacSum $RD
                MOV v8, #0

$LOOP.1         ADD v8, $RD, v8
                SUBS $RD, $RD, #1
                BNE $LOOP.1
                MOV $RD, v8
                MEND
```

1. $LOOP 是参数吗？宏标签
2. 如果宏定义中去掉$LOOP, 用确定lable “loop”替代“$LOOP.1”, 程序运行的结果会有变化吗？
~~多次引用，标签重复~~
3. $LOOP.1的作用？局部标签，宏定义体内有效

```
                AREA mycode, CODE
                ENTRY
                CODE32
START
                MOV R0, #100
add100          MacSum R0    ; 宏标签，调用宏
STOP
                b  STOP
                END
```

> 不建议在宏内使用固定寄存器，例如v8。
>
> 一般定义变量使用

数值计算

64位整数加（减）法

- 实现64位数0x2200330044和0x9876543210的加（减）法，相加后的结果保存在起始地址为0x20000000的存储空间里。设数据存储采用小端格式。
- 如何用32位加法器实现64位加法运算?

```
LDR R0, =0x00000022 ；加载第一个数的高32位放到R0中
LDR R1, =0x00330044  ；加载第一个数的低32位放到R1中
LDR R2, =0x00000098
LDR R3, =0x76543210
LDR R6, =0x20000000  ；把存储结果的内存地址放到R6中
ADDS R4, R1, R3（ SUBS R4, R1, R3 ） ;低32位加（减），R4存储低32位
ADC R5, R0, R2 （ SUBC R5, R0, R2 ） ;高32位加（减），R5存储高32位
STMIA R6!,{R4,R5}  ；将R4，R5的数据存储到R6指向的地址上，
                   ；R6值更新
```

- 一般寄存器大的放高位，这里反了
- 两数相减大于0，借位为1
- 借位：A-B-！C

64位乘法

2个无符号64位数a 和 b相乘，得到64位结果

- 函数参数的传递方式
  - 寄存器 r0-r3 （数目<=4）
  - 堆栈 （>4）
- 返回

  r0, r1
- 算法:

  `H1L1 *H2L2=L1*L2 +(L1*H2)<<32+ (H1*L2)<<32+(H1*H2)<<64`

```
a_0     RN  0       ;a low
a_1 RN  1   ;a high
b_0     RN  2   ;b low
b_1 RN  3   ;b high
c_0     RN  4   ;c low low
c_1 RN  5   ;c low high
c_2     RN  12  ;c high low
c_3 RN  14  ;c high high
Mul_64 to 64
        stmfd   sp!, {r4,r5,lr}
        umull   c_0,c_1, a_0, b_0   ;low *low
        mla c_1, a_0,b_1,c_1    ;low *high
        mla     c_1, a_1, b_0, c_1  ;high* low

        mov     r0, c_0
        mov r1, c_1         ;return
```

2个有符号64位数a 和 b相乘，得到128位结果

- 算法:

`(S)H1L1 *(S)H2L2=L1*L2 +(L1*(S)H2)<<32+((S)H1*L2)<<32+((S)H1*(S)H2)<<64`

- 无符号乘法: UMULL, UMLAL
- 有符号乘法: SMULL, SMLAL
- 有符号* 无符号?

64位乘法

设 A为无符号数， B为有符号数，用SMULL或SMLAL计算，则结果如何?

（1）如果A[31]=0，则`(S)A * (S)B= A*(S)B`

（2）如果A[31]=1, 则`(S)A * (S)B= (A-232)*(S)B`

```
                                    `A*(S)B = (S)A * (S)B+(1<<32) *(S)B`
```

宏定义 USMLAL（无符号*有符号）

```
MACRO
    USMLAL  $cl, $ch, $a, $b
    ;signed $ch, $cl +=unsigned $a *signed $b
    SMULL       $cl, $ch, $a, $b
    ;c= (signed)a *(signed)b
    TST     $a, #1<<31
    ;if(signed)a<0
    ADDNE       $ch, $ch, $b
    ;c+=(b<<32)
    MEND
```

```
    smul_64_to_128
    stmfd   sp!, {r4, r5, lr}
    umull   c_0, c_1, a_0, b_0  ;low*low
    mov     c_2, #0
    usmlal  c_1, c_2, a_0, b_1  ;low*high
    mov c_3, #0
    usmlal  c_1, c_3, b_0, a_1  ;high*low
    mov a_0, c_2, ASR #31
    adds    c_2, c_2, c_3
    adc c_3, a_0, c_3, ASR #31
    smlal   c_2, c_3, a_1, b_1  ;high*high
    mov r0, c_0
    mov r1, c_1
    mov r2, c_2
    mov r3, c_3
    ldmfd   sp!,{r4, r5, pc}
```

宏定义 USMLAL（无符号*有符号）

```
    MACRO
    USMLAL  $cl, $ch, $a, $b
    ;signed $ch, $cl +=unsigned $a *signed $b
    SMULL       $cl, $ch, $a, $b
    ;c= (signed)a *(signed)b
    TST     $a, #1<<31
    ;if(signed)a<0
    ADDNE       $ch, $ch, $b
    ;c+=(b<<32)
    MEND
```

浮点计算

- IEEE754浮点格式

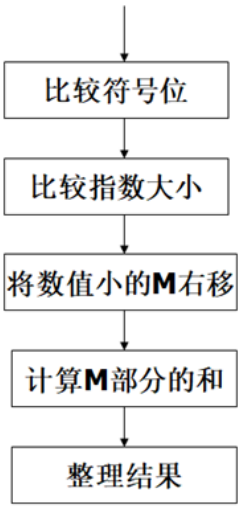
符号 S	指数 E	尾数 M

- 单精度
  E=8位， M=23位
- 扩展单精度
  E>=11位， M=31位
- 双精度
  E>=11位， M=52位
- 扩展双精度
  E>=15位，M>63位

- 浮点数与实数的转换
  - 单精度转换公式
  - `V=(-1)^S*2^(E（值）-127)*(1+M) （指数位!=0)`
  - `V=(-1)^S*2^(1-127)*M （指数位=0)`

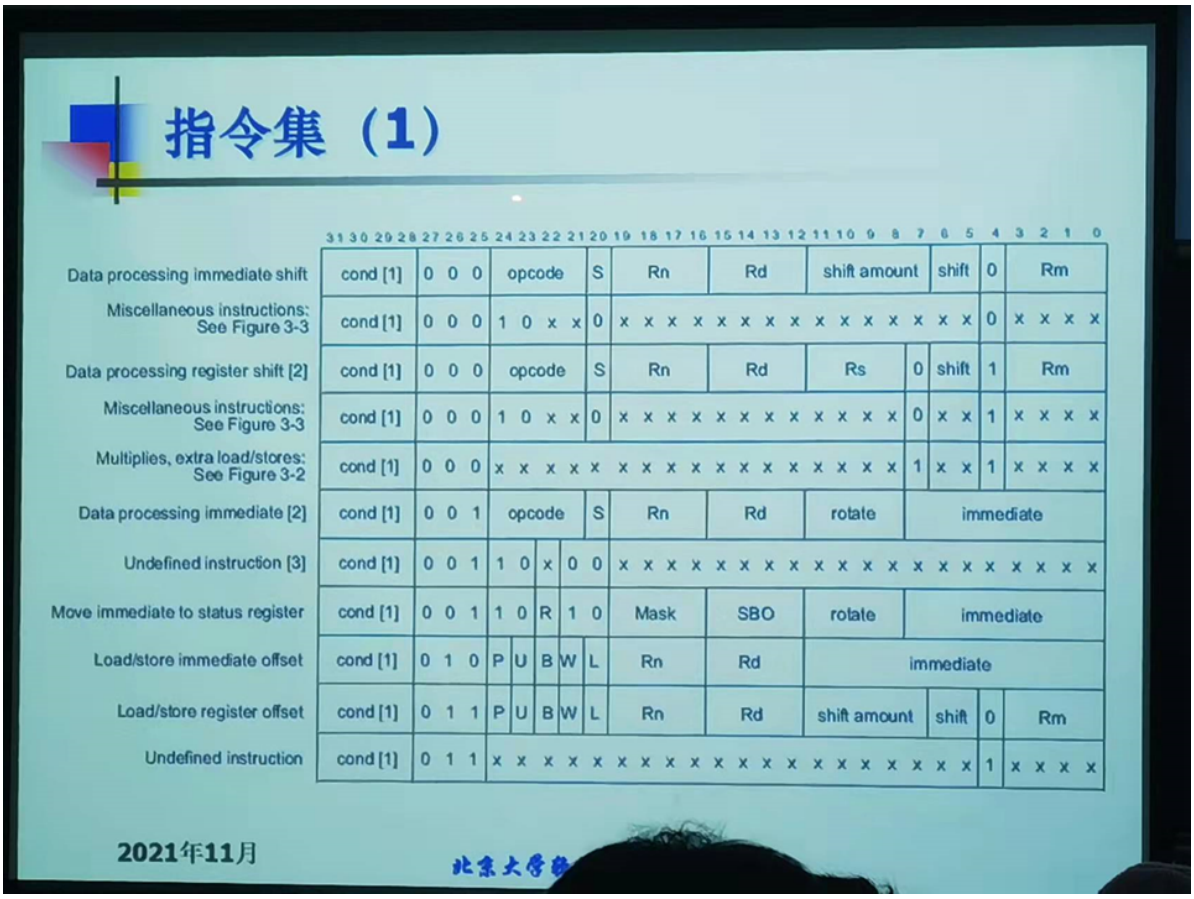
0x00280000（浮点数？）
转换成二进制
00000000010100000000000000000000
我们将其分段：
符号位 指数部分（8位） 尾数部分
0 00000000 01010000000000000000000
符号位=0;
指数部分=0;
尾数部分M:
0.01010000000000000000000=0.3125
该浮点数的十进制为:
(-1)^0*2^(-126)*0.3125
=3.673419846319648462402301678819 5e-39

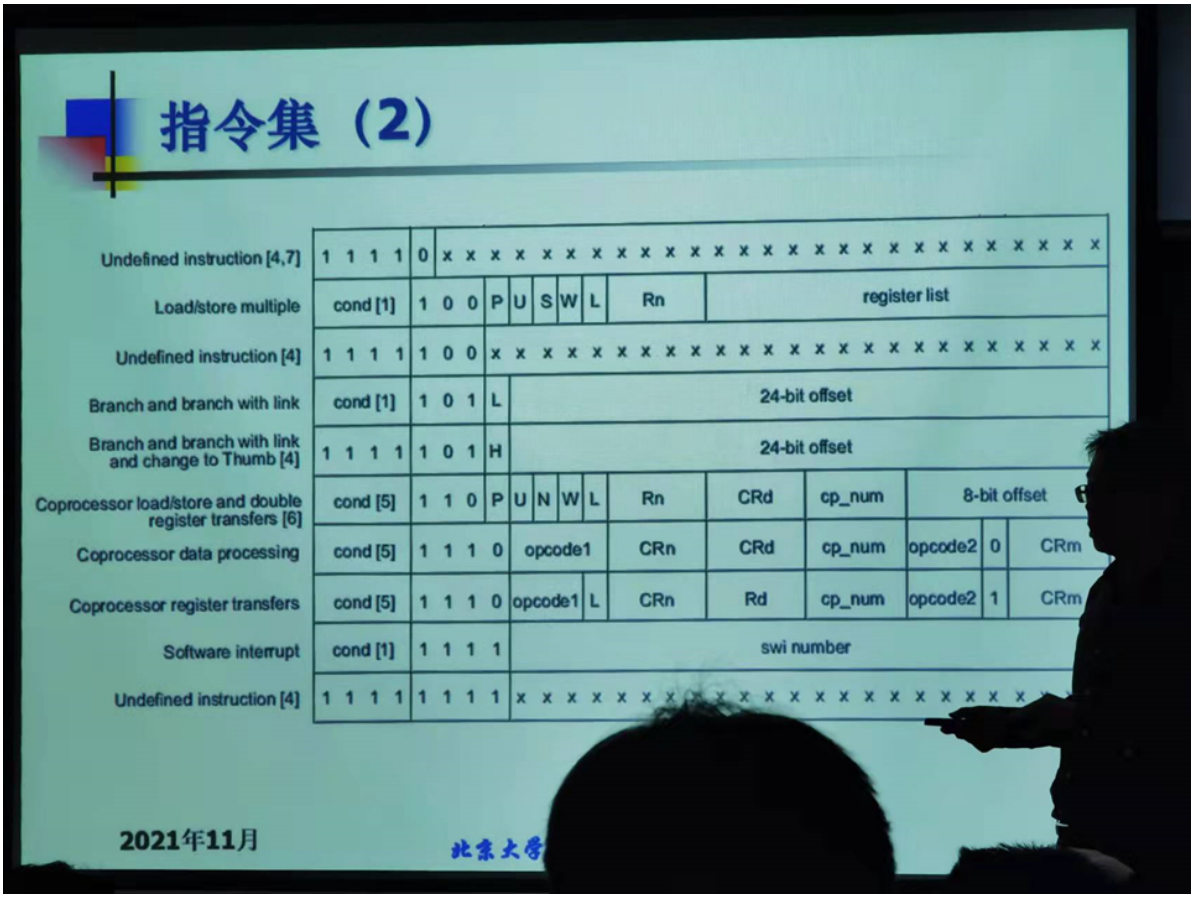
浮点加法




```
FADD32
        SUBS    R2,R0,R1        ;f1-f2
        SUBCC   R0,R0,R2        ;if(f1<f2){ r0 =f2;
        ADDCC   R1,R1,R2        ;   r1=f1}
        MOV     R2,R0,LSR #23   ;f2>>23  ->r2
        SUB     R3,R2,R1,LSR #23 ; difference of E bits
        MOV     R12,#0x80000000
        ORR     R0,R12,R0,LSL #8
        ORR     R1,R12,R1,LSL #8
        ADD     R12,R0,R1,LSR R3  ;( f1>>r3)+f2, M
        SUB     R2,R2,#0x00000001
        MOV     R0,R12,LSR #8
        ADC     R0,R0,R2,LSL #23 ;E
        BX      R14
```

小结





分析下列程序的功能

```
    AREA    Block, CODE, READONLY    ; name this block of code
num     EQU     20                      ; set number of words to be copied
    ENTRY                           ; mark the first instruction to call
start
    LDR     r0, =src                ; r0 = pointer to source block
    LDR     r1, =dst                ; r1 = pointer to destination block
    MOV     r2, #num                ; r2 = number of words to copy
    MOV     sp, #0x400              ; Set up stack pointer (r13)
blockcopy
    MOVS    r3,r2, LSR #3     ; Number of eight word multiples
    BEQ     copywords               ; Less than eight words to move?
    STMFD   sp!, {r4-r11}           ; Save some working registers

octcopy  LDMIA   r0!, {r4-r11}            ; Load 8 words from the source
    STMIA   r1!, {r4-r11}           ; and put them at the destination
    SUBS    r3, r3, #1              ; Decrement the counter
    BNE     octcopy                 ; ... copy more
    LDMFD   sp!, {r4-r11}      ; Don't need these now - restore
                                            ; originals
copywords
    ANDS    r2, r2, #7              ; Number of odd words to copy
    BEQ     stop                    ; No words left to copy?

wordcopy    LDR     r3, [r0], #4     ; Load a word from the source  and
       STR     r3, [r1], #4             ; store it to the destination
       SUBS    r2, r2, #1               ; Decrement the counter
       BNE     wordcopy                 ; ... copy more
stop            MOV     r0, #0x18               ;
       LDR     r1, =0x20026          ; ADP_Stopped_ApplicationExit
       SWI     0x123456                 ; ARM semihosting SWI
```

```
      AREA    BlockData, DATA, READWRITE
src               DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst               DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                END
```

C程序

```
int main(void)
{   int i, j;
    unsigned char Inimage[480][640];
    unsigned char Tmpimg[120][160];
    for(i=0;i<120;i++)
      for(j=0;j<160;j++)
                Tmpimg[i][j]=Inimage[i*4+2][j*4+2];
    memcpy(Inimage,Tmpimg,160*120);
    return 1;
}
```

汇编

```
Imageheight equ 120
Imagewidth  equ 160
InImagewidth    equ        640
        area    reset , code , readonly
        entry
        mov r6, #0
Framecircle
    mov r5, #0
Linecircle
;i->r5; j->r6; Tmpimg-> r8; Inimage->r9;
    ldr r8, =Tmpimg
    ldr r9, =Inimage
;Tmpimg[i][j]->r10;
    mov r0, #Imagewidth
    mul r10, r6, r0   ;i*Imagewidth
    add r10, r10, r5   ;i*Imagewidth+j
    add r10,r8

;Inimage[i*4+2][j*4+2]-> r11
    mov r0, #InImagewidth
    mov r1, r6, lsl #2 ; i*4
    add r1, r1, #2  ;i*4+2
    mul r11, r1, r0 ;(i*4+2)*InImagewidth
    mov r1, r5, lsl #2 ;j*4
    add r1, r1, #2    ;j*4+2
    add r11, r11, r1; (i*4+2)*InImagewidth+4*j+2
    add r11, r9
    ldrb    r0, [r11] ;read Inimage
    strb    r0, [r10] ;write Tmpimg
;circle control
        add r5, r5,#1
        cmp r5, #Imagewidth
      bmi  Linecircle
        add r6, r6, #1
        cmp r6, #Imageheight
```

```
        bmi Framecircle

; memcpy
    ldr     r10,    =160*120
memcpy
    ldmia   r8! ,   {r0-r7}
    stmia   r9! ,   {r0-r7}
    subs    r10,r10, #8*4
    bgt memcpy
    ;end

    area  imscale, data, readwrite

Tmpimg  space       160*120
Inimage space       640*480

    end
```

ARM程序基础

参考内容：

- ARM Architecture Reference Manual（V5）, Second Edition
- Andrew N. Sloss 等，沈建华译，ARM嵌入式系统开发-软件设计与优化，北京航空航天大学出版社，2005年5月

函数调用

源文件内

```
    AREA  function, CODE, READONLY
    ENTRY
start
        MOV     r0, #10
        MOV     r1, #3      ;参数
        BL      doadd           ; call

    ……

doadd                               ;函数
        ADD     r0, r0, r1          ;结果
        MOV     pc, lr              ;返回

        END
```

不同源文件

- File1

```
AREA Main, CODE, READONLY
    IMPORT   Testprg
    ENTRY
main        ……
        BL Testprg
        ……
        END
```

- File2

```
    AREA  Test, CODE, READONLY
    EXPORT   Testprg
   Testprg
        ADD  r0, r1, r0
        MOV pc,lr
        END
```

> 汇编import的函数文件需要加export

ATPCS（ARM-Thumb Procedure Call Standard）

- 寄存器约定
  - R0-r3 (a1-a4) 传递参数
  - R4-r11 (v1-v8)保存局部变量 （r4-r7 for Thumb)
  - R12 (ip) 过程间自定义数据交换
  - R13 (sp) 数据堆栈指针
  - R14 (lr) 保存子程序的返回地址
  - R15（pc)
- ATPCS规定数据栈为FD（满递减）类型，并且对数据栈的操作是8字节对齐。

> 向堆栈写数据，指针减小；数据栈8字节对齐

- 参数传输规则
  - 小于等于4， r0-r3,依次
  - 大于4，存入数据栈，最后一个数据先入栈

`function( char a, int *b, int c, short d, long f)`

> - 参数次序与所使用的寄存器?
>
> - 寄存器的分配与类型（浮点除外）有关?

- 浮点参数传递(顺序传递)
  - FPA (Floating Point Arithmetic)：f0-f7 (s0-s7, d0-d7,e0-e7)
  - VFP (d0-d15, s0-s31)

`float function(int v1, float v2, char *v3, int v4, float v5, double v6, short v7);`

- v1, v3, v4, v7 => r0, r1, r2, r3
- v2, v5 => s0, s1
- v6 => d0

> 继承协处理器
>
> 注意参数传递：整形（字符），float，double

- 参数返回规则
  - 结果为32bit整数时，通过r0传递
  - 结果为64bit整数时，通过r0和R1传递
  - 结果为浮点数时，通过浮点寄存器返回(f0,d0)
  - 更多的数据通过内存返回

C调用汇编

```
#include <stdio.h>
extern void strcopy(char *d, const char *s); //ASM
int main()
{       const char *srcstr = "First string - source ";
        char dststr[] = "Second string - destination ";
        strcopy(dststr,srcstr);
        printf("After copying:\n");
        printf("  %s\n  %s\n",srcstr,dststr);
        return (0);
}
```

```
        AREA    SCopy, CODE, READONLY
            EXPORT strcopy
strcopy
        LDRB r2, [r1],#1  ; Load byte and update address.
        STRB r2, [r0],#1  ;
        CMP r2, #0        ; Check for zero terminator.
            BNE strcopy       ; Keep going if not.
            MOV pc,lr         ; Return.
            END
```

参数传递过程?

汇编调用C

```
    AREA f, CODE, READONLY
        IMPORT  g
    ENTRY
        LDR r4, =data
        LDR  r0, [r4], #4
    LDR  r1, [r4], #4
    LDR  r2, [r4], #4
    LDR  r3, [r4], #4
    LDR  r5, [r4]
    STR r5, [sp, #-4]!
    BL  g
    ADD sp, sp, #4
    END
```

```
int g(int a, int b, int c, int d, int e)
    {  return a + b + c + d + e;}
```

> STR r5, [sp, #-4]! //栈初始满的，需要减去4字节腾出空间

C++调用汇编

```
struct S {
        S(int s) {i=s; }
        int i;
    }

extern "C" void asmfunc(S *);

int f()
{
        S s(2);
        asmfunc(&s);
    return s.i * 3;
}
```

```
AREA   Asm, CODE
        EXPORT asmfunc
asmfunc          ; the definition of the Asm function
        LDR r1, [r0]
    ADD r1, r1, #5
        STR r1, [r0]
        MOV pc, lr
        END
```

汇编调用C++

```
    AREA  Asm, CODE, READONLY
        IMPORT cppfunc
    ENTRY
    MOV    r0,#2
        STR    r0,[sp,#-4]!
        MOV    r0,sp
        BL     cppfunc
    LDR    r0, [sp], #4
        ADD    r0, r0, r0,LSL #1
    END
```

```
extern "C" void cppfunc(S * p) {
 // Definition of the C++ function to be called from ASM.
    p->i += 5;}
```

嵌入汇编

```
void my_strcpy(const char *src, char *dst)
{   int ch;
        __asm{
            loop:
            LDRB    ch, [src], #1
                    STRB    ch, [dst], #1
            CMP     ch, #0
                    BNE     loop
        }
}
```

- 嵌入 vs 汇编程序
  - 不支持 LDR Rn,= XXX 和 ADR, ADRL 伪指令
  - 不支持 BX
  - 用"&"替代 "0x" 表示16位数据

> [src], [dst] ?
>
> ch ?
>
> 循环结束条件?
>
> c中的变量?

- 小心使用寄存器，尽量不用 R0-R3，lr，ip 和CPSR 中的NZCV 标志位；不用r0-r3，用v1-v8

```
__asm
{
    MOV R0, x
    ADD y, R0,  y
}
```

- 不要使用寄存器替代变量

```
int bad_f(int x) // x in R0
{          ……
            __asm {
                ADD R0, R0, #1 ; wrongly asserts that x is still in R0
            }
          return x; // x in R0
}
```

- 无需保存和恢复寄存器

```
int f(int x){
           __asm{
                STMFD sp!, {R0} ; save R0 - illegal: read before write
                ADD R0, x, 1
                EOR x, R0, x
                LDMFD sp!, {R0} ;  restore R0 - not needed.
         }
         return x;
}
```

- 汇编语言用","作为操作数分隔符。如果有C 或C++表达式作为操作数，必须用"( )"将其归约为一个汇编操作数。

```
__asm {ADD x, y, (f()+z)}
```

异常处理

异常向量表

向量地址	异常中断类型	异常中断模式	优先级
0x0	Reset	特权（SVC)	1
0x4	Undefined Instruction	未定义指令中止模式	6
0x08	SWI	特权模式	6
0x0c	Prefetch Abort	中止模式	5
0x10	Data Abort	中止模式	2
0x14	Reserved	未使用	未使用
0x18	IRQ中断	IRQ模式	4
0x1c	FIQ快速中断	FIQ模式	3

确定优先级的作用?

- 异常向量表特点
  - 异常事件与异常处理程序之间的对应关系；
  - 向量表大小32个字节，每个异常向量占据4个字节；
  - 每个字存放PC赋值的语句，或跳转指令；
  - 通常存放在存储器地址的低端（或利用寄存器设置）
- 向量表-LDR间接寻址

```
Vectors         LDR     PC, Reset_Addr          ; 0x00
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP                                 ; Reserved Vector
                LDR     PC, IRQ_Addr
                LDR     PC, FIQ_Addr
        ……                                  ; IRQ_Entry, 9条指令
        Reset_Addr          DCD     Reset_Handler
        Undef_Addr          DCD     Undef_Handler
        SWI_Addr            DCD     SWI_Handler
        PAbt_Addr           DCD     PAbt_Handler
        DAbt_Addr           DCD     DAbt_Handler
                    DCD     0                           ; Reserved Address
        IRQ_Addr            DCD     IRQ_Handler
        FIQ_Addr            DCD     FIQ_Handler
```

> LDR PC, Reset_Addr 会相对寻址 Ldr [pc, #Reset_Addr to pc]
>
> 偏移不能超过4k

```
LDR    PC, Reset_Addr  =>
       PC      <-  [pc，  #offset of Reset_Addr to pc];
```

```
Reset_Addr          DCD    Reset_Handler =>
       [Reset_Addr]=Reset_Handler;
```

```
 PC      <-  Reset_Handler;   Jump to Reset_Handler
```

向量表二进制格式

```
0x00000000   E59FF03C   LDR        PC,[PC,#0x003C]
0x00000004   E59FF03C   LDR        PC,[PC,#0x003C]
0x00000008   E59FF03C   LDR        PC,[PC,#0x003C]
0x0000000C   E59FF03C   LDR        PC,[PC,#0x003C]
0x00000010   E59FF03C   LDR        PC,[PC,#0x003C]
0x00000014   E1A00000   NOP
0x00000018   E59FF03C   LDR        PC,[PC,#0x003C]
0x0000001C   E59FF03C   LDR        PC,[PC,#0x003C]
```

- 向量表-B跳转

```
Vectors
            B      Reset_Handler  ;0x00
            B      Undef_Handler
            B      SWI_Handler
            B      PAbt_Handler
            B      DAbt_Handler
            NOP                          ; Reserved Address
            B      IRQ_Handler
            B      FIQ_Handler
```

- B
  - 简单
  - 跳转范围受限 （±32MB)
- 使用LDR指令
  - 跳转范围不受限
  - 额外空间存地址
  - 地址存放在4KB范围以内

在这里，”B” 能否改成”BL”？

- 向量表-伪指令LDR

```
Vectors
             LDR PC, = Reset_Handler  ;0x00
             LDR PC, = Undef_Handler
            LDR PC, = SWI_Handler
             LDR PC, = PAbt_Handler
             LDR PC, = DAbt_Handler
            NOP                           ; Reserved Address
            LDR PC, = IRQ_Handler
            LDR PC, = FIQ_Handler
                    ……                                ; IRQ_Entry, 9条指令
        ltorg
        ;Reset_Addr         DCD     Reset_Handler
        ;Undef_Addr         DCD     Undef_Handler
```

```
        ;SWI_Addr           DCD     SWI_Handler
        ;PAbt_Addr          DCD     PAbt_Handler
        ;DAbt_Addr          DCD     DAbt_Handler
```

- 向量表/异常服务程序

```
        import  ISR_IRQ_Handler
Vectors
            B     Reset_Handler
            B     Undef_Handler
            B     SWI_Handler
            B     PAbt_Handler
            B     DAbt_Handler
            NOP
            B     ISR_IRQ_Handler
            B     FIQ_Handler
```

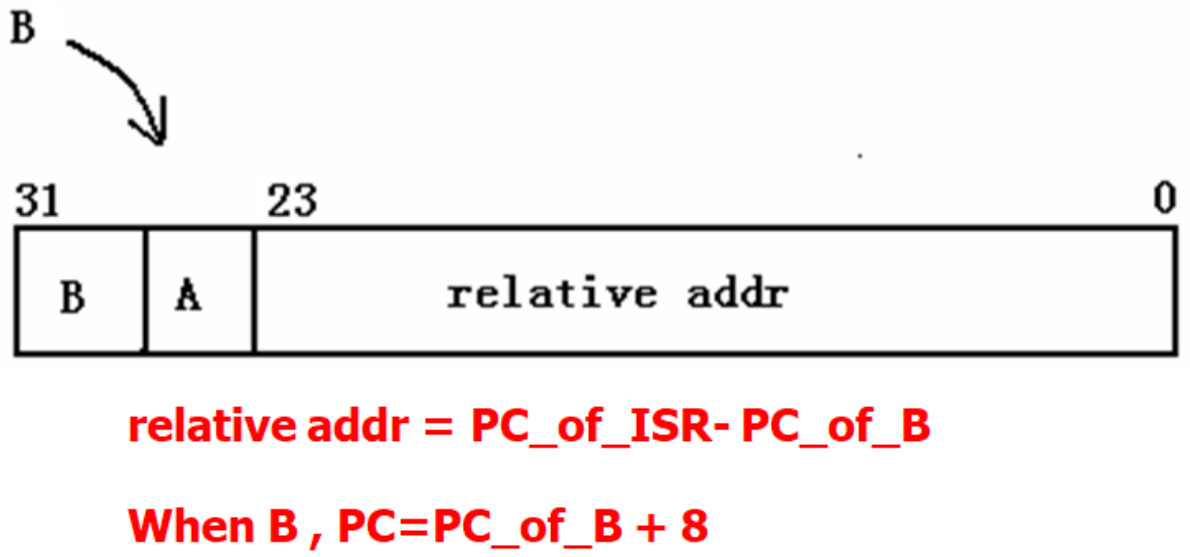
> 中断服务程序一般没有参数不需要传回值，或通过内存传递

能否在程序运行时注册（安装）异常服务程序?

注册异常服务程序

能直接写入文本格式程序源码?

程序运行时，将指令 “B ISR_IRQ_Handler ”写入向量表中?



> 构建二进制指令，取高26位，偏移地址：中断向量-中断服务程序入口地址

- 步骤

  (1) 读取中断处理程序的地址addr1

  (2) 将addr1减去该中断对应的中断向量的地址vector1

  (3) addr=Addr1-vector1-8 (允许指令预取)

  (4) addr LSR #2

  (5) if(addr and 0xff000000 ==0)

  (6) addr or 0xea00 0000

（7）结果写回中断向量表

- 安装函数

```
unsigned Install_Handler (unsigned *handlerloc, unsigned *vector)
{
    unsigned vec, oldvec;
    vec = ((unsigned)handlerloc - (unsigned)vector - 0x8)>>2;
    if ((vec & 0xFF000000) != 0)
     { return 0;}
    vec = 0xEa000000 | vec;
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
```

```
unsigned *irqvec = (unsigned *)0x18;
static unsigned pIRQ_Handler = (unsigned)ISR_IRQ_handler
Install_Handler (&pIRQ_Handler, irqvec);
```

中断向量表的基地址?

上述例程适用条件？跳转地址和向量表偏移地址不能超过正负32M

如果不能确定异常服务程序的地址范围，如何处理?

如果向量表采用"LDR"语句，需要修改"DCD" 后的地址, 如何做?

- 获取服务程序的入口地址;
- 确定异常向量对应的"DCD" 分配的存储地址;
- 将服务程序入口地址写入对应的存储单元;

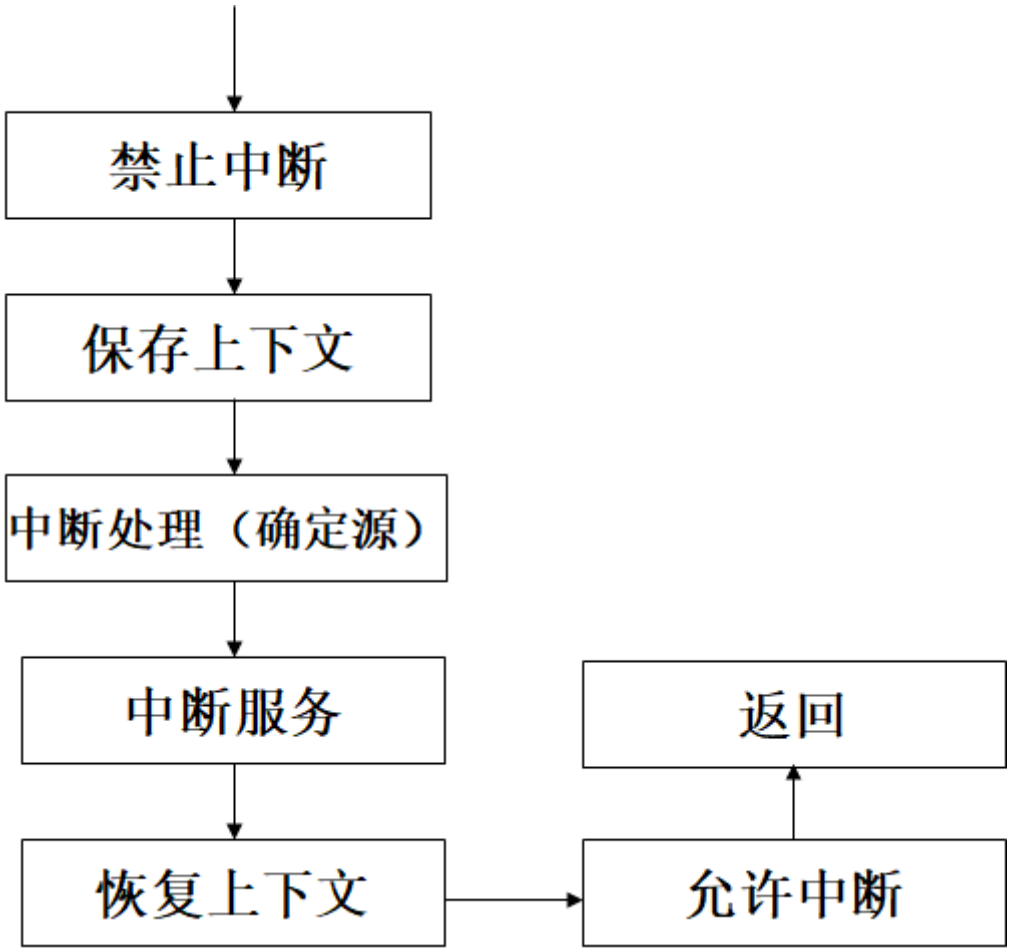
中断向量扩展

实际处理器通常有多个外部中断 (IRQ, 外设， IO)，这些中断与其对应的服务程序如何关联?

```
INTOFFSET           EQU    0X4A000014  ;Address of Interrupt
                            ;offset Register
IntVTAddress        EQU     0x33FFFF20

;Interrupt Vector Table Address
HandleEINT0         EQU    IntVTAddress
HandleEINT1         EQU    IntVTAddress +4
……
HandleTIMER0        EQU    IntVTAddress +4*10
HandleTIMER1        EQU    IntVTAddress +4*11
……
HandleUART1         EQU    IntVTAddress +4*23
……
HandleUART0         EQU    IntVTAddress +4*28
HandleADC           EQU    IntVTAddress +4*31
```

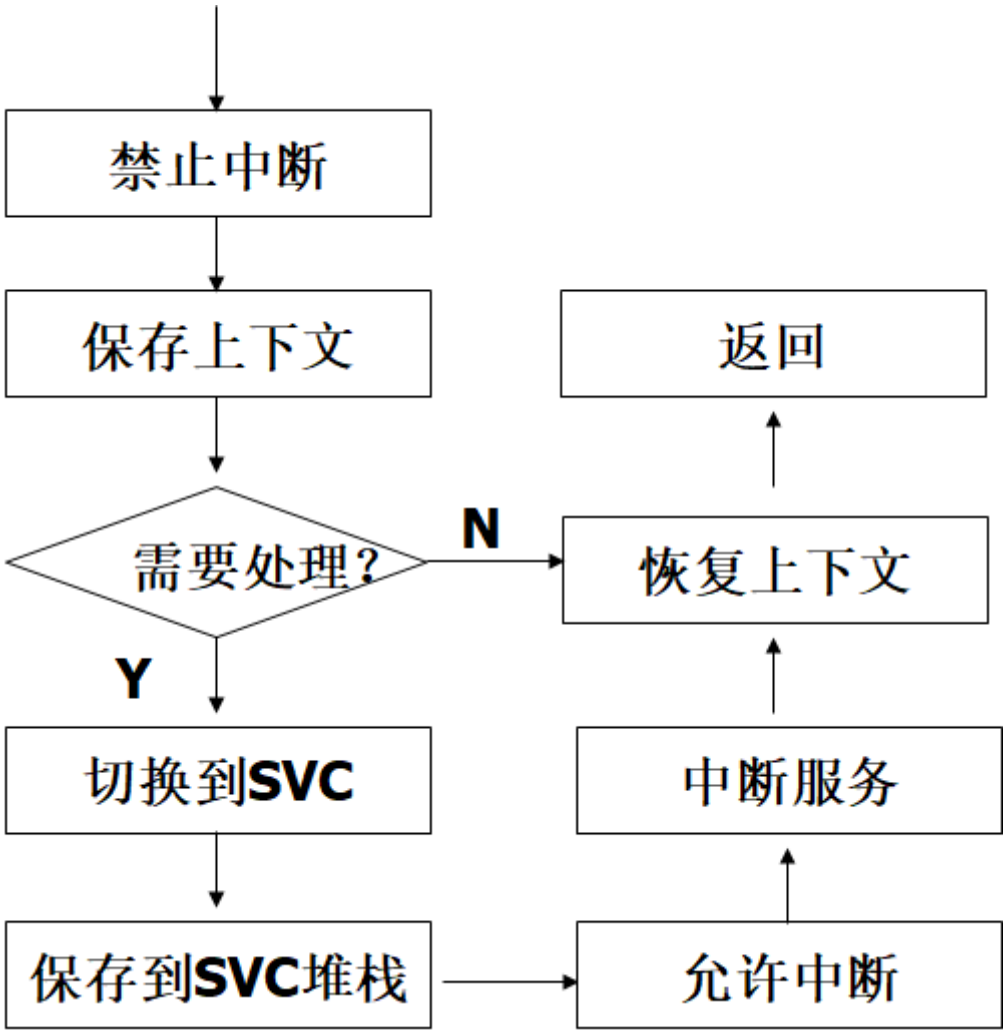
```
IRQ_Entry
                        sub sp,sp,#4       ;reserved for PC
                    stmfd   sp!,{r8-r9}
                    ldr r9,=INTOFFSET  ;中断序号
                    ldr r9,[r9]
                    ldr r8,=HandleEINT0 ;中断扩展表首地址
                    add r8,r8,r9,lsl #2 ; ?
                    ldr r8,[r8]   ;Eintx_ Entry Add ->r8
                    str r8,[sp,#8] ;r8->SP(-#4)

                    ldmfd   sp!,{r8-r9,pc};Eintx_ Entry Add ->pc
```

非嵌套中断




```
interrupt_handler
    sub r14, r14, #4 ; ?
    stmfd   sp!, {r0-r3, r12, r14}
    ldr r1, =IRQStatus
    ldr r0,[r1]
    tst r0, #0x0080  ;测试中断源
    blne    my_isr1
    tst r0, #0x0001 ; 测试中断源
    blne    my_isr2
    ldmfd   sp!, {r0-r3, r12, r14}^
    ldr pc, lr
```

嵌套中断

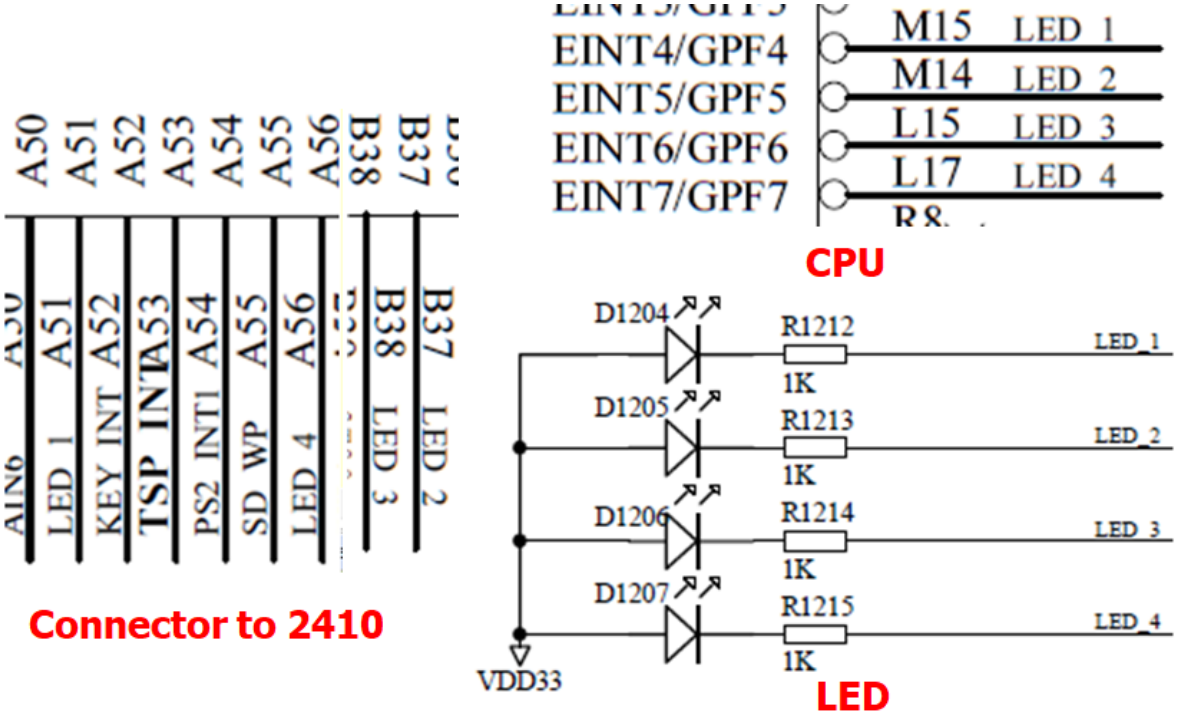


外设访问

如何控制实验板上LED显示?

LED连接

Mainboard



- LED连接
  - LED1 -> GPF4
  - LED2 -> GPF5
  - LED3 -> GPF6
  - LED4 -> GPF7
- LED 控制
  - n0 -> on
  - n1 -> off
- GPF控制
  - **Registers**

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configure the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undefined
GPFUP	0x56000058	R/W	Pull-up disable register for port F	0x0

GPFCON	Bit	Description
GPF7	[15:14]	00 = Input 01 = Output 10 = EINT7 11 = Reserved
GPF6	[13:12]	00 = Input 01 = Output 10 = EINT6 11 = Reserved
GPF5	[11:10]	00 = Input 01 = Output 10 = EINT5 11 = Reserved
GPF4	[9:8]	00 = Input 01 = Output 10 = EINT4 11 = Reserved

GPFDAT	Bit
GPF[7:0]	[7:0]

GPFUP	Bit	Description
GPF[7:0]	[7:0]	0: enabled 1: disabled

- The value of Registers
  - GPFCON (4-7 Output)： 0b 0101 0101 XXXX XXXX
  - GPFUP (4-7 No concern): 0b XXXX XXXX

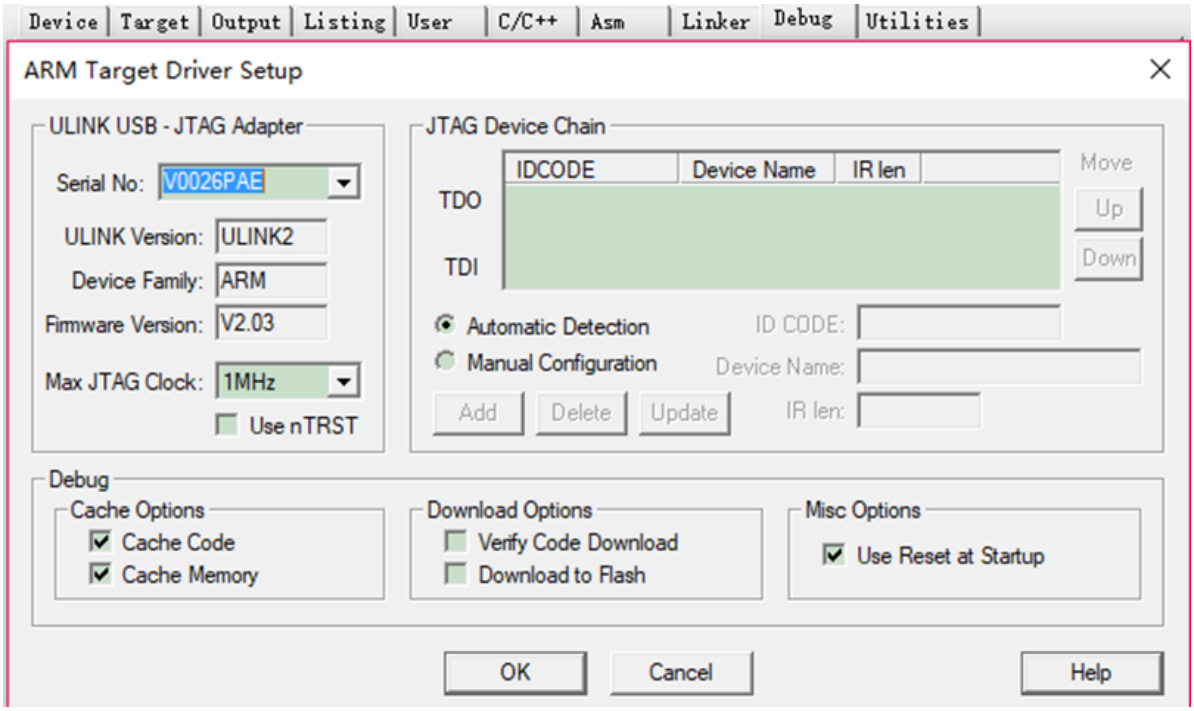
- GPFDAT (4-7 on): 0b 0000 XXXX
- GPFDAT (4-7 off): 0b 1111 XXXX

C Code

```
int *rGPFCON = (int *) 0x56000050;
int *rGPFDAT = (int *) 0x56000054;

void main( void)
{
    *rGPFCON=0x5500;
    while(1)
    {
        *rGPFDAT = 0x00;
        delay(1000);
        *rGPFDAT = 0xf0;
        delay(1000);
    }
}
```

如何连接到目标板?

Emulator



如何在目板RAM中运行程序?

Run in RAM

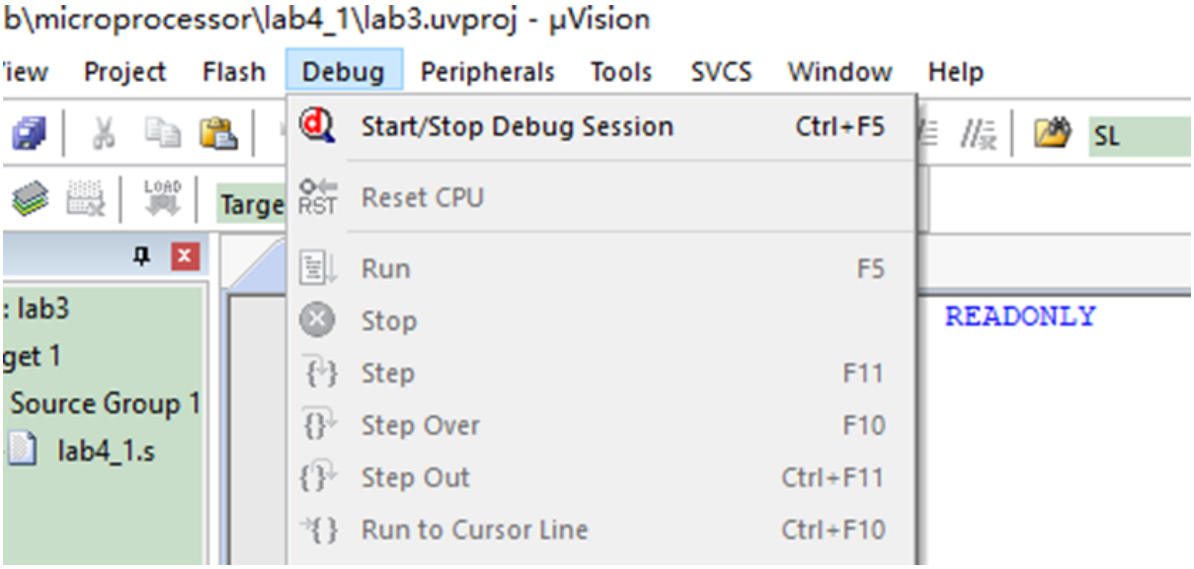
- Linker Configure
  - RO Base=0x30000000
  - RW Base=0x30100000
- DRAM 初始化
  - Debug Init

```
//MEMMAP=0x01;
map 0x48000000, 0x60000000 read write ;
_WDWORD(0x53000000,0x00000000);         //watchdog Timer Control register
_WDWORD(0x4a000008,0xffffffff);                 //Interrupt Mask Control
_WDWORD(0x4a00001c,0x000007ff);         //Interrupt sub mask
_WDWORD(0x4c000014,0x03);                       //Clock Divider Control Register
_WDWORD(0x4c000004,0x5c042);                //MPLL Configuration Register
_WDWORD(0x48000000,0x22111110);         //Bus Width and Wait Status Ctrl
_WDWORD(0x48000004,0x00000700);         //Bank 0 Control Register
_WDWORD(0x48000008,0x00000700);         //Bank 1 Control Register
_WDWORD(0x4800000c,0x00000700);         //Bank 2 Control Register
_WDWORD(0x48000010,0x00000700);         //Bank 3 Control Register
_WDWORD(0x48000014,0x00000700);         //Bank 4 Control Register
_WDWORD(0x48000018,0x00000700);         //Bank 5 Control Register
_WDWORD(0x4800001c,0x00018005);         //Bank 6 Control Register
_WDWORD(0x48000020,0x00000700);         //Bank 7 Control Register
_WDWORD(0x48000024,0x008e0459);         //SDRAM Refresh Control Register
_WDWORD(0x48000028,0x000000b2);         //Flexible Bank Size Register
_WDWORD(0x4800002c,0x00000030);         //Bank 6 Mode Register
_WDWORD(0x48000030,0x00000030);         //Bank 7 Mode Register
_WDWORD(0x56000014,0x01);                   //Port B Data
_WDWORD(0x56000020,0xaaaa55aa);         //Port C Control
_WDWORD(0x56000028,0x0ffff);                    //Pull-up Control C
_WDWORD(0x56000024,0x00000000);         //Port C Data
_WDWORD(0x56000070,0x00280000);         //Port H Control
_WDWORD(0x56000078,0x00000000);         //Pull-up Control H
```

```
//pc=0x30000000
```

需要在代码中加入初始化，上电后才能正常访问RAM

- 下载到RAM



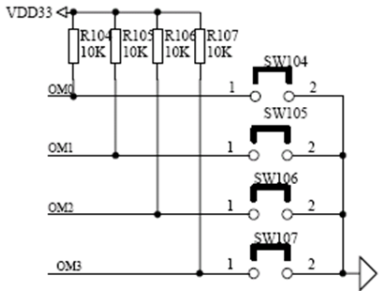
如何实现上电启动?

Run in FLASH

- Linker Configure
  - RO Base=0x00000000
  - RW Base=0x30000000
- Output Configure
  - Create HEX file
- DRAM 初始化
  - Initialization by code

Nor Flash

- Set Jumper　(boot from Nor Flash)(16bit)
  - SW104: OM0 (Open)
  - SW105: OM1 (Close)

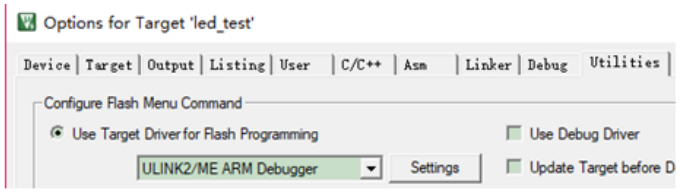


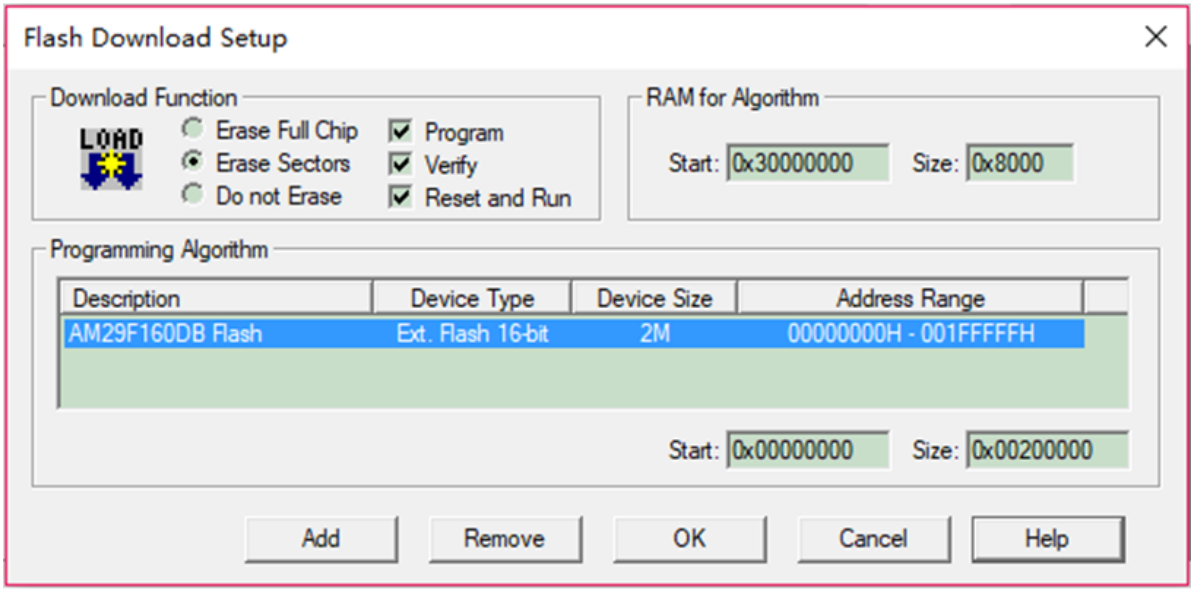
OM1 (Operating Mode 1)	OM0 (Operating Mode 0)	Booting ROM Data width
0	0	Nand Flash Mode
0	1	16-bit
1	0	32-bit
1	1	Test Mode

Linker CMD

```
LR_ROM1 0x00000000
{    ; load region
     ER_ROM1 0x00000000 0x0200000
    {  ; load address = execution address
    *.o (RESET, +First)
```

```
    *(InRoot$$Sections)
    .ANY (+RO)
   }
  RW_RAM1 0x30000000 0x4000000
  {  ; RW data
    .ANY (+RW +ZI)
   }
   RW_IRAM1 0x40000000 0x00001000
   {
    .ANY (+RW +ZI)
   }
}
```





小结

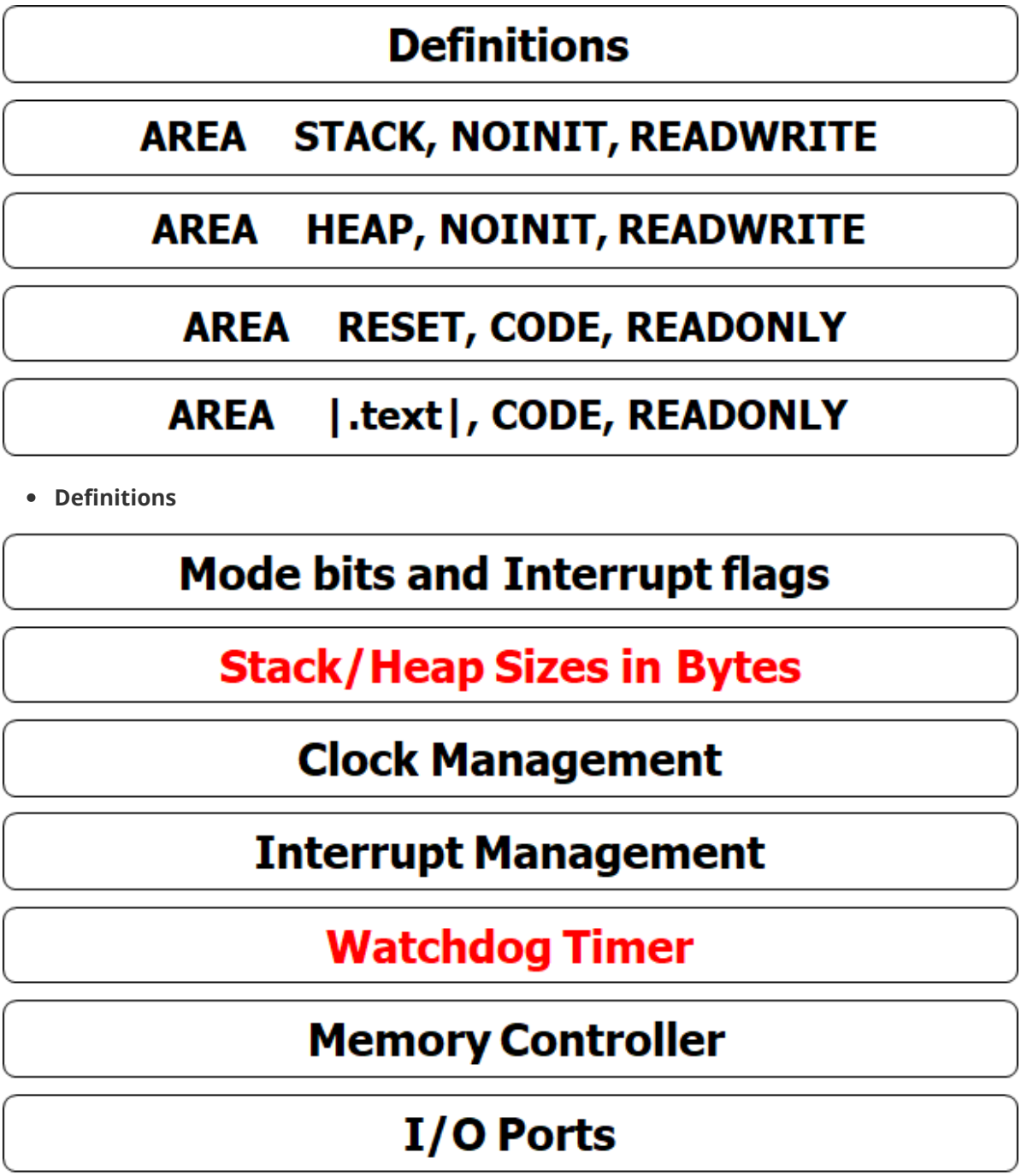
- 函数调用
- 异常向量表
- 异常服务程序注册
- 外设访问

ARM程序分析

启动程序

Start.s

- **结构**



- **Definitions**

- PRESERVE8 8字节对齐

```
;/*****************************************************************************/
;/* S3C2410A.S: Startup file for Samsung S3C410A                              */
;/*****************************************************************************/
;/* <<< Use Configuration Wizard in Context Menu >>>                          */

;/*****************************************************************************/
;/* This file is part of the uVision/ARM development tools.                   */
;/* Copyright (c) 2005-2006 Keil Software. All rights reserved.               */
;/* This software may only be used under the terms of a valid, current,       */
;/* end user licence from KEIL for a compatible version of KEIL software      */
;/* development tools. Nothing else gives you the right to use this software. */
;/*****************************************************************************/


; *** Startup Code (executed after Reset) ***
```

```
; Standard definitions of Mode bits and Interrupt (I & F) flags in PSRs

Mode_USR        EQU     0x10
Mode_FIQ        EQU     0x11
Mode_IRQ        EQU     0x12
Mode_SVC        EQU     0x13
Mode_ABT        EQU     0x17
Mode_UND        EQU     0x1B
Mode_SYS        EQU     0x1F

I_Bit           EQU     0x80            ; when I bit is set, IRQ is disabled
F_Bit           EQU     0x40            ; when F bit is set, FIQ is disabled


;// <h> Stack Configuration (Stack Sizes in Bytes)
;//   <o0> Undefined Mode      <0x0-0xFFFFFFFF:8>
;//   <o1> Supervisor Mode     <0x0-0xFFFFFFFF:8>
;//   <o2> Abort Mode          <0x0-0xFFFFFFFF:8>
;//   <o3> Fast Interrupt Mode <0x0-0xFFFFFFFF:8>
;//   <o4> Interrupt Mode      <0x0-0xFFFFFFFF:8>
;//   <o5> User/System Mode    <0x0-0xFFFFFFFF:8>
;// </h>

;   Stack/Heap Definition 堆栈定义
UND_Stack_Size  EQU     0x00000000
SVC_Stack_Size  EQU     0x00000008
ABT_Stack_Size  EQU     0x00000000
FIQ_Stack_Size  EQU     0x00000000
IRQ_Stack_Size  EQU     0x00000080
USR_Stack_Size  EQU     0x00000400


; ----------------------------------------------------------------------------
Stack_Size      EQU     (UND_Stack_Size + SVC_Stack_Size + ABT_Stack_Size + \
                         FIQ_Stack_Size + IRQ_Stack_Size + USR_Stack_Size)

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size

; ARM 栈初始位于栈顶
Stack_Top       EQU     Stack_Mem + Stack_Size

;// <h> Heap Configuration
;//   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF>
;// </h>

Heap_Size       EQU     0x00000000

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
Heap_Mem        SPACE   Heap_Size

; ----------------------------------------------------------------------------



; Clock Management definitions
; Clock定义
```

```
CLK_BASE        EQU     0x4C000000      ; Clock Base Address
LOCKTIME_OFS    EQU     0x00            ; LOCKTIME Offset
MPLLCON_OFS     EQU     0x04            ; MPLLCON Offset
UPLLCON_OFS     EQU     0x08            ; UPLLCON Offset
CLKCON_OFS      EQU     0x0C            ; CLKCON Offset
CLKSLOW_OFS     EQU     0x10            ; CLKSLOW Offset
CLKDIVN_OFS     EQU     0x14            ; CLDKIVN Offset
CAMDIVN_OFS     EQU     0x18            ; CAMDIVN Offset



;// <e> Clock Management
;//   <h> MPLL Settings
;//   <i> Mpll = (m * Fin) / (p * 2^s)
;//     <o1.12..19> MDIV: Main divider <0x0-0xFF>
;//                 <i> m = MDIV + 8
;//     <o1.4..9>   PDIV: Pre-divider  <0x0-0x3F>
;//                 <i> p = PDIV + 2
;//     <o1.0..1>   SDIV: Post Divider <0x0-0x03>
;//                 <i> s = SDIV
;//   </h>
;//   <h> UPLL Settings
;//   <i> Upll = ( m * Fin) / (p * 2^s),Uclk must be 48MHZ to USB device
;//     <o2.12..19> MDIV: Main divider <0x1-0xF8>
;//                 <i> m = MDIV + 8,if Fin=12MHZ MDIV could be 0x38
;//     <o2.4..9>   PDIV: Pre-divider  <0x1-0x3E>
;//                 <i> p = PDIV + 2,if Fin=12MHZ PDIV could be 0x2
;//     <o2.0..1>   SDIV: Post Divider <0x0-0x03>
;//                 <i> s = SDIV ,if Fin=12MHZ SDIV could be 0x2
;//   </h>
;//   <h>LOCK TIME
;//      <o5.0..11>  LTIME CNT: MPLL Lock Time Count  <0x0-0xFFF>
;//      <o5.12..23>  LTIME CNT: UPLL Lock Time Count  <0x0-0xFFF>
;//   </h>
;//   <h> Master Clock
;//   <i> PLL Clock:  FCLK = FMPLL
;//   <i> Slow Clock: FCLK = Fin / (2 * SLOW_VAL), SLOW_VAL > 0
;//   <i> Slow Clock: FCLK = Fin, SLOW_VAL = 0
;//     <o4.7>      UCLK_ON: UCLK ON
;//                 <i> 0: UCLK ON(UPLL is also turned on) 1: UCLK OFF (UPLL is
also turned off)
;//     <o4.5>      MPLL_OFF: Turn off PLL
;//                 <i> 0: Turn on PLL.After PLL stabilization time (minimum
300us), SLOW_BIT can be cleared to 0. 1: Turn off PLL. PLL is turned off only
when SLOW_BIT is 1.
;//     <o4.4>      SLOW_BIT: Slow Clock
;//     <o4.0..2>   SLOW_VAL: Slow Clock divider    <0x0-0x7>
;//   </h>
;//   <h> CLOCK DIVIDER CONTROL
;//     <o6.1>   HDIVN
;//                  <i> 0: HCLK = FCLK/1, 01 : HCLK = FCLK/2
;//     <o6.0>      PDIVN
;//                  <i> 0: PCLK has the clock same as the HCLK/1,1: PCLK has the
clock same as the HCLK/2
;//    </h>
;//  <h> Clock Generation
;//     <o3.18>     SPI          <0=> Disable  <1=> Enable
;//     <o3.17>     IIS          <0=> Disable  <1=> Enable
```

```
;//     <o3.16>     IIC          <0=> Disable  <1=> Enable
;//     <o3.15>     ADC          <0=> Disable  <1=> Enable
;//     <o3.14>     RTC          <0=> Disable  <1=> Enable
;//     <o3.13>     GPIO         <0=> Disable  <1=> Enable
;//     <o3.12>     UART2        <0=> Disable  <1=> Enable
;//     <o3.11>     UART1        <0=> Disable  <1=> Enable
;//     <o3.10>     UART0        <0=> Disable  <1=> Enable
;//     <o3.9>      SDI          <0=> Disable  <1=> Enable
;//     <o3.8>      PWMTIMER     <0=> Disable  <1=> Enable
;//     <o3.7>      USB device   <0=> Disable  <1=> Enable
;//     <o3.6>      USB host     <0=> Disable  <1=> Enable
;//     <o3.5>      LCDC         <0=> Disable  <1=> Enable
;//     <o3.4>      NAND FLASH Controller       <0=> Disable  <1=> Enable
;//     <o3.3>      POWER-OFF    <0=> Disable  <1=> Enable
;//     <o3.2>      IDLE BIT     <0=> Disable  <1=> Enable
;//     <o3.0>      SM_BIT       <0=> Disable  <1=> Enable
;//   </h>
;// </e>
CLK_SETUP       EQU     1
MPLLCON_Val     EQU     0x0005C080
UPLLCON_Val     EQU     0x00028080
CLKCON_Val      EQU     0x0007FFF0
CLKSLOW_Val     EQU     0x00000004
LOCKTIME_Val    EQU     0x00FFFFFF
CLKDIVN_Val     EQU     0X00000000



;Interrupt  definitions
INTOFFSET          EQU    0X4A000014                    ;Address of Interrupt
offset Register

;//<e> Interrupt Vector Table
;//  <o1.0..31> Interrupt Vector address     <0x20-0x3fffff78>
;//            <i> You could define Interuupt Vctor Table address.
;//            <i> The Interrupt Vector Table address must be word aligned
adress.
;//</e>
IntVT_SETUP      EQU     1
IntVTAddress    EQU     0x33FFFF20

; Watchdog Timer definitions
WT_BASE         EQU     0x53000000      ; WT Base Address
WTCON_OFS       EQU     0x00            ; WTCON Offset
WTDAT_OFS       EQU     0x04            ; WTDAT Offset
WTCNT_OFS       EQU     0x08            ; WTCNT Offset

;// <e> Watchdog Timer
;//   <o1.5>      Watchdog Timer Enable/Disable
;//   <o1.0>      Reset Enable/Disable
;//   <o1.2>      Interrupt Enable/Disable
;//   <o1.3..4>   Clock Select
;//               <0=> 1/16  <1=> 1/32  <2=> 1/64  <3=> 1/128
;//               <i> Clock Division Factor
;//   <o1.8..15>  Prescaler Value <0x0-0xFF>
;//   <o2.0..15>  Time-out Value  <0x0-0xFFFF>
;// </e>
WT_SETUP        EQU     1
```

```
WTCON_Val       EQU     0x00008021
WTDAT_Val       EQU     0x00008000


; Memory Controller definitions
MC_BASE         EQU     0x48000000      ; Memory Controller Base Address

;// <e> Memory Controller
MC_SETUP        EQU     1

;//   <h> Bank 0
;//     <o0.0..1>   PMC: Page Mode Configuration
;//                 <0=> 1 Data  <1=> 4 Data  <2=> 8 Data  <3=> 16 Data
;//     <o0.2..3>   Tpac: Page Mode Access Cycle
;//                 <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> 6 clks
;//     <o0.4..5>   Tcah: Address Holding Time after nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o0.6..7>   Toch: Chip Select Hold on nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o0.8..10>  Tacc: Access Cycle
;//                 <0=> 1 clk   <1=> 2 clks  <2=> 3 clks  <3=> 4 clks
;//                 <4=> 6 clk   <5=> 8 clks  <6=> 10 clks <7=> 14 clks
;//     <o0.11..12> Tcos: Chip Select Set-up nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o0.13..14> Tacs: Address Set-up before nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//   </h>
;//
;//   <h> Bank 1
;//     <o8.4..5>   DW: Data Bus Width
;//                 <0=> 8-bit   <1=> 16-bit  <2=> 32-bit  <3=> Rsrvd
;//     <o8.6>      WS: WAIT Status
;//                 <0=> WAIT Disable
;//                 <1=> WAIT Enable
;//     <o8.7>      ST: SRAM Type
;//                 <0=> Not using UB/LB
;//                 <1=> Using UB/LB
;//     <o1.0..1>   PMC: Page Mode Configuration
;//                 <0=> 1 Data  <1=> 4 Data  <2=> 8 Data  <3=> 16 Data
;//     <o1.2..3>   Tpac: Page Mode Access Cycle
;//                 <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> 6 clks
;//     <o1.4..5>   Tcah: Address Holding Time after nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o1.6..7>   Toch: Chip Select Hold on nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o1.8..10>  Tacc: Access Cycle
;//                 <0=> 1 clk   <1=> 2 clks  <2=> 3 clks  <3=> 4 clks
;//                 <4=> 6 clk   <5=> 8 clks  <6=> 10 clks <7=> 14 clks
;//     <o1.11..12> Tcos: Chip Select Set-up nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o1.13..14> Tacs: Address Set-up before nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//   </h>
;//
;//   <h> Bank 2
;//     <o8.8..9>   DW: Data Bus Width
;//                 <0=> 8-bit   <1=> 16-bit  <2=> 32-bit  <3=> Rsrvd
;//     <o8.10>     WS: WAIT Status
```

```
;//                 <0=> WAIT Disable
;//                 <1=> WAIT Enable
;//     <o8.11>     ST: SRAM Type
;//                 <0=> Not using UB/LB
;//                 <1=> Using UB/LB
;//     <o2.0..1>   PMC: Page Mode Configuration
;//                 <0=> 1 Data  <1=> 4 Data  <2=> 8 Data  <3=> 16 Data
;//     <o2.2..3>   Tpac: Page Mode Access Cycle
;//                 <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> 6 clks
;//     <o2.4..5>   Tcah: Address Holding Time after nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o2.6..7>   Toch: Chip Select Hold on nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o2.8..10>  Tacc: Access Cycle
;//                 <0=> 1 clk   <1=> 2 clks  <2=> 3 clks  <3=> 4 clks
;//                 <4=> 6 clk   <5=> 8 clks  <6=> 10 clks <7=> 14 clks
;//     <o2.11..12> Tcos: Chip Select Set-up nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o2.13..14> Tacs: Address Set-up before nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//   </h>
;//
;//   <h> Bank 3
;//     <o8.12..13> DW: Data Bus Width
;//                 <0=> 8-bit   <1=> 16-bit  <2=> 32-bit  <3=> Rsrvd
;//     <o8.14>     WS: WAIT Status
;//                 <0=> WAIT Disable
;//                 <1=> WAIT Enable
;//     <o8.15>     ST: SRAM Type
;//                 <0=> Not using UB/LB
;//                 <1=> Using UB/LB
;//     <o3.0..1>   PMC: Page Mode Configuration
;//                 <0=> 1 Data  <1=> 4 Data  <2=> 8 Data  <3=> 16 Data
;//     <o3.2..3>   Tpac: Page Mode Access Cycle
;//                 <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> 6 clks
;//     <o3.4..5>   Tcah: Address Holding Time after nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o3.6..7>   Toch: Chip Select Hold on nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o3.8..10>  Tacc: Access Cycle
;//                 <0=> 1 clk   <1=> 2 clks  <2=> 3 clks  <3=> 4 clks
;//                 <4=> 6 clk   <5=> 8 clks  <6=> 10 clks <7=> 14 clks
;//     <o3.11..12> Tcos: Chip Select Set-up nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o3.13..14> Tacs: Address Set-up before nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//   </h>
;//
;//   <h> Bank 4
;//     <o8.16..17> DW: Data Bus Width
;//                 <0=> 8-bit   <1=> 16-bit  <2=> 32-bit  <3=> Rsrvd
;//     <o8.18>     WS: WAIT Status
;//                 <0=> WAIT Disable
;//                 <1=> WAIT Enable
;//     <o8.19>     ST: SRAM Type
;//                 <0=> Not using UB/LB
;//                 <1=> Using UB/LB
;//     <o4.0..1>   PMC: Page Mode Configuration
```

```
;//                 <0=> 1 Data  <1=> 4 Data  <2=> 8 Data  <3=> 16 Data
;//     <o4.2..3>   Tpac: Page Mode Access Cycle
;//                 <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> 6 clks
;//     <o4.4..5>   Tcah: Address Holding Time after nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o4.6..7>   Toch: Chip Select Hold on nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o4.8..10>  Tacc: Access Cycle
;//                 <0=> 1 clk   <1=> 2 clks  <2=> 3 clks  <3=> 4 clks
;//                 <4=> 6 clk   <5=> 8 clks  <6=> 10 clks <7=> 14 clks
;//     <o4.11..12> Tcos: Chip Select Set-up nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o4.13..14> Tacs: Address Set-up before nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//   </h>
;//
;//   <h> Bank 5
;//     <o8.20..21> DW: Data Bus Width
;//                 <0=> 8-bit   <1=> 16-bit  <2=> 32-bit  <3=> Rsrvd
;//     <o8.22>     WS: WAIT Status
;//                 <0=> WAIT Disable
;//                 <1=> WAIT Enable
;//     <o8.23>     ST: SRAM Type
;//                 <0=> Not using UB/LB
;//                 <1=> Using UB/LB
;//     <o5.0..1>   PMC: Page Mode Configuration
;//                 <0=> 1 Data  <1=> 4 Data  <2=> 8 Data  <3=> 16 Data
;//     <o5.2..3>   Tpac: Page Mode Access Cycle
;//                 <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> 6 clks
;//     <o5.4..5>   Tcah: Address Holding Time after nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o5.6..7>   Toch: Chip Select Hold on nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o5.8..10>  Tacc: Access Cycle
;//                 <0=> 1 clk   <1=> 2 clks  <2=> 3 clks  <3=> 4 clks
;//                 <4=> 6 clk   <5=> 8 clks  <6=> 10 clks <7=> 14 clks
;//     <o5.11..12> Tcos: Chip Select Set-up nOE
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     <o5.13..14> Tacs: Address Set-up before nGCSn
;//                 <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//   </h>
;//
;//   <h> Bank 6
;//     <o10.0..2>  BK76MAP: Bank 6/7 Memory Map
;//                 <0=> 32M  <1=> 64M <2=> 128M <4=> 2M   <5=> 4M   <6=> 8M
<7=> 16M
;//     <o8.24..25> DW: Data Bus Width
;//                 <0=> 8-bit   <1=> 16-bit  <2=> 32-bit  <3=> Rsrvd
;//     <o8.26>     WS: WAIT Status
;//                 <0=> WAIT Disable
;//                 <1=> WAIT Enable
;//     <o8.27>     ST: SRAM Type
;//                 <0=> Not using UB/LB
;//                 <1=> Using UB/LB
;//     <o6.15..16> MT: Memory Type
;//                 <0=> ROM or SRAM
;//                 <3=> SDRAM
;//     <h> ROM or SRAM
```

```
;//       <o6.0..1>   PMC: Page Mode Configuration
;//                   <0=> 1 Data  <1=> 4 Data  <2=> 8 Data  <3=> 16 Data
;//       <o6.2..3>   Tpac: Page Mode Access Cycle
;//                 <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> 6 clks
;//       <o6.4..5>   Tcah: Address Holding Time after nGCSn
;//                   <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//       <o6.6..7>   Toch: Chip Select Hold on nOE
;//                   <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//       <o6.8..10>  Tacc: Access Cycle
;//                   <0=> 1 clk   <1=> 2 clks  <2=> 3 clks  <3=> 4 clks
;//                   <4=> 6 clk   <5=> 8 clks  <6=> 10 clks <7=> 14 clks
;//       <o6.11..12> Tcos: Chip Select Set-up nOE
;//                   <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//       <o6.13..14> Tacs: Address Set-up before nGCSn
;//                   <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     </h>
;//     <h> SDRAM
;//       <o6.0..1>   SCAN: Columnn Address Number
;//                   <0=> 8-bit   <1=> 9-bit   <2=> 10-bit  <3=> Rsrvd
;//       <o6.2..3>   Trcd: RAS to CAS Delay
;//                   <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> Rsrvd
;//       <o10.4>     SCKEEN: SCLK Selection (Bank 6/7)
;//                   <0=> Normal
;//                   <1=> Reduced Power
;//       <o10.5>     SCLKEN: SDRAM power down mode (Bank 6/7)
;//                   <0=> DISABLE
;//                   <1=> ENABLE
;//       <o10.7>     BURST_EN: ARM core burst operation (Bank 6/7)
;//                   <0=> DISABLE
;//                   <1=> ENABLE
;//       <o11.0..2>  BL: Burst Length
;//                   <0=> 1
;//       <o11.3>     BT: Burst Type
;//                   <0=> Sequential
;//       <o11.4..6>  CL: CAS Latency
;//                   <0=> 1 clk   <1=> 2 clks  <2=> 3 clks
;//       <o11.7..8>  TM: Test Mode
;//                   <0=> Mode Register Set
;//       <o11.9>     WBL: Write Burst Length
;//                   <0=> 0
;//     </h>
;//   </h>
;//
;//   <h> Bank 7
;//     <o10.0..2>  BK76MAP: Bank 6/7 Memory Map
;//                 <0=> 32M  <1=> 64M <2=> 128M <4=> 2M   <5=> 4M   <6=> 8M
<7=> 16M
;//     <o8.28..29> DW: Data Bus Width
;//                 <0=> 8-bit   <1=> 16-bit  <2=> 32-bit  <3=> Rsrvd
;//     <o8.30>     WS: WAIT Status
;//                 <0=> WAIT Disable
;//                 <1=> WAIT Enable
;//     <o8.31>     ST: SRAM Type
;//                 <0=> Not using UB/LB
;//                 <1=> Using UB/LB
;//     <o7.15..16> MT: Memory Type
;//                 <0=> ROM or SRAM
;//                 <3=> SDRAM
```

```
;//     <h> ROM or SRAM
;//       <o7.0..1>   PMC: Page Mode Configuration
;//                   <0=> 1 Data  <1=> 4 Data  <2=> 8 Data  <3=> 16 Data
;//       <o7.2..3>   Tpac: Page Mode Access Cycle
;//                 <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> 6 clks
;//       <o7.4..5>   Tcah: Address Holding Time after nGCSn
;//                   <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//       <o7.6..7>   Toch: Chip Select Hold on nOE
;//                   <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//       <o7.8..10>  Tacc: Access Cycle
;//                   <0=> 1 clk   <1=> 2 clks  <2=> 3 clks  <3=> 4 clks
;//                   <4=> 6 clk   <5=> 8 clks  <6=> 10 clks <7=> 14 clks
;//       <o7.11..12> Tcos: Chip Select Set-up nOE
;//                   <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//       <o7.13..14> Tacs: Address Set-up before nGCSn
;//                   <0=> 0 clk   <1=> 1 clk   <2=> 2 clks  <3=> 4 clks
;//     </h>
;//     <h> SDRAM
;//       <o7.0..1>   SCAN: Columnn Address Number
;//                   <0=> 8-bit   <1=> 9-bit   <2=> 10-bit  <3=> Rsrvd
;//       <o7.2..3>   Trcd: RAS to CAS Delay
;//                   <0=> 2 clks  <1=> 3 clks  <2=> 4 clks  <3=> Rsrvd
;//       <o10.4>     SCLKEN: SCLK Selection (Bank 6/7)
;//                   <0=> Normal
;//                   <1=> Reduced Power
;//       <o10.5>     SCLKEN: SDRAM power down mode (Bank 6/7)
;//                   <0=> DISABLE
;//                   <1=> ENABLE
;//       <o10.7>     BURST_EN: ARM core burst operation (Bank 6/7)
;//                   <0=> DISABLE
;//                   <1=> ENABLE
;//       <o12.0..2>  BL: Burst Length
;//                   <0=> 1
;//       <o12.3>     BT: Burst Type
;//                   <0=> Sequential
;//       <o12.4..6>  CL: CAS Latency
;//                   <0=> 1 clk   <1=> 2 clks  <2=> 3 clks
;//       <o12.7..8>  TM: Test Mode
;//                   <0=> Mode Register Set
;//       <o12.9>     WBL: Write Burst Length
;//                   <0=> 0
;//     </h>
;//   </h>
;//
;//   <h> Refresh
;//     <o9.23>     REFEN: SDRAM Refresh
;//                 <0=> Disable <1=> Enable
;//     <o9.22>     TREFMD: SDRAM Refresh Mode
;//                 <0=> CBR/Auto Refresh
;//                 <1=> Self Refresh
;//     <o9.20..21> Trp: SDRAM RAS Pre-charge Time
;//                 <0=> 2 clks
;//                 <1=> 3 clks
;//                 <2=> 4 clks
;//                 <3=> Rsrvd
;//     <o9.18..19> Tsrc: SDRAM Semi Row cycle time
;//                <i> SDRAM Row cycle time: Trc=Tsrc+Trp
;//                 <0=> 4 clks  <1=> 5 clks  <2=> 6 clks  <3=> 7 clks
```

```
;//     <o9.0..10>  Refresh Counter <0x0-0x07FF>
;//                 <i> Refresh Period = (2^11 - Refresh Count + 1) / HCLK
;//   </h>
BANKCON0_Val    EQU     0x00000700
BANKCON1_Val    EQU     0x00000700
BANKCON2_Val    EQU     0x00000700
BANKCON3_Val    EQU     0x00000700
BANKCON4_Val    EQU     0x00000700
BANKCON5_Val    EQU     0x00000700
BANKCON6_Val    EQU     0x00018008
BANKCON7_Val    EQU     0x00018008
BWSCON_Val      EQU     0x00000000
REFRESH_Val     EQU     0x00ac0000
BANKSIZE_Val    EQU     0x00000000
MRSRB6_Val      EQU     0x00000020
MRSRB7_Val      EQU     0x00000000

;// </e> End of MC



; I/O Ports definitions
PIO_BASE        EQU     0x56000000      ; PIO Base Address
PCONA_OFS       EQU     0x00            ; PCONA Offset
PCONB_OFS       EQU     0x10            ; PCONB Offset
PCONC_OFS       EQU     0x20            ; PCONC Offset
PCOND_OFS       EQU     0x30            ; PCOND Offset
PCONE_OFS       EQU     0x40            ; PCONE Offset
PCONF_OFS       EQU     0x50            ; PCONF Offset
PCONG_OFS       EQU     0x60            ; PCONG Offset
PCONH_OFS       EQU     0x70            ; PCONH Offset
PCONJ_OFS       EQU     0xD0            ; PCONJ Offset
PUPB_OFS        EQU     0x18            ; PUPB Offset
PUPC_OFS        EQU     0x28            ; PUPC Offset
PUPD_OFS        EQU     0x38            ; PUPD Offset
PUPE_OFS        EQU     0x48            ; PUPE Offset
PUPF_OFS        EQU     0x58            ; PUPF Offset
PUPG_OFS        EQU     0x68            ; PUPG Offset
PUPH_OFS        EQU     0x78            ; PUPH Offset
PUPJ_OFS        EQU     0xD8            ; PUPJ Offset


;// <e> I/O Configuration
PIO_SETUP       EQU     1

;//   <e> Port A
;//     <o1.0>      PA0  <0=> Output   <1=> ADDR0
;//     <o1.1>      PA1  <0=> Output   <1=> ADDR16
;//     <o1.2>      PA2  <0=> Output   <1=> ADDR17
;//     <o1.3>      PA3  <0=> Output   <1=> ADDR18
;//     <o1.4>      PA4  <0=> Output   <1=> ADDR19
;//     <o1.5>      PA5  <0=> Output   <1=> ADDR20
;//     <o1.6>      PA6  <0=> Output   <1=> ADDR21
;//     <o1.7>      PA7  <0=> Output   <1=> ADDR22
;//     <o1.8>      PA8  <0=> Output   <1=> ADDR23
;//     <o1.9>      PA9  <0=> Output   <1=> ADDR24
;//     <o1.10>      PA0  <0=> Output   <1=> ADDR25
;//     <o1.11>      PA1  <0=> Output   <1=> ADDR26
```

```
;//     <o1.12>      PA2  <0=> Output   <1=> nGCS[1]
;//     <o1.13>      PA3  <0=> Output   <1=> nGCS[2]
;//     <o1.14>      PA4  <0=> Output   <1=> nGCS[3]
;//     <o1.15>      PA5  <0=> Output   <1=> nGCS[4]
;//     <o1.16>      PA6  <0=> Output   <1=> nGCS[5]
;//     <o1.17>      PA7  <0=> Output   <1=> CLE
;//     <o1.18>      PA8  <0=> Output   <1=> ALE
;//     <o1.19>      PA9  <0=> Output   <1=> nFWE
;//     <o1.20>      PA0  <0=> Output   <1=> nFRE
;//     <o1.21>      PA1  <0=> Output   <1=> nRSTOUT
;//     <o1.22>      PA2  <0=> Output   <1=> nFCE
;//   </e>
PIOA_SETUP      EQU     0
PCONA_Val       EQU     0x000003FF

;//   <e> Port B
;//     <o1.0..1>        PB0  <0=> Input   <1=> Output  <2=> TOUT0    <3=>
Reserved
;//     <o1.2..3>        PB1  <0=> Input   <1=> Output  <2=> TOUT1    <3=>
Reserved
;//     <o1.4..5>        PB2  <0=> Input   <1=> Output  <2=> TOUT2    <3=>
Reserved
;//     <o1.6..7>        PB3  <0=> Input   <1=> Output  <2=> TOUT3    <3=>
Reserved
;//     <o1.8..9>        PB4  <0=> Input   <1=> Output  <2=> TCLK[0]  <3=>
Reserved
;//     <o1.10..11>      PB5  <0=> Input   <1=> Output  <2=> nXBACK   <3=>
Reserved
;//     <o1.12..13>      PB6  <0=> Input   <1=> Output  <2=> nXBREQ   <3=>
Reserved
;//     <o1.14..15>      PB7  <0=> Input   <1=> Output  <2=> nXDACK1  <3=>
Reserved
;//     <o1.16..17>      PB8  <0=> Input   <1=> Output  <2=> nXDREQ1  <3=>
Reserved
;//     <o1.18..19>      PB9  <0=> Input   <1=> Output  <2=> nXDACK0  <3=>
Reserved
;//     <o1.20..21>      PB10 <0=> Input   <1=> Output  <2=> nXDREQ0  <3=>
Reserved
;//     <h> Pull-up Resistors
;//       <o2.0>    PB0 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.1>    PB1 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.2>    PB2 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.3>    PB3 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.4>    PB4 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.5>    PB5 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.6>    PB6 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.7>    PB7 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.8>    PB8 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.9>    PB9 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.10>   PB10 Pull-up       <0=> Enabled  <1=> Disabled
;//     </h>
;//   </e>
PIOB_SETUP      EQU     0
PCONB_Val       EQU     0x000007FF
PUPB_Val        EQU     0x00000000

;//   <e> Port C
```

```
;//     <o1.0..1>          PC0  <0=> Input    <1=> Output  <2=> LEND
<3=> Reserved
;//     <o1.2..3>          PC1  <0=> Input    <1=> Output  <2=> VCLK
<3=> Reserved
;//     <o1.4..5>          PC2  <0=> Input    <1=> Output  <2=> VLINE
<3=> Reserved
;//     <o1.6..7>          PC3  <0=> Input    <1=> Output  <2=> VFRAME
<3=> Reserved
;//     <o1.8..9>          PC4  <0=> Input    <1=> Output  <2=> VM
<3=> Reserved
;//     <o1.10..11>        PC5  <0=> Input    <1=> Output  <2=> LCDVF2     <3=>
Reserved
;//     <o1.12..13>        PC6  <0=> Input    <1=> Output  <2=> LCDVF1    <3=>
Reserved
;//     <o1.14..15>        PC7  <0=> Input    <1=> Output  <2=> LCDVF0   <3=>
Reserved
;//     <o1.16..17>        PC8  <0=> Input    <1=> Output  <2=> VD[0]
<3=> Reserved
;//     <o1.18..19>        PC9  <0=> Input    <1=> Output  <2=> VD[1]
<3=> Reserved
;//     <o1.20..21>        PC10 <0=> Input    <1=> Output  <2=> VD[2]
<3=> Reserved
;//     <o1.22..23>        PC11  <0=> Input   <1=> Output  <2=> VD[3]
<3=> Reserved
;//     <o1.24..25>        PC12  <0=> Input   <1=> Output  <2=> VD[4]
<3=> Reserved
;//     <o1.26..27>        PC13  <0=> Input   <1=> Output  <2=> VD[5]
<3=> Reserved
;//     <o1.28..29>        PC14  <0=> Input   <1=> Output  <2=> VD[6]
<3=> Reserved
;//     <o1.30..31>        PC15  <0=> Input   <1=> Output  <2=> VD[7]
<3=> Reserved
;//     <h> Pull-up Resistors
;//       <o2.0>     PC0 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.1>     PC1 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.2>     PC2 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.3>     PC3 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.4>     PC4 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.5>     PC5 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.6>     PC6 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.7>     PC7 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.8>     PC8 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.9>     PC9 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.10>    PC10 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.11>    PC11 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.12>    PC12 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.13>    PC13 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.14>    PC14 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.15>    PC15 Pull-up        <0=> Enabled  <1=> Disabled
;//     </h>
;//   </e>
PIOC_SETUP      EQU     0
PCONC_Val       EQU     0xAAAAAAAA
PUPC_Val        EQU     0x00000000

;//   <e> Port D
;//     <o1.0..1>          PD0  <0=> Input    <1=> Output  <2=> VD[8]
<3=> Reserved
```

```
;//     <o1.2..3>          PD1  <0=> Input    <1=> Output  <2=> VD[9]
<3=> Reserved
;//     <o1.4..5>          PD2  <0=> Input    <1=> Output  <2=> VD[10]
<3=> Reserved
;//     <o1.6..7>          PD3  <0=> Input    <1=> Output  <2=> VD[11]
<3=> Reserved
;//     <o1.8..9>          PD4  <0=> Input    <1=> Output  <2=> VD[12]
<3=> Reserved
;//     <o1.10..11>        PD5  <0=> Input    <1=> Output  <2=> VD[13]
<3=> Reserved
;//     <o1.12..13>        PD6  <0=> Input    <1=> Output  <2=> VD[14]
<3=> Reserved
;//     <o1.14..15>        PD7  <0=> Input    <1=> Output  <2=> VD[15]
<3=> Reserved
;//     <o1.16..17>        PD8  <0=> Input    <1=> Output  <2=> VD[16]
<3=> Reserved
;//     <o1.18..19>        PD9  <0=> Input    <1=> Output  <2=> VD[17]
<3=> Reserved
;//     <o1.20..21>        PD10 <0=> Input    <1=> Output  <2=> VD[18]
<3=> Reserved
;//     <o1.22..23>        PD11  <0=> Input   <1=> Output  <2=> VD[19]
<3=> Reserved
;//     <o1.24..25>        PD12  <0=> Input   <1=> Output  <2=> VD[20]
<3=> Reserved
;//     <o1.26..27>        PD13  <0=> Input   <1=> Output  <2=> VD[21]
<3=> Reserved
;//     <o1.28..29>        PD14  <0=> Input   <1=> Output  <2=> VD[22]
<3=> nSS1
;//     <o1.30..31>        PD15  <0=> Input   <1=> Output  <2=> VD[23]
<3=> nSS0
;//     <h> Pull-up Resistors
;//       <o2.0>     PD0 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.1>     PD1 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.2>     PD2 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.3>     PD3 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.4>     PD4 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.5>     PD5 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.6>     PD6 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.7>     PD7 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.8>     PD8 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.9>     PD9 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.10>    PD10 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.11>    PD11 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.12>    PD12 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.13>    PD13 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.14>    PD14 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.15>    PD15 Pull-up        <0=> Enabled  <1=> Disabled
;//     </h>
;//   </e>
PIOD_SETUP      EQU     0
PCOND_Val       EQU     0x00000000
PUPD_Val        EQU     0x00000000

;//   <e> Port E
;//     <o1.0..1>          PE0  <0=> Input    <1=> Output  <2=> I2SLRCK
<3=> Reserved
;//     <o1.2..3>          PE1  <0=> Input    <1=> Output  <2=> I2SSCLK
<3=> Reserved
```

```
;//     <o1.4..5>          PE2  <0=> Input    <1=> Output  <2=> CDCLK
<3=> Reserved
;//     <o1.6..7>          PE3  <0=> Input    <1=> Output  <2=> I2SDI
<3=> nSS0
;//     <o1.8..9>          PE4  <0=> Input    <1=> Output  <2=> I2SDO
<3=> I2SSDI
;//     <o1.10..11>        PE5  <0=> Input    <1=> Output  <2=> SDCLK
<3=> Reserved
;//     <o1.12..13>        PE6  <0=> Input    <1=> Output  <2=> SDCMD
<3=> Reserved
;//     <o1.14..15>        PE7  <0=> Input    <1=> Output  <2=> SDDAT0
<3=> Reserved
;//     <o1.16..17>        PE8  <0=> Input    <1=> Output  <2=> SDDAT1
<3=> Reserved
;//     <o1.18..19>        PE9  <0=> Input    <1=> Output  <2=> SDDAT2
<3=> Reserved
;//     <o1.20..21>        PE10 <0=> Input    <1=> Output  <2=> SDDAT3
<3=> Reserved
;//     <o1.22..23>        PE11  <0=> Input   <1=> Output  <2=> SPIMISO0
<3=> Reserved
;//     <o1.24..25>        PE12  <0=> Input   <1=> Output  <2=> SPIMOSI0
<3=> Reserved
;//     <o1.26..27>        PE13  <0=> Input   <1=> Output  <2=> SPICLK0
<3=> Reserved
;//     <o1.28..29>        PE14  <0=> Input   <1=> Output  <2=> IICSCL
<3=> Reserved
;//     <o1.30..31>        PE15  <0=> Input   <1=> Output  <2=> IICSDA
<3=> Reserved
;//     <h> Pull-up Resistors

;//       <o2.0>     PE0 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.1>     PE1 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.2>     PE2 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.3>     PE3 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.4>     PE4 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.5>     PE5 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.6>     PE6 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.7>     PE7 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.8>     PE8 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.9>     PE9 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.10>    PE10 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.11>    PE11 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.12>    PE12 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.13>    PE13 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.14>    PE14 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.15>    PE15 Pull-up        <0=> Enabled  <1=> Disabled
;//     </h>
;//   </e>
PIOE_SETUP      EQU     0
PCONE_Val       EQU     0x00000000
PUPE_Val        EQU     0x00000000

;//   <e> Port F
;//     <o1.0..1>        PF0  <0=> Input   <1=> Output  <2=> EINT[0]  <3=>
Reserved
;//     <o1.2..3>        PF1  <0=> Input   <1=> Output  <2=> EINT[1]  <3=>
Reserved
```

```
;//     <o1.4..5>        PF2  <0=> Input   <1=> Output  <2=> EINT[2]  <3=>
Reserved
;//     <o1.6..7>        PF3  <0=> Input   <1=> Output  <2=> EINT[3]  <3=>
Reserved
;//     <o1.8..9>        PF4  <0=> Input   <1=> Output  <2=> EINT[4]  <3=>
Reserved
;//     <o1.10..11>      PF5  <0=> Input   <1=> Output  <2=> EINT[5]  <3=>
Reserved
;//     <o1.12..13>      PF6  <0=> Input   <1=> Output  <2=> EINT[6]  <3=>
Reserved
;//     <o1.14..15>      PF7  <0=> Input   <1=> Output  <2=> EINT[7]  <3=>
Reserved
;//     <h> Pull-up Resistors
;//       <o2.0>    PF0 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.1>    PF1 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.2>    PF2 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.3>    PF3 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.4>    PF4 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.5>    PF5 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.6>    PF6 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.7>    PF7 Pull-up        <0=> Enabled  <1=> Disabled
;//     </h>
;//   </e>
PIOF_SETUP      EQU     1
PCONF_Val       EQU     0x0000511A
PUPF_Val        EQU     0x00000000

;//   <e> Port G
;//     <o1.0..1>          PG0  <0=> Input    <1=> Output  <2=> EINT[8]   <3=>
Reserved
;//     <o1.2..3>          PG1  <0=> Input    <1=> Output  <2=> EINT[9]   <3=>
Reserved
;//     <o1.4..5>          PG2  <0=> Input    <1=> Output  <2=> EINT[10]   <3=>
nSS0
;//     <o1.6..7>          PG3  <0=> Input    <1=> Output  <2=> EINT[11]   <3=>
nSS1
;//     <o1.8..9>          PG4  <0=> Input    <1=> Output  <2=> EINT[12]   <3=>
LCD_PWRDN
;//     <o1.10..11>        PG5  <0=> Input    <1=> Output  <2=> EINT[13]   <3=>
SPIMISO1
;//     <o1.12..13>        PG6  <0=> Input    <1=> Output  <2=> EINT[14]   <3=>
SPIMOSI1
;//     <o1.14..15>        PG7  <0=> Input    <1=> Output  <2=> EINT[15]   <3=>
SPICLK1
;//     <o1.16..17>        PG8  <0=> Input    <1=> Output  <2=> EINT[16]   <3=>
Reserved
;//     <o1.18..19>        PG9  <0=> Input    <1=> Output  <2=> EINT[17]   <3=>
Reserved
;//     <o1.20..21>        PG10 <0=> Input    <1=> Output  <2=> EINT[18]   <3=>
Reserved
;//     <o1.22..23>        PG11  <0=> Input   <1=> Output  <2=> EINT[19]   <3=>
TCLK1
;//     <o1.24..25>        PG12  <0=> Input   <1=> Output  <2=> EINT[20]   <3=>
XMON
;//     <o1.26..27>        PG13  <0=> Input   <1=> Output  <2=> EINT[21]   <3=>
nXPON
;//     <o1.28..29>        PG14  <0=> Input   <1=> Output  <2=> EINT[22]   <3=>
YMON
```

```
;//     <o1.30..31>        PG15  <0=> Input   <1=> Output  <2=> EINT[23]   <3=>
nYPON
;//     <h> Pull-up Resistors
;//       <o2.0>     PG0 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.1>     PG1 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.2>     PG2 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.3>     PG3 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.4>     PG4 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.5>     PG5 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.6>     PG6 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.7>     PG7 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.8>     PG8 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.9>     PG9 Pull-up         <0=> Enabled  <1=> Disabled
;//       <o2.10>    PG10 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.11>    PG11 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.12>    PG12 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.13>    PG13 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.14>    PG14 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.15>    PG15 Pull-up        <0=> Enabled  <1=> Disabled
;//     </h>
;//   </e>
PIOG_SETUP      EQU     0
PCONG_Val       EQU     0x00000000
PUPG_Val        EQU     0x00000000

;//   <e> Port H
;//     <o1.0..1>        PH0  <0=> Input   <1=> Output  <2=> nCTS0    <3=>
Reserved
;//     <o1.2..3>        PH1  <0=> Input   <1=> Output  <2=> nRTS0    <3=>
Reserved
;//     <o1.4..5>        PH2  <0=> Input   <1=> Output  <2=> TXD[0]    <3=>
Reserved
;//     <o1.6..7>        PH3  <0=> Input   <1=> Output  <2=> RXD[0]    <3=>
Reserved
;//     <o1.8..9>        PH4  <0=> Input   <1=> Output  <2=> TXD[1]  <3=>
Reserved
;//     <o1.10..11>      PH5  <0=> Input   <1=> Output  <2=> RXD[1]   <3=>
Reserved
;//     <o1.12..13>      PH6  <0=> Input   <1=> Output  <2=> TXD[2]   <3=> nRTS1
;//     <o1.14..15>      PH7  <0=> Input   <1=> Output  <2=> RXD[2]  <3=> nCTS1
;//     <o1.16..17>      PH8  <0=> Input   <1=> Output  <2=> UCLK    <3=>
Reserved
;//     <o1.18..19>      PH9  <0=> Input   <1=> Output  <2=> CLKOUT0  <3=>
Reserved
;//     <o1.20..21>      PH10 <0=> Input   <1=> Output  <2=> CLKOUT1  <3=>
Reserved
;//     <h> Pull-up Resistors
;//       <o2.0>     PH0 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.1>     PH1 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.2>     PH2 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.3>     PH3 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.4>     PH4 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.5>     PH5 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.6>     PH6 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.7>     PH7 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.8>     PH8 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.9>     PH9 Pull-up        <0=> Enabled  <1=> Disabled
;//       <o2.10>    PH10 Pull-up       <0=> Enabled  <1=> Disabled
```

```
;//     </h>
;//   </e>
PIOH_SETUP      EQU     0
PCONH_Val       EQU     0x000007FF
PUPH_Val        EQU     0x00000000

;// </e>



                PRESERVE8


; Area Definition and Entry Point
;  Startup Code must be linked first at Address at which it expects to run.


                AREA    RESET, CODE, READONLY
                IMPORT INISDRAM
                ARM


; Exception Vectors
;  Mapped to Address 0.
;  Absolute addressing mode must be used.
;  Dummy Handlers are implemented as infinite loops which can be modified.


; -------------------------------------------------------------------------

Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP                             ; Reserved Vector
                LDR     PC, IRQ_Addr
                LDR     PC, FIQ_Addr
; -------------------------------------------------------------------------



                IF      IntVT_SETUP <> 0

;Interrupt Vector Table Address
HandleEINT0       EQU    IntVTAddress
HandleEINT1       EQU    IntVTAddress +4
HandleEINT2       EQU    IntVTAddress +4*2
HandleEINT3       EQU    IntVTAddress +4*3
HandleEINT4_7     EQU    IntVTAddress +4*4
HandleEINT8_23    EQU    IntVTAddress +4*5
HandleReserved    EQU    IntVTAddress +4*6
HandleBATFLT      EQU    IntVTAddress +4*7
HandleTICK        EQU    IntVTAddress +4*8
HandleWDT         EQU    IntVTAddress +4*9
HandleTIMER0      EQU    IntVTAddress +4*10
HandleTIMER1      EQU    IntVTAddress +4*11
```

```
HandleTIMER2      EQU    IntVTAddress +4*12
HandleTIMER3      EQU    IntVTAddress +4*13
HandleTIMER4      EQU    IntVTAddress +4*14
HandleUART2       EQU    IntVTAddress +4*15
HandleLCD         EQU    IntVTAddress +4*16
HandleDMA0        EQU    IntVTAddress +4*17
HandleDMA1        EQU    IntVTAddress +4*18
HandleDMA2        EQU    IntVTAddress +4*19
HandleDMA3        EQU    IntVTAddress +4*20
HandleMMC         EQU    IntVTAddress +4*21
HandleSPI0        EQU    IntVTAddress +4*22
HandleUART1       EQU    IntVTAddress +4*23
;HandleReserved       EQU    IntVTAddress +4*24
HandleUSBD        EQU    IntVTAddress +4*25
HandleUSBH        EQU    IntVTAddress +4*26
HandleIIC         EQU    IntVTAddress +4*27
HandleUART0       EQU    IntVTAddress +4*28
HandleSPI1        EQU    IntVTAddress +4*39
HandleRTC         EQU    IntVTAddress +4*30
HandleADC         EQU    IntVTAddress +4*31

; ---------------------------------------------------------------------------
; 分析寄存器过程
IRQ_Entry
                sub sp,sp,#4       ;reserved for PC
                stmfd   sp!,{r8-r9}

                ldr r9,=INTOFFSET  ;中断序号
                ldr r9,[r9]
                ldr r8,=HandleEINT0    ;中断扩展首地址
                add r8,r8,r9,lsl #2    ;偏移量*4 + 基地址
                ldr r8,[r8]            ;Eintx_Entry Add -> r8
                str r8,[sp,#8]         ;r8 -> sp(-#4)
                ldmfd   sp!,{r8-r9,pc}  ;Eintx_Entry Add -> pc

; ---------------------------------------------------------------------------


                ENDIF

Reset_Addr      DCD     Reset_Handler
Undef_Addr      DCD     Undef_Handler
SWI_Addr        DCD     SWI_Handler
PAbt_Addr       DCD     PAbt_Handler
DAbt_Addr       DCD     DAbt_Handler
                DCD     0                       ; Reserved Address
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler

Undef_Handler   B       Undef_Handler
SWI_Handler     B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler

                IF      IntVT_SETUP <> 1
IRQ_Handler     B       IRQ_Handler
                ENDIF
```

```
                IF      IntVT_SETUP <> 0
IRQ_Handler     B       IRQ_Entry
                ENDIF

FIQ_Handler     B       FIQ_Handler



; Memory Controller Configuration
                IF      MC_SETUP <> 0
MC_CFG
                DCD     BWSCON_Val
                DCD     BANKCON0_Val
                DCD     BANKCON1_Val
                DCD     BANKCON2_Val
                DCD     BANKCON3_Val
                DCD     BANKCON4_Val
                DCD     BANKCON5_Val
                DCD     BANKCON6_Val
                DCD     BANKCON7_Val
                DCD     REFRESH_Val
                DCD     BANKSIZE_Val
                DCD     MRSRB6_Val
                DCD     MRSRB7_Val
                ENDIF


; Clock Management Configuration
                IF      CLK_SETUP <> 0
CLK_CFG
                DCD     LOCKTIME_Val
                DCD     CLKDIVN_Val
                DCD     MPLLCON_Val
                DCD     UPLLCON_Val
                DCD     CLKSLOW_Val
                DCD     CLKCON_Val
                ENDIF


; Reset Handler

                EXPORT  Reset_Handler
Reset_Handler

                IF      WT_SETUP <> 0
                LDR     R0, =WT_BASE
                LDR     R1, =WTCON_Val
                LDR     R2, =WTDAT_Val
                STR     R2, [R0, #WTCNT_OFS]
                STR     R2, [R0, #WTDAT_OFS]
                STR     R1, [R0, #WTCON_OFS]
                ENDIF


                IF      CLK_SETUP <> 0
                LDR     R0, =CLK_BASE
                ADR     R8, CLK_CFG
                LDMIA   R8, {R1-R6}
```

```
                STR     R1, [R0, #LOCKTIME_OFS]
                STR     R2, [R0, #CLKDIVN_OFS]
                STR     R3, [R0, #MPLLCON_OFS]
                STR     R4, [R0, #UPLLCON_OFS]
                STR     R5, [R0, #CLKSLOW_OFS]
                STR     R6, [R0, #CLKCON_OFS]
                ENDIF


                IF      MC_SETUP <> 0
                ADR     R14, MC_CFG
                LDMIA   R14, {R0-R12}
                LDR     R14, =MC_BASE
                STMIA   R14, {R0-R12}
                ENDIF


                IF      PIO_SETUP <> 0
                LDR     R14, =PIO_BASE

                IF      PIOA_SETUP <> 0
                LDR     R0, =PCONA_Val
                STR     R0, [R14, #PCONA_OFS]
                ENDIF

                IF      PIOB_SETUP <> 0
                LDR     R0, =PCONB_Val
                LDR     R1, =PUPB_Val
                STR     R0, [R14, #PCONB_OFS]
                STR     R1, [R14, #PUPB_OFS]
                ENDIF

                IF      PIOC_SETUP <> 0
                LDR     R0, =PCONC_Val
                LDR     R1, =PUPC_Val
                STR     R0, [R14, #PCONC_OFS]
                STR     R1, [R14, #PUPC_OFS]
                ENDIF

                IF      PIOD_SETUP <> 0
                LDR     R0, =PCOND_Val
                LDR     R1, =PUPD_Val
                STR     R0, [R14, #PCOND_OFS]
                STR     R1, [R14, #PUPD_OFS]
                ENDIF

                IF      PIOE_SETUP <> 0
                LDR     R0, =PCONE_Val
                LDR     R1, =PUPE_Val
                STR     R0, [R14, #PCONE_OFS]
                STR     R1, [R14, #PUPE_OFS]
                ENDIF

                IF      PIOF_SETUP <> 0
                LDR     R0, =PCONF_Val
                LDR     R1, =PUPF_Val
                STR     R0, [R14, #PCONF_OFS]
                STR     R1, [R14, #PUPF_OFS]
```

```
                ENDIF

                IF      PIOG_SETUP <> 0
                LDR     R0, =PCONG_Val
                LDR     R1, =PUPG_Val
                STR     R0, [R14, #PCONG_OFS]
                STR     R1, [R14, #PUPG_OFS]
                ENDIF

                IF      PIOH_SETUP <> 0
                LDR     R0, =PCONH_Val
                LDR     R1, =PUPH_Val
                STR     R0, [R14, #PCONH_OFS]
                STR     R1, [R14, #PUPH_OFS]
                ENDIF

                ENDIF

; ini sdram

                BL INISDRAM


; Setup Stack for each mode

                LDR     R0, =Stack_Top

;  Enter Undefined Instruction Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_UND:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #UND_Stack_Size

;  Enter Abort Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_ABT:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #ABT_Stack_Size

;  Enter FIQ Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #FIQ_Stack_Size

;  Enter IRQ Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #IRQ_Stack_Size

;  Enter Supervisor Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #SVC_Stack_Size

;  Enter User Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_USR
                MOV     SP, R0
                SUB     SL, SP, #USR_Stack_Size
```

```
; Enter the C code
; ！！！注意这里的__main是跳转到arm库函数中的init，然后再到用户写的main
                IMPORT  __main
                LDR     R0, =__main
                BX      R0


; User Initial Stack & Heap
                AREA    |.text|, CODE, READONLY

                IMPORT  __use_two_region_memory
                EXPORT  __user_initial_stackheap
__user_initial_stackheap

                LDR     R0, =  Heap_Mem
                LDR     R1, =(Stack_Mem + USR_Stack_Size)
                LDR     R2, = (Heap_Mem +      Heap_Size)
                LDR     R3, = Stack_Mem
                BX      LR


                END
```

Stack/Heap Definition

```
UND_Stack_Size      EQU     0x00000000
SVC_Stack_Size      EQU     0x00000008
ABT_Stack_Size      EQU     0x00000000
FIQ_Stack_Size      EQU     0x00000000
IRQ_Stack_Size      EQU     0x00000400
USR_Stack_Size      EQU     0x00010000

Stack_Size      EQU     (UND_Stack_Size + SVC_Stack_Size\
     + ABT_Stack_Size + FIQ_Stack_Size + IRQ_Stack_Size \
                + USR_Stack_Size)

Heap_Size       EQU     0x00010000
```

Clock

```
CLK_BASE                EQU     0x4C000000          ; Base Address
LOCKTIME_OFS        EQU     0x00                ; LOCKTIME Offset
MPLLCON_OFS     EQU     0x04            ; MPLLCON Offset
UPLLCON_OFS         EQU     0x08                ; UPLLCON Offset
CLKCON_OFS          EQU     0x0C                ; CLKCON Offset
CLKSLOW_OFS         EQU     0x10                ; CLKSLOW Offset
CLKDIVN_OFS         EQU     0x14                ; CLDKIVN Offset

CLK_SETUP               EQU     1
MPLLCON_Val         EQU     0x0005C080
UPLLCON_Val         EQU     0x00028080
CLKCON_Val          EQU     0x0007FFF0
CLKSLOW_Val         EQU     0x00000004
LOCKTIME_Val        EQU     0x00FFFFFF
CLKDIVN_Val         EQU     0x00000000
```

Watchdog Timer

```
WT_BASE           EQU     0x53000000      ; WT Base Address
WTCON_OFS       EQU     0x00              ; WTCON Offset
WTDAT_OFS      EQU     0x04             ; WTDAT Offset
WTCNT_OFS       EQU     0x08              ; WTCNT Offset

WT_SETUP            EQU     1
WTCON_Val       EQU     0x00008021
WTDAT_Val           EQU     0x00008000
```

Memory Controller

```
MC_BASE             EQU     0x48000000      ; Base Address
MC_SETUP            EQU     1

BANKCON0_Val    EQU     0x00000700  ; Bank Control
BANKCON1_Val    EQU     0x00000700
BANKCON2_Val    EQU     0x00000700
BANKCON3_Val    EQU     0x00000700
BANKCON4_Val    EQU     0x00000700
BANKCON5_Val    EQU     0x00000700
BANKCON6_Val    EQU     0x00018008
BANKCON7_Val    EQU     0x00018008
BWSCON_Val        EQU     0x00000000    ; Bus Width & Wait Control
REFRESH_Val        EQU     0x00ac0000   ; Refresh
BANKSIZE_Val      EQU     0x00000000    ; Banksize
MRSRB6_Val         EQU     0x00000020   ; SDRAM Mode Register Set
MRSRB7_Val         EQU     0x00000000
```

IO PORT

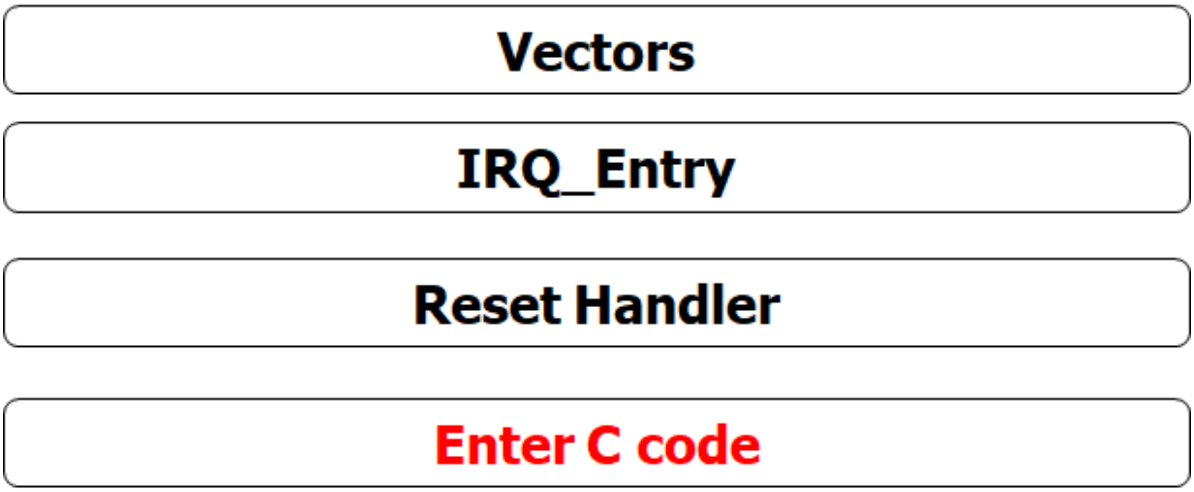
```
PIO_BASE            EQU     0x56000000          ; PIO Base Address
PCONA_OFS           EQU     0x00        ; PCONA Offset
……
PCONJ_OFS           EQU     0xD0    ; PCONJ Offset
PUPB_OFS            EQU     0x18                ; PUPB Offset
……
PUPJ_OFS            EQU     0xD8                ; PUPJ Offset
PIO_SETUP           EQU     1

PIOA_SETUP      EQU     0
PCONA_Val           EQU     0x000003FF

PIOB_SETUP      EQU     0
PCONB_Val           EQU     0x000007FF
PUPB_Val            EQU     0x00000000
```

Area, Stack/Heap

```
        AREA    STACK, NOINIT, READWRITE, ALIGN=3

Stack_Mem       SPACE   Stack_Size

Stack_Top           EQU     Stack_Mem + Stack_Size

        AREA        HEAP, NOINIT, READWRITE, ALIGN=3
Heap_Mem        SPACE   Heap_Size
```

- Arm栈满减

RESET



Vectors

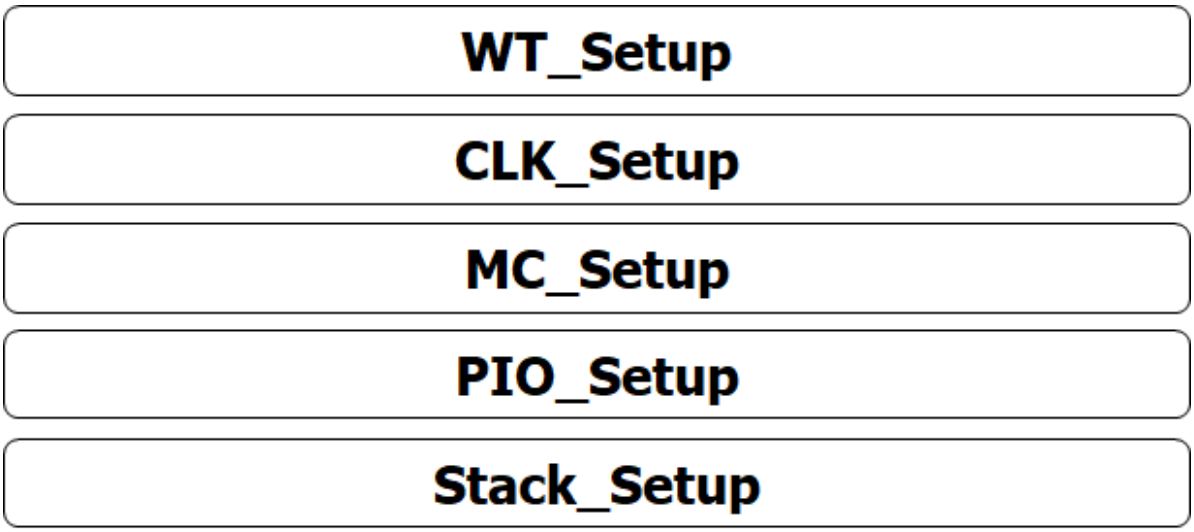
```
Vectors

    LDR     PC, Reset_Addr
            LDR     PC, Undef_Addr
            LDR     PC, SWI_Addr
            LDR     PC, PAbt_Addr
            LDR     PC, DAbt_Addr
            NOP                             ; Reserved Vector
            LDR     PC, IRQ_Addr
            LDR     PC, FIQ_Addr
```

IRQ_Entry

```
IRQ_Entry
                    sub sp,sp,#4 ;reserved for PC
                stmfd   sp!,{r8-r9}

                ldr r9,=INTOFFSET
                ldr r9,[r9]
                ldr r8,=HandleEINT0
                add r8,r8,r9,lsl #2
                ldr r8,[r8]
                str r8,[sp,#8]

                ldmfd   sp!,{r8-r9,pc}
```

> LDMFD->LDMIA
>
> STMFD->STMDB
>
> 只用r8，r9可以少备份

Reset_Handler



WT_Setup MC_Setup

```
    IF      WT_SETUP <> 0
    LDR     R0, =WT_BASE
        LDR     R1, =WTCON_Val
    LDR     R2, =WTDAT_Val
            STR     R2, [R0, #WTCNT_OFS]
            STR     R2, [R0, #WTDAT_OFS]
            STR     R1, [R0, #WTCON_OFS]
    ENDIF

    IF      MC_SETUP <> 0
    ADR         R14, MC_CFG
            LDMIA  R14, {R0-R12}
            LDR         R14, =MC_BASE
            STMIA  R14, {R0-R12}
    ENDIF
```

CLK_Setup

```
    IF      CLK_SETUP <> 0
    LDR     R0, =CLK_BASE
    ADR     R8, CLK_CFG
    LDMIA   R8, {R1-R6}
            STR     R1, [R0, #LOCKTIME_OFS]
            STR     R2, [R0, #CLKDIVN_OFS]
            STR     R3, [R0, #MPLLCON_OFS]
            STR     R4, [R0, #UPLLCON_OFS]
            STR     R5, [R0, #CLKSLOW_OFS]
            STR     R6, [R0, #CLKCON_OFS]
    ENDIF
```

PIO_Setup

```
    IF      PIO_SETUP <> 0
    LDR     R14, =PIO_BASE
    IF      PIOA_SETUP <> 0
    LDR     R0, =PCONA_Val
    STR     R0, [R14, #PCONA_OFS]
    ENDIF
    IF      PIOB_SETUP <> 0
            LDR     R0, =PCONB_Val
        LDR     R1, =PUPB_Val
        STR     R0, [R14, #PCONB_OFS]
        STR     R1, [R14, #PUPB_OFS]
            ENDIF
    ……
    ENDIF
```

Stack_Setup

```
; Setup Stack for each mode
            LDR     R0, =Stack_Top
;  Enter Undefined Instruction Mode and set its Stack Pointer
    MSR     CPSR_c, #Mode_UND:OR:I_Bit:OR:F_Bit
            MOV     SP, R0
            SUB         R0, R0, #UND_Stack_Size
```

Enter C Code

```
    IMPORT      __main
            LDR             R0, =__main
            BX              R0
```

> Msr： 通用寄存器写到状态寄存器
>
> Mrs：反过来
>
> __mian调用arm库函数
>
> __main -> init -> main

__user_initial_stackheap

```
    EXPORT  __user_initial_stackheap
__user_initial_stackheap
                LDR     R0, =  Heap_Mem
                LDR     R1, =(Stack_Mem + USR_Stack_Size)
                LDR     R2, = (Heap_Mem +      Heap_Size)
                LDR     R3, = Stack_Mem
                BX          LR
```

__user_initial_stackheap(C)

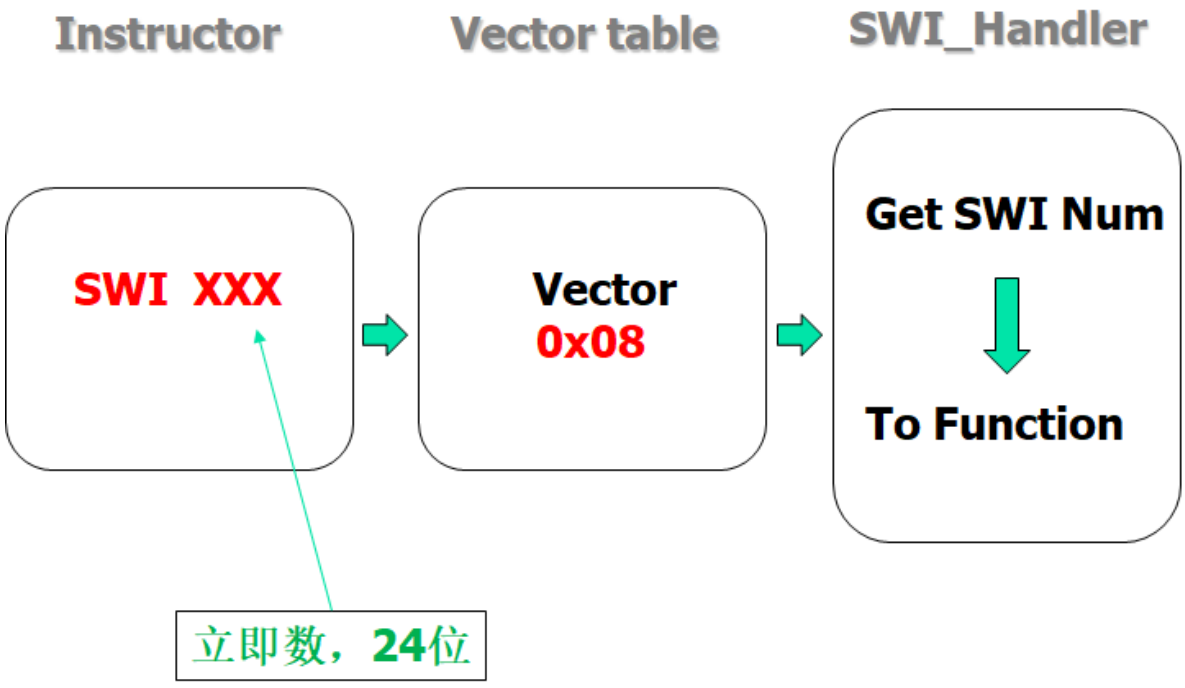
```
 struct __initial_stackheap {
    unsigned heap_base;
        unsigned stack_base;
    unsigned heap_limit;
    unsigned stack_limit;
    };

_value_in_regs struct __initial_stackheap
    __user_initial_stackheap(void);
/*
        __user_initial_stackheap(unsigned SP)
     {
    struct __initial_stackheap config;
    config.stack_base = SP;
    config.stack_limit = SP-STACK_SIZE;
    config.heap_base  = (unsigned)(USER_HEAP_ADDRESS)
    config.heap_limit = ((unsigned)(USER _HEAP_ADDRESS))
        +USER_SRAM_SIZE;
    return config;
}
*/
```

如果调试时无法进入main，可能的原因是什么?

软中断

如何通过软中断方式调用一个函数，该函数计算4个整数的和。

过程



> SWI：软中断指令
>
> XXX：中断信号

SWI_Handler

Get SWI Num

```
        AREA TopSwi, CODE, READONLY
        IMPORT      C_SWI_Handler
        EXPORT      SWI_Handler
SWI_Handler
        STMFD      sp!,{r0-r12,lr}
        LDR        r0,[lr,#-4]          ;获取 SWI 指令
        BIC        r0,r0,#0xff000000    ; 参数1，NUM
        MOV         R1, SP                  ;参数2，传递堆栈指针
        BL C_SWI_Handler                ;To Function
        LDMFD        sp!, {r0-r12,pc}^
        END
```

> r0,[lr,#-4] 把swi指令读到r0
>
> BIC清除高8位，得到中断号
>
> 这里用BL是因为lr已经备份到栈里面了
>
> 通常减8获取上一条指令，SWI特殊需要减4

Function (C)

```
void C_SWI_handler (int swi_num, int *reg )
{   switch (swi_num)
    {
        case 0 :
            ……                /* SWI number 0 code */
                    break;
            case 1 :
            ……           /* SWI number 1 code */
                    break;
        ……
            default :
            break; /* Unknown SWI - report error */
    }
    return;
}
```

> R0:中断号
>
> R1:栈指针

Function (asm)

```
        AREA SecondSwi, CODE, READONLY
         EXPORT     C_SWI_Handler
C_SWI_Handler
        STMFD      sp!,{r0-r12,lr}
    CMP    r0,#MaxSWI          ; Range check?
        LDRLE  pc, [pc,r0,LSL #2] ;(PC-〉DCD SWInum0)
    B       SWIOutOfRange
SWIJumpTable    DCD    SWInum0
        DCD    SWInum1
```

```
SWInum0   ; SWI number 0 code
            B     EndofSWI
SWInum1   ; SWI number 1 code
            B     EndofSW
EndofSW SUB lr, lr, #4
        LDMFD         sp!, {r0-r12,pc}^
        END
```

SWI异常

- 汇编

```
MOV   r0, #0x18
LDR   r1, =0x20026
SWI   0x123456
NOP
```

注：在实验中发现，如果紧随SWI后是无效指令（例如：literal pool），则出现异常，不能正确执行SWI。为什么？

- C

> Swi（0）关键字，产生软中断，0为中断信号
>
> 函数参数存到r0

函数

声明

```
__swi(0) int add_four(int, int, int, int);    〈- 寄存器传数据
```

调用

```
#include <stdio.h>
#include "swi.h"
unsigned *swi_vec = (unsigned *)0x08;
extern void SWI_Handler(void);
int main( void )
{
    int res;

    Install_Handler( (unsigned) SWI_Handler, swi_vec );

    res=add_four(3,4,5,6);
```

```
    return 0;
}
```

- Install_Handler注册中断服务程序

函数体

```
void C_SWI_Handler( int swi_num, int *ptr )
{
    switch( swi_num )
    {
    case    0:
        ptr[0] = ptr[0] +ptr[1]+ptr[2]+ptr[3];
            break;

    case   1:
        //(next page)
            break;
    }
}
```

- 之所以用栈指针是因为在软中断发生r0到r12和lr都存到了栈中，最低的地址就是r0；且返回结果就是栈底，即r0

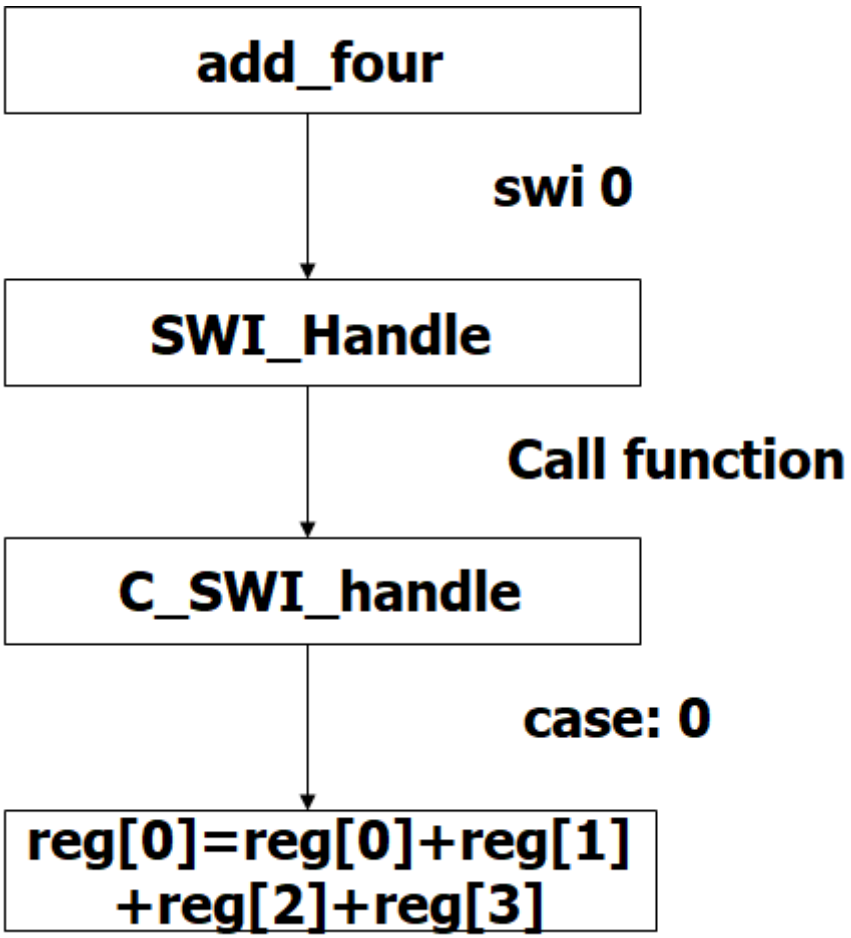
返回结构体

```
struct four_results
{
    int a;
    int b;
    int c;
    int d;
};

__swi(1)  struct four_results many_operations(int, int, int, int);
```

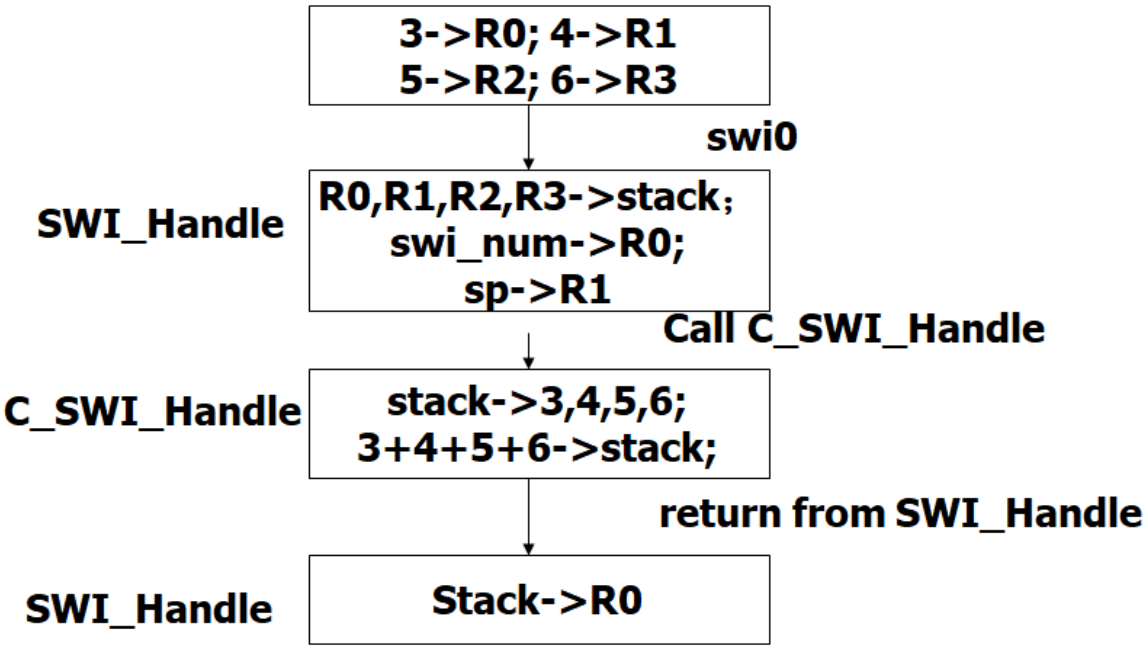
函数体

```
case    1:

    int w, x, y, z;

    w = ptr[0];
            x = ptr[1];
            y = ptr[2];
            z = ptr[3];

        ptr[0] = w + x + y + z;
            ptr[1] = w - x - y - z;
            ptr[2] = w * x * y * z;
            ptr[3] =(w + x) * (y - z);
```

add_four



参数传递和结果返回



Build

- armasm
- input file
- -o filename //output file name
- -16|-32 //Thumb or ARM Instruction
- -cpu cpu // Set the target CPU

- -bigend|-littleend
- -FPU name // target float-point unit
- -apcs // ARM/Thumb Procedure Call Standard

> 编译时的-16 | -32 与code里的命令不一样时，优先code

armlink

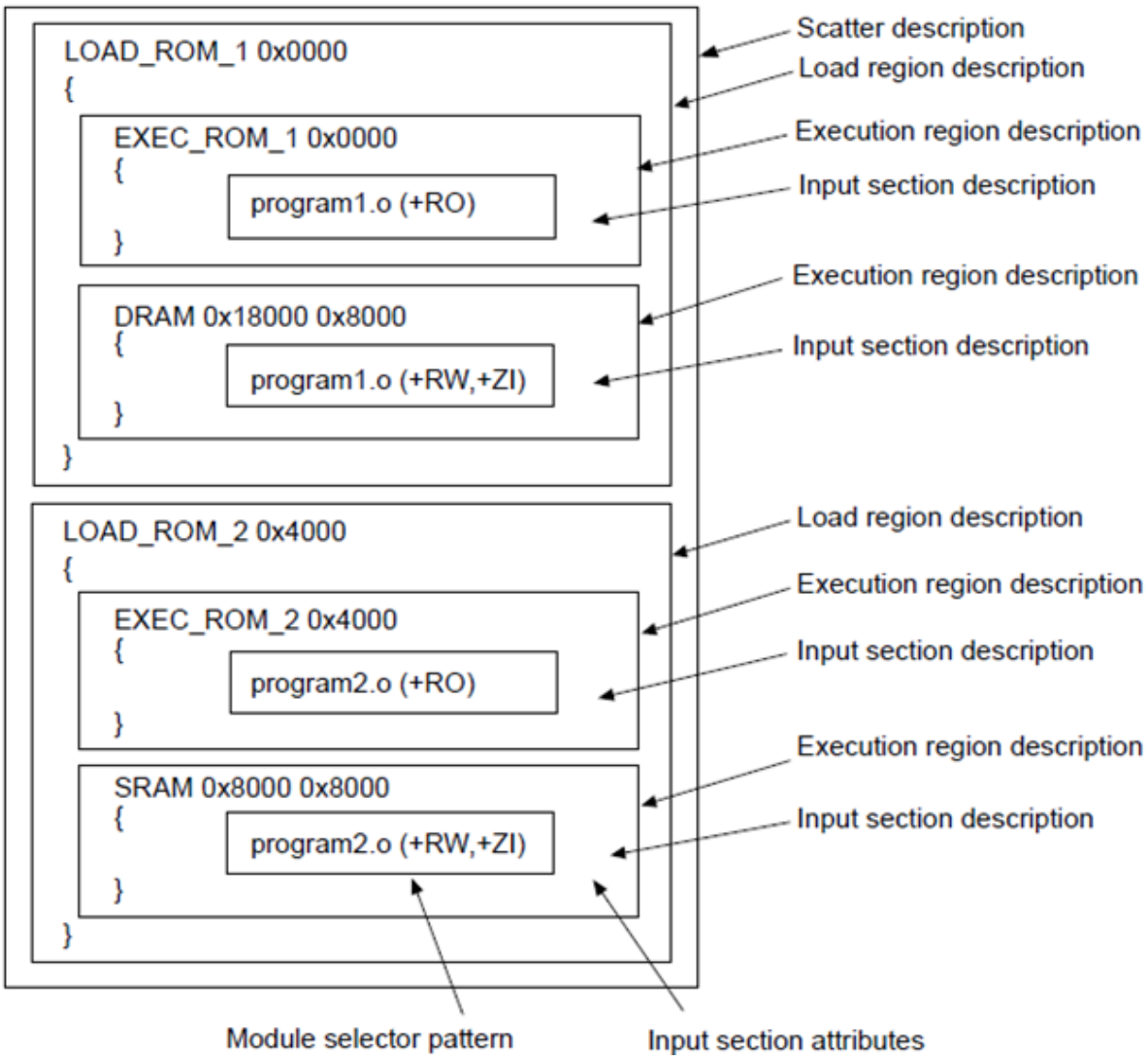
- -output file
  - 指定了输出文件名，该文件可能是部分链接的目标文件，也可能是可执行映像文件。
- -elf
  - 生成ELF格式的映像文件，armlink所支持的唯一的一种输出格式。
- -reloc
  - 生成可重定址的映像。程序地址使用相对地址
- -ro-base address
  - 包含有RO(Read-Only属性)输出段的加载地址和运行地址设置为address.
- -ropi
  - 使得包含有RO输出段的加载域和运行域是位置无关的。
- rw-base address
  - 设置包含RW(Read/Write属性)输出段的域的运行时地址.
- -rwpi
  - 使得包含有RW和ZI(Zero Initialization，初始化为0)属性的输出段的加载和运行时域为位置无关的。
- -split
  - 将包含RO和RW属性的输出段的加载域，分割成2个加载域。
- -scatter file
  - 使用在file中包含的分组和定位信息来创建映像内存映射。
- -debug
  - 使输出文件包含调试信息，调试信息包括，调试输入段，符号和字符串表。
- -entry location
  - 指定映像文件中唯一的初始化入口点。
    - 数值地址，如-entry 0x0；
    - 符号所代表的地址处，如：-entry int_handler
    - 段内偏移量，如：-entry offset+object(section)
- -first section-id
  - 将被选择的输入段放在运行域的开始
    - symbol 选择定义symbol的段。
    - object(section) 从目标文件中选择段放在映像文件的开始位置。例如： -first init.o(init)
    - object 选择只有一个输入段的目标文件。如： -first init.o
- -last section-id
  - 将所选择的输入段放在运行域的最后
- -map
  - 创建映像文件的信息图

Scatter

如何理解和编写.sct文件?

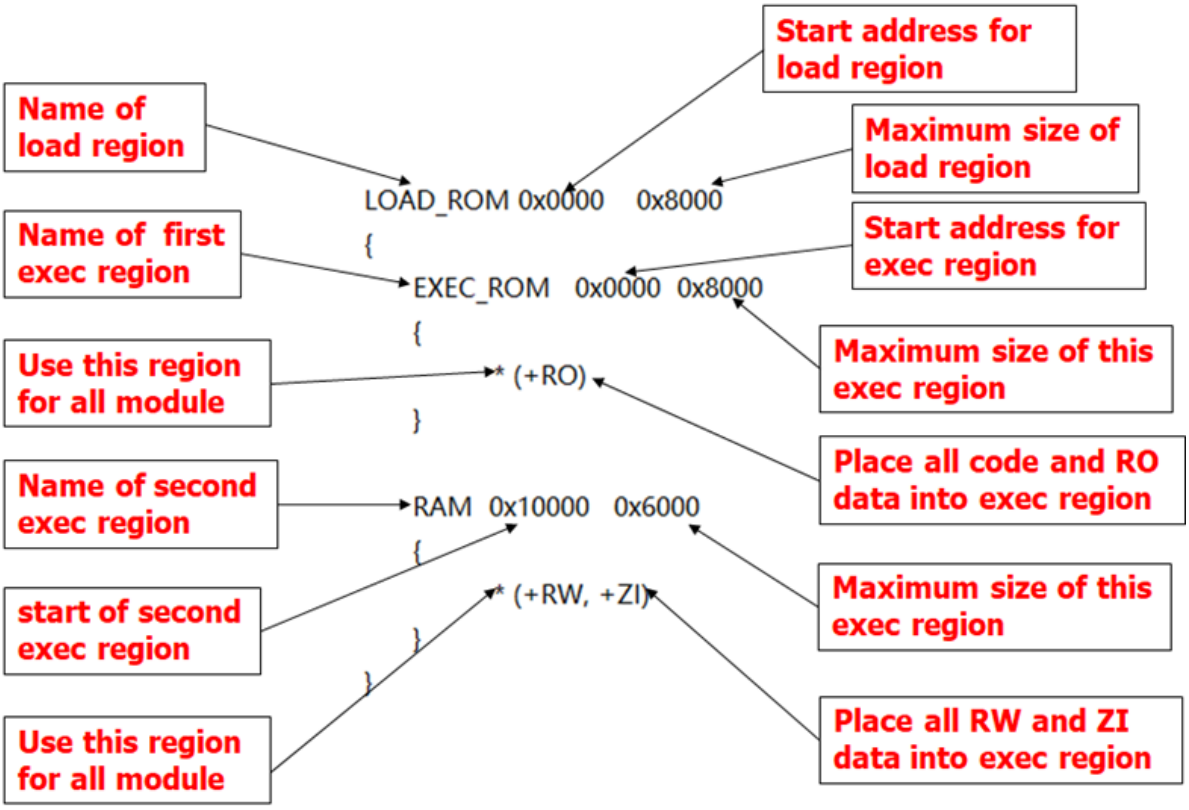
> 分散加载文件
>
> 使用者：liner用的

组成

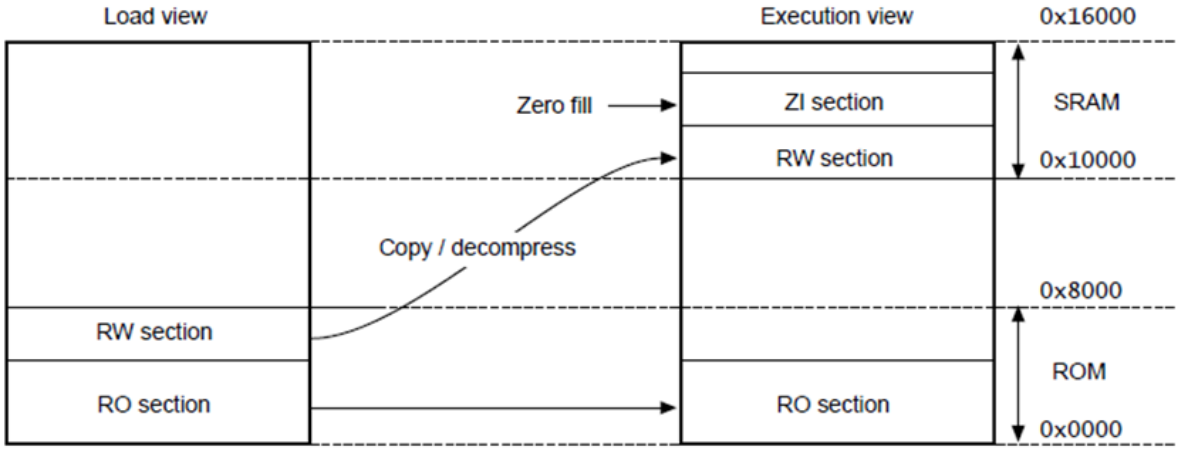


示例一：

文件



映射



- A root region is a region with the same load and execution address.
- The initial entry point of the image must be in a root region.
- If the initial entry point is not in a root region, the link fails and the linker gives an error message.

示例二：

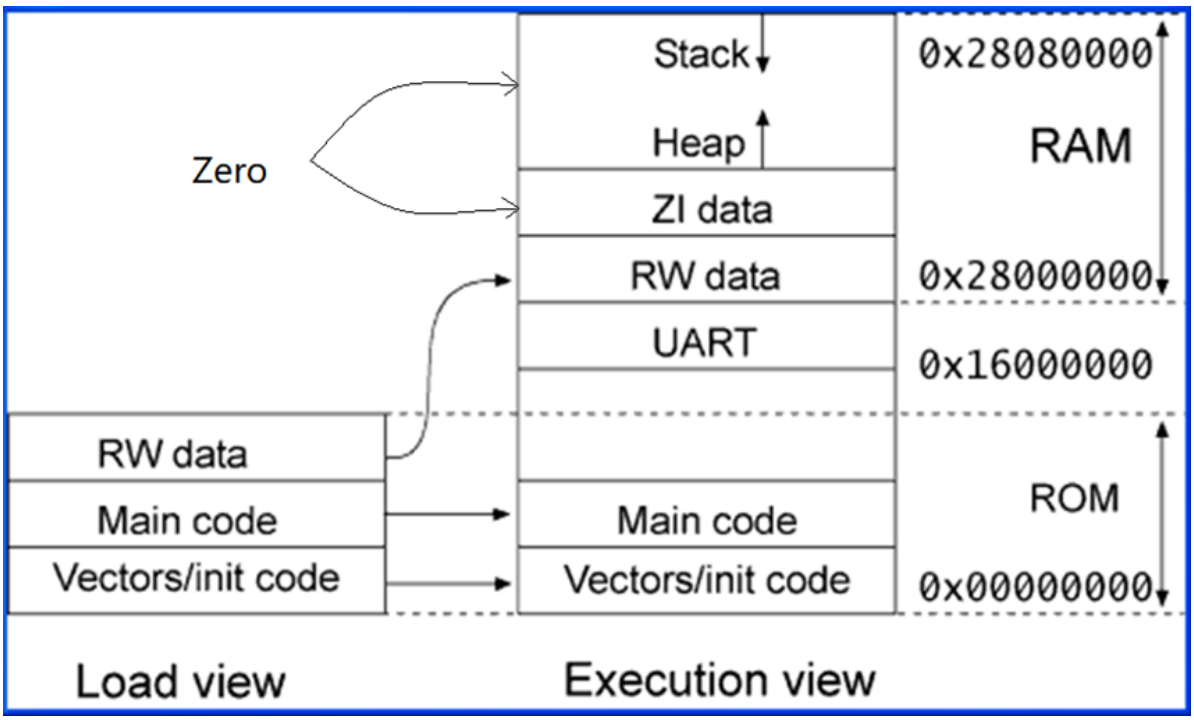
文件

```
ROM_LOAD 0x0
{
    ROM_EXEC 0x0
    {    vectors.o (Vect, +First)
                * (+RO)
    }
    RAM 0x28000000      FIXED
    {   * (+RW,+ZI) }
```

```
    HEAP +0 UNINIT
    {   heap.o (+ZI) }
    STACKS 0x28080000 UNINIT
    {   stack.o (+ZI) }
    UART0 0x16000000 UNINIT
     {  uart.o (+ZI) }
   }
```

> * (+RO) 所有RO属性的值
>
> UART0

映射



At a specific address without scatter-loading

main.c

```c
#include <stdio.h>
extern int sqr(int n1);
int gSquared __attribute__((at(0x5000))); //Place at 0x5000
int main()
{
     gSquared=sqr(3);
     printf("Value squared is: %d\n", gSquared);
}
```

function.c

```c
int sqr(int n1)
{
      return n1*n1;
}
```

- 栈里全局变量，全局变量可以直接指定地址；局部变量需要用栈

In a named section with scatter-loading

main.c

```
#include <stdio.h>
extern int sqr(int n1);
int gsquaed__attribute__((section(“foo”))); //place in section “foo”
int main()
{
     gSquared=sqr(3);
     printf("Value squared is: %d\n", gSquared);
}
```

scatter.sct

```
LR1 0x0000 0x20000
{  ……
   ER3 0x10000 0x2000{
     function.o
     *(foo) ; Place gSquared in ER3
    }
     ……}
```

At a specific address with scatter-loading

main.c

```
#include <stdio.h>
extern int sqr(int n1);
const int gValue __attribute__((section(".ARM.__at_0x10000"))) = 3;
int main()
{
     gSquared=sqr(gValue);
     printf("Value squared is: %d\n", gSquared);
}
```

scatter.sct

```
LR1 0x0000 0x20000
{  ……
   ER2 +0
   {
    function.o
    *(.ARM.__at_0x10000) ; Place gValue at 0x10000
    }
     ……}
```

Placement a function a specific address

function.c

```
int sqr(int n1) __attribute__((section(".ARM.__at_0x20000")));
int sqr(int n1)
{
      return n1*n1;
}

// place flash_key in a section called .ARM.__at_0x8000
long flash_key __attribute__((section(".ARM.__at_0x8000")));
```

scatter.sct

```
ER_FLASH 0x8000 0x2000
{
        *(+RW)
       *(.ARM.__at_0x8000) ; key
}
```

> C代码全局变量的设置地址一般放在sct文件中设置

place a named section with scatter-loading

init.c

```
int foo() __attribute__((section("INIT")));
int foo() {  return 1; }
int bar() {  return 2; }
```

data.c

```
const long padding=123;
int z=5;
```

scatter.sct

```
LR1 0x0 0x10000
{ ; Root Region, containing init code
      ER1 0x0 0x2000
     {
            init.o (INIT, +FIRST) ; place init code at exactly 0x0
    *(+RO) ; rest of code and read-only data
     }
; RW & ZI data to be placed at 0x400000
     RAM_RW 0x400000 (0x1FF00-0x2000)
     {   *(+RW)  }
    RAM_ZI +0 { *(+ZI)
      }
; execution region at 0x1FF00, maximum space available for table is 0xFF
    DATABLOCK 0x1FF00 0xFF
    { data.o(+RO-DATA) ; place RO data between 0x1FF00 and 0x1FFFF}
}
```

Placement of unassigned sections with the .ANY

scatter.sct

```
lr1 0x8000 1024
{
    er1 +0 512
    {
        .ANY1(+RO) ; evenly distributed with er3
    }
    er2 +0 256
    {
        .ANY2(+RO) ; Highest priority, so filled first
    }
    er3 +0 256
    {
        .ANY1(+RO) ; evenly distributed with er1
    }
}
```

Placement of sections with overlays

scatter.sct

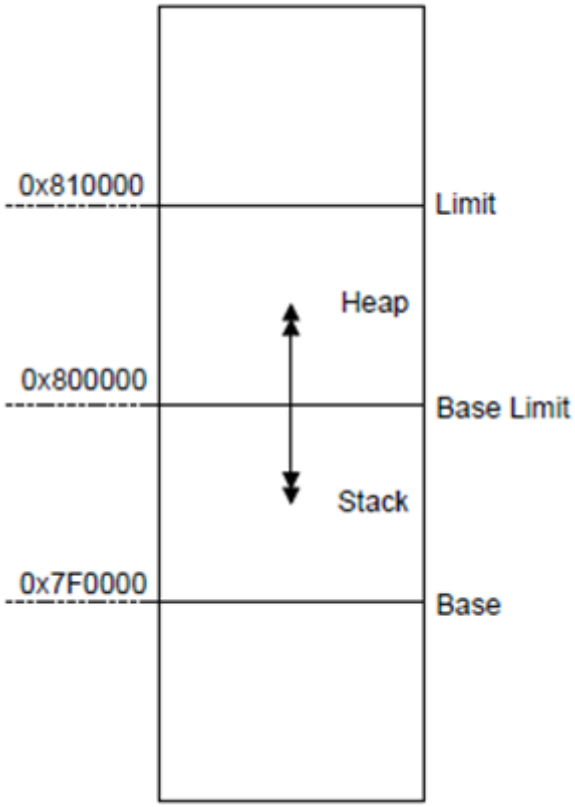
```
EMB_APP 0x8000
{
    …
    STATIC_RAM 0x0 ; contains most of the RW and ZI code/data
    {
        * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY ; start address of overlay…
    {
        module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
        module2.o (+RW,+ZI)
    }
    … ; rest of scatter-loading description
}
```

Reserving an empty region

scatter.sct

```
EMB_APP 0x8000
{
    STACK 0x800000 EMPTY -0x10000 ; region  ;ends at 0x800000 because of the
negative
    ; length. The start of the region is    ;calculated using the length.
    {
    ; Empty region for placing the stack
    }
    HEAP +0 EMPTY 0x10000 ;
```

```
    ;region starts at the end of previous
    ; region. End of region calculated using
    ; positive length
    {
    ; Empty region for placing the heap
    }
    … ; rest of scatter-loading description
}
```



placing ARM C library code

scatter.sct

```
LR1 0x0
{
    ROM1 0
    {   * (InRoot$$Sections)
    * (+RO)
    }
    ROM2 0x1000
    {   *armlib/c_* (+RO) ; all ARM-supplied C library functions}
    ROM3 0x2000
    {   *armlib/h_* (+RO) ; just the ARM-supplied __ARM_*
    ; redistributable library functions
    }
    RAM1 0x3000
    {   *armlib* (+RO) ; all other ARM-supplied library code
    ; for example, floating-point libraries
     }
     RAM2 0x4000
     { * (+RW, +ZI)  }
}
```

placing ARM C++ library code

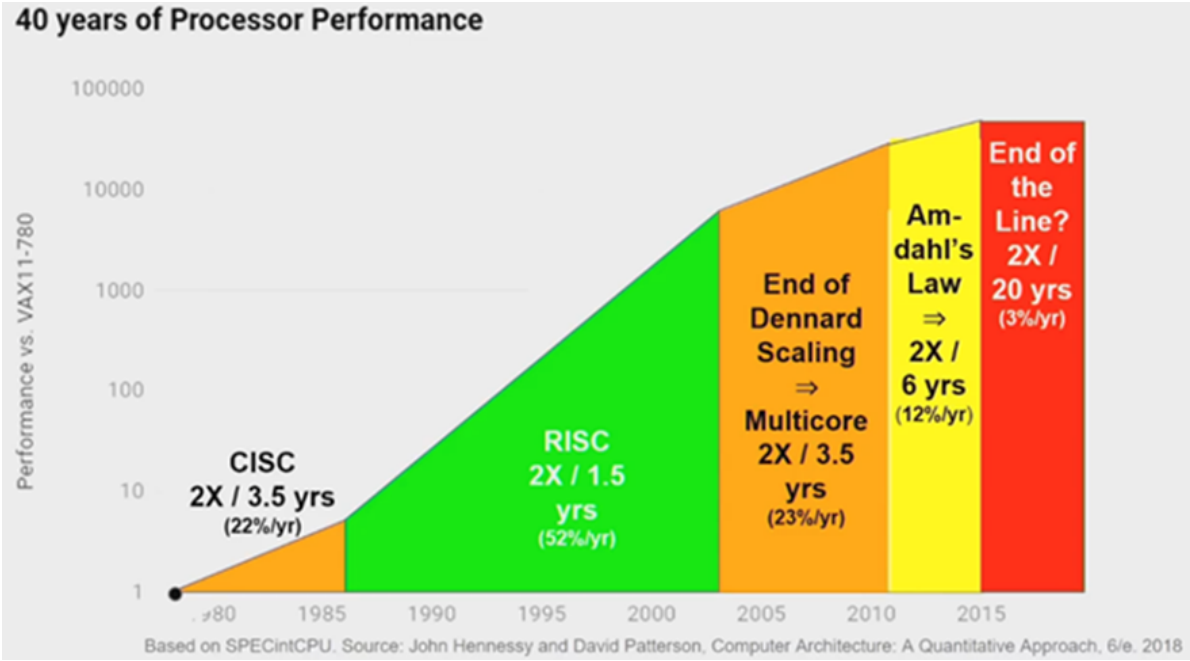
```
LR 0x0
{
    ER1 0x0
    {   *armlib*(+RO)  }
    ER2 +0
    {   *cpplib*(+RO)
    *(.init_array) ; Section .init_array must be placed explicitly,
    ; otherwise it is shared between two regions, and
    ; the linker is unable to decide where to place it.
    }
    ER3 +0
    {   *(+RO) }
    ER4 +0
    {   *(+RW,+ZI) }
}
```

Creation of regions on page boundaries

```
LR1 GetPageSize() + SizeOfHeaders()
{
    ER_RO +0
    {   *(+RO) }
    ER_RW AlignExpr(+0, GetPageSize())
    {   *(+RW)  }
    ER_ZI AlignExpr(+0, GetPageSize())
   {    *(+ZI) }
}
```

程序优化

问题



Hennessy and Patterson ,A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development, lecture, June 3, 2018

- Speed on different environments

Matrix Multiply: relative speedup to a Python version (18 core Intel)

Version	Speed-up	Optimization
Python	1	
C	47	Translate to static, compiled language
C with parallel loops	366	Extract parallelism
C with loops & memory optimization	6,727	Organize parallelism and memory access
Intel AVX instructions	62,806	Use domain-specific HW

概述

程序优化

- 指在不改变程序功能的情况下，根据处理器及系统的特性，通过修改原来程序的算法、结构，或利用软件开发工具对程序进行改进。使修改后的程序运行速度更快或占用空间更小或能耗最低。

优化原则

- 等效原则，优化前后程序实现的功能一致。
- 有效原则，优化后要比优化前运行速度快或占用存储空间小或能耗低，或三者兼有。
- 经济原则，优化程序要付出较小的代价，取得较好的结果。

优化方法

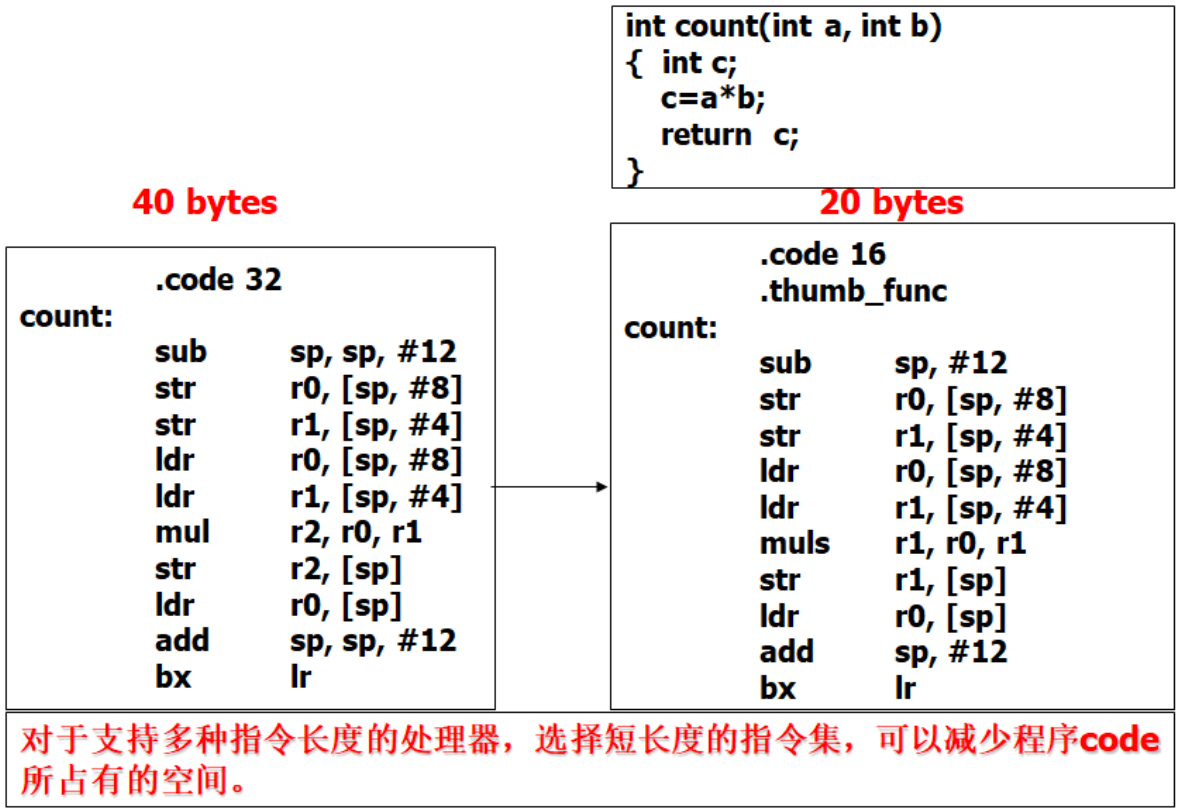
- 算法优化
  - 选择一种高效的算法
  - 对算法进行优化
  - 例：在数据搜索时，二分查找法要比顺序查找法快
- 数据结构优化
  - 采用访问比较快的数据结构
  - 例：在一些无序的数据中多次进行插入、删除数据项操作，那么采用链表结构就会比较快
- 编译优化
  - 编译器有不同级别的优化选项，选用一种合适的优化方式。
  - 针对体系结构进行了优化设计
- 代码优化
  - 采用汇编语言或更精简的程序代码来代替原有的代码

空间优化

Code 空间

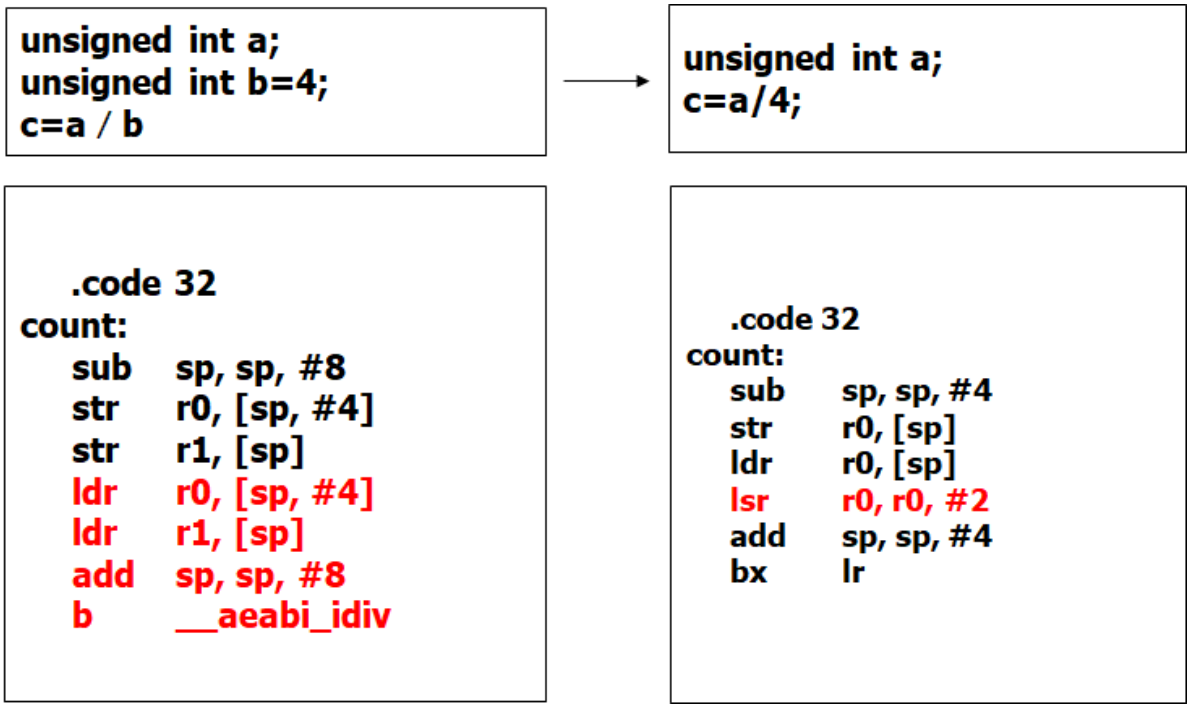
选择短长度指令集

- ARM与THUMB



> 减少rom空间
>
> 通过状态寄存器T来判断哪种指令

运算替换



```
b=pow(a，3);
```

→

```
b=a*a*a;
```

```
        .code 32
count:
        push    {r11, lr}
        sub     sp, sp, #8
        str     r0, [sp, #4]
        str     r1, [sp]
        ldr     r0, [sp, #4]
        ldr     r1, [sp]
        bl      pow
        add     sp, sp, #8
        pop     {r11, lr}
        bx      lr
```

```
        .code 32
count:
        sub     sp, sp, #4
        str     r0, [sp]
        ldr     r0, [sp]
        mul     r1, r0, r0
        mul     r2, r1, r0
        mov     r0, r2
        add     sp, sp, #4
        bx      lr
```

用指令直接支持的计算，替换需要用库函数实现的运算。可以减少程序的指令数。

循环替换

```
unsigned int checksum(unsigned int *a,
                      unsigned int N)
{
  unsigned int check=0;
  unsigned i;
  for(i=0;i<N;i++)
  {
    check+=*a++;
  }
  return  check;
}
```

```
        .code 32
checksum:
        sub     sp, sp, #16
        str     r0, [sp, #12]
        str     r1, [sp, #8]
        mov     r0, #0
        str     r0, [sp, #4]
        str     r0, [sp]
.LBB1_1:
        ldr     r0, [sp]
        ldr     r1, [sp, #8]
        cmp     r0, r1
        bhs     .LBB1_4
        ldr     r0, [sp, #12]
        add     r1, r0, #4
        str     r1, [sp, #12]
        ldr     r0, [r0]
        ldr     r1, [sp, #4]
        add     r0, r1, r0
        str     r0, [sp, #4]
        ldr     r0, [sp]
        add     r0, r0, #1
        str     r0, [sp]
        b       .LBB1_1
.LBB1_4:
        ldr     r0, [sp, #4]
        add     sp, sp, #16
        bx      lr
```

24条

```
unsigned int checksum(unsigned int *a,
                      unsigned int N)
{
        unsigned int check=0;
        do{
            check+=*a++;
           N--;
         }while(N>0);
}
```

```
        .code 32
checksum:
        sub     sp, sp, #16
        str     r0, [sp, #12]
        str     r1, [sp, #8]
        mov     r0, #0
        str     r0, [sp, #4]
.LBB1_1:
        ldr     r0, [sp, #12]
        add     r1, r0, #4
        str     r1, [sp, #12]
        ldr     r0, [r0]
        ldr     r1, [sp, #4]
        add     r0, r1, r0
        str     r0, [sp, #4]
        ldr     r0, [sp, #8]
        sub     r0, r0, #1
        str     r0, [sp, #8]
        ldr     r0, [sp, #8]
        cmp     r0, #0
        bne     .LBB1_1
        ldr     r0, [sp, #4]
        add     sp, sp, #16
        bx      lr
```

21条

用do…… while 替换 for(;;);

数据类型

```
int a;
c=a/4;
```

→

```
unsigned int a;
c=a/4;
```

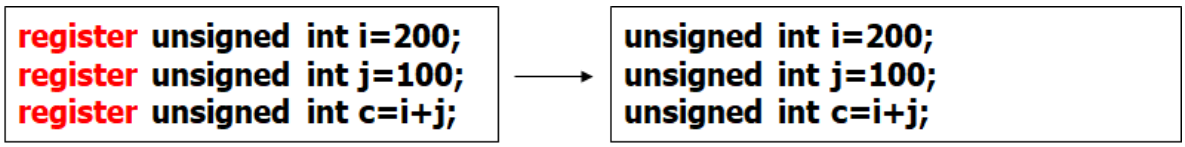
```
    . code 32
count:
    sub   sp, sp, #8
    str   r0, [sp, #4]
    ldr   r0, [sp, #4]
    asr   r1, r0, #31
    add   r0, r0, r1, lsr #30
    asr   r0, r0, #2
    str   r0, [sp]
    ldr   r0, [sp]
    add   sp, sp, #8
    bx    lr
```

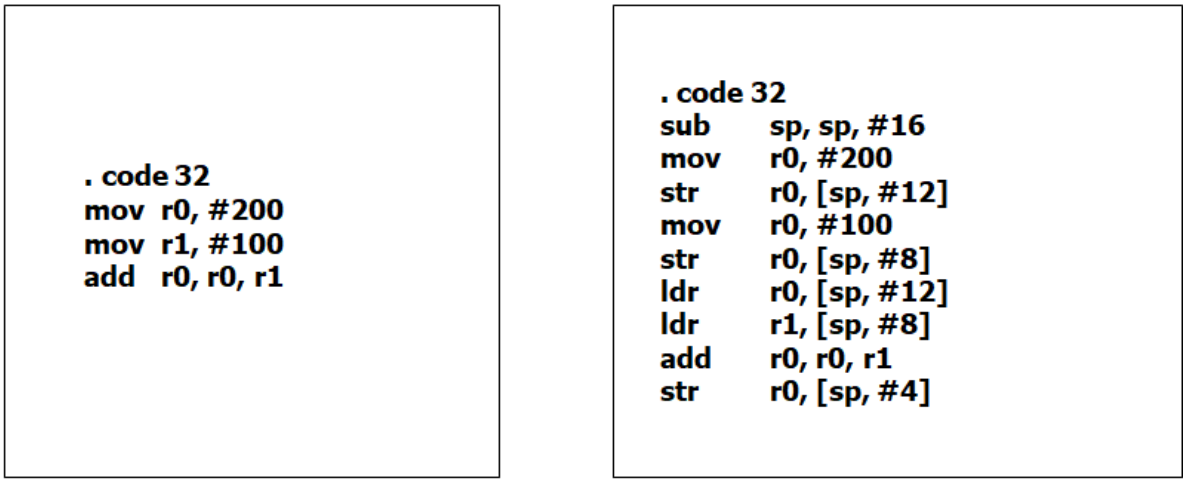
10条

```
    . code 32
count:
    sub   sp, sp, #4
    str   r0, [sp]
    ldr   r0, [sp]
    lsr   r0, r0, #2
    add   sp, sp, #4
    bx    lr
```

6条

尽量使用无符号整数

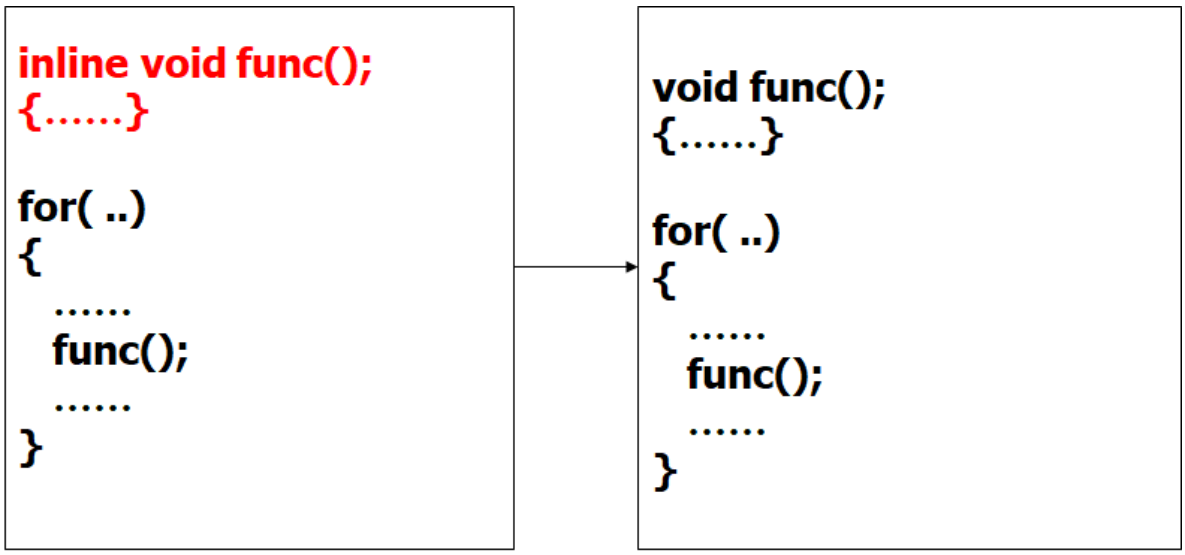




使用 “register”修饰符，将局部变量存到寄存器中；

> - const表示只读
> - Volatile（挥发）直接访问物理单元或IO端口，不是cache，保证读取数据的一致性
> - Register尽量将变量存进寄存器，否则局部变量存进栈

内联函数



取消内联函数，可以减少代码所占有的空间

Data 空间

结构体

- 设处理器数据总线宽度为BUS_W，结构体中成员数据类型的最大宽度为 DATA_W，该结构体的自然边界为N，则：
  - N=min(BUS_W, DATA_W)
- 结构体内：
  - 成员类型宽度小于自然边界，以该类型宽度对齐；

    - 成员类型宽度大于等于自然边界，以自然边界对齐。
- 结构体之间：
    - 结构体的自然边界是该结构体中最大数据成员的自然边界。
    - 结构体占用的空间是其自然边界的整数倍。

```
struct  mydata{
        char a;
        int b;
        char c;
        short d;
}
```

add	+3	+2	+1	0
+0	pad	pad	pad	a[7:0]
+4	b[31:24]	b[23:16]	b[15:8]	b[7:0]
+8	d[15:8]	d[7:0]	pad	c[7:0]

```
struct{
        char a;
        char c;
        short d;
        int b;
}
```

add	+3	+2	+1	0
+0	d[15:8]	d[7:0]	c[7:0]	a[7:0]
+4	b[31:24]	b[23:16]	b[15:8]	b[7:0]

数据压缩

```
_packed struct{
        char a;
        int b;
        char c ;
        shrot d;
}
```

add	+3	+2	+1	0
+0	b[23:16]	b[15:8]	b[7:0]	a[7:0]
+4	d[15:8]	d[7:0]	c[7:0]	b[31:24]

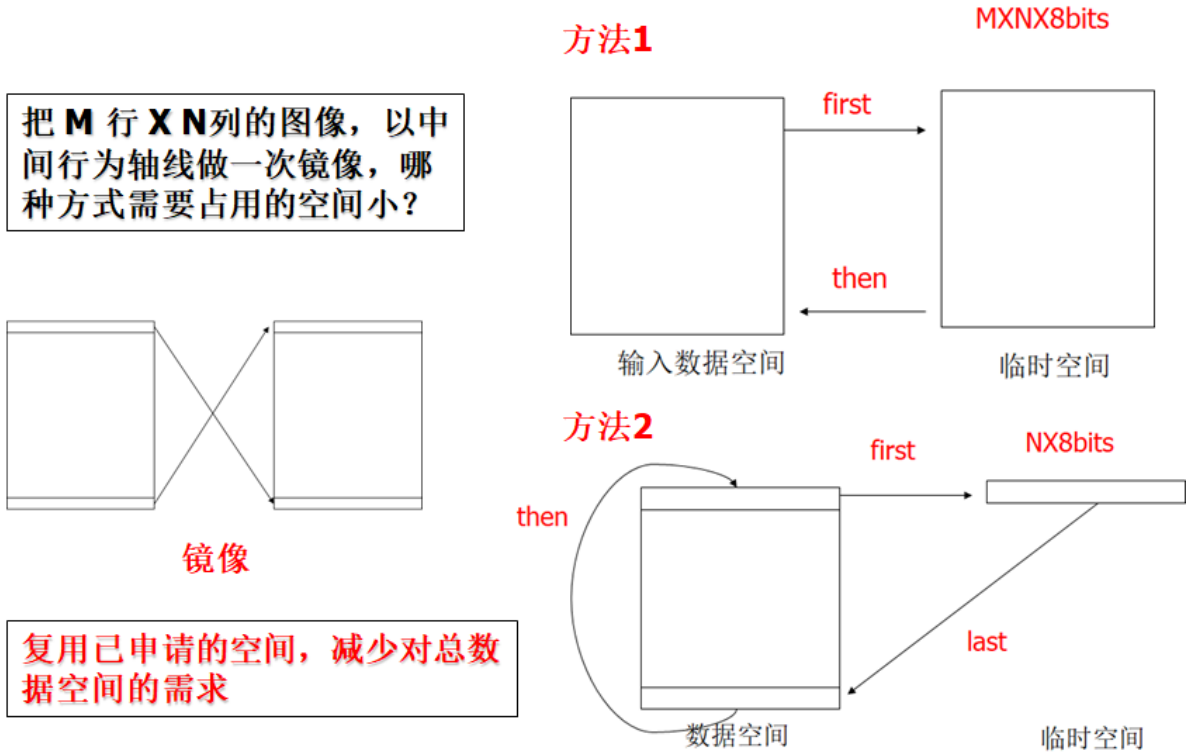
访问结构中数据的效率低。

数据类型

- 每个像素为 8bit，大小为 M 行 X N列的图像，选择不同的数据类型所占用的空间。

```
unsinged char image[M][N];

short image[M][N];

int image[M][N];
```

空间复用



静态分配与动态分配

下面两个使用数据空间的方式哪个更经济?

```
void imageproc(……)
{
    char a[M][N];
    char b[M][N];
    char c[M][N];
    ……
    func1(a, b);
    ……
    func2(b, c);
    ……
}
```

```
Void imageproc(……)
{
    char *a, *b, *c;
    a=(char *)malloc(…);
    b=(char *)malloc(…);
  ……
    func1(a, b);
    free(a);
   ……
   c=(char *)malloc(…);
   func2(b, c);
    ……
    free(b);
    free(c);
}
```

采用动态分配内存空间并及时释放，将更有效地利用存储空间。

> S3c2410A 的堆heap size初始为0，动态分配时需要注意修改，左边栈右边堆
>
> 软中断进入svc异常模式，不同模式对应不同堆栈，软中断的栈svc

降低能耗

能耗与功率

能耗： W=Pt ，P：系统功率；t：工作时间	
P= Ps+Pd •Ps: 静态功率 •Pd: 动态功率	P=Pc+Pm+Pp •Pc: CPU功率 •Pm: 存储器功率 •Pp: 其他外设功率

功率状态

功率： Dynamic>Standby（待机）>Sleep(idle)>off

能耗管理

- 处理器工作状态（功耗状态，多核模式）；
- 系统工作状态（外设）；
- 系统工作时间。

程序优化

- 程序执行功耗
  - 减少指令数，减少执行时间（t)
  - 选用功耗低的指令(P)
- 系统管理（P）
  - 控制处理器
    - 降低主频
    - 状态管理
    - 模式管理
  - 控制系统
    - 减少内存访问次数
    - 关断空闲外设

比较下列指令的功耗

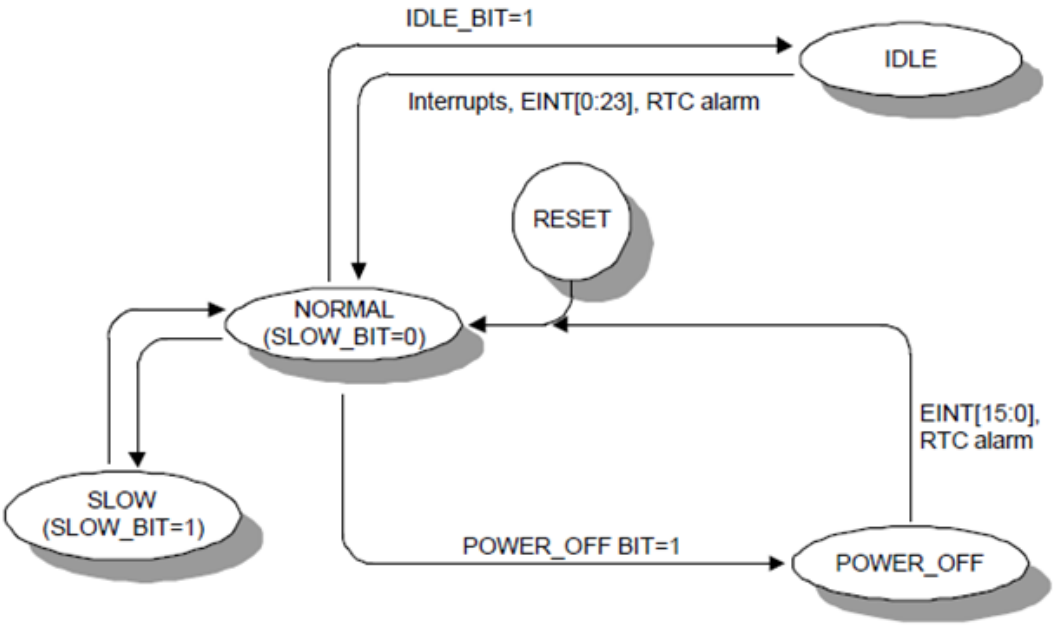
a=2*b
add r0, r0, r0
mul r0, r0, r1
mov r0, r0, lsl #1

add r0, r0, r0
add r0, r1, r2

mul r0, r1, r0
mul r0, r1, r2

> 寄存器善良影响功耗
>
> Mul > Add
>
> Add r0,r1,r2 > add r0,r0,r0

功耗状态(S3C2410)



Mode	ARM920T	AHB Modules (1) /WDT	Power Management	GPIO	32.768kHz RTC clock	APB Modules (2) & USBH/LCD/NAND
NORMAL	O	O	O	SEL	O	SEL
IDLE	X	O	O	SEL	O	SEL
SLOW	O	O	O	SEL	O	SEL
POWER_OFF	OFF	OFF	Wait for wake-up event	Previous state	O	OFF

> 这里power-off对应sleep，idle对应standby，只能外部事件中断唤醒，slow可通过程序进入normal

存储访问

- 比较下列两组代码的功耗　　那种情况功耗大，为什么？

```
for(L = 0; M > L; L++)
{
  *pu32temp +=  *array++;
}
```

VS

```
register int temp= *ptemp;
for (L = 0; M> L;L++)
{
  temp +=  *array++;
}
*ptemp = temp;
```

- 哪个**scatter**文件生成程序的功耗

```
ROM_LOAD 0x0
{
    ROM_EXEC 0x00
    {
      vectors.o (Vect, +First)
      * (+RO)
    }
}
```

VS

```
ROM_LOAD 0x0
{
        ROM_EXEC 0x00
        {
        vectors.o (Vect, +First)
        }

        RAM_EXEC 0x100000
        {
        * (+RO)
        }
}
```

区别？

- 结论
  - 减少存储器访问，从而降低存储器的功耗

提高速度

目标

- 减少程序运行时实际执行的指令数；
- 减少指令执行时间。

途径

- 算法
- 数据类型
- 循环体
- 计算
- 存储访问

对于处理器，如果某类型数的存访地址是自然边界的整数倍，则访问效率最高。

变量

共10条语句

```
int checksum(int *data)
{
    char i;
    int sum=0;
    for(i=0;i<64;i++)
    {
     sum+=data[i];
   }
    return sum;
}
```

```
checksum
      mov r2, r0 ;r2=data
      mov r0,#0 ;sum=0
      mov r1, #0;i=0;
checksum_loop
      ldr r3, [r2, r1, LSL #2]
      add r1, r1, #1 ;i=i+1
      and r1, r1, #0xff;i=(char)r1
      cmp r1, #0x40;i<64?
       add r0, r3, r0
       bcc checksum_loop
       mov pc, r14
```

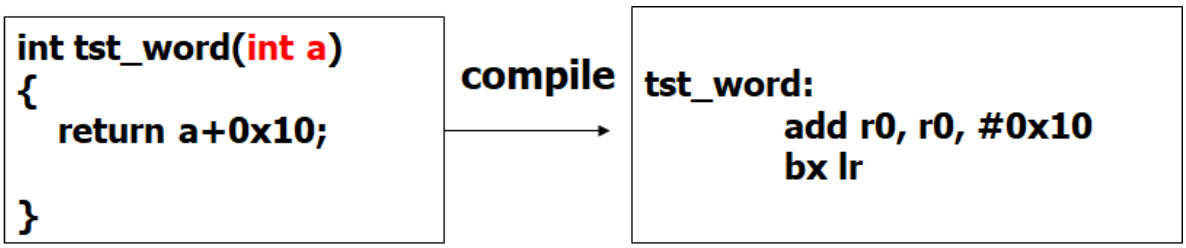
共9条语句，少了一条

```
int checksum(int *data)
{
    unsigned int i;
    int sum=0;
    for(i=0;i<64;i++)
    {
     sum+=data[i];
   }
    return sum;
}
```

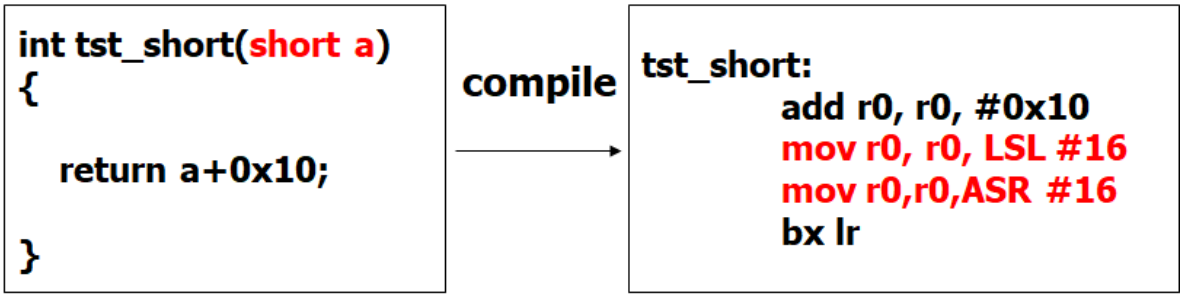
```
checksum
      mov r2, r0 ;r2=data
      mov r0,#0 ;sum=0
      mov r1, #0;i=0
checksum_loop
      ldr r3, [r2, r1, LSL #2]
      add r1, r1, #1 ;i=i+1
        cmp r1, #0x40;i<64?
       add r0, r3, r0
       bcc checksum_loop
       mov pc, r14
```

- 对于char 数据类型，在i 和64比较之前，编译器增加了AND指令来保证i的范围在0-255之间;
- 计算过程中的局部变量应尽量避免使用char 和short 数据类型，除非需要使用char 和short的数据溢出特性。
- 存在主存中的数组和全局变量尽可能使用小尺寸的数据类型，以节省存储空间

参数



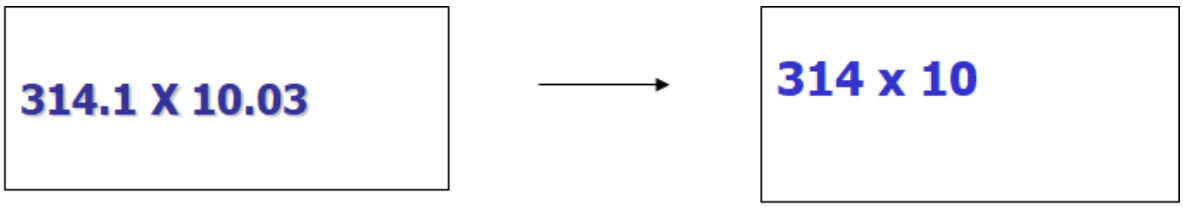
多出2条指令

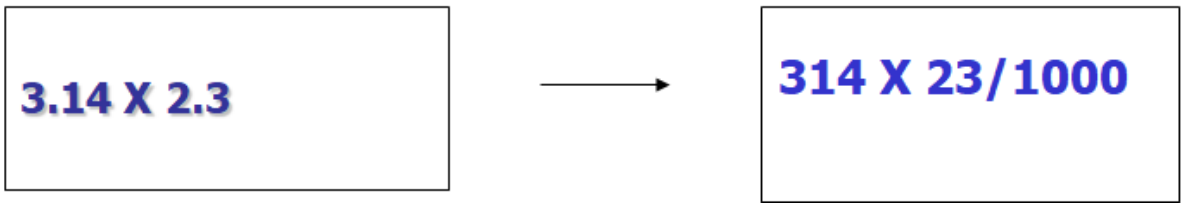


- 宽参数传递，被调用者把参数缩小到正确的范围。
- 宽返回，调用者把返回值缩小到正确的范围。
- 窄参数传递，调用者把参数缩小到正确的范围。
- 窄返回，被调用者把参数缩小到正确的范围。
- GCC是宽参数传递和宽返回。
- armcc 是窄参数传递与窄返回
- 尽量用int 或 unsigned int 型参数。
- 对于返回值尽量避免使用char和short类型。
- 防止编译器做不必要的类型转换

类型转换

对于没有浮点运算指令的处理器，把浮点数转换成整数，用整数运算替代浮点运算，提高程序的执行速度。





有符号数与无符号数

```
int average(int a, int b)
{
    return(a+b)/2;
}
```

→

```
average
    add   r0, r0, r1; r0=a+b
    add   r0, r0, r0, lsr #31
                ;if(r<0) r0++
    mov  r0, r0, asr #1; >>1
    mov  pc, r14
```

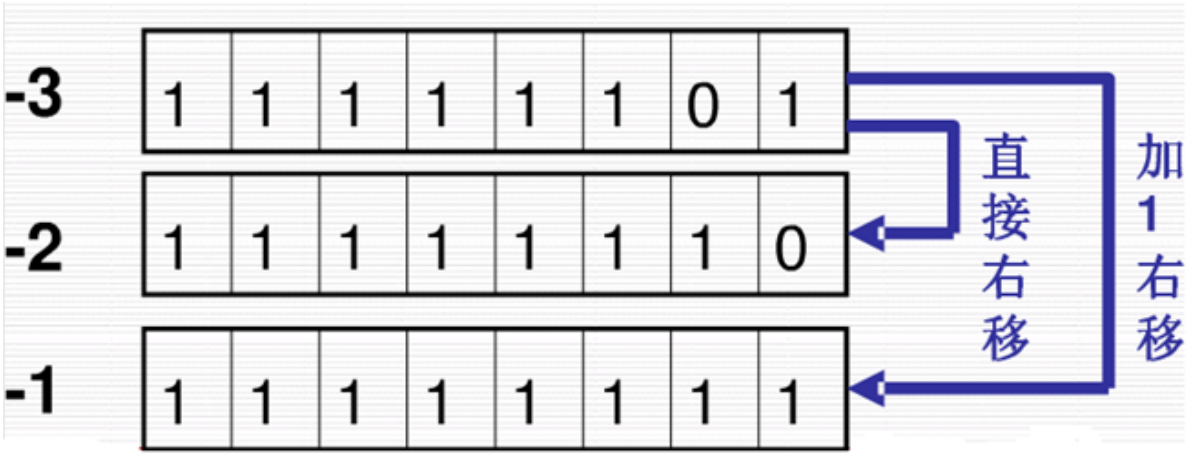
```
int average(unsigned int a,
            unsigned int b)
{
    return(a+b)/2;
}
```

→

```
average
    add   r0, r0, r1; r0=a+b
    mov  r0, r0, asr #1; >>1
    mov  pc, r14
```

对于非负数的计算，采用无符号数类型效率更高

- ARM C中， 如果x 是负数，则x/2不是右移一位，而是加1后右移。例如：-3/2=-1。



- 用右移计算除法时要考虑符号位
- 对于非负数的计算，采用无符号数类型效率更高。

循环体

- 减少使整个循环过程中执行的指令数；
- 增加循环过程中连续顺序执行的指令数。

循环展开

代码长度变化？实际执行指令数变化？

```
int checksum(int *data,
             unsigned int N)
{
   int sum=0;
   do
   {
     sum+=*(data++);
     N--;
   } while(N!=0);
    return sum;
}
```

```
int checksum(int *data,
             unsigned int N)
{
   int sum=0;
   do
   {
     sum+=*(data++);
     sum+=*(data++);
     sum+=*(data++);
     sum+=*(data++);
     N-=4;
   } while(N!=0);
    return sum;
}
```

```
int checksum(int *data,
             unsigned int N)
{
   int sum=0;
   do
   {
     sum+=*(data++);
     sum+=*(data++);
     sum+=*(data++);
     sum+=*(data++);
     N-=4;
   } while(N!=0);
    return sum;
}
```

```
checksum
      mov r2, #0 ;
checksum_loop
      ldr r3, [r0], #4
      subs r1, r1, #4 ;N-=4
      add r2, r3, r2
      ldr r3, [r0], #4
      add r2, r3, r2
      ldr r3, [r0], #4
      add r2, r3, r2
      ldr r3, [r0], #4
      add r2, r3, r2
      bne checksum_loop
      mov r0, r2
      mov pc, r14
```

- 通过循环展开，可以减少循环开销，提高程序执行速度；
- 展开循环可以减少程序的跳转，从而降低流水线中断的次数，提高程序执行的效率。

循环展开后，循环开销从4N个周期，减少到4N/4=N个周期，提高速度。

固定次数循环

```
int checksum(int *data)
{
    unsigned int i;
    int sum=0;
    for(i=0;i<64;i++)
    {
      sum+=*(data++);
    }
    return sum;
}
```

```
int checksum(int *data)
{
    unsigned int i;
    int sum=0;
    for(i=64;i>0;i--)
    {
      sum+=*(data++);
    }
    return sum;
}
```

- 递加循环时增加一条CMP指令
- 尽量采用递减循环

不定次数循环

```
int checksum(int *data,
             unsigned int N)
{
   int sum=0;
    for(;N!=0;N--)
    {
      sum+=*(data++);
    }
    return sum;
}
```

9条指令

```
int checksum(int *data,
             unsigned int N)
{
   int sum=0;
   do
   {
     sum+=*(data++);
   } while(--N!=0);
    return sum;
}
```

7条指令

do – while 比 for(;;) 更有效

减少重复运算

```
for (i = 0; i < N; i++)
{
    array[i] = a/b + 4;
}
```

```
temp = a/b + 4;
for (i = 0; i < LEN; i++)
{
    array[i] = temp;
}
```

```
for (i = 0;
    i < num * 4 + 2;  i++)
{
        array[i] = 0;
}
```

```
m = num * 4 + 2;
for (i = 0; i < m; i++)
{
        array[i] = 0;
}
```

将不变计算移到循环体外，减少重复计算。

计算替换

```
for (i = 0; i < M; i++)
{
        ……
        s = 14 * i;
        ……
}
```

→

```
s = 0;
for (i = 0; i < M; i++)
{
        ……
        s+= 14;
        ……
}
```

```
int a[H][W];
int i，j;
for (i = 0; i < H; i++)
  for(j=0;j<W;j++)
        a[i][j]=0;
```

→

```
int a[H][W];
int i，j;
regiser int *p1
for (i = 0, p1=a; i < H; i++,p1+=W)
{
    register int *p2;
   for(j=0,p1=p1;j<W;p2++)
        *p2=0;
}
```

利用加法替代乘法和索引

软件流水线

```
for (i = 0; i < H; i++)
        z[i]=x[i]*y[i];
```

→

```
  i=0                    数据依赖，必须串行执行
loop:
  ld x[i]
  ld y[i]
  mul
  st z[i]
  if( ++i<W) goto loop
```

```
  i=0
  ld x[0]
  ld y[0]
loop:
  mul
  ld x[i+1]
  ld y[i+1]
  st z[i]
  if( ++i<W) goto loop
```

mul, ld 可以并行
st, mul可以并行

改变指令顺序，减少指令之间的依赖

计算替换

- 减少除法

```
if((a / b)>c)
        a=a / 4
```

→

```
if(a>(b*c))
        a=a>>2
```

- 减少乘方

```
b=pow(a，2)；
```

→

```
b=a*a；
```

- 求余处理

```
b=a%8;
```

→

```
b=a&7;
```

用运算量小但功能相同的表达式替换原来复杂的的表达式

查找表

减少了多少计算量？

```
for (L = 0; M> L; L++)

{
   *ptemp=A*sin(L*pi/M);
}
```

→

```
for(i=0;i<360;i++)
   asin[i]=sin(i*pi/180);
for (L = 0; M > L; L++)
{
   *ptemp=A *Asin[L&359];
}
```

- 对既消耗时间又消费资源的运算，应尽量使用查表的方式，并且将数据表置于程序存储区，但这需要预先计算出表的所有项。
- 如果表很大，则直接生成所需的表比较困难，此时可以在启动时的初始化函数中先计算，然后在数据存储器中生成所需的表，以后在程序运行直接查表就可以了，减少了程序执行过程中重复计算的工作量。

存储管理

局部变量全局变量

2次访问外存

```
void f(int *a);
int g(int a);
int test1 (int i)
{
  f(&i);
  i+=g(i);
  i+=g(i);
  return i;
}
```

compile →

```
test1
  stmdb sp!, {r0, lr}
  mov r0,sp
  bl f
  ldr r0, [sp,#0]
  bl g
  ldr r1,[sp,#0]
  add r0, r0, r1
  add sp, sp, #4
  ldmia sp!, {pc}
```

1次访问外存

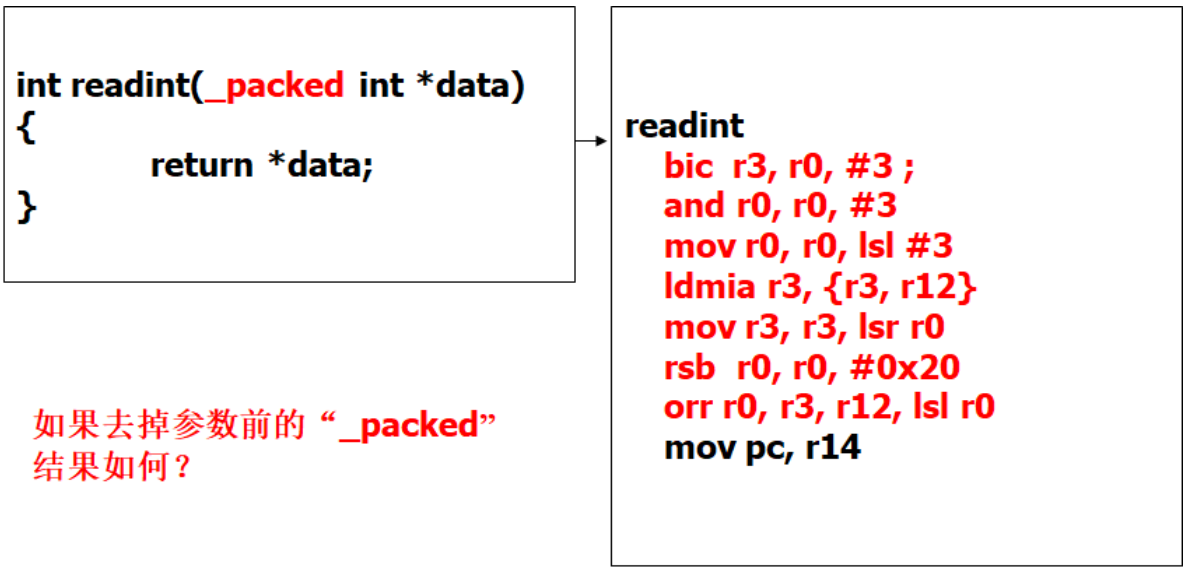
```
void f(int *a);
int g(int a);
int test2 (int i)
{
  int tmp=i;
  f(&tmp);
  i=tmp;
  i+=g(i);
  return i;
}
```

compile →

```
test2
  stmdb sp!, {r4, lr}
  str r0, [sp, #-4]!
  mov r0,sp
  bl f
  ldr r4, [sp,#0]
  mov r0, r4
  bl g
  add r0, r0, r4
  add sp, sp, #4
  ldmia sp!, {r4,pc}
```

用局部变量替换全局变量，减少程序访问存储器的次数

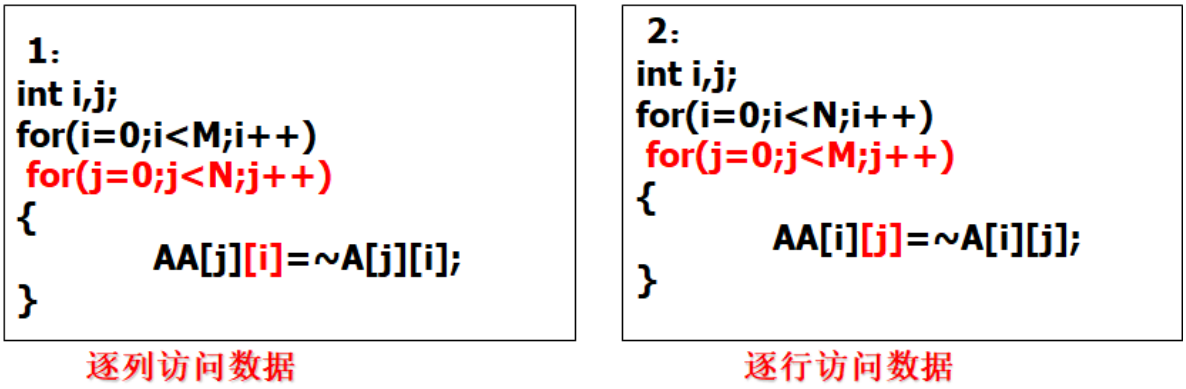
边界不对齐



在内存空间没有严格限制的情况下，尽量使数据对齐。

巧用Cache

一幅M行, N列的图像数据 AA，在内存中按行的顺序连续存放，下列哪段程序速度更快?

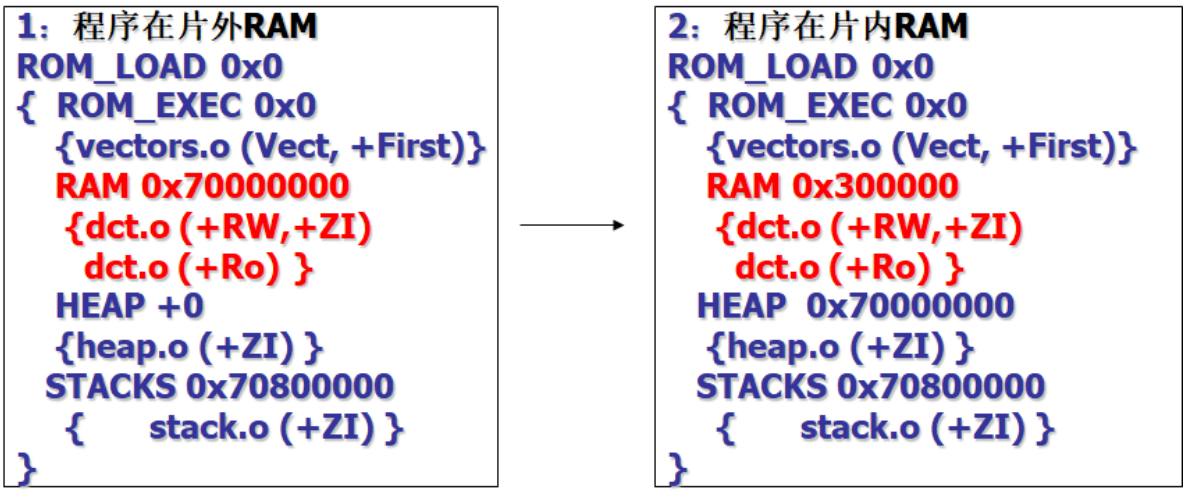


访问连续的数据区，充分发挥数据Cache的优势。

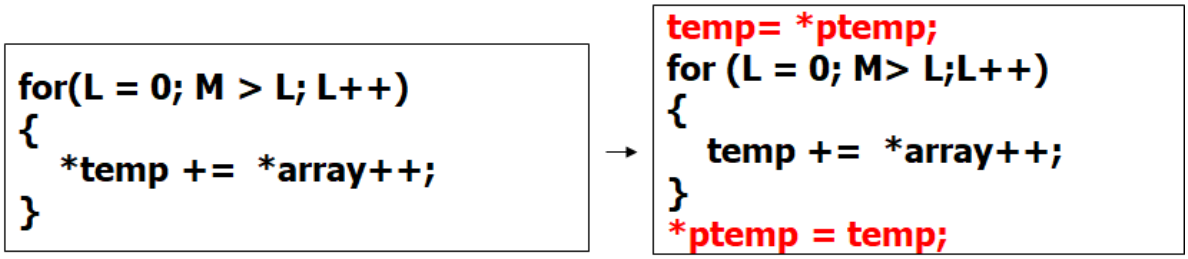
右边是连续访问，提高cache命中率

TCM

- Atmel 公司的AT91SAM9G45内有64KB, 映射在内空间的起始地址为0x300000。设系统DDR2 RAM的容量为64MB, 映射到内存空间的起始地址为：0x70000000。
- 用该处理器实现DCT变换运算，连接时采用下列哪个.sct文件所生成的程序执行速度更快
- 将计算量较大的代码和常用数据放到片内RAM中



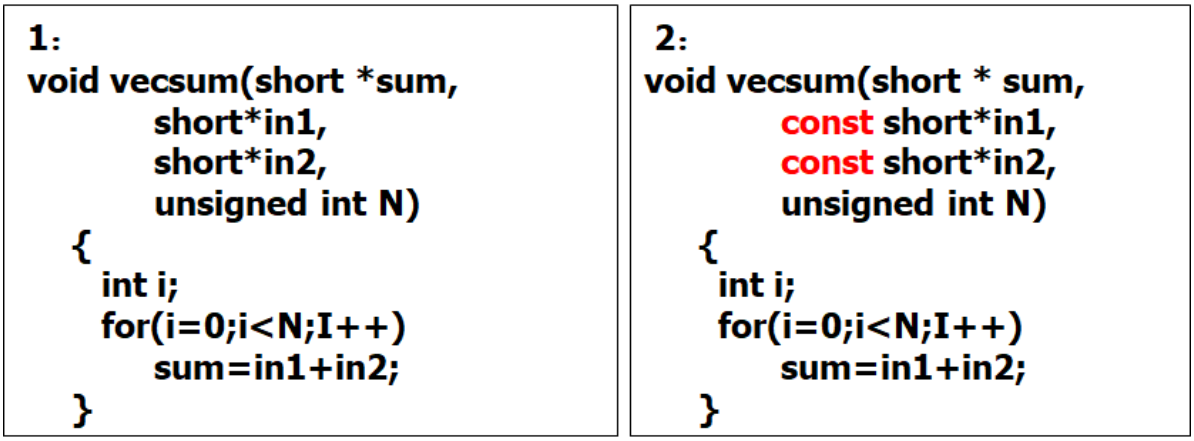
减少存储器访问



- 访问片外RAM或者Flash中的数据时，当需要多次读取或修改时，应遵循“读——改——写”模式，即首先读取片外RAM或者Flash中的数据，将其保存在片内RAM中，针对本地变量进行计算，计算完毕后再写回到片外RAM或者Flash中；而不是每修改一次就进行回写操作。

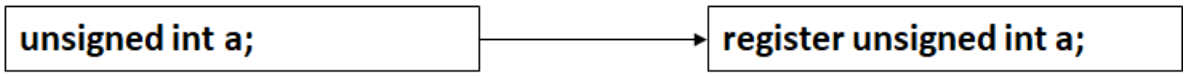
去除相关性

哪种更有效?

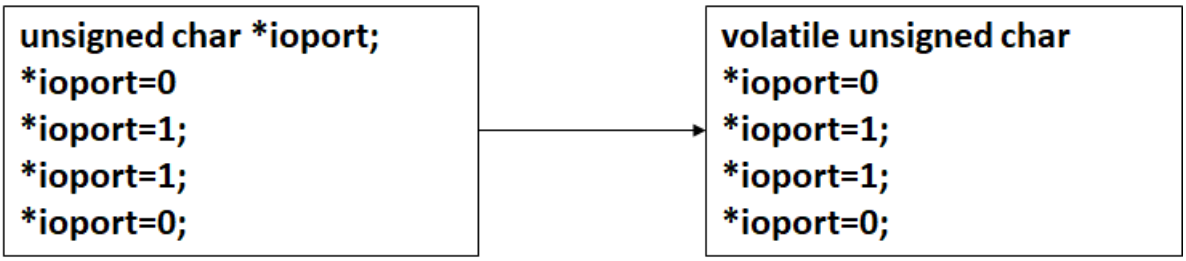


- 编译器不能确定写sum对in1和in2有无影响，从而in1和in2的读操作必须等到写sum操作完成之后才能进行，降低了流水效率 。
- 使用了关键字const，消除了指令之间的相关，从而使编译器能够判别内存操作之间的相关性，找到更好的指令执行方案。
- 消除数据之间的相关性，可以更有效地利用流水线提高程序的执行速度。

Register

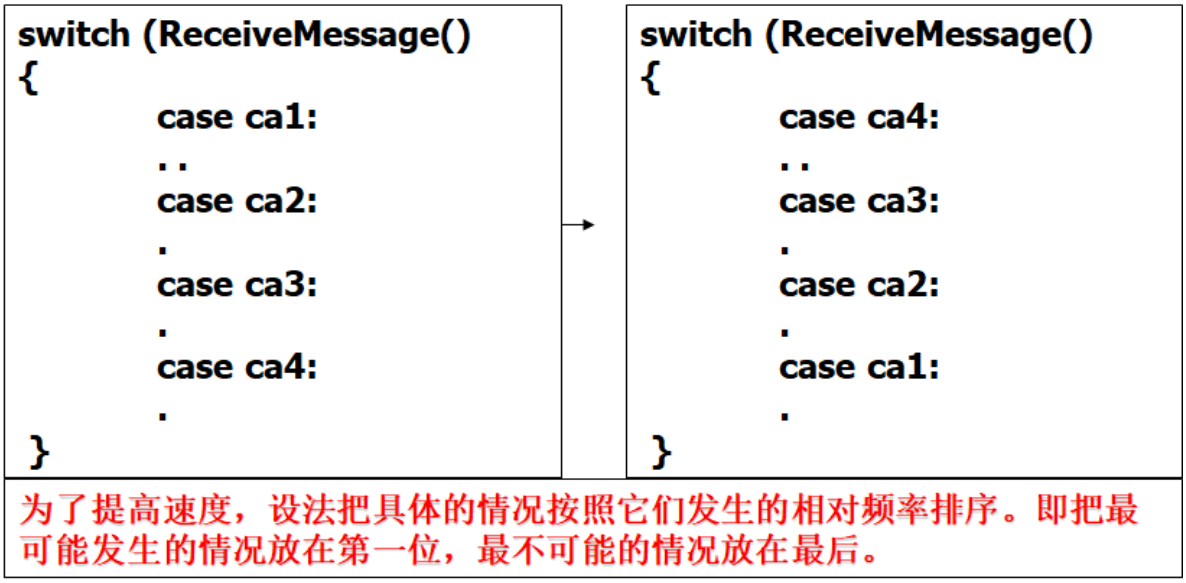


volatile



分支跳转

- Switch-case
- if- else
- 如果不同case值出现的频率不同，且可以预估。例如：ca1<ca2<ca3<ca4, 则下判断语句，哪一个平均速度快?



- 如何验证空间优化后的效果?
- 测试系统功耗的方法?
- 怎样观察程序的执行时间?

risc-v程序开发

RISC-V概述

业界动态与发展过程

- RISC-I
  - David A. Patterson and Carlo H. Sequin, RISC I: A reduced instruction set VLSI computer. In ISCA, 1981
- RISC-II
  - Manolis G. H Katevenis, Robert W. Sherburne, Jr., David A. Patterson and Carlo H. Sequin, The RISC II micro-architecture. In Proceedings VLSI 83 Conference, 1983
- RISC-III(SOAR)
  - David Ungar, Ricki Blau, Peter Foley, Dain Samples, and Patterson, Architecture of SOAR: Smalltalk on a RISC. In ISCA, 1984
- RISC-IV(SPUR)

    - David D. Lee, Shing I. Kong, Mark D. Hill, Georges S. Taylor, David A. Hodges, Randy H. Katz, and David A. Patterson. A VLSI chip set for a multiprocessor workstation, IEEE JSSC, 1989
- RISC-V
    - Krste Asanovic, 2010
    - V1.0, 2014
    - “V” , The fifth version, variation, vectors

	ARM Cortex-A5	RISC-V Rocket	Ratio
寄存器宽度	32	64	2
主频	>1Ghz	>1GHz	1
Dhrystone	1.57DMIPS/MHz	1.72DMIPS/Hz	1.1
面积（不包含Cache）	0.27mm2	0.14mm2	0.5
面积（16KBCache）	0.53mm2	0.39mm2	0.7
动态功耗	<0.08 mW/MHz	0.034 mW/MHz	>0.4

RISC-V 特点

成本

简洁性

cost ≈ f(die area²)

基本的RISC-V指令数目仅有40多条

相同大小缓存（**16KiB**）的**RISC-V Rocket** 处理器和采用相同技术（**TSMC40GPLUS**）的**ARM-32 Cortex A5** 处理器相比，**RISC-V** 晶粒的大小是**0.27mm²**，而**ARM-32** 晶粒的大小是**0.53mm²**。由于面积大一倍，**ARM-32 Cortex A5** 的晶粒成本是**RISC-V Rocket** 的约**4**倍。

指令集文档

ISA	Pages	Words	Hours to read	Weeks to read
RISC-V	236	76,702	6	0.2
ARM-32	2736	895,032	79	1.9
x86-32	2198	2,186,259	182	4.5

性能

CoreMark Testing

$$\frac{instructions}{program} \times \frac{average\ clock\ cycles}{instruction} \times \frac{time}{clock\ cycle} = \frac{time}{program}$$

Cortex-A9

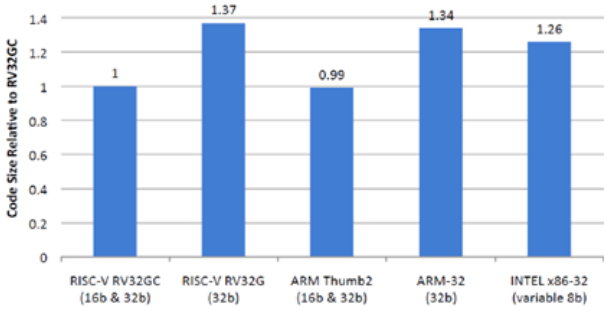
$$\frac{32.27\ B\ instructions}{program} \times \frac{0.79\ clock\ cycles}{instruction} \times \frac{0.71\ ns}{clock\ cycle} = \frac{18.15\ secs}{program}$$

Risc-V Boom

$$\frac{29.51\ B\ instructions}{program} \times \frac{0.72\ clock\ cycles}{instruction} \times \frac{0.67\ ns}{clock\ cycle} = \frac{14.26\ secs}{program}$$

程序大小

Code Size



RISC-V 生态

架构和实现的分离

不需要为特定实现进行架构优化

今天的流水线处理器使用硬件预测器预测分支结果，这种方法的准确度可以超过 90%，并且适用于任何大小的流水线。他们只需要一种机制来刷新和重启流水线。

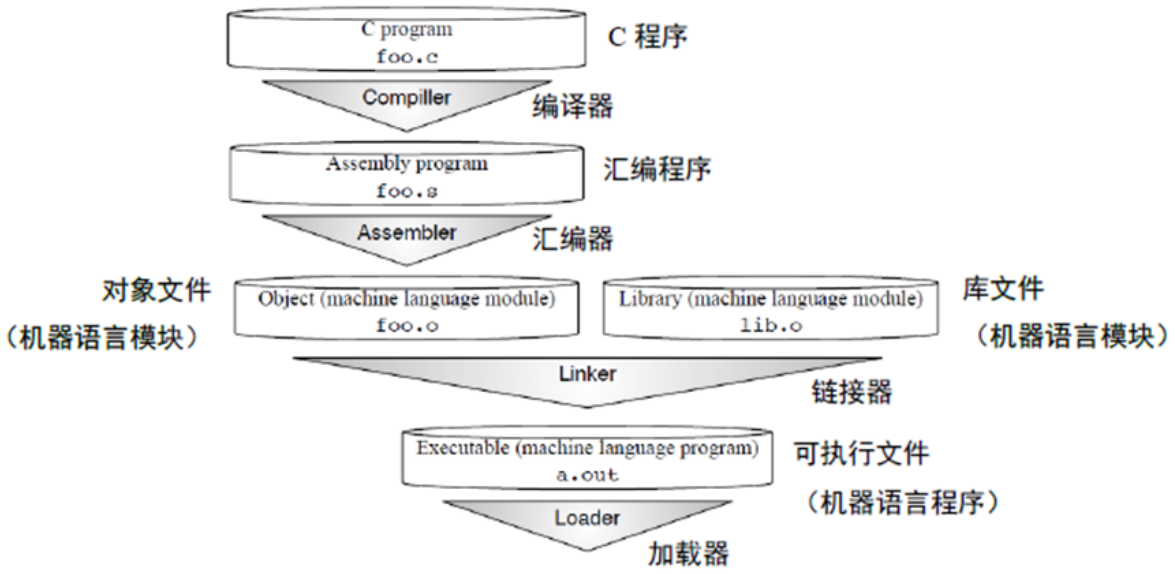
易于编程/编译/链接

简化工具链

- 更多（**32** 个）通用寄存器
- 单周期指令
- **PC** 相关的分支和数据寻址，支持位置无关代码（**PIC**）

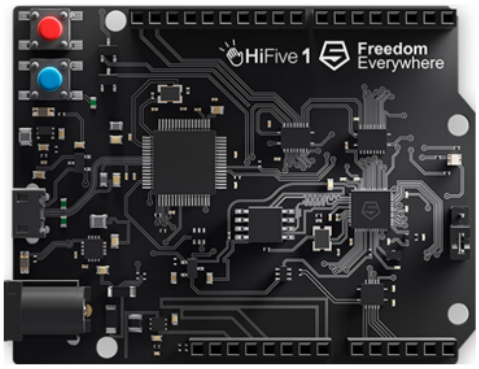
RISC-V Software Development

开发过程



开发板

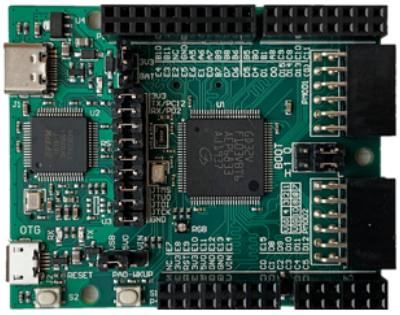
- **HiFive1**



https://www.sifive-china.com/site/HiFive1RevB

http://www.gd32mcu.com/cn/download/8

- **GD32VF103-EVAL**



https://www.nucleisys.com/developboard.php

开发环境和工具

IDE

- Nuclei Studio IDE, Nuclei

https://www.nucleisys.com/download.php

- Freedom Studio, SiFive

https://www.sifive-china.com/site/Software_tools

- RiscFree, Ashling

http://tools.emdoor.com/products/compiler/ashling/1693.html

- Embedded Studio, segger

https://www.segger.com/products/development-tools/embedded-studio/editions/risc-v/

- IAR Embedded Workbench，IAR System

https://www.iar.com/iar-embedded-workbench/

平台

- Windows
- macOS
- Ubuntu

https://www.sifive-china.com/site/Software_tools

RISC-V ISA

架构特点

特性	x86或ARM架构	RISC-V
模块化	不支持	支持模块化可配置的指令子集
可扩展性	不支持	支持可扩展定制指令
指令数目	指令数繁多 不同的架构分支彼此不兼容	一套指令集支持所有架构。基本指令子集仅40余条指令，以此为共有基础，加上其他常用模块子集指令总指令数也仅几十条
易实现性	硬件实现得复杂度高	硬件设计与编译器实现非常简单 1) 仅支持小端格式 2) 存储器访问指令一次只访问一个元素 3) 去除存储器访问指令的地址自增自减模式 4) 规整的指令编码格式 5) 简化的分支跳转指令与静态预测机制 6) 不使用分支延迟槽（Delay Slot） 7) 不使用指令条件码（Conditional Code） 8) 运算指令的结果不产生异常（Exception） 9) 16位的压缩指令有其对应的普通32位指令 10) 不使用零开销硬件循环

指令集模块

基本指令集	指令数	描述
RV32I	47	32位地址空间、整数指令，32个寄存器
RV32E	47	RV32I子集，仅支持16个
RV64I	59	64位地址空间、整数指令，一部分32位整数指令
RV128I	71	128位地址空间、整数指令，一部分64、32位整数指令
扩展指令集		
M	8	整数乘法与除法
A	11	存储器原子（atom）操作
F	26	单精度浮点
D	26	双精度浮点
C	46	压缩指令，16bit
Q	28	四精度浮点运算扩展
L		10进制浮点扩展
Zifencei	1	指令获取界限

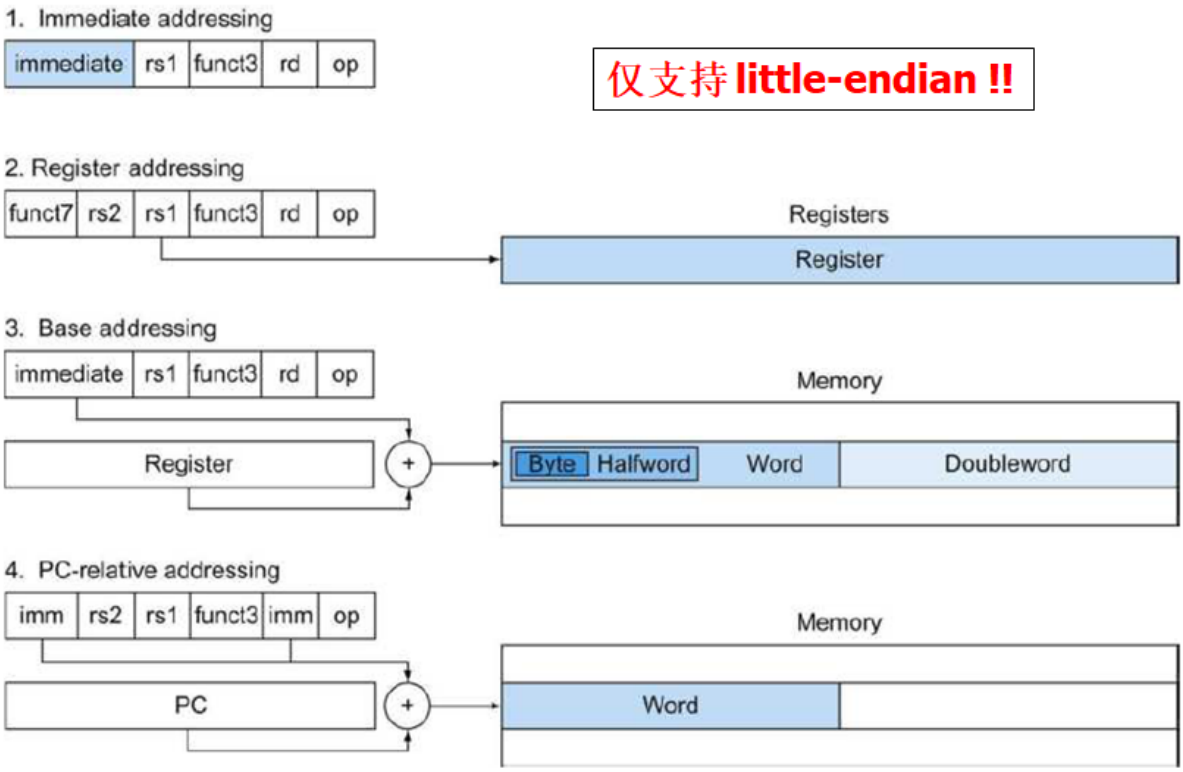
Registers

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero 硬编码 0	—
x1	ra	Return address 返回地址	No
x2	sp	Stack pointer 栈指针	Yes
x3	gp	Global pointer 全局指针	—
x4	tp	Thread pointer 线程指针	—
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries 临时寄存器	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register 保存寄存器	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments 函数参数	No
x18–27	s2–11	Saved registers 保存寄存器	Yes
x28–31	t3–6	Temporaries 临时寄存器	No

- RV32I有32个通用寄存器，每一个寄存器是32位，寄存器x0置恒定值0，其他31个寄存器 x1-x31是可以读写的普通通用寄存器。
- RV64I同样拥有32个通用寄存器，且x0恒为0，但每个通用寄存器是64位。
- 控制和状态寄存器（Control and Status Register，CSR）用于配置或记录处理器内核运行状态
- 采用独立程序计数器寄存器pc，不能用指令直接改写。

> CSR寄存器是处理器内核内部的寄存器，使用专有的12位地址编码空间，对一个hart，可以配置4k的CSR寄存器。

addressing mode



Unprivileged /privileged Mode

Levels

等级	编码	名称	缩写
0	00	用户/应用模式（user/application Mode）	U
1	01	监督者模式（supervisor Mode）	S
2	10	管理者模式（Hypervisor Mode）	H
3	11	机器模式（Machine Mode）	M

Different Packages

模式数量	支持模式	目标应用
1	M	简单嵌入式系统
2	M，U	安全嵌入式系统
3	M，S，U	支持Unix，Linux，Windows等操作系统
4	M，H，S，U	支持虚拟机系统

RISC-V Processor

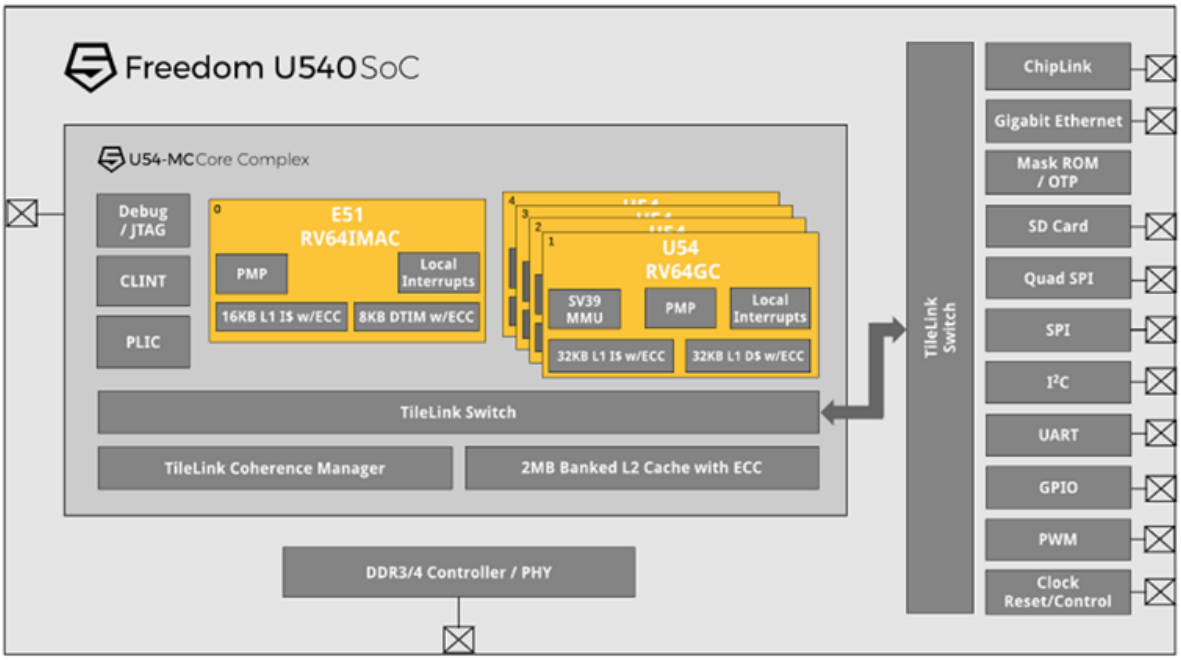
RISC-V core

- Rocket，Berkeley
- Boom（Out-of-Order Machine），Berkeley

	ARM Cortex-A9	RISC-V BOOM-2W
ISA	32bit ARM v7	64bit RISC-V（RV64G）
微架构	2wide，4发射乱序处理器	2wide，3发射乱序处理器
流水线级数	8	6
性能	3.59 CoreMarks/MHz	3.91 CoreMarks/MHz
工艺	台积电40nm	台积电40nm
面积（含32K缓存）	约2.5 mm2	约1.00 mm2
主频	1.4GHz	1.5GHz
功耗	0.5-1.9 W (2 cores + L2)@ TSMC 40nm, 0.8-2.0 GHz	0.25 W (1 core + L1)@ TSMC 45nm, 1 GHz

- SiFive

- RISC-V SOC

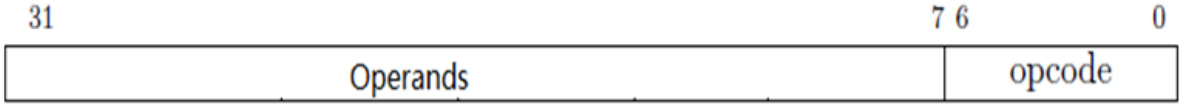


- Nuclei（芯来科技）

RV32I

RV32I指令

指令长度



- **RV32I指令集指令长度是32位；**
- **RV64I和RV128I指令集中的指令长度也是32位；**
- **扩展指令集“M”、“A”、“F”、“Q”、“D”、以及CSR寄存器访问指令的长度是32位；**
- **扩展指令集“C”的指令长度是16位；**
- **在程序内存中，32位指令必须位于4字节对齐边界，16位指令必须是2字节边界对齐。**

指令类型

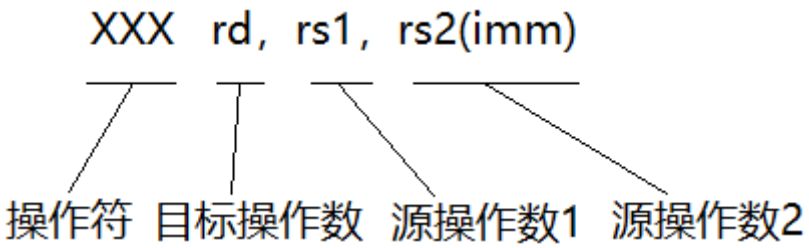
- 数值计算、存储访问、跳转控制、CSR访问、其他指令

RV32I-运算

助记符

功能	操作	助记符	解释
整数计算（Integer Computation）	addition (immediate)	add(i)	“+”运算（立即数）
	subtract	sub	“-”运算
	and (immediate)	and (i)	“与”运算
	or (immediate	or (i)	“或”运算
	xor (immediate)	xor (i)	“异或”运算
	shift left logical (immediate)	sll (i)	逻辑左移
	shift right arithmetic (immediate)	sra (i)	算术右移
	shift right logical(immediate)	srl (i)	逻辑右移
	load upper immediate	lui	装载立即数到寄存器的高20位
	add upper immediate to pc	auipc	把立即数加到程序指针的高20位
	set less than immediate unsigned	slt(i)(u)	根据比较结果设置寄存器值

格式



算术运算

指令	示例	操作
加法	**add t0,t1,t2**	**t0=t1+t2**；
减法	**sub t0,t1,t2**	**t0=t1-t2**；
立即数加法	**addi t0,t1,200**	**t0=t1+200**；

- RV32I算术运算指令中，立即数的数值范围是imm[11:0]，12位

逻辑指令

指令	示例	操作
与	**and t0,t1,t2**	**t0=t1&t2** ；按位与
或	**or t0,t1,t2**	**t0=t1|t2**；按位或
异或	**xor t0,t1,t2**	**t0=t1^t2**；按位异或
立即数与	**andi t0,t1,200**	**t0=t1&200**；按位与
立即数或	**ori t0,t1,200**	**t0=t1|200**；按位或
立即数异或	**xori t0,t1,200**	**t0=t1^200**；按位异或

- RV32I逻辑运算指令中立即数的数值范围是imm[11:0],12位

移位指令

指令	示例	操作
逻辑左移	**sll t0,t1,t2**	**t0=t1<<t2**；低位补0
逻辑右移	**srl t0,t1,t2**	**t0=t1>>t2**；高位补0
算术右移	**sar t0,t1,t2**	**t0=t1>>t2**；负数高位补1，正数高位补0
立即数逻辑左移	**slli t0,t1,10**	**t0=t1<<10**；低位补0
立即数逻辑右移	**srli t0,t1,10**	**t0=t1>>10**；高位补0
立即数算术右移	**srai t0,t1,10**	**t0=t1>>10**；负数高位补1，正数高位补0

- RV32I移位操作指令中立即数的数值范围imm[4:0],5位

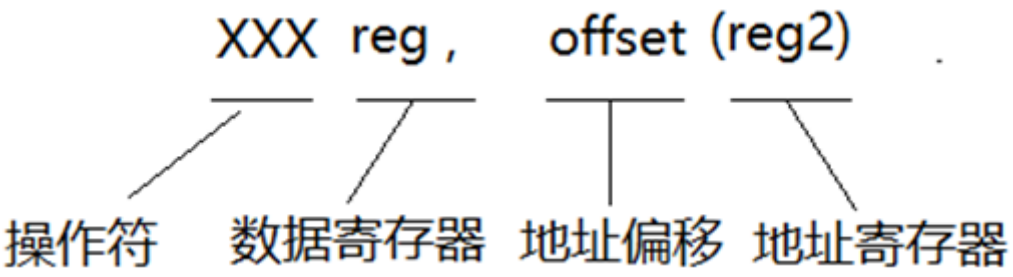
RV32I-储存访问

助记符

功能	操作	助记符	解释
装载与存储（Load and Store）	load byte （unsigned）	lb（u）	读取1字节（8位）
	load halfword （unsigned）	lh（u）	读取2字节（16位）
	load word	lw	读取4字节（32位）
	store byte	sb	保存1字节（8位）
	store halfword	sh	保存2字节（16位）
	store word	sw	保存4字节（32位）

> 选项U，设计扩展时才考虑

指令格式



访存

指令	示例	操作
装载字（32位）	lw t0, 50(t1)	t0=memory[t1+50]；将起始地址t1+50内存中4字节数写入t0。
装载半字（16位）	lh t0, 50(t1)	t0=memory[t1+50]；将起始地址t1+50内存中2字节数写入t0低16位；正数，高16位补0；负数，高16位补1。
装载半字（无符号）	lhu t0, 50(t1)	t0=memory[t1+50]；将起始地址t1+50内存中2字节数写入t0低16位；高16位补0。
装载字节（8位）	lb t0, 50(t1)	t0=memory[t1+50]；将地址t1+50内存中1字节数写入t0；正数，高24位补0；负数，高24位补1。
装载字节（无符号）	lbu t0, 50(t1)	t0=memory[t1+50]；将地址t1+50内存1字节数写入t0；高24位补0。
写字	sw t0, 50(t1)	memory[t1+50]=t0；将t0中数据写入起始地址t1+50内存中。
写半字	sh t0, 50(t1)	memory[t1+50]=t0；将t0中数据低16位写入起始地址t1+50内存中。
写字节	sb t0, 50(t1)	memory[t1+50]=t0；将t0中数据低8位写入地址t1+50内存中。

> 50不是4的倍数，有问题，lw是4字节对齐，lh是2字节对齐，lb无对齐

- RV32I load和store指令中偏移量数值范围offset[11:0], 12位；
- “lui”指令将立即数装载到目标寄存器的高20位，目标寄存器的低12位置0；

```
lui     x5, 0x12345 ; x5=0x12345000
addi    x5, x5, 0x678 ; x5=0x12345678
```

- “auipc”指令将立即数加到pc的高20位；

```
auipc    t0, 0x12345 ;        执行后t0=0x12345000+pc
```

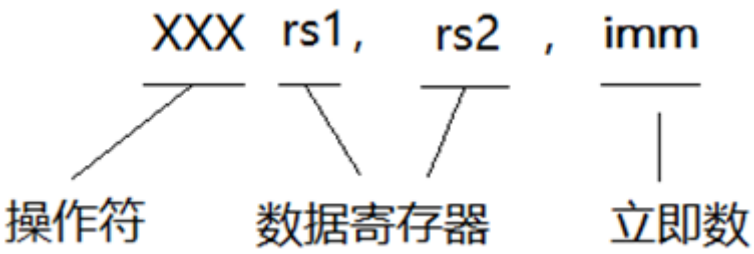
- “lui”和“auipc”指令中立即数的范围是imm[19:0],20位。

RV32I-跳转

助记符

功能	操作	助记符	解释
控制转移（Control transfer）	branch equal	beq	如果相等则跳转
	branch not equal	bne	如果不等则跳转
	branch less than （unsigned）	blt（u）	如果小于则跳转（无符号数比较）
	branch greater and equal （unsigned）	bge（u）	如果不小于则跳转
	jump and link	Jal	可返回的立即数跳转
	jump and link register	jalr	可返回的寄存器跳转

指令格式



分支跳转

指令	示例	操作
相等时分支	beq t0,t1,200	if（t0==t1) go to pc+200；pc相对跳转
不等时分支	bne t0,t1,200	if（t0！=t1) go to pc+200
小于时分支	blt t0,t1,200	if（t0<t1) go to pc+200
大于等于时分支	bge t0,t1,200	if（t0>=t1) go to pc+200
小于时分支（无符号）	bltu t0,t1,200	if（t0<t1) go to pc+200；无符号数比较
大于小于时分支 （无符号）	bgeu t0,t1,200	if（t0>=t1) go to pc+200；无符号数比较

无条件跳转

指令	示例	操作
带返回跳转	jal ra, 200	ra= pc+4，保存下条指令指针 pc=pc+200， pc 相对跳转
带返回跳转（寄存器）	jalr ra, 200 （t0）	ra= pc+4，保存下条指令指针 pc=t0+200，寄存器相对跳转

- 无条件跳转指令包含两个操作数，返回指针寄存器（ra）和跳转目标地址。
- 对于指令jal，跳转目标地址是语句中的立即数表示与当前PC值之和。立即数的范围是imm[20:1],20位。
- 对于指令jalr，跳转的目标地址为地址寄存器（t0)中的值与偏移量之和。偏移量的数值范围是offset[11:0],12位。

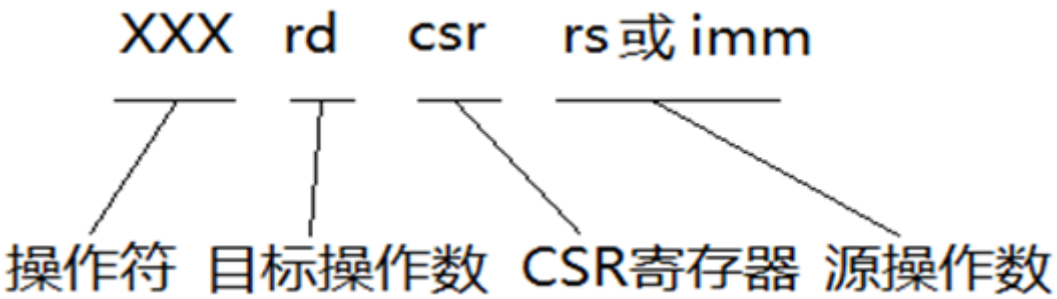
```
1.start:
2.  add x10, x10, x22
3.  lw x9, 0(x10)
4.  bne x9, x24, end; if(x9!=x24) PC=PC+8
5.  addi x12, x12, 1
6.  beq x0, x0, start ; pc=pc-16
7. end:
```

RV32I-CSR

助记符

功能	操作	助记符	解释
CSR访问	control status register read & clear bit (immediate)	csrrc(i)	CSR寄存器读并清除位（立即数）
	control status register read & set bit (immediate)	csrrs(i)	CSR寄存器读并置位（立即数）
	control status register read & write (immediate)	csrrw(i)	CSR寄存器读写（立即数）

格式



指令	示例	操作
先读后清除CSR	csrrc t0, 0x123，t1	t0=[0x123]; [0x123]=t0 & （~t1）; 把0x123中的值读入t0，然后用计算得到结果更新0x123中的值。
先读后置位CSR	csrrs t0,0x123,t1	t0=[0x123];[0x123]=t0 | t1; 把0x123中的值读入t0，然后用计算得到结果更新0x123中的值。
先读后写CSR	csrrw t0,0x123,t1	t0=[0x123];[0x123]=t1; 把0x123中的值读入t0，然后将t1中的值写入0x123中。
立即数先读后清除CSR	csrrci t0, 0x123，20	t0=[0x123]; [0x123]=t0 & (~20); 把0x123中的值读入t0，然后用计算得到结果更新0x123中的值。
立即数先读后置位CSR	csrrsi t0,0x123,20	t0=[0x123];[0x123]=t0 | 20; 把0x123中的值读入t0，然后用计算得到结果更新0x123中的值。
立即数先读后写CSR	csrrwi t0,0x123,20	t0=[0x123];[0x123]=20; 把0x123中的值读入t0，然后将20中的值写入0x123中。

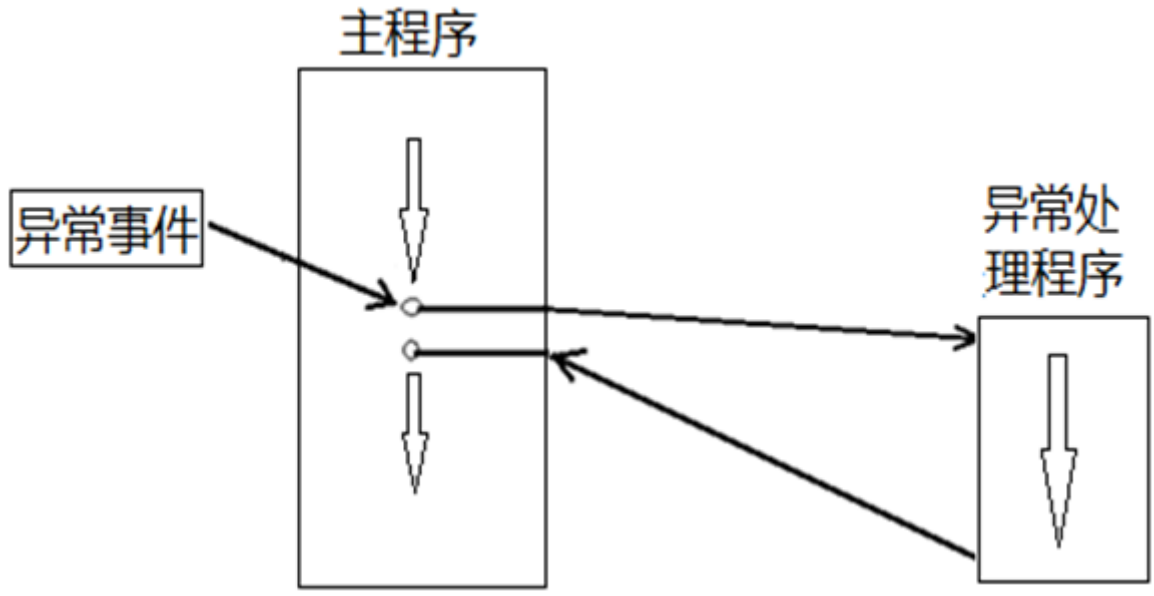
RV32I-其他

助记符

功能	操作	助记符	解释
其他（Miscellaneous）	fence |load & store	fence	同步内存访问和IO操作
	fence.instruction & data	fence.i	同步指令流
	envirmonent break	ebreak	环境断点
	envirmonent call	ecall	环境调用
	wait for interrupt	wfi	等待中断
	wait for exception	wfe	等待异常

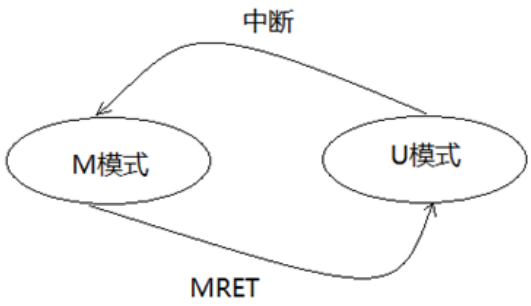
异常与中断

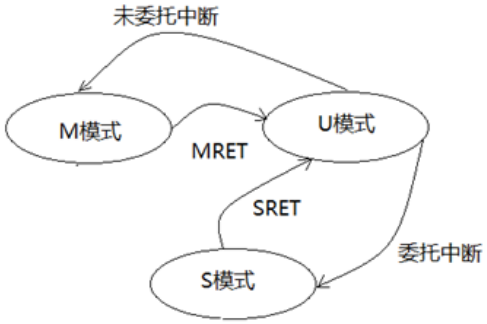
异常响应过程



- 通过CSR寄存器管理处理器异常和中断事件的响应和处理过程。

特权模式转换





- 机器（M）和用户（U）两种模式的RISC-V处理器，如果在用户模式下发生异常事件或中断，并且处理器打开中断使能，则处理器进入机器（M）模式，并执行异常或中断处理程序。异常或中断处理程序完成后通过指令返回到U模式。
- 机器（M）、监督者（S）和用户（U）三种模式的RISC-V处理器，异常委托机制，选择性地将中断和异常交给S 模式处理，不需要进入M 模式。处理完成后通过指令返回到U模式。

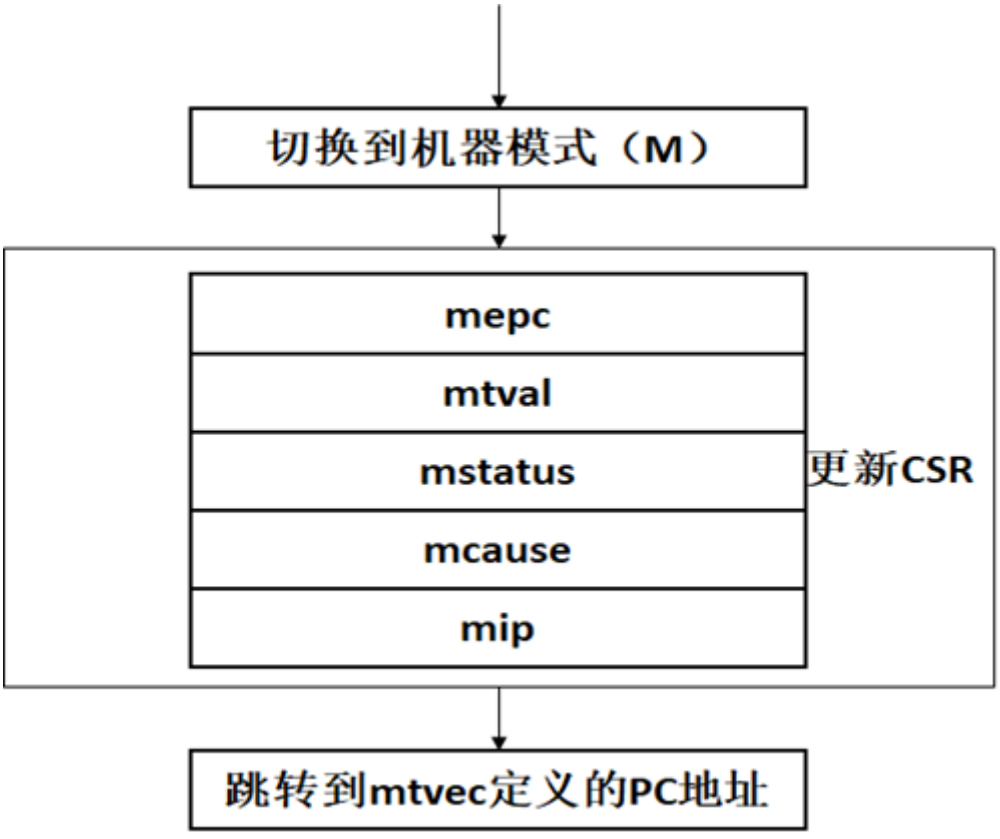
> 不设委托模式，任何模式中断都回到机器模式，自己模式不能处理自己的异常，必须委托高一级处理

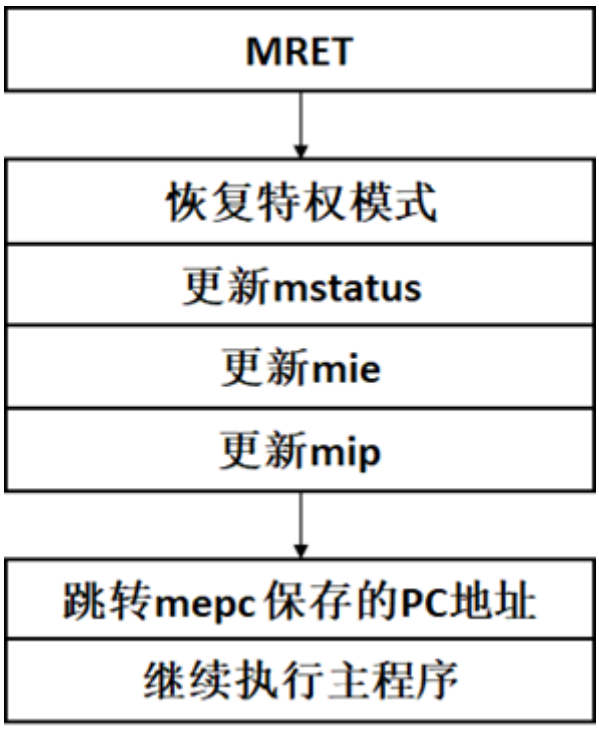
操作	助记符	解释
Machine-mode trap return	mret	M模式异常返回
Supervisor-mode trap return	sret	S模式异常返回
Supervisor-mode fence.virtual memory address	sfence	S模式内存访问同步

异常和中断CSR

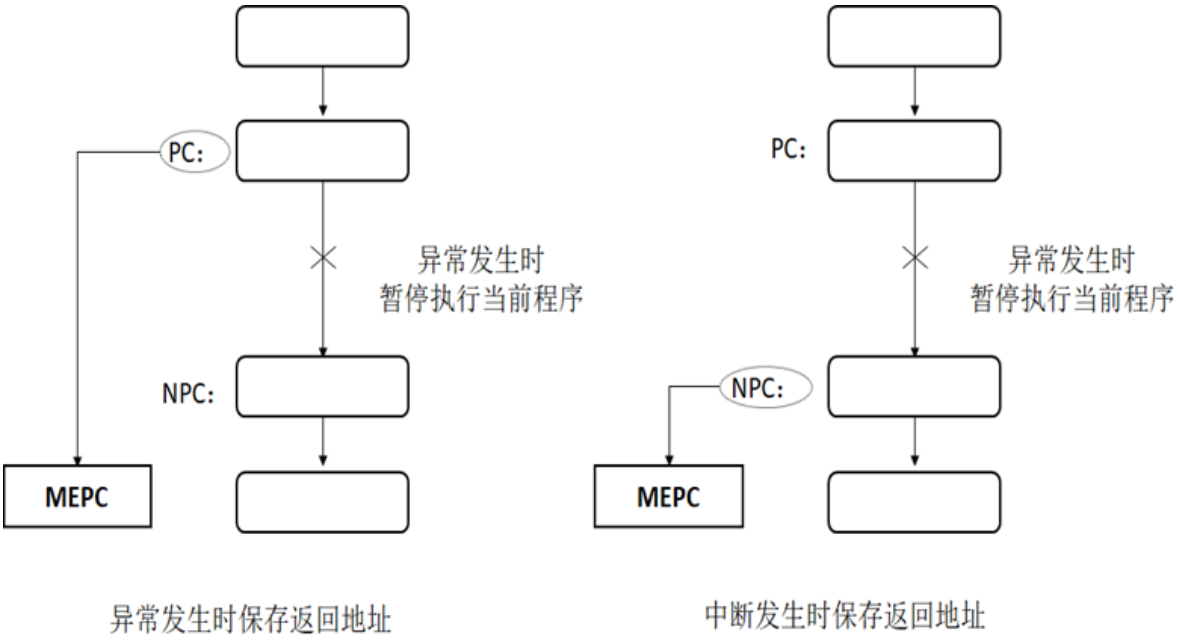
符号	名称	功能描述	CSR空间地址
mstatus	机器模式状态寄存器 （Machine Status Register）	寄存器中MIE和MPIE用于中断全局使能	0x300
mcause	机器模式异常原因寄存器 （Machine Cause Register）	进入异常的原因	0x342
mtvec	机器模式异常入口基地址寄存器 （Machine Trap-Vector Based-Address Register）	中断向量表基地址，进入异常的PC地址	0x305
mtval	机器模式异常值寄存器 （Machine Trap Value Register）	进入异常的信息	0x343
mepc	机器模式异常PC寄存器 （Machine Exception Program Counter）	保存异常返回地址	0x341
mie	机器模式中断使能寄存器 （Machine Interrupt Enable Register）	中断局部使能	0x304
mip	机器模式中断等待寄存器 （Machine Interrupt Pending Register）	中断等待状态	0x344

进入异常/中断





异常 vs 中断



GD32VF103

芯片

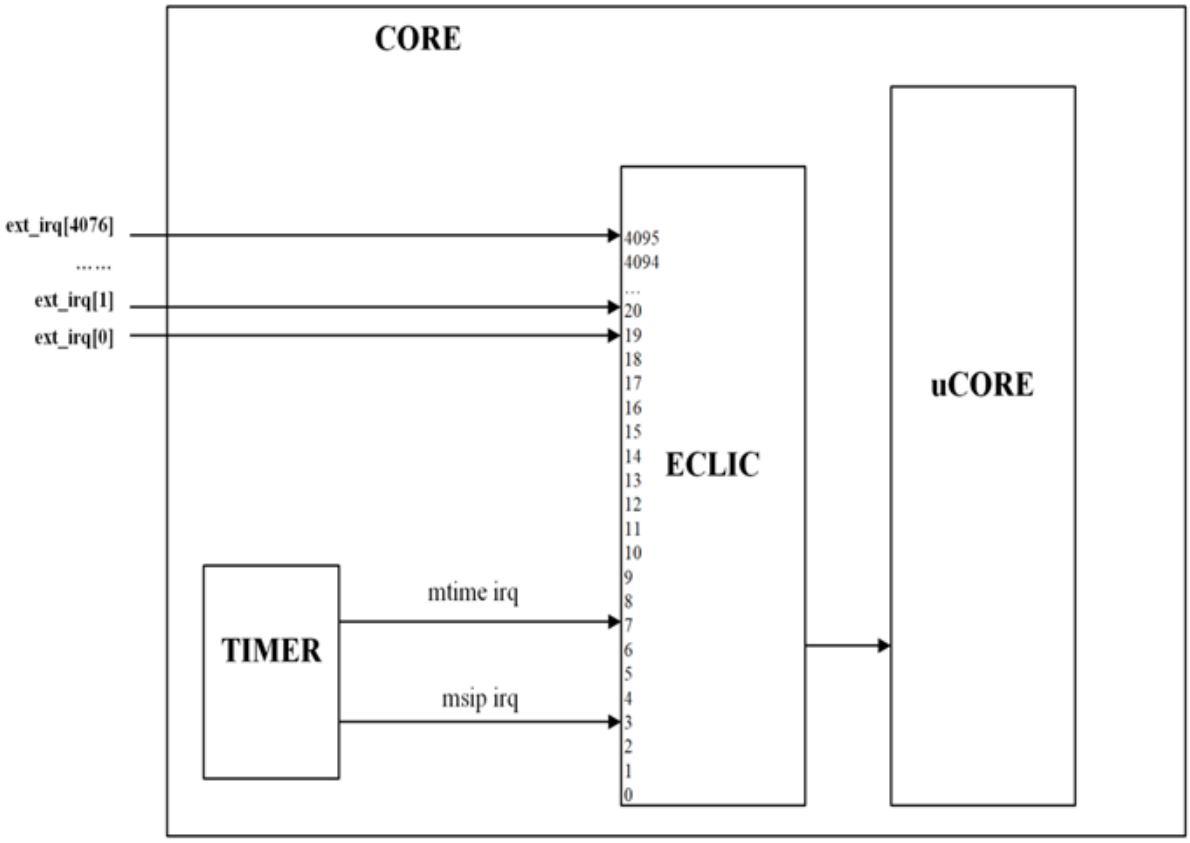
- BumleBee内核，32位RISC-V通用微控制器；
- 三条高速总线，即指令（I）总线、数据（D）总线和系统（System）总线；
- 采用哈佛体系结构，内存映射和高达4GB的内存空间；
- 32 KB的片上SRAM，地址0x2000 0000；
- 128KB 主FLASH ，地址0x0800 0000；
- 18KB 启动区，地址 0x000 000；
- 支持字节、半字（16位）和字（32位）访问

BumbleBee

- 支持RV32IMAC指令子集，机器模式（M）和用户模式（U）
- 两个私有64位计数器单元，时钟计数器（Timer) 和指令计数器（Counter)。
- 增强内核中断控制器（ECLIC）
- 软件中断、计时器中断和外部中断、支持16个中断级别、支持向量中断处理机制。
- 低功耗管理，支持WFI与WFE指令进入休眠模式，支持浅与深两级休眠模式。
- 不支持虚拟地址管理单元（MMU），所有地址访问操作都使用物理地址。

中断管理

内核中断控制器（Enhanced Core Local Interrupt Controller，ECLIC)



ECLIC 特点

- 可以支持4096个中断源（Interrupt Source），并为每个中断分配唯一编号（ID）；
- 控制每一个中断使能（IE）位，标志每一个中断的状态（IP）
- 设置每一个中断的电平或边沿属性（Level or Edge-Triggered）
- 设置中断级别和优先级（Level and Priority），
- 可以选择向量或非向量（Vector or Non-Vector Mode）中断响应方式。

ECLIC 寄存器

CLIC寄存器映射在处理器内存地址空间，以访存方式进行读写。

偏移量	属性	名称	宽度
0x0000	可读可写	中断设置 cliccfg	8位
0x0004	只读，写忽略	中断信息 clicinfo	32位
0x000b	可读可写	阈值等级寄存器 mth	8位
0x1000+4*i	可读可写	中断标志寄存器 clicintip[i]	8位
0x1001+4*i	可读可写	中断使能寄存器 clicintie[i]	8位
0x1002+4*i	可读可写	中断属性寄存器 clicintattr[i]	8位
0x1003+4*i	可读可写	中断控制寄存器 clicintctl[i]	8位

Timer

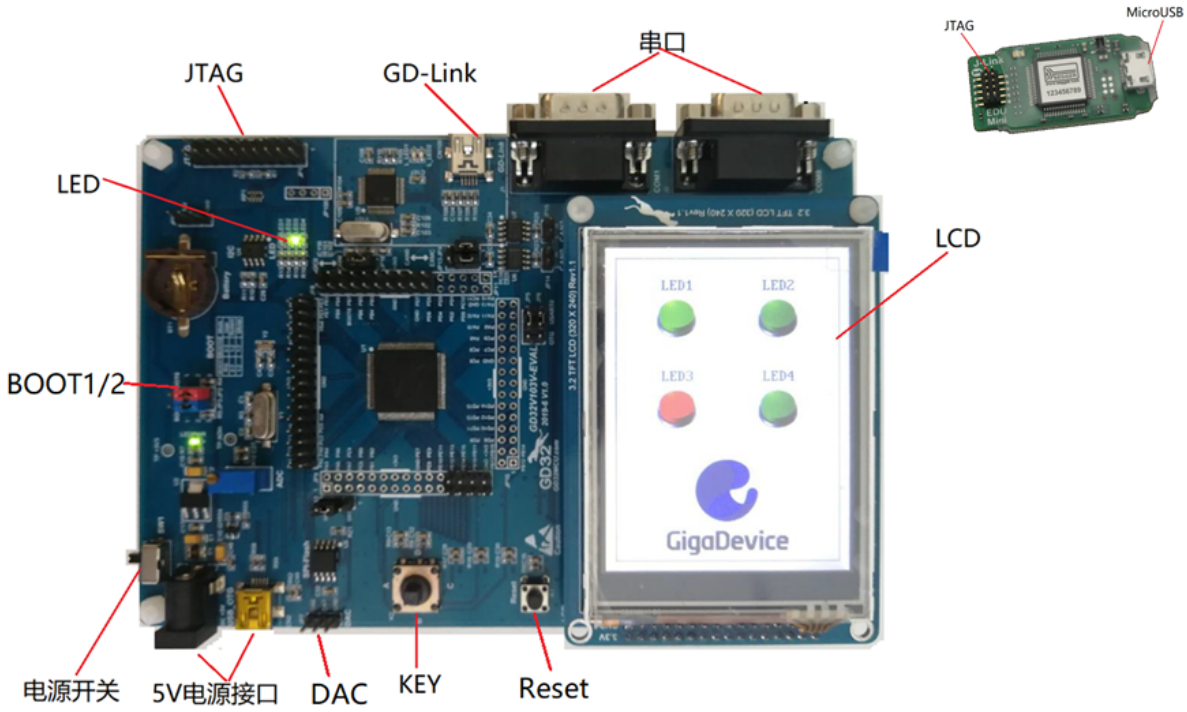
- TIMER采用自动增加计数模式，其计数寄存器（mtime）由两个32位寄存器{mtime_hi，mtime_lo}拼成，分别保存计数的高32位和低32位。

偏移量	属性	名称	功能描述
0x0	可读写	mtime_lo	计时器mtime的低32位值
0x4	可读写	mtime_hi	计时器mtime的高32位值
0x8	可读写	mtimecmp_lo	计时器比较值mtimecmp低32位
0xC	可读写	mtimecmp_hi	计时器比较值mtimecmp高32位
0xFF8	可读写	mstop	计时器的暂停控制
0xFFC	可读写	msip	产生软件中断

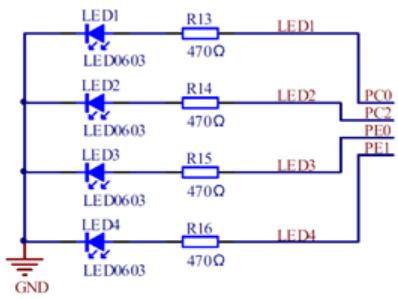
键盘中断

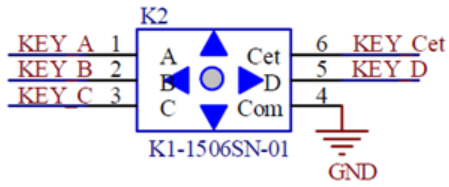
GD32VF103 -EVAL

EVAL Board



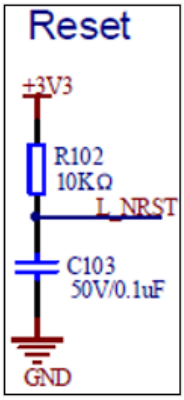
LED和KEY

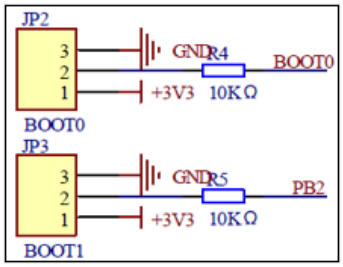




功能	引脚	描述
LED	PC0	LED1
	PC2	LED2
	PE0	LED3
	PE1	LED4
RESET		K1-Reset
KEY	PA0	KEY_A
	PC13	KEY_B
	PB14	KEY_C
	PC5	KEY_D
	PC4	KEY_Cet

- 启动
  - 复位

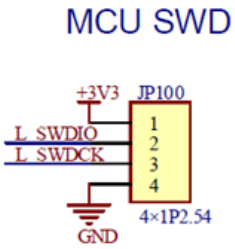


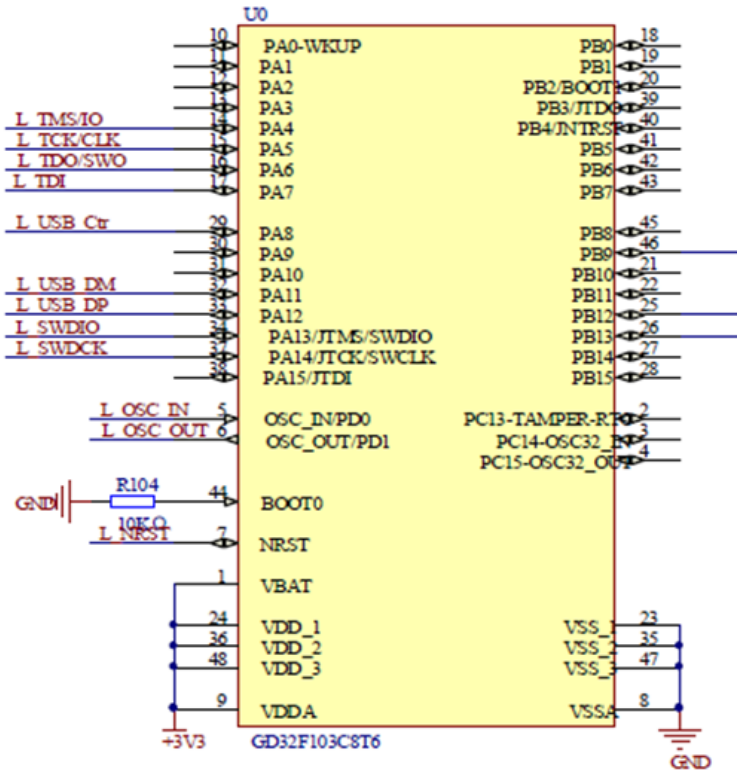


  - 启动选择

Selected boot source	Boot mode selection pins	
	Boot1	Boot0
Main Flash Memory	x	0
Boot loader	0	1
On-chip SRAM	1	1

GD-Linker & Jtag



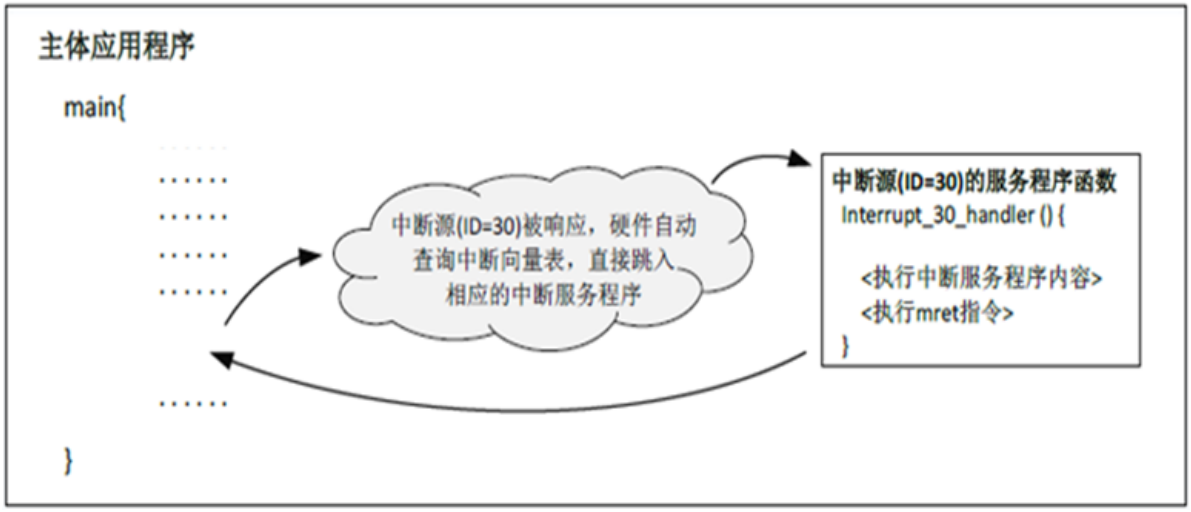


中断处理

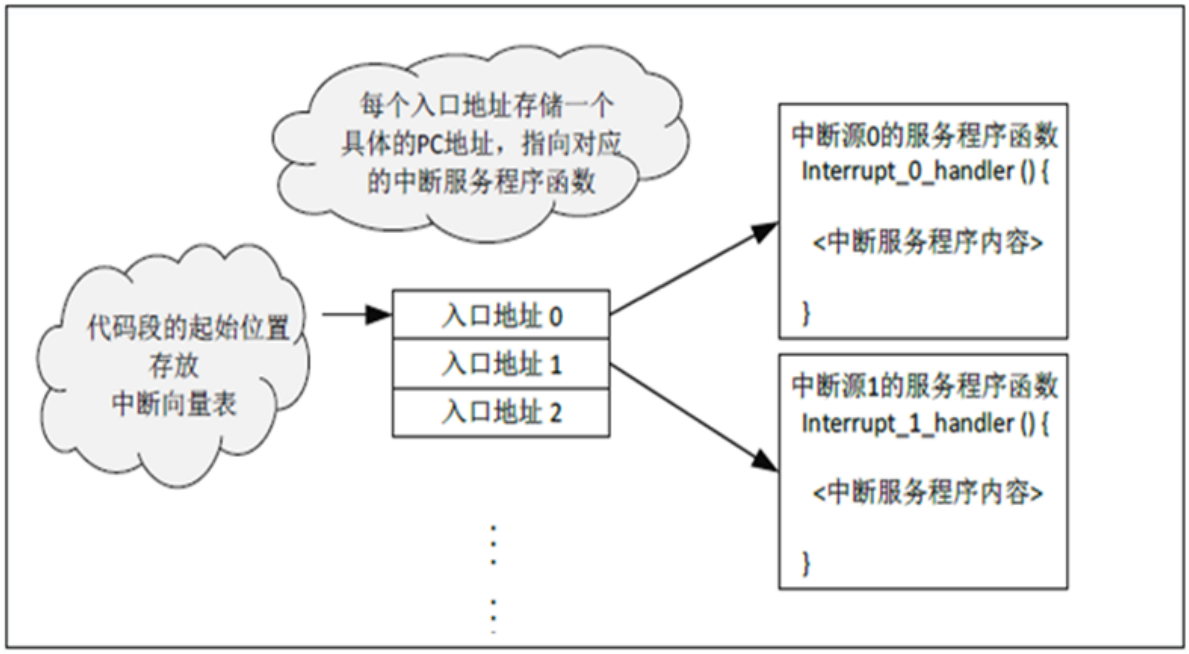
中断处理模式

- 向量中断处理模式和非向量中断处理模式。

向量中断

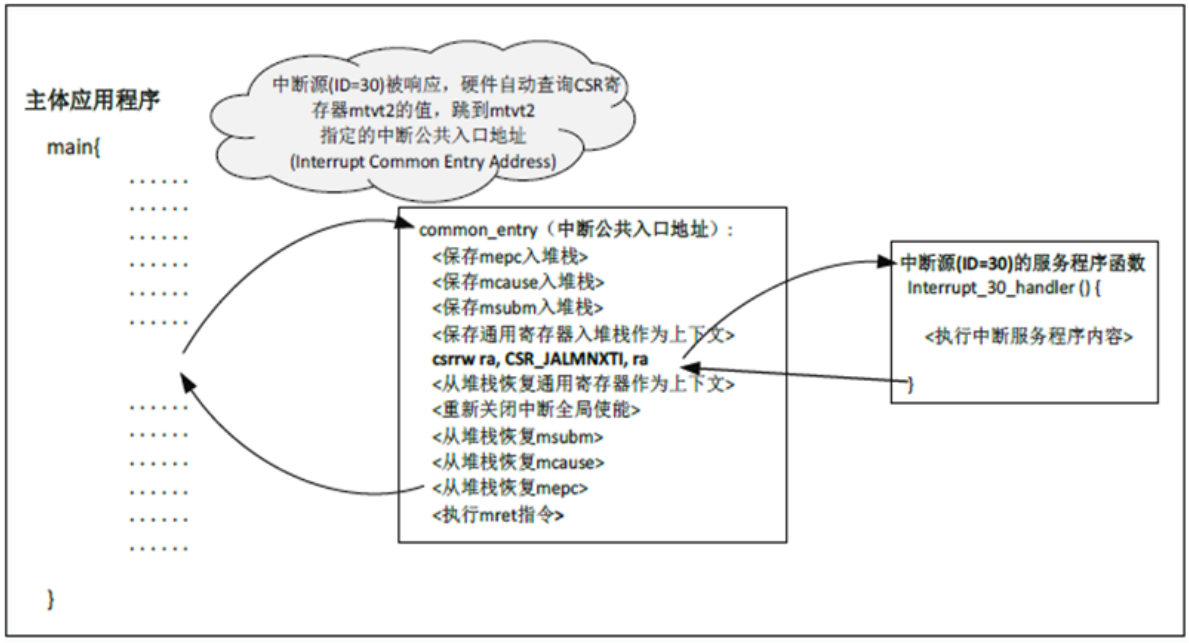


向量表



向量表中内容，中断服务程序地址或跳转语句

非向量中断

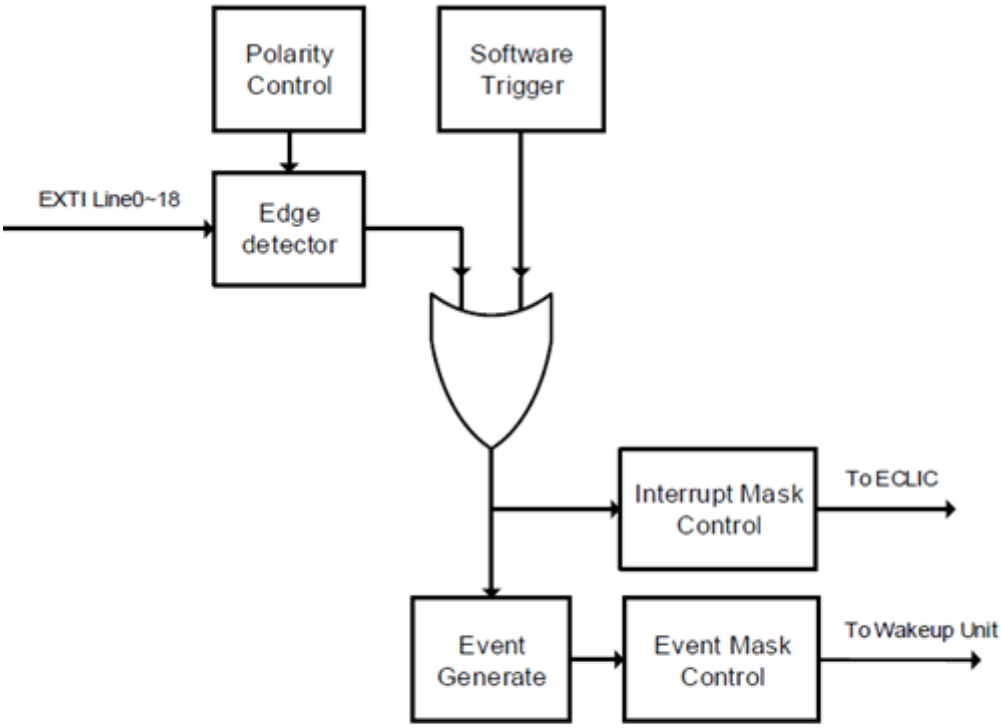


Bumblebee 非向量中断响应过程

中断服务程序函数：保存程序现场、处理中断事务和恢复程序现场。

GD32VF103中断机制

- 三层管理机制
- RISC-V异常处理；
- BumbleBee内核中断控制模块ECLIC；
- 外部中断和事件控制器（External interrupt and event，EXTI）。



GD32VF103 EXTI中断接口

EXTI寄存器组

名称	地址偏移	初始值	功能
中断使能 EXTI_INTEN	0x00	0x0000 0000	Bit 18:0, INT18:0使能 0：关闭；1：使能
事件使能 EXTI_EVEN	0x04	0x0000 0000	Bit 18:0, EV18:0使能 0：关闭；1：使能
上升沿触发 EXTI_RTEN	0x08	0x0000 0000	Bit 18:0, RT18:0使能 0：关闭；1：使能
下降沿触发 EXTI_FTEN	0x0c	0x0000 0000	Bit 18:0, FT18:0使能 0：关闭；1：使能
软中断事件 EXTI_SWIEV	0x10	0x0000 0000	Bit 18:0, SWI18:0使能 0：关闭；1：使能
中断状态 EXTI_PD	0x14	0x0000 0014	Bit 18:0, PD18:0使能 0：无触发；1：触发

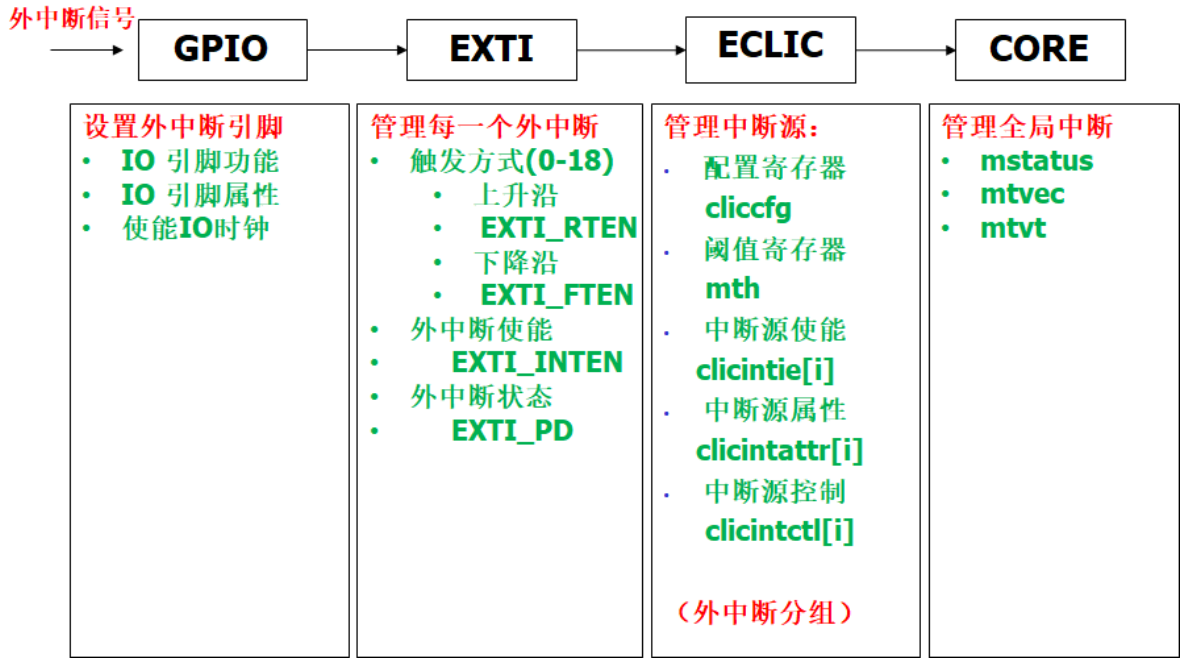
GD32VF103中断向量

- GD32VF103支持内核定时器中断，USART、I2C、ADC、DAC、FMC、SPI和RTC等所有芯片集成外设中断，以及EXIT[0:4]和EXTI[10:15]共11个外中断线。

Vector Number	Interrupt Description	Vector Address
3	CLIC_INT_SFT	0x0000_000C
7	CLIC_INT_TMR	0x0000_001C
17	CLIC_INT_BWEI	0x0000_0044
18	CLIC_INT_PMOVI	0x0000_0048
19	WWDGT interrupt	0x0000_004C
20	LVD from EXTI interrupt	0x0000_0050
21	Tamper interrupt	0x0000_0054
22	RTC global interrupt	0x0000_0058
23	FMC global interrupt	0x0000_005C
24	RCU global interrupt	0x0000_0060
25	EXTI Line0 interrupt	0x0000_0064
30	DMA0 channel0 global interrupt	0x0000_0078
37	ADC0 and ADC1 global interrupt	0x0000_0094
38	CAN0 TX interrupts	0x0000_0098
43	TIMER0 break interrupt	0x0000_00AC
50	I2C0 event interrupt	0x0000_00C8
54	SPI0 global interrupt	0x0000_00D8
56	USART0 global interrupt	0x0000_00E0
86	USBFS global interrupt	0x0000_0158

系统设置

外中断请求路径



EXIT设置

初始化

```
void exti_init(exti_line_enum linex, exti_mode_enum mode, exti_trig_type_enum
trig_type)
{switch (mode) {    //选则请求模式：中断/事件？
        case EXTI_INTERRUPT:
                EXTI_INTEN |= (uint32_t) linex;  break
        case EXTI_EVENT:
                EXTI_EVEN |= (uint32_t) linex;   break;
        ….    }
   switch (trig_type) {//选择触边沿：上升/下降/上升和下降
     case EXTI_TRIG_RISING:
                EXTI_RTEN |= (uint32_t) linex;
                EXTI_FTEN &= ~(uint32_t) linex;  break;
        case EXTI_TRIG_FALLING:
                EXTI_RTEN &= ~(uint32_t) linex;
                EXTI_FTEN |= (uint32_t) linex;  break;
        case EXTI_TRIG_BOTH:
                EXTI_RTEN |= (uint32_t) linex;
                EXTI_FTEN |= (uint32_t) linex;  break;
    ……}
}
```

使能

```
void exti_interrupt_enable(exti_line_enum linex)
{
    EXTI_INTEN |= (uint32_t) linex;  //使能外中断: x=0...18
}
void exti_event_enable(exti_line_enum linex)
{
    EXTI_EVEN |= (uint32_t) linex;    //事件使能
}
```

读取状态

```
FlagStatus exti_flag_get(exti_line_enum linex)
{
    if (RESET != (EXTI_PD & (uint32_t) linex)) {
        return SET;
    } else {
        return RESET;
    }
}
```

ECLIC设置

初始化

```
void eclic_init(uint32_t num_irq) {
        typedef volatile uint32_t vuint32_t;
        *(volatile uint8_t*) (ECLIC_ADDR_BASE + ECLIC_CFG_OFFSET) = 0;//清除设置
        *(volatile uint8_t*) (ECLIC_ADDR_BASE + ECLIC_MTH_OFFSET) = 0; //清除设置
         vuint32_t * ptr;   //清除IP/IE/ATTR/CTRL位
         vuint32_t * base = (vuint32_t*) (ECLIC_ADDR_BASE +
ECLIC_INT_IP_OFFSET);
         vuint32_t * upper = (vuint32_t*) (base + num_irq * 4);
         for (ptr = base; ptr < upper; ptr = ptr + 4) {
        *ptr = 0; }
         eclic_set_nlbits(ECLIC_GROUP_LEVEL2_PRIO2);
}
```

中断使能

```
void eclic_irq_enable(uint32_t source, uint8_t level, uint8_t priority) {
    eclic_enable_interrupt(source);   //使能
    eclic_set_int_level(source, level); //设定源级
    eclic_set_int_priority(source, priority); //设定优先级
}
```

设置中断级

```
uint8_t eclic_set_int_level(uint32_t source, uint8_t level) { //设定中断级
    uint8_t nlbits = eclic_get_nlbits(); // 获取特定位
    if (nlbits > ECLICINTCTLBITS) { nlbits = ECLICINTCTLBITS; }
     level = level << (8 - nlbits); //移位
     uint8_t current_intctrl = eclic_get_intctrl(source); //写入控制位
     current_intctrl = current_intctrl << nlbits; //移位
      current_intctrl = current_intctrl >> nlbits; //移位
     eclic_set_intctrl(source, (current_intctrl | level));
     return level;
}
```

设置优先级

```
uint8_t eclic_set_int_priority(uint32_t source, uint8_t priority) {
    uint8_t nlbits = eclic_get_nlbits();
    if (nlbits >= ECLICINTCTLBITS) {nlbits = ECLICINTCTLBITS; return 0;}
    priority = priority << (8 - ECLICINTCTLBITS);
    uint8_t current_intctrl = eclic_get_intctrl(source);
    current_intctrl = current_intctrl >> (8-nlbits);
    current_intctrl = current_intctrl << (8-nlbits);
    eclic_set_intctrl(source, (current_intctrl | priority));
     return priority;}
```

初始相关寄存器

```
void eclic_set_intctrl(uint32_t source, uint8_t intctrl) {  //中断源控制寄存器
    *(volatile uint8_t*) (ECLIC_ADDR_BASE + ECLIC_INT_CTRL_OFFSET + source * 4)
=
     intctrl;
}
void eclic_set_intattr(uint32_t source, uint8_t intattr) {  //中断源属性寄存器
    *(volatile uint8_t*) (ECLIC_ADDR_BASE + ECLIC_INT_ATTR_OFFSET + source * 4)
=
            intattr;
}
void eclic_set_eclicCfg(uint8_t eclicCfg) { // 配置寄存器
     *(volatile uint8_t*) (ECLIC_ADDR_BASE + ECLIC_CFG_OFFSET) = eclicCfg;
}
void eclic_set_mth(uint8_t mth) { //阈值
     *(volatile uint8_t*) (ECLIC_ADDR_BASE + ECLIC_MTH_OFFSET) = mth;
}
void eclic_enable_interrupt(uint32_t source) { //使能中断源
       *(volatile uint8_t*) (ECLIC_ADDR_BASE + ECLIC_INT_IE_OFFSET + source * 4)
= 1;
}
```

CORE设置

全局中断

```
void eclic_global_interrupt_enable()   //使能全局中断
{
     set_csr(mstatus, MSTATUS_MIE);
     return;
}
```

设置中断处理方式（ECLIC/普通）

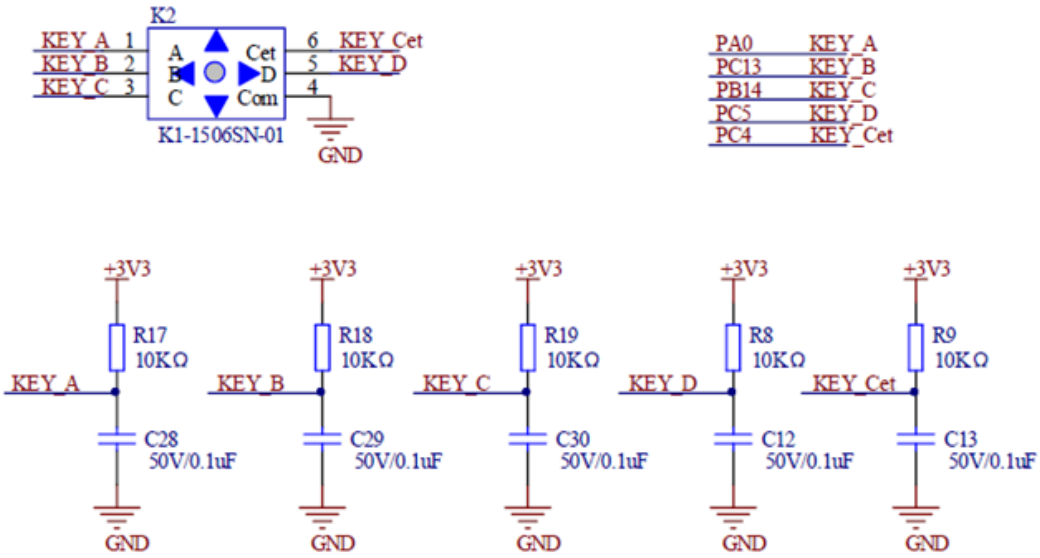
```
void eclic_mode_enable() {
    uint32_t mtvec_value = read_csr(mtvec);
    mtvec_value = mtvec_value & 0xFFFFFFC0;
    mtvec_value = mtvec_value | 0x00000003;    //ECLIC中断方式
    write_csr(mtvec, mtvec_value);
}
```

设置向量响应方式（向量/非向量）

```
void eclic_set_shv(uint32_t source, uint8_t shv) { //shv=1,  向量中断
    uint8_t attr = eclic_get_intattr(source);
    if (shv) {
        attr |= 0x01;
        eclic_set_intattr(source, attr);}
      }
```

键盘中断

原理图



KEY_B 连接到PC13。PC13 也是处理器的外中断EXTI_13的输入线。设置GPIO控制器的寄存器，将PC13引脚配置成中断输入线。

中断向量表

```
.section .vectors, "ax"              //向量段
    ……
  .weak   EXTI5_9_IRQHandler         //声明中断ID5至ID9处理程序
    ……
  .weak   EXTI10_15_IRQHandler       //声明中断ID10至ID15处理程序
    ……
.globl vector_base
vector_base:                        //中断向量表基地址
  j Reset_Handler                   //跳转到复位处理程序
```

```
  .align    2                //四字节对齐
   ……
  .word     EXTI5_9_IRQHandler      //中断ID5至ID9处理程序入口地址
   ……
  .word     EXTI10_15_IRQHandler    //中断ID10到15处理程序入口地址，Int59
   ……
```

选择向量EXTI13中断模式

```
#define ECLIC_ADDR_BASE      0xd2000000 //ECLIC 映射基地址
#define ECLIC_INT_ATTR_OFFSET  _AC(0x1002,UL) //中断属性寄存器偏移
eclic_set_shv(EXTI10_15, 1) ;  //设为向量中断型，EXTI10_15=59
```

开启中断

```
//使能全局中断
global_interrupt_enable()
//函数调用
eclic_enable_interrupt(eclic_set_shv(EXTI10_15, 1)) ;
```

设置向量表基址(ECLIC)

```
 //将向量表基地址写入寄存器t0
la t0, vector_base
//写入CSR寄存器，ECLIC 向量基址寄存器
csrw CSR_MTVT, t0
```

中断服务程序

```
__attribute__((interrupt)) void EXTI10_15_IRQHandler()
{
    if(RESET != exti_interrupt_flag_get(EXTI_13)){ //EXTI_13?
        exti_interrupt_flag_clear(EXTI_13); //清除中断状态
        led_flash(2);                       //执行中断任务，LED闪烁
        }
    eclic_global_interrupt_enable();     //打开全局中断使能
    return;
}
```

主程序

```
int main(void)
{
    //初始化连接LED的GPIO引脚
    led_init();     //初始化LED
    //使能引脚时钟
    rcu_periph_clock_enable(RCU_GPIOC); //开启GPIO时钟
    rcu_periph_clock_enable(RCU_AF);    //开启功能引脚时钟
    eclic_set_shv(EXTI10_15, 1) ;   //设为向量中断型，EXTI10_15=59
    global_interrupt_enable();  //使能全局中断
    eclic_priority_group_set(ECLIC_PRIGROUP_LEVEL3_PRIO1);//设优先级
    gpio_init(GPIOC, GPIO_MODE_IN_FLOATING, GPIO_OSPEED_50MHZ,
GPIO_PIN_13); //设置PC13属性
```

```
    gpio_exti_source_select(GPIO_PORT_SOURCE_GPIOC,
GPIO_PIN_SOURCE_13);//将PC13设为中断
    eclic_irq_enable(EXTI10_15_IRQn, 1, 1);      //使能中断 EXTI10-15
    exti_init(EXTI_13, EXTI_INTERRUPT, EXTI_TRIG_FALLING); //设置触发方式
    exti_interrupt_flag_clear(EXTI_13);  //清除中断状态位
    while(1){__asm("wfi");  }          //等待
}
```

小结

Lab

- **在评估版上调试键盘中断程序（可选触屏）**

Embedded studio

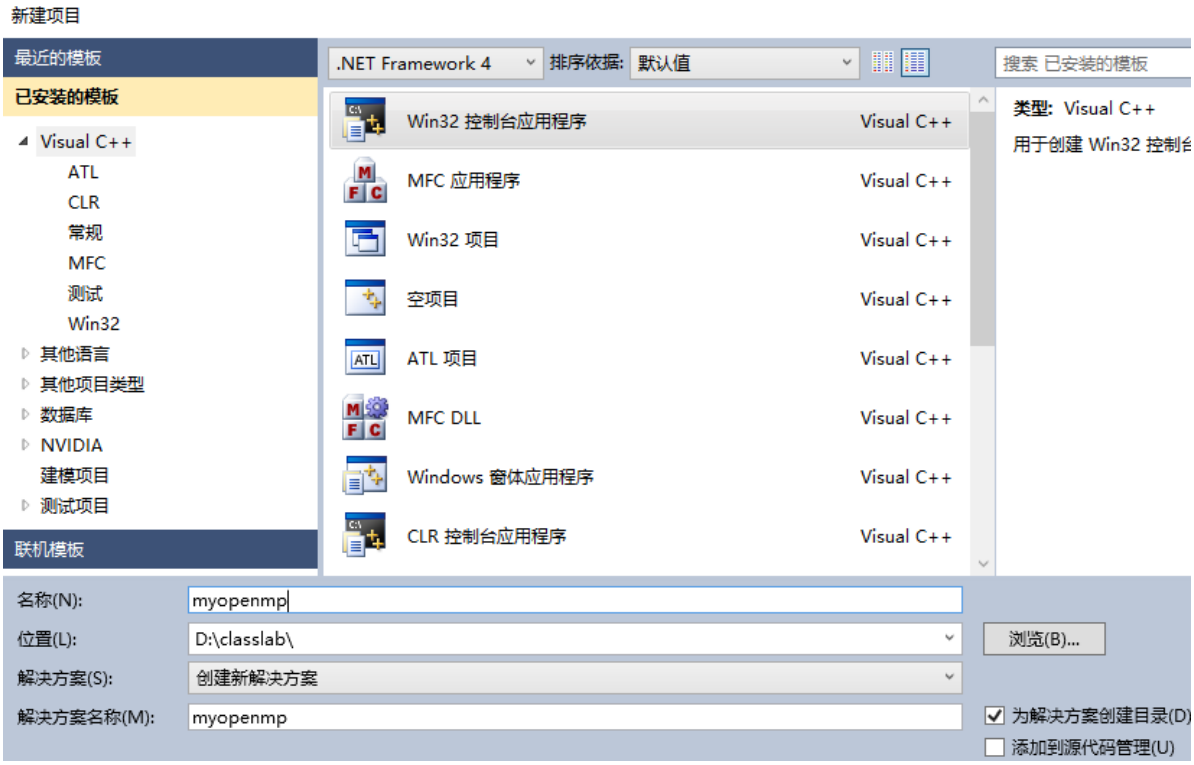
- 在三实验中选择一个，编译，调试。
- 修改参考程序（界面或功能），重新编译，调试。

课后

- 安装 OpenMP开发环境，并调试运行下列程序。

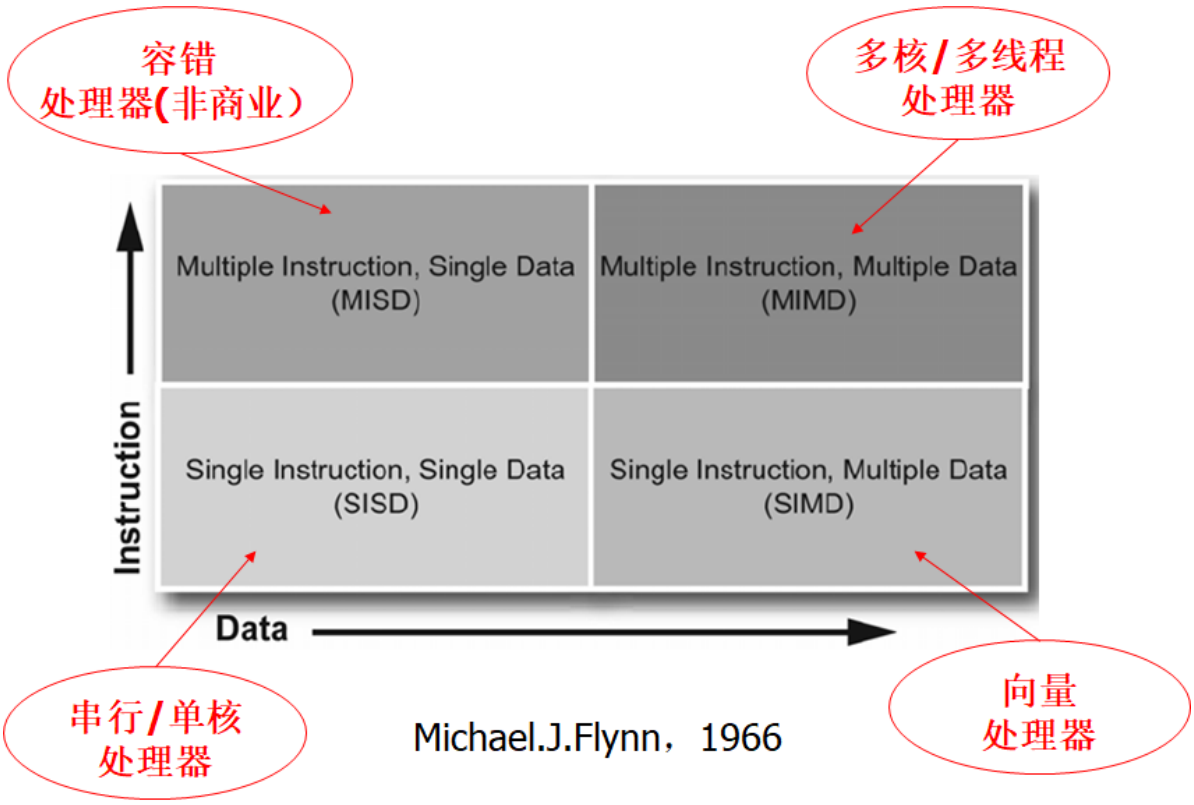
```
#include "stdafx.h"
#include "omp.h"
int _tmain(int argc, _TCHAR* argv[])
{
    printf("Hello from serial.\n");
    printf("Thread number = %d\n",
    omp_get_thread_num());  //serial
    #pragma omp parallel        //parallel
    {
    printf("Hello from parallel.Thread number = %d\n",
        omp_get_thread_num());
    }
    printf("Hello from serial again.\n");
    return;
}
```

Programming with VS

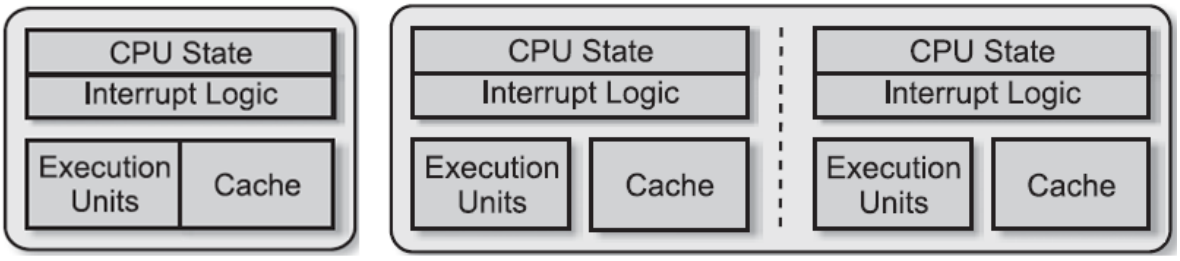


多核处理器-OpenMP

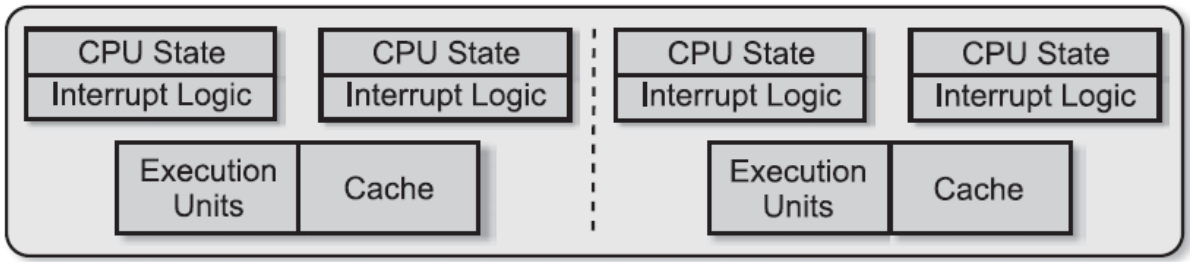
并行计算



处理器结构

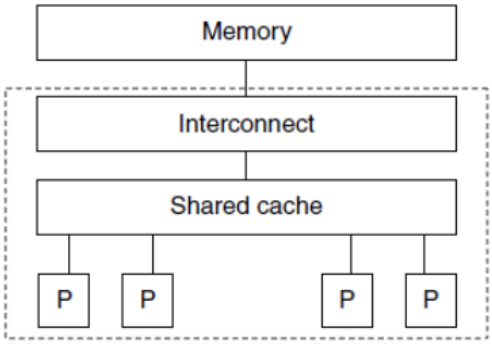


单核处理器　　多核处理器

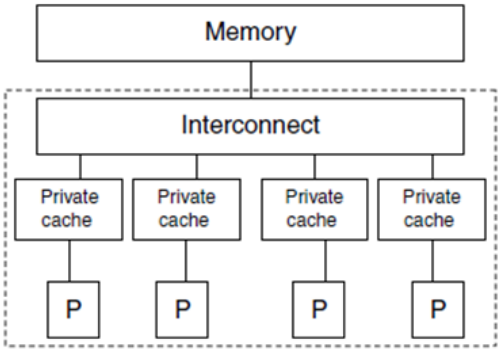


多线程多核处理器

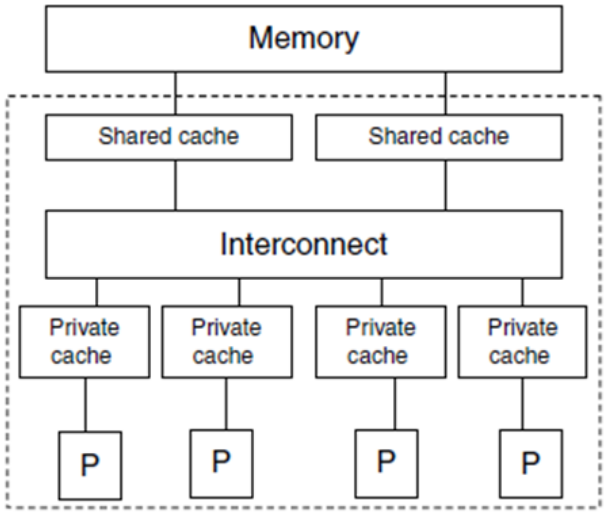
存储结构



Shared cache only for all cores

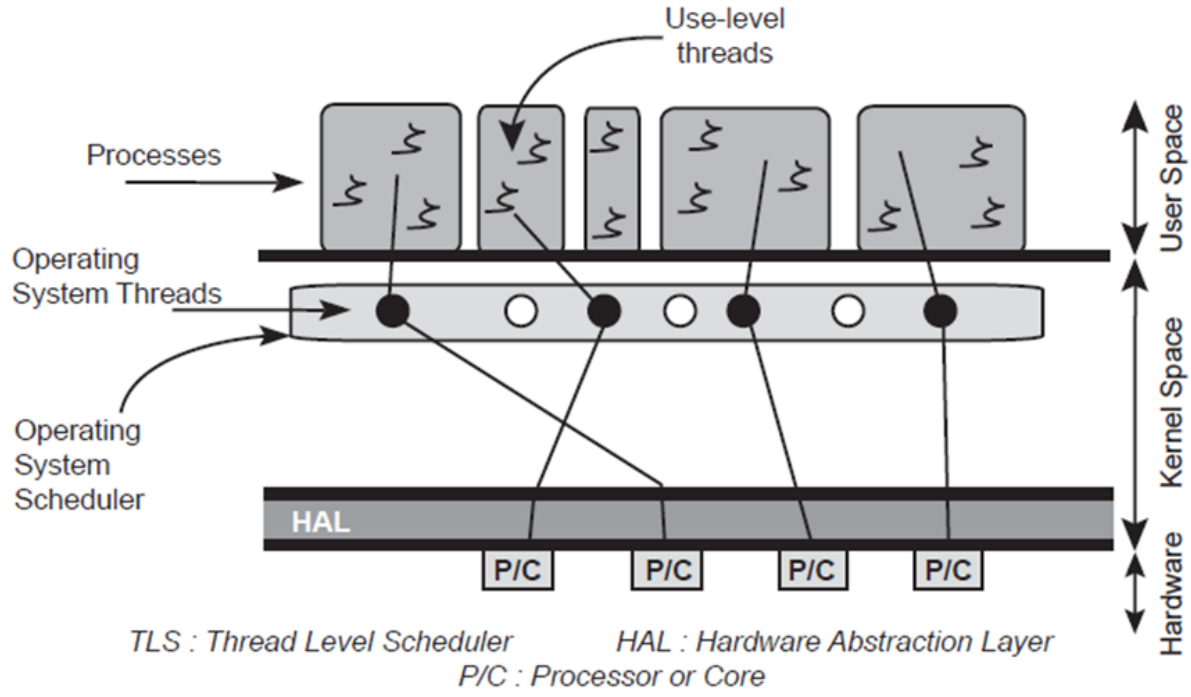


Private cache only for each cores



Multi-level architecture with both private and shared cache

线程映射



Mapping Threads to Processors

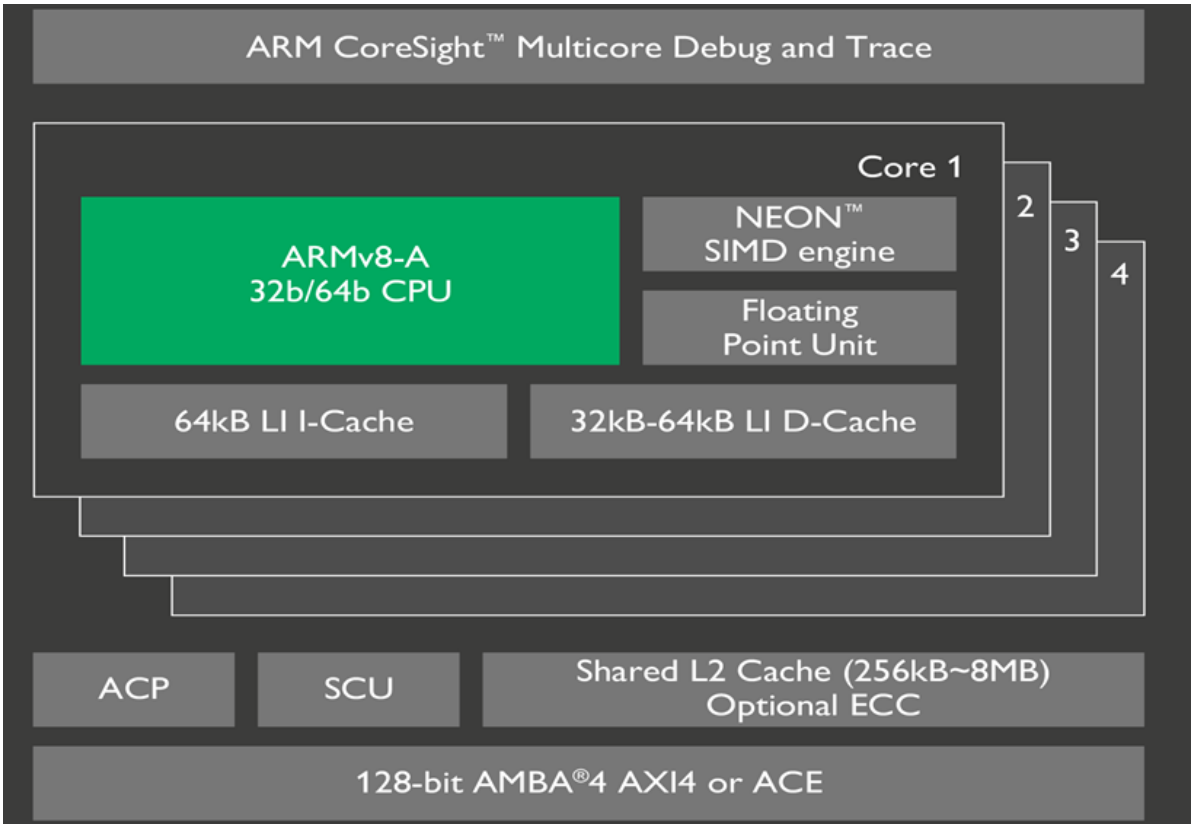
- HAL：硬件抽象层
- 多核必须操作系统支持

ARM MPcore

Accelerator Coherency Port, ACP

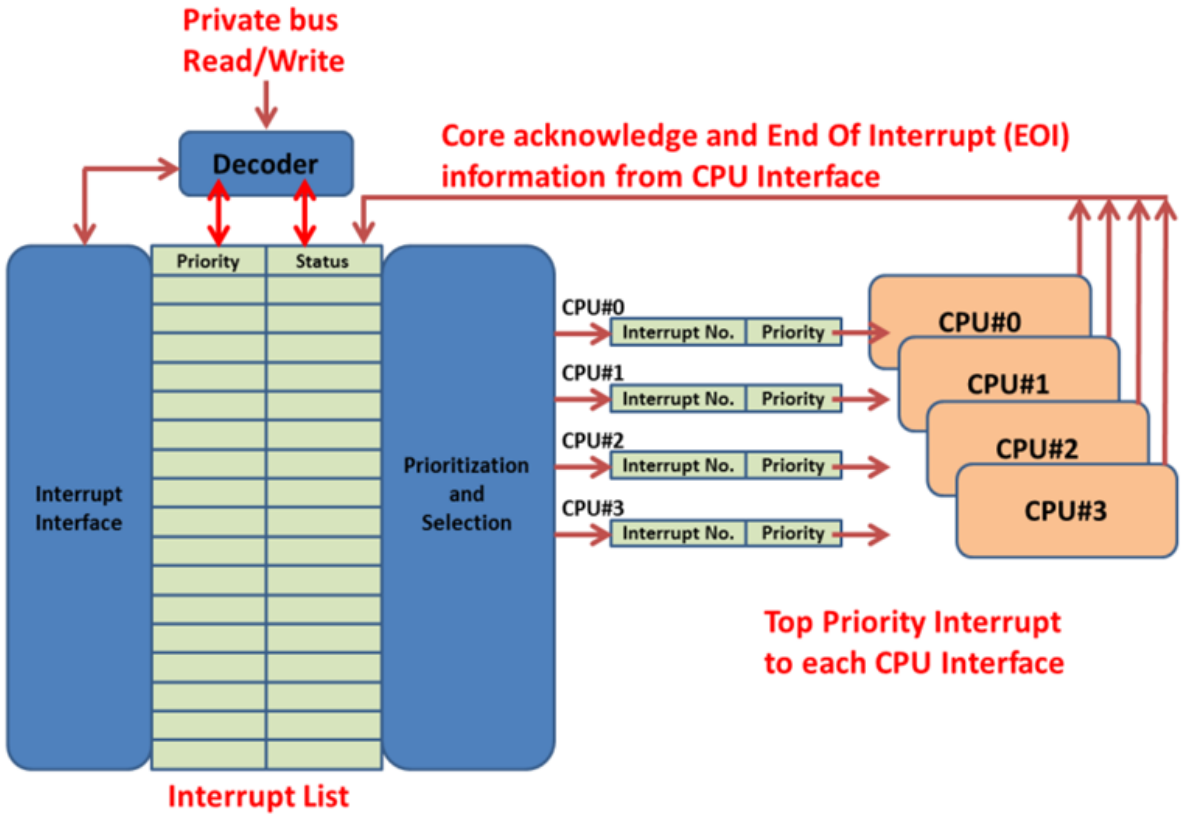
Snoop Control Unit，SCU

MPIDR：内核 ID 寄存器



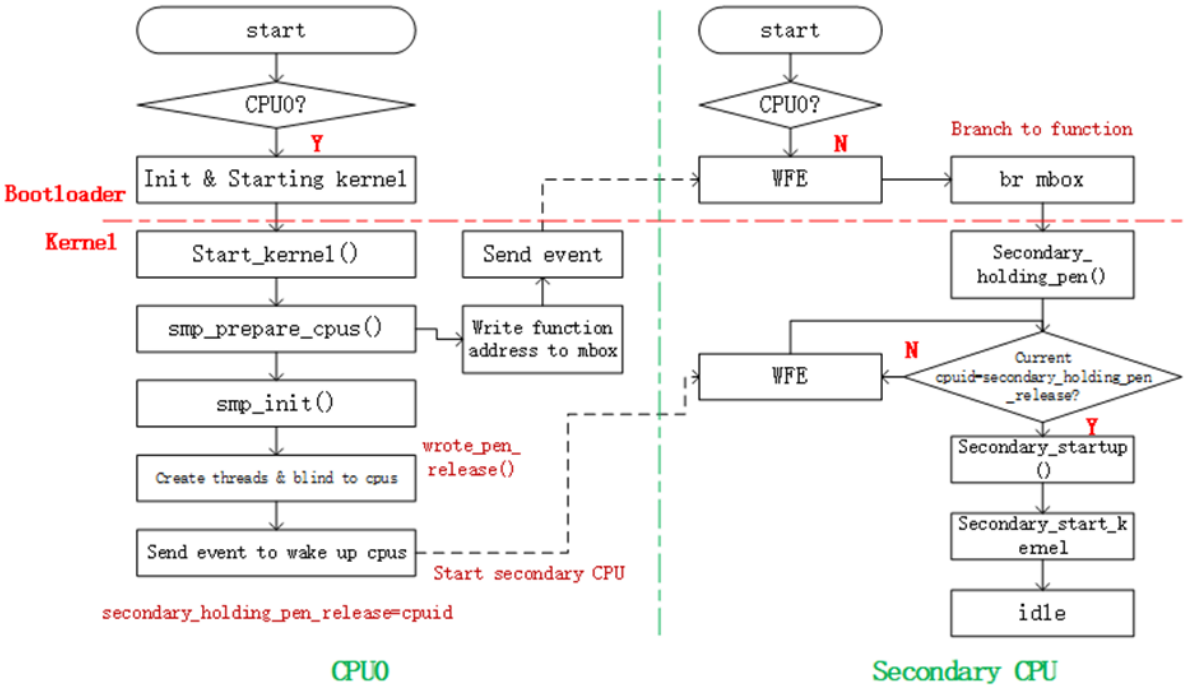
> Advanced Custom Extension (ACE)
>
> 提供一个额外的PE（process element）识别机制属性：MPIDR_EL1是一个64位的寄存器
> MPIDR_EL1是一个64位的寄存器。
>
> 域值：
>
> [63:40]：Reserved, RES0；
>
> [39:32]：Affinity level3；
>
> [31]：Reserved, RES1.
>
> [30]：U 表示一个单处理器系统，与多处理器系统中的pe0不同。这个位的可能值是:0b0 ： 处理器是多处理器系统的一部分。0b1 ： 处理器是单处理器系统的一部分[29:25]：Reserved, RES0.
>
> [24]：MT指示关联的最低级别是否由使用多线程类型方法实现的逻辑PEs组成。这个位的值可能是：0b0 当PEs的性能关联级别最低，或者使用MPIDR_EL1.MT的PEs被设置为1时，级别0的不同值或者级别1的相同值或更高级别，是相互独立的0b1 当PEs的性能关联级别最低，或者使用MPIDR_EL1.MT的PEs被设置为1时，级别0的不同值或者级别1的相同值或更高级别的相关性非常高。
>
> [23:16]：Affinity level 2.
>
> [15:8]：Affinity level 1. [11:8], 0x00-0x07: Core0-core7
>
> [7:0]：Affinity level 0. 这个Affinity等级对于确定PE行为最为重要。更高级别的affinity 等级的重要性越低。分配给MPIDR的值的域的集合{Aff2, Aff1, Aff0}在整个系统中必须是不同的访问MPIDR_EL1：MRS , MPIDR_EL1

Interrupt



Power on flow

可以用CPU1作为启动核吗?



Bootloader

```
    /*CPU0 */
    mrs  x4, mpidr_el1    //读取 寄存器
    tst    x4,#15               //testwether the current cpu is CPU0,
      //ie. mpidr_el1=15
    b.eq 2f
    /*  Secondary CPUs */
1:
              wfe
    ldr x4, mbox
    cbz x4, 1b        //if x4==0(ie. The value in address of mbox

                        //is 0) dead loop,or jump to x4
    br x4                // branch to thegiven address
2:
                ……                  //UART initialisation
```

- mbox的地址在Makefile中写定，保存在dts 文件中。
- dts即Device Tree Source 设备树源码, DeviceTree是一种描述硬件的数据结构.

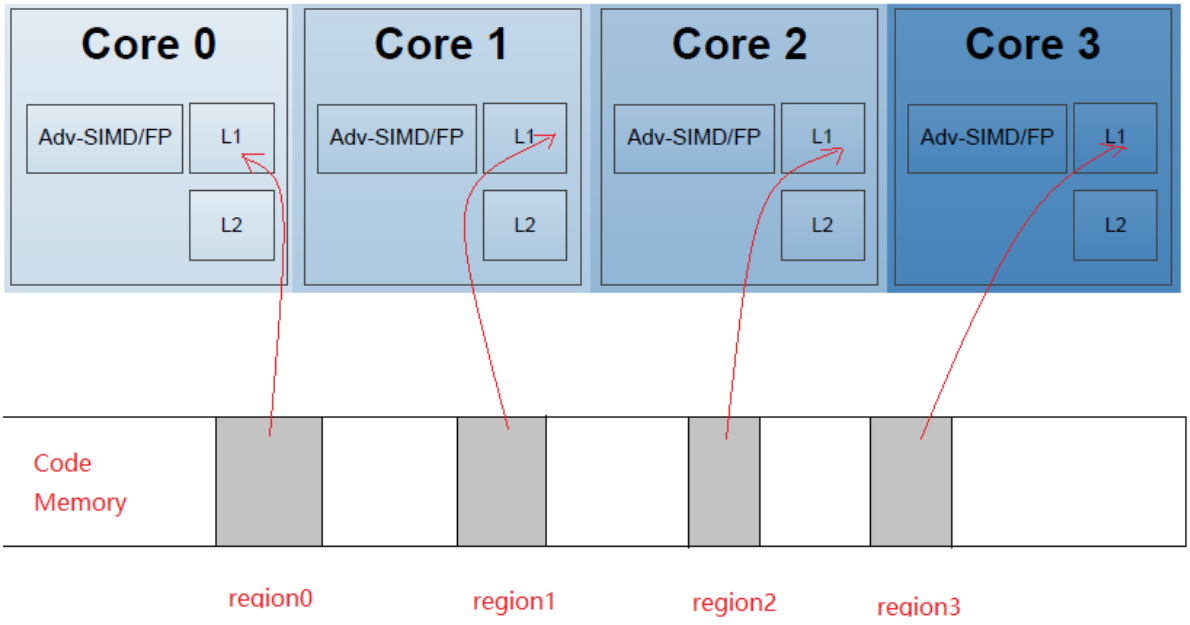
> el0：用户模式
>
> el1：管理员模式（什么中断，复位以后的模式，最常用的）
>
> el2：虚拟机
>
> el3：更高级的安全模式

Secondary_holding_pen()

```
/* provides a"holding pen" to hold all secondary cores */
ENTRY(secondary_holding_pen)
    bl      el2_setup                          // Drop to EL1
    mrs  x0, mpidr_el1
    and  x0, x0, #15                        // CPU number
    adr   x1, 1b
    ldp   x2, x3, [x1]
    sub   x1, x1, x2
    add  x3, x3, x1
pen:
    ldr    x4, [x3]
    cmp x4,x0
    b.eq secondary_startup
    wfe
    b       pen
    ENDPROC(secondary_holding_pen)
```

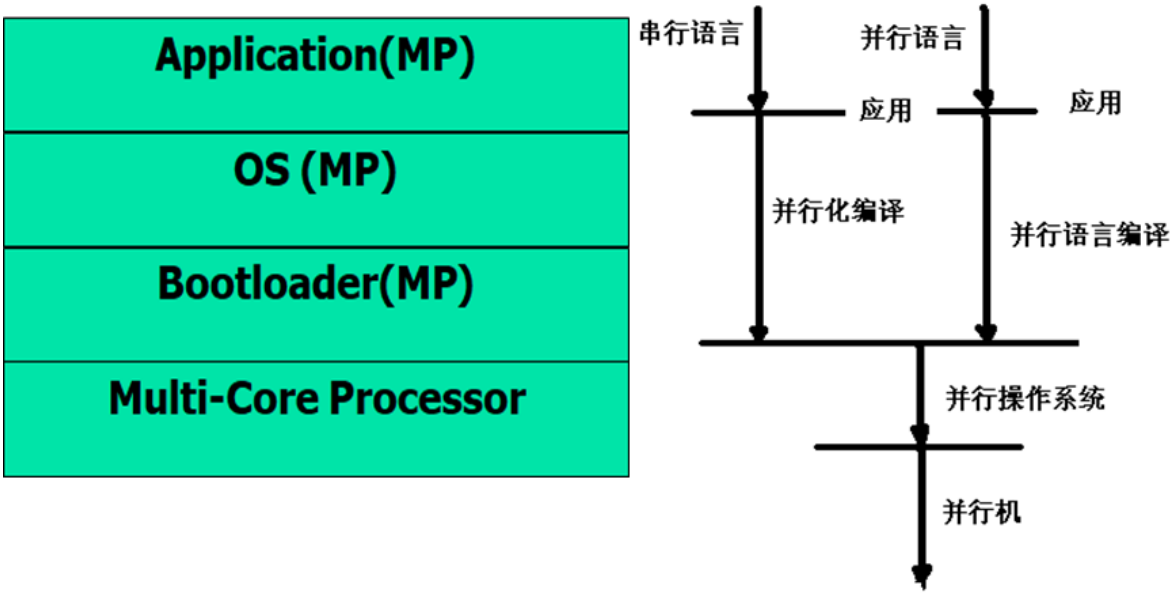
- ldp x1 , x0, [x1] ; 将[x1]中的值取出来, 放入x1 和 x0.

Fetching Instructions



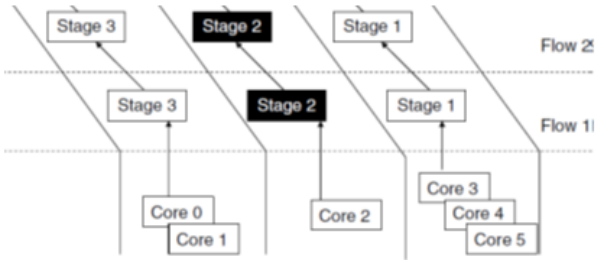
软件架构

运行环境　　开发方法



并行方法

- **Functional Decomposition**
  - Aims at dividing up the overall computation along functional dimensions.
  - **Example : h(x)=f(x)+g(x) => f, g** parallel
- **Data-Based Decomposition**
  - Targets the parallel execution of the same computation but on multiple data instances.
  - **Example : H=F+G, H， F** and **G** are matrix, element parallel
- **Helper computation**
  - Stages from different flows are independent, thus can be executed in parallel
  - **Example:**

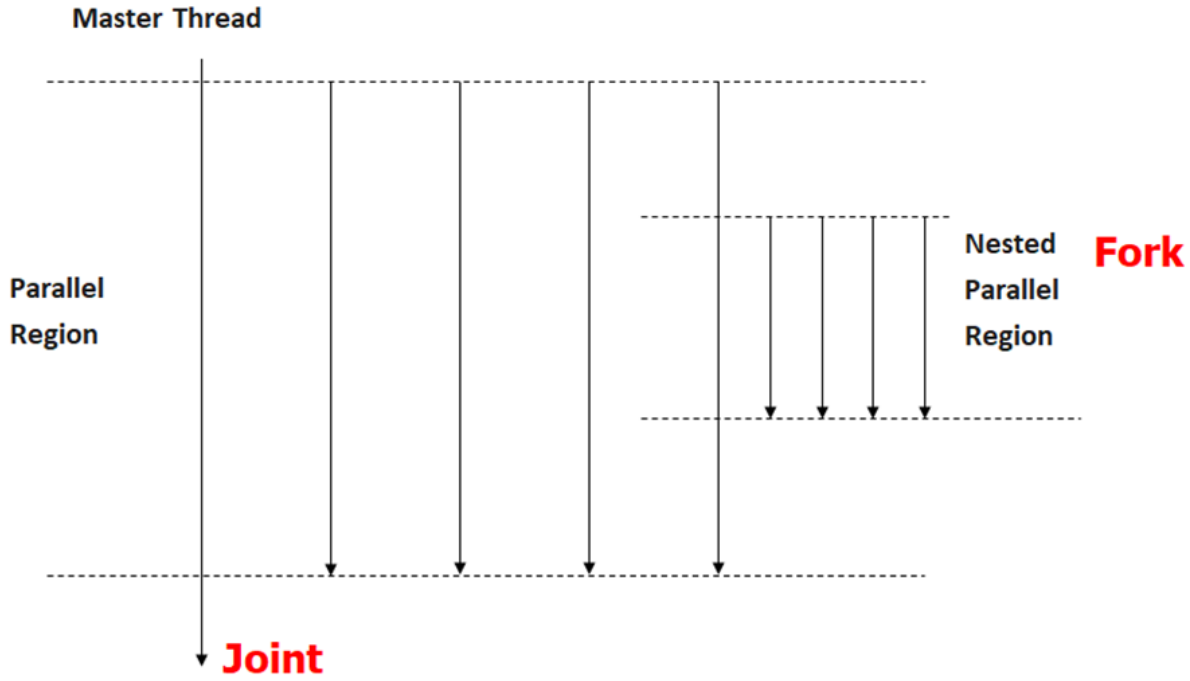


OpenMP

- 直接控制共享内存式并行编程的应用程序接口（API）
- 由三个主要的API组成:
  - 编译指令
  - 运行库 Runtime Library Routines
  - 环境变量 Environment Variables
- **特点**
  - 可移植性Portable:
  - 适合C/C++和Fortran的API
  - 已在多种主要系统实现（ Unix/Linux platforms and Windows NT ）
  - 由主要的硬件及软件厂家共同制定和支持
  - 主流 SMP 并行程序开发库

程序结构

Fork-Joint Mode



编程

- Implementation
  - #pragma
  - Run time library
- Using Environment varies to control program execution threads.

For example: OMP_NUM_THREADS

例程

```
#include "stdafx.h"
#include "omp.h"
int _tmain(int argc, _TCHAR* argv[])
{
    printf("Hello from serial.\n");
    printf("Thread number = %d\n",
    omp_get_thread_num());  //serial
    #pragma omp parallel        //parallel
    {
    printf("Hello from parallel.Thread number = %d\n",
        omp_get_thread_num());
    }
    printf("Hello from serial again.\n");
    return;
}
```

指令

- **#pragma**
  - parallel：表示这段代码将被并行执行。
  - for：表示将循环计算任务分配到多个线程中并行执行。
  - sections：用于实现多个结构块语句的任务分担。
  - parallel sections：类似于parallel for；
  - single：表示一段只被单个线程执行的代码；

- critical：保证每次只有一个OpenMP线程进入；
- flush：保证各个OpenMP线程的数据映像的一致性；
- barrier：用于并行域内代码的线程同步，线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行；atomic：用于指定一个数据操作需要原子性地完成；
- master：用于指定一段代码由主线程执行；
- Thread private：用于指定一个或多个变量是线程专用。

后面会解释线程专有和私有的区别。

循环

The results of two program ?

```
int step = 100;
int _tmain(int argc, _TCHAR*
argv[])
{
   int i;

   for (i = 0; i< step; i++)
  {
     printf("i = %d\n", i);
  }
  return 0;
}
```

```
int step = 100;
int _tmain(int argc, _TCHAR*
argv[])
{
   int i;
  #pragma omp parallel for
  for (i = 0; i< step; i++)
  {
     printf("i = %d\n", i);
  }
  return 0;
}
```

能从结果中估算出线程数或处理器的核数？

Are these two code segments equivalent?

```
#pragma omp parallel
{
     #pragma omp for
     for (i=0;i< MAX; i++)
     {     res[i] = huge();
     }
}
```

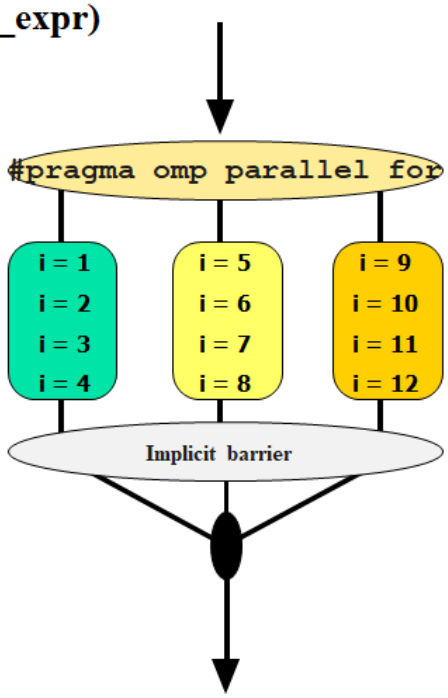
```
#pragma omp parallel for
for (i=0;i< MAX; i++)
{     res[i] = huge();
}
```

“#pragma” available scope?

#pragma omp end parallel

```
#pragma omp parallel for [clause[clause…]]
For(index=start ; index < end ; increment_expr)
{
      body of the loop;
}
```

```
#pragma omp parallel for
   for(i = 1, i < 13, i++)
      c[i] = a[i] + b[i]
```



Loop restrictions

- The loop variable must be type of *signed integer* (v2.5) .
- The comparison operation must be in the form *loop_variable*
  - <, <=, >, or >=*loop_invariant_integer.*
- The increment portion of the for loop must be *integer addition* or *integer subtraction* and by *a loop_invariant_value*.
- The loop must be *a single entry* and *single exit,* meaning no jump from the inside of loop to outside or outside to the inside are permitted with the exception of the exit statement. No use *break, return and goto*.

变量

Any problem for parallelizing next loop?

```
bool calsum(unsigned char *image, unsigned int *sum)
{   int i;
    unsigned in hist[256];
    for(i=255;i>=0;i--)
    {   hist[i]=0;
        *(sum+i)=0;
    }
    i=IMAGE_WIDTH*IMAGE_HEIGHT;
    //需要用for
    do{  i--;
        hist[*image++]++;
    }while(i>0);
     image=image-IMAGE_WIDTH*IMAGE_HEIGHT;
    *sum=hist[0];
    //for循环不能将i计算
    for( i=1;i<256;i++)
    {   *(sum+i)=hist[i]+*(sum+i-1); }
}
```

Challenge?

Loop-carrier dependency

下列程序哪个执行时间短?

```
ifirst = 10;
int k[600];
for(j = 0; j <= 60; j++)
{
    for(i=0;i<6000000;i++)
    {
      i2 = i-(i/600)*600;
      k[i2] = ifirst + i;
    }
}
```

```
ifirst = 10;
int k[600];
for(j = 0; j <= 60; j++)
{
#pragma omp parallel for
    for(i=0;i<6000000;i++){
      i2 = i-(i/600)*600;
      k[i2] = ifirst + i;
    }
}
```

clock_t t=clock(); 获取当前时钟，单位是**ms**

发现什么问题？为什么？

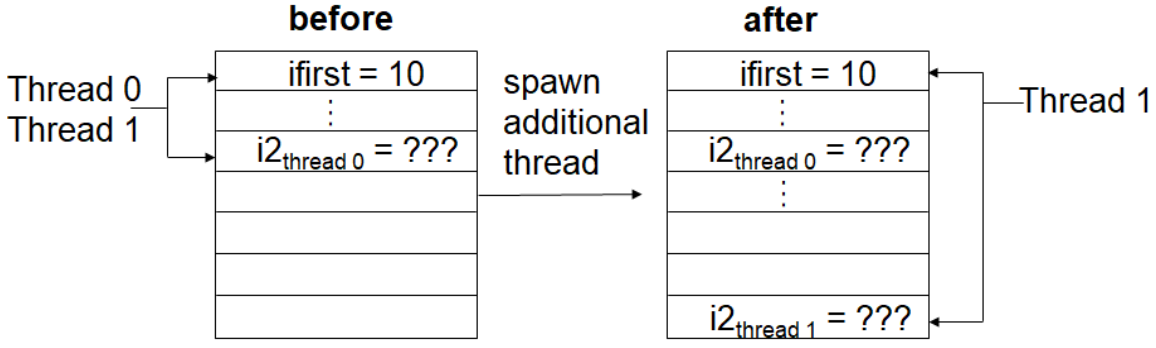
> 共享数据打架，并行反而更慢

- In parallel region, default behavior is that all variables are *shared* except loop index
  - All threads read and write the same memory location for each variable
  - This is ok if threads are accessing different elements of an array
  - Problem if threads write same scalar or array element
  - Loop index is *private*, so each thread has its own copy

看看下列程序的执行时间

```
ifirst = 10;
int k[600];
for(j = 0; j <= 60; j++)
{
#pragma omp parallel for private(i2)
    for(i=0;i<6000000;i++){
     i2 = i-(i/600)*600;
     k[i2] = ifirst + i;
    }
}
```

private



How about variables in function?

```
for(i = 0; i < 100; i++)
{
    mycalc(i,x,y);
}
```

how about x, y ?

试试这个程序

```
float sum = 0.0;
float a[10000],b[10000];
#pragma omp parallel for private(sum)
   for(int i=0; i<10000; i++) {
     sum += a[i] * b[i];
   }
  return sum;
```

编译错误?

试试（shared）

```
float sum = 0.0;
float a[10000],b[10000];
For(int j=0;j<10;j++)
{
    sum=0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<10000; i++) {
     sum += a[i] * b[i];
   }
    printf("j=%d, sum=%f\n",j,sum);
}
  return sum;
```

每次结果相同？为什么?

```
float sum = 0.0;
float a[10000],b[10000];
for(int j=0;j<10;j++)
{
    sum=0.0;
#pragma omp parallel for
   for(int i=0; i<10000; i++) {
     sum += a[i] * b[i];
   }
    printf("j=%d, sum=%f\n",j,sum);
}
  return sum;
```

Critical

Solution 1

```
float sum = 0.0;
flaot a[10000],b[10000];
for(int j=0;j<10;j++)
{
    sum=0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
    #pragma omp critical    //让每次输出结果相同
        sum += a[i] * b[i];
    }
    printf("j=%d, sum=%f\n",j,sum);
}
return sum;
```

Critical Construct

```
#pragma omp critical [(lock_name)]
```

Defines a critical region on a structured block

Threads wait their turn –at a time, only one calls consum() thereby protecting RES from race conditions

Naming the critical construct RES_lock is optional

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (RES_lock)
    consum (B, RES);
  }
}
```

> 加锁：RES_lock

```
void dequeue(NODE * node)
{
   #pragma omp critical(x)
   {
    node=node->next;
   }
}

void do_work(NODE *node)
{
 #pragma omp critical(x)
   {
    node->next->data=fn1(node->data);
    node=dequeue(node);
   }
}
```

What's wrong? Deadlock!

Atomic

```
#pragma omp parallel for shared(x, y, index, n)
   for (i = 0; i < n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
   }
```

- Special case of a critical section
- Applies only to **simple update of memory location**

reduction

Solution 2

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
   for(int i=0; i<N; i++) {
    sum += a[i] * b[i];
   }
  return sum;
}
```

- Each thread performs **its own reduction** (sum, in this case);
- Results from all threads are automatically reduced (summed) **at the end of the loop**

线程

Any program paralleled?

Fibonacci Sequence 0, 1, 1, 2, 3, 5, 8, 13.....

```
a[1] = 0;
a[2] = 1;
for(i = 3; i <= 100; i++){
    a[i] = a[i-1] + a[i-2];
}
```

程序运行结果?

```
doube a[100];
a[0] = 0.0;
a[1] = 1.0;
for (i = 2; i < 100; i++)
    a[i] = 0;
#pragma omp parallel for
for (i = 2; i < 100; i++)
{
    a[i] = a[i - 1] + a[i - 2];
    printf("a[%d]= %f\n",i,a[i]);
}
```

为什么?

- **Data on one thread can be dependent on data on another thread**
- **This can result in wrong answers**
  - **thread 0 may require a variable that is calculated on thread 1**
  - **answer depends on timing – When thread 0 does the calculation, has thread 1 calculated it's value yet?**
- **parallelize on 2 threads（for example）**
  - **thread 0 gets i = 3 to 51**
  - **thread 1 gets i = 52 to 100**
  - **look carefully at calculation for i = 52 on thread 1**
    - **what will be values of for i -1 and i - 2 ?**

如何解决？　　取消并行？

线程

- Assigning Iterations
  - Which Schedule to Use

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

- Assigning Iterations
  - **schedule(static [,chunk])**
    - Blocks of iterations of size “chunk” to threads
    - Round robin distribution

```
#pragma omp parallel for schedule (static, 8)
for (i = 0; i <20; i++){
    a[i] =i*i;
    printf("static_a[%d]= %d, num=%d\n", i,a[i],    omp_get_thread_num());
}
```

- Iterations are divided into chunks of 8
- If start = 0, then first chunk is
  - thread0{0, 1,2,3,4,5,6,7}
  - thread1{8, 9,2,10,11,12,13,14,15}
  - thread2{16,16,18,19}
- **schedule(dynamic[,chunk])**
  - Threads grab “chunk” iterations
  - When done with iterations, thread requests next set

```
#pragma omp parallel for schedule (dynamic, 8)
for (i = 0; i <20; i++){
    a[i] =i*i;
    printf("static_a[%d]= %d, num=%d\n", i,a[i],    omp_get_thread_num());
}
```

- Iterations are divided into chunks of 8
- If start = 0, then first chunk is
  - thread0{0, 1,2,3,4,5,6,7}
  - thread2{8, 9,2,10,11,12,13,14,15}
  - thread3{16,16,18,19}
- **schedule(guided[,chunk])**

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”

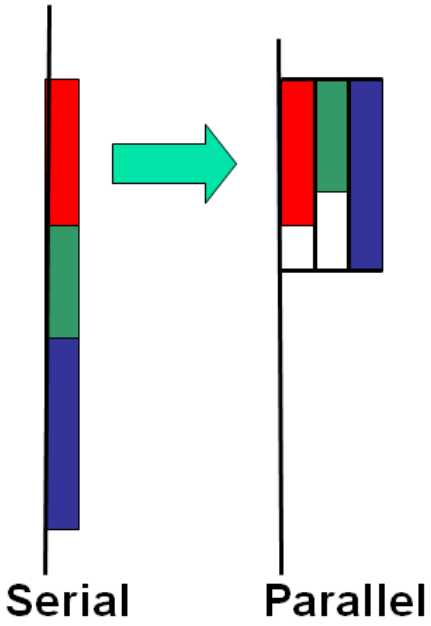
```
#pragma omp parallel for schedule (guided)
for (i = 0; i <20; i++){
    a[i] =i*i;
    printf("static_a[%d]= %d, num=%d\n", i,a[i],    omp_get_thread_num());
}
```

- thread0{0, 1,2,3,4}
- thread1{9, 10,11}
- thread2{5,6,7,8} {14,15,16,17,18,19}
- thread3{12,13}

Section

```
#pragma omp parallel sections
#pragma omp section
 init_field(a);
#pragma omp section
 check_grid(x);
#pragma omp section
 process_data(y);
#pragma omp end parallel sections
```

- **In parallel region, have several independent tasks;**
- **Divide code into sections;**
- **Each section is executed on a different thread.**



Barrier

```
#pragma omp parallel private(myid,istart,iend)
myrange(myid,istart,iend);
for(i=istart; i<=iend; i++){
   a[i] = a[i] - b[i];
}
#pragma omp barrier
myval[myid] = a[istart] + a[0]
```

- **Barrier synchronizes threads**
- Here barrier assures that a[istart] or a[0] is available before computing myval

Nowait

```
#pragma omp for nowait
   for(...)
     {...};
```

```
#pragma single nowait
{ [...] }
```

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```

To remove implicit barrier, Used when threads would wait between independent computations .

Master

```
#pragma omp parallel private(myid, istart, iend)
 myrange(myid, istart, iend);
 for(i=istart; i<=iend; i++){
    a[i] = a[i] - b[i];
 }
 #pragma omp barrier
 #pragma omp master
 fwrite(fid, sizeof(float), iend-istart+1, a);
 #pragma omp end master
 do_work(istart, iend);
 #pragma omp end parallel
```

- If you want part of code to be executed only on master thread, use master directive
- “non-master” threads will skip over master region and continue

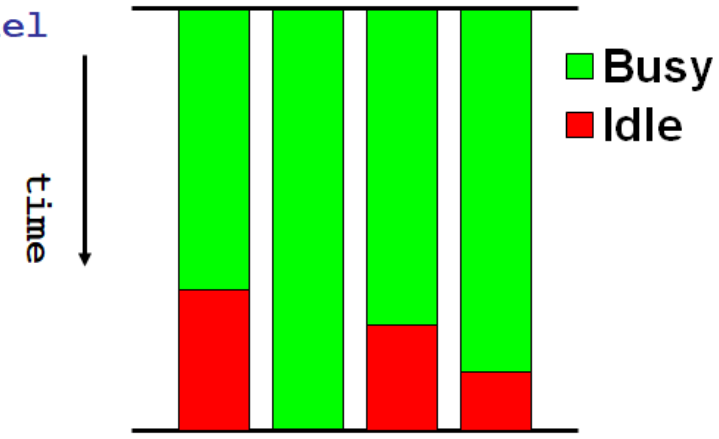
Single

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp single
   {
     ExchangeBoundaries();
   }  // threads wait here for single
   DoManyMoreThings();
}
```

- Denotes block of code to be executed by only one thread
- First thread to arrive is chosen
- Implicit barrier at end

Unequal work loads lead to idle threads and wasted time.

```
#pragma omp parallel
{

#pragma omp for
  for( ; ; ){

  }

}
```

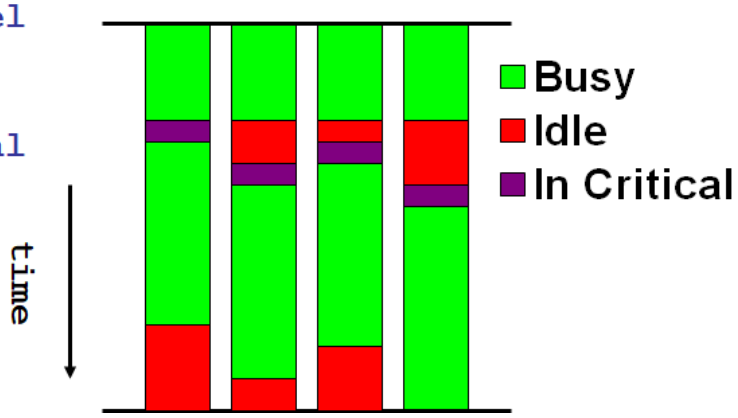


Load Balancing

```
#pragma omp parallel for
for(int i=0;i<100;i++)
  if (i<50)
    smallwork()
  else
    bigwork();
-----------------------------------------------------
#pragma omp parallel for
for(int i=0;i<100;i++)
  if (i%2)
    smallwork()
  else
    bigwork();
```

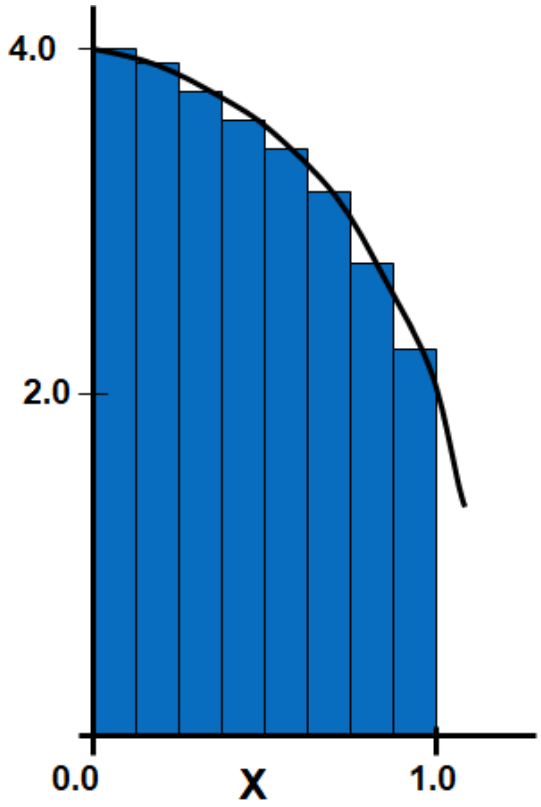
Lost time waiting for locks

```
#pragma omp parallel
{

#pragma omp critical
  {
    ...
  }
  ...
}
```



示例

Numerical Integration



$$f(x) = \frac{4.0}{(1+x^2)}$$

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

- **Parallelize the numerical integration code using OpenMP**
- **What variables can be shared?**
- **What variables need to be private?**
- **What variables should be set up for reductions?**

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
 #pragma omp parallel for private(x) reduction(+:sum)
   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
```

```
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

小结

请下载并安装CUDA

Heterogeneous Multi-core

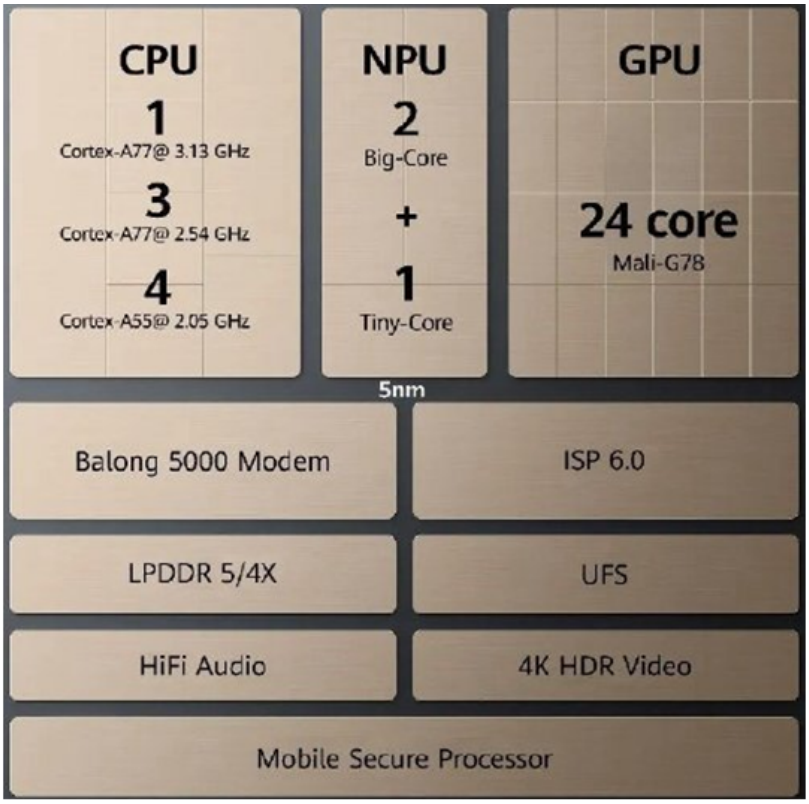
reference

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

http://blog.csdn.net/kkk584520/article/details/9413973

AMP

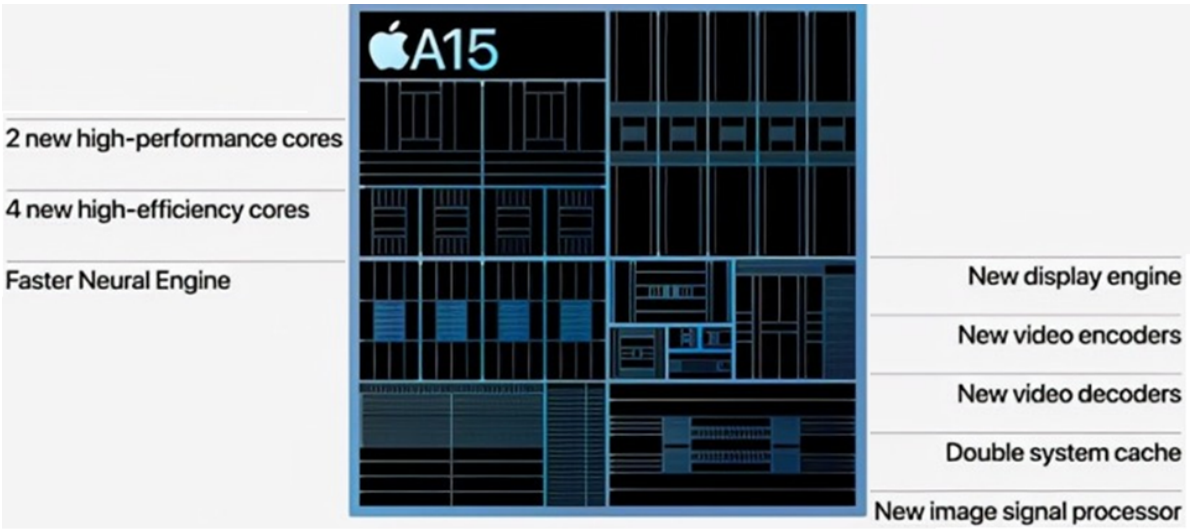
Kirin9000



2020年
5nm

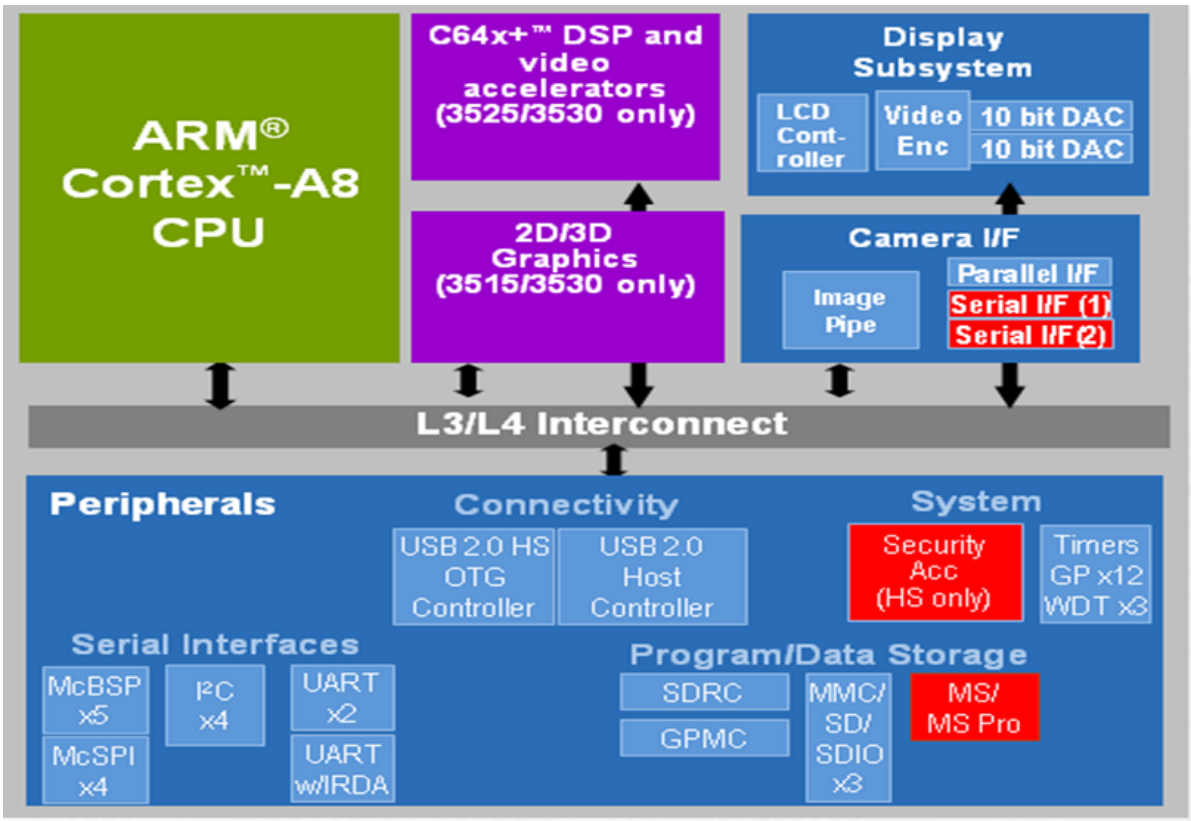
https://www.sohu.com/na/430711564_442150

A15 bionic

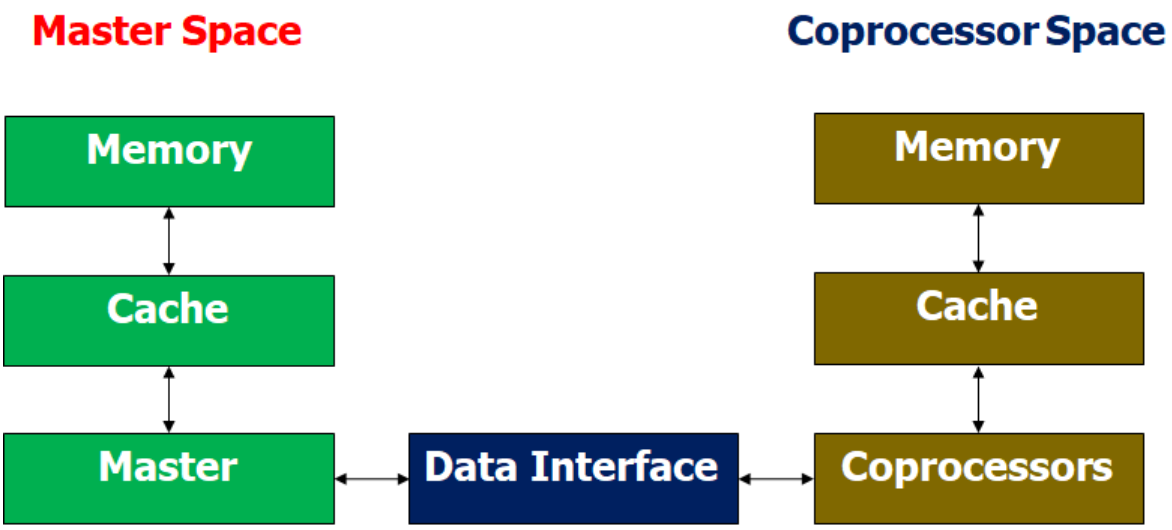


https://new.qq.com/omn/20211025/20211025A077G200.html

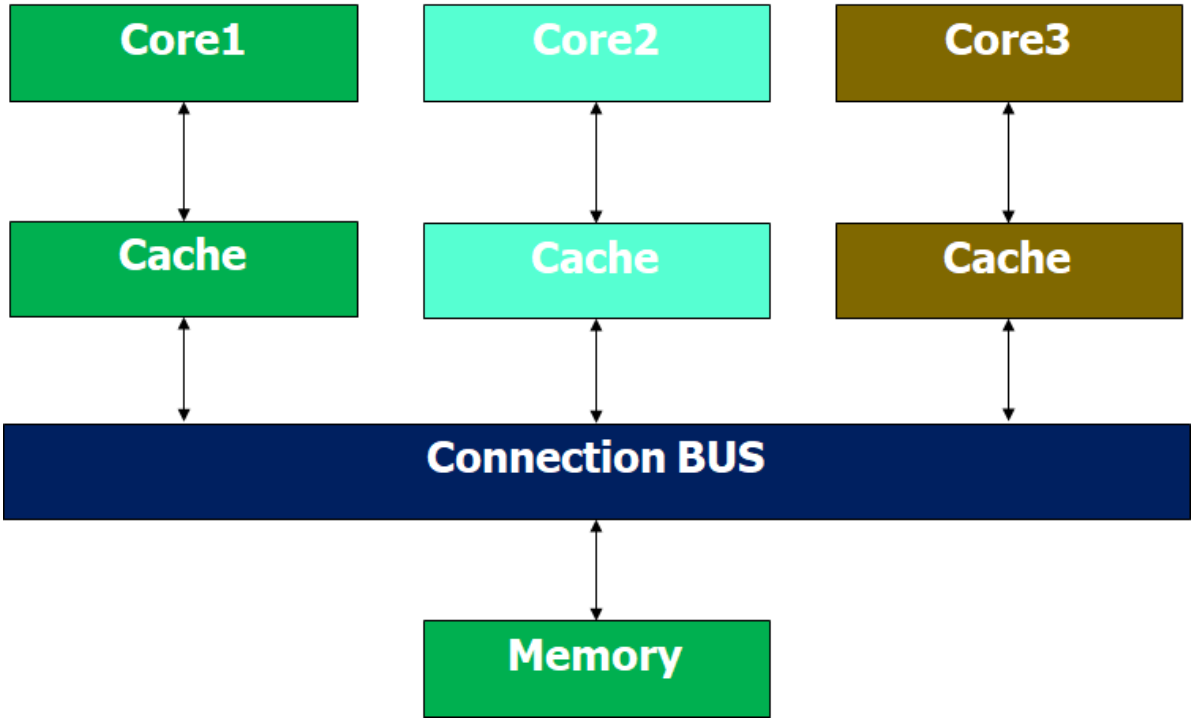
OMAP3530 (Ti)



AMP 存储架构



Data Interface: High Speed Parallel Interface
: Dural Port Memory
: Bus



AMP编程

Multi-thread for AMP?

- The method of Programming for SMP?

  Multi- thread (parallelizing).

- Is “Multi-thread”method available for AMP programming? Why?

  No!

  Different architecture processors with different Instructions .

- Programming for different processors?

Independent

Integrated

- How to download binary code to different processors?

Master and Coprocessor independently by tools

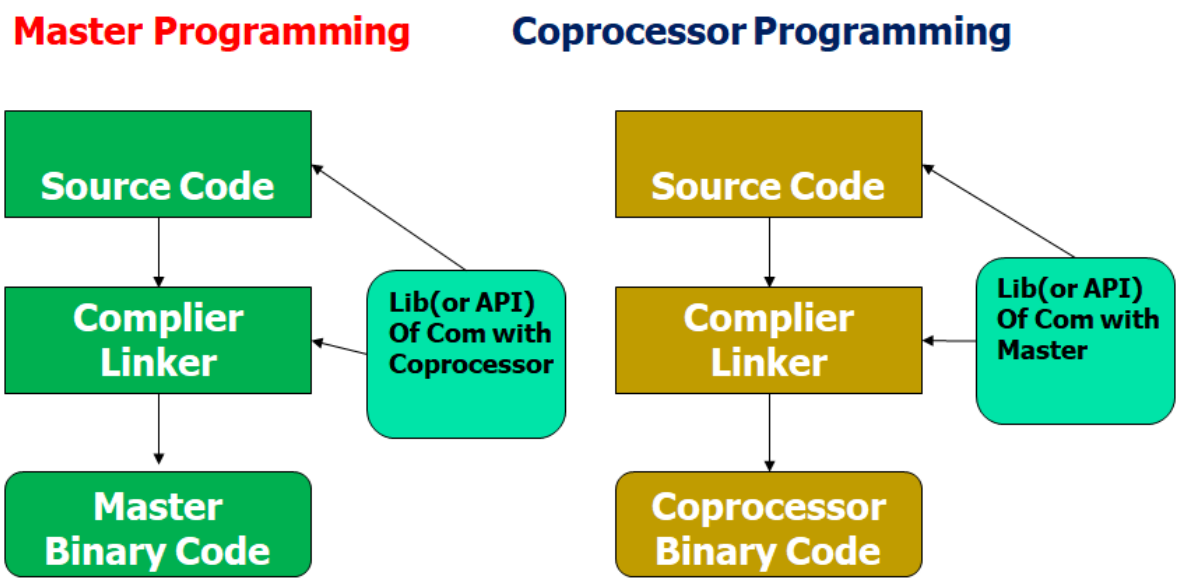
From Master to coprocessor

- How to start up the different processors?

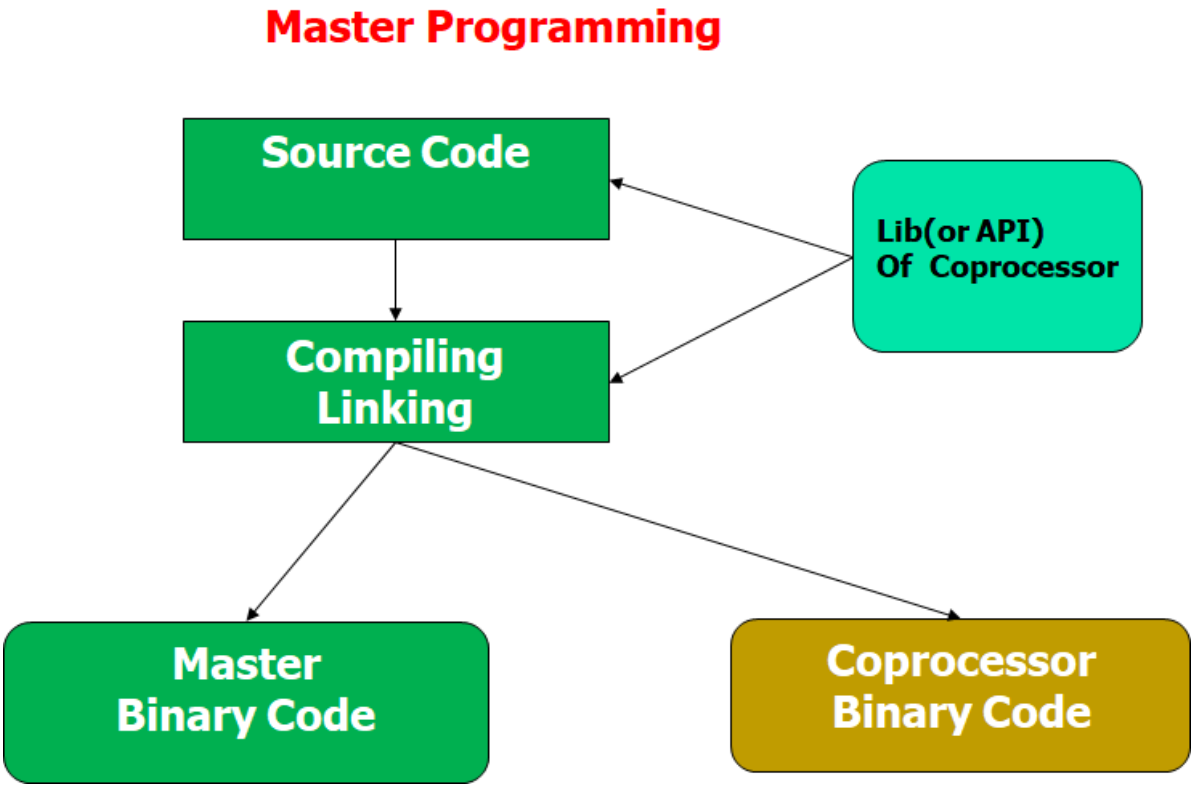
Master – Power(reset)

Coprocessor – Set by master

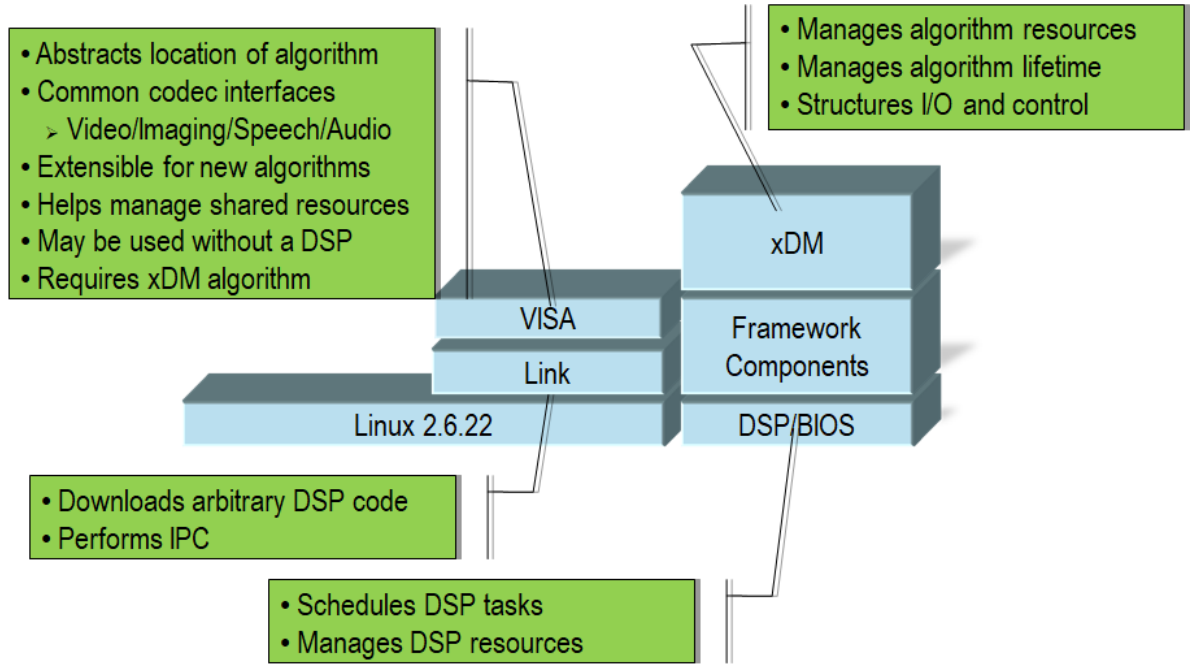
Independent Programming



Integrated Programming



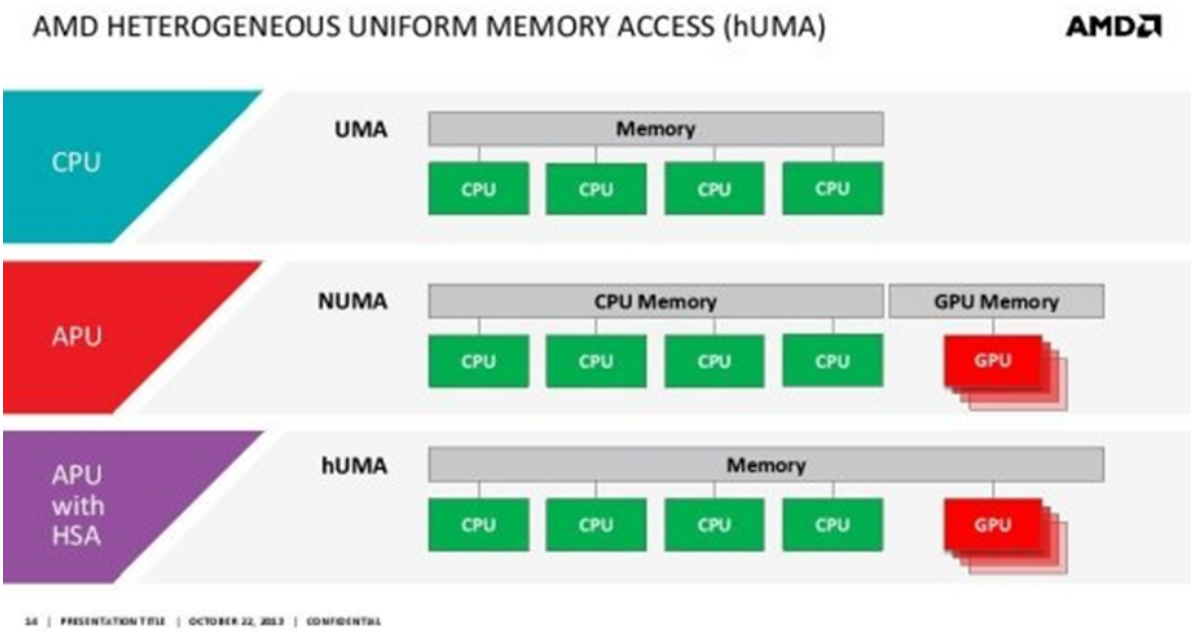
Programming OMAP3530



- xDM算法接口；IPC：Inter-Process Communication

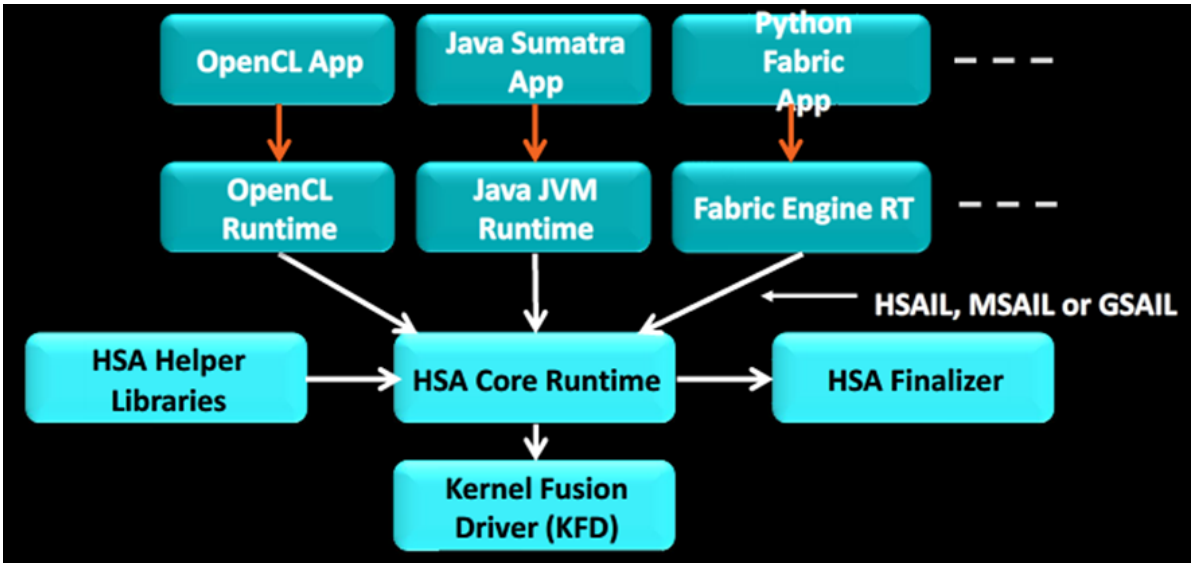
HSA

HSA（Heterogeneous System Architecture）



- APU：Accelerated Processing Unit ; NUMA: Non Uniform Memory Access Architecture; UMA Uniform Memory Architecture

Programming



GPU

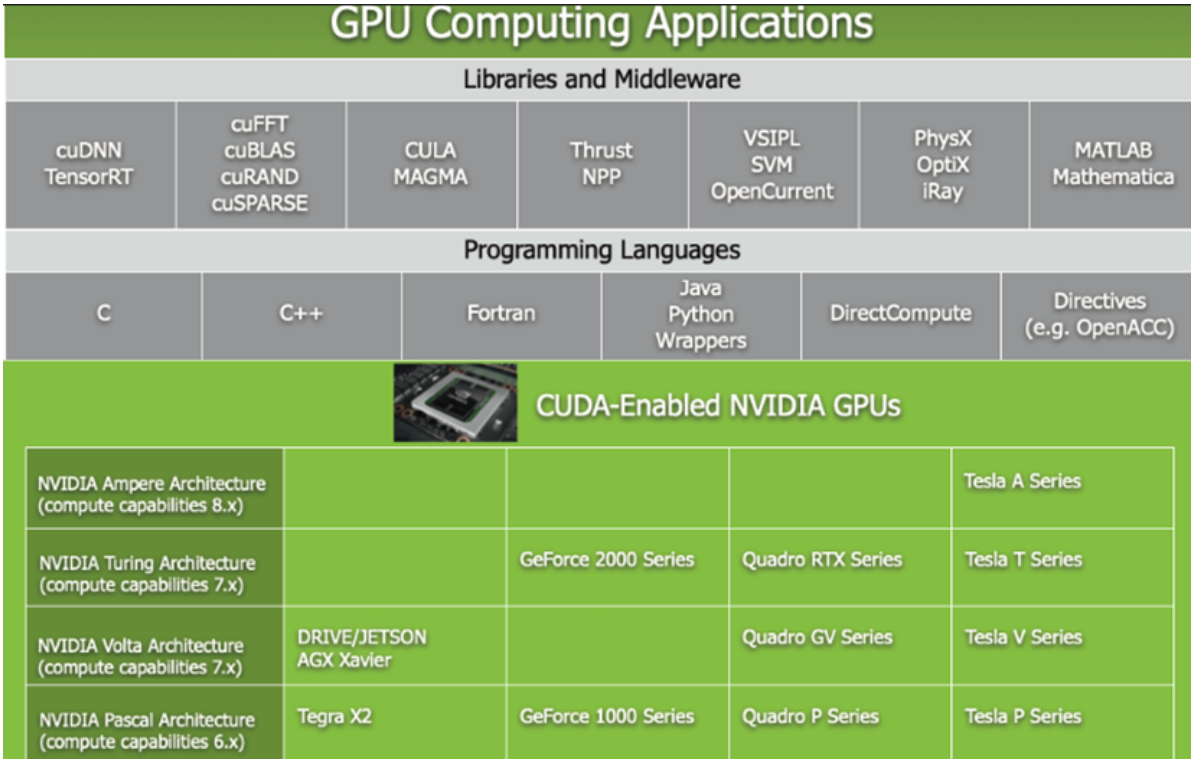
Reference

- Manuals
- Programming Guide
- Best Practice Guide
- Books
- CUDA By Examples, Tsinghua University Press
- Training videos
- GTC talks online: optimization, advanced optimization + hundreds of other GPU computing talks http://www.gputechconf.com/gtcnew/on-demand-gtc.php
- NVIDIA GPU Computing webinars http://developer.nvidia.com/gpu-computing-webinars

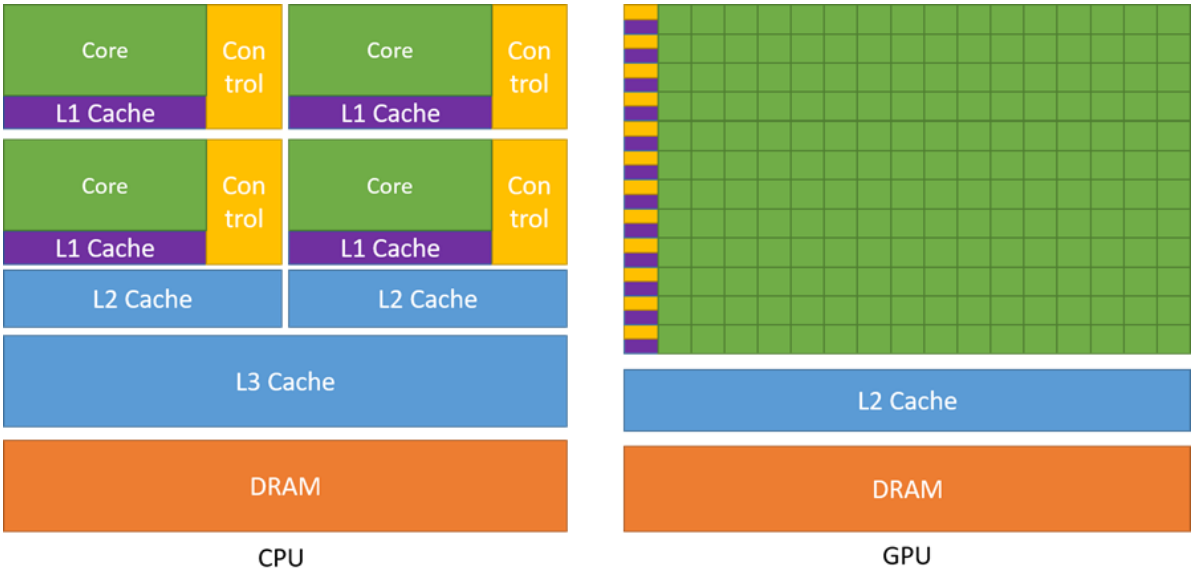
GPU

- GPUs are massively multithreaded many core chips
- Hundreds of scalar processors
- Tens of thousands of concurrent threads
- n TFLOP peak performance
- Fine-grained data-parallel computation
- Suppliers
- Intel: Iris™ Pro Graphics[, i740
- NVIDIA: Geforce, Telsa, Tegra
- AMD(ATI)： RX, FirePro
- Matrox
- 3dfx
- SiS
- VIA

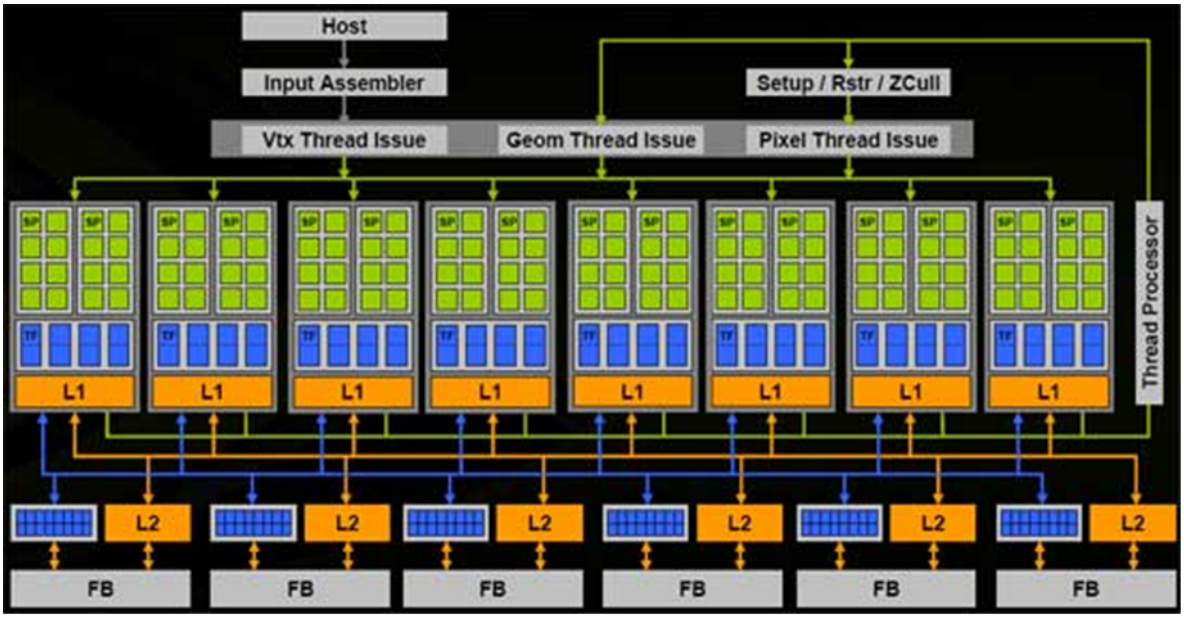
GPU Applications



CPU vs GPU



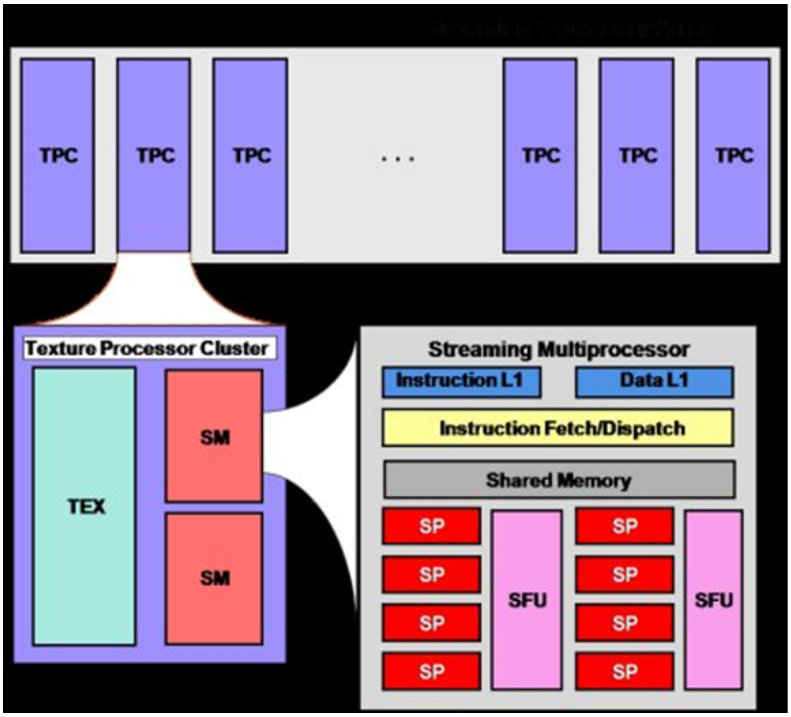
A GPU Device(G80)



SP: Stream Processor
TP: Texture Processor
FB: Frame Buffer
http://news.mydrivers.com/1/152/152896_all.htm

- Vtx =Vertex , 线的顶点；Geom=Geometry

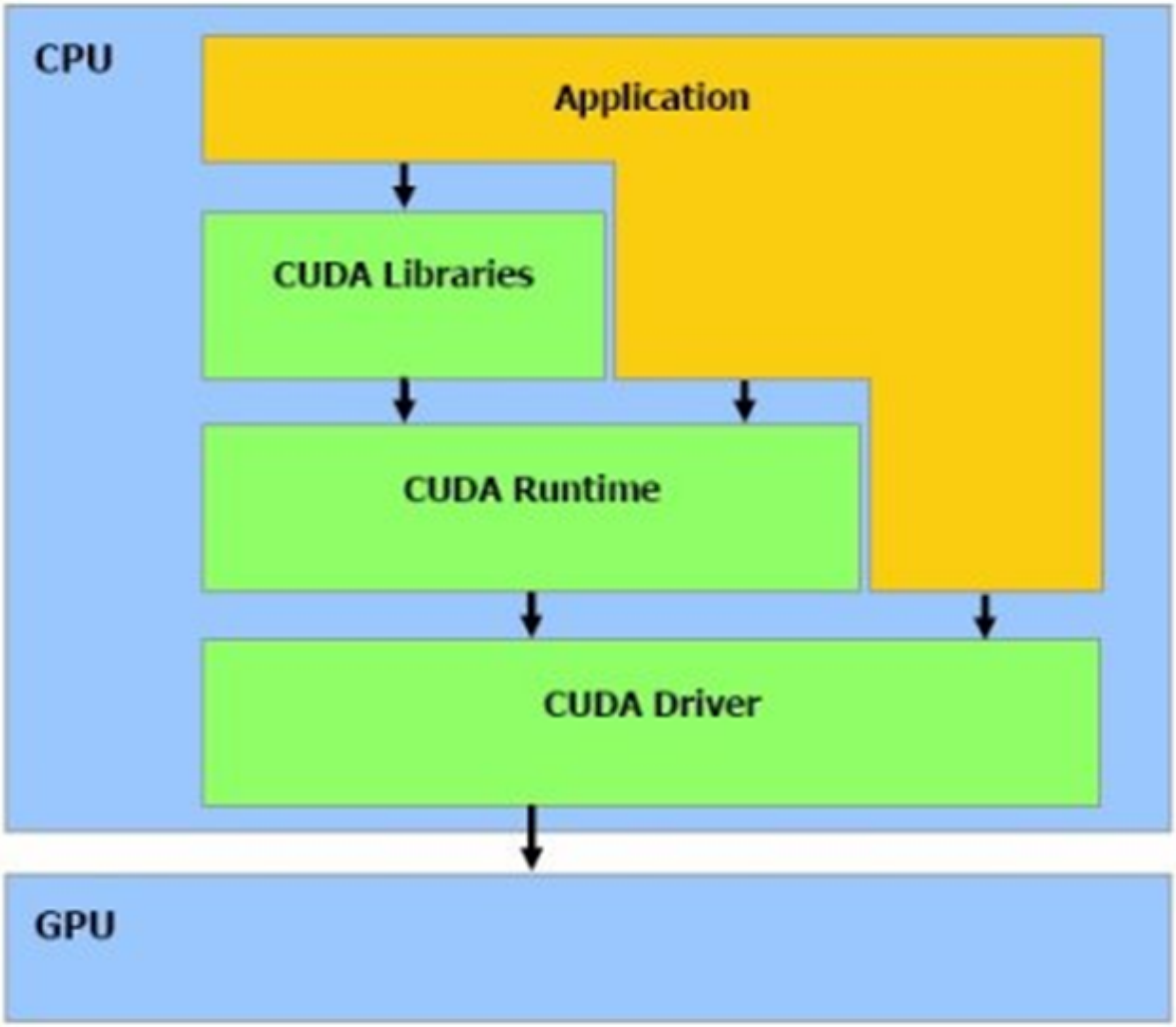
Texture Processor Cluster



SM: Stream Multi-Processor

SFU: Special Function Unit

GPU in System



GPU in System

Connected to CPU chipset by PCIe

PCIe x16 Gen 2: 8 GB/s in one direction, 16 GB/s bi-directionally

http://arstechnica.com/old/content/2004/07/pcie.ars

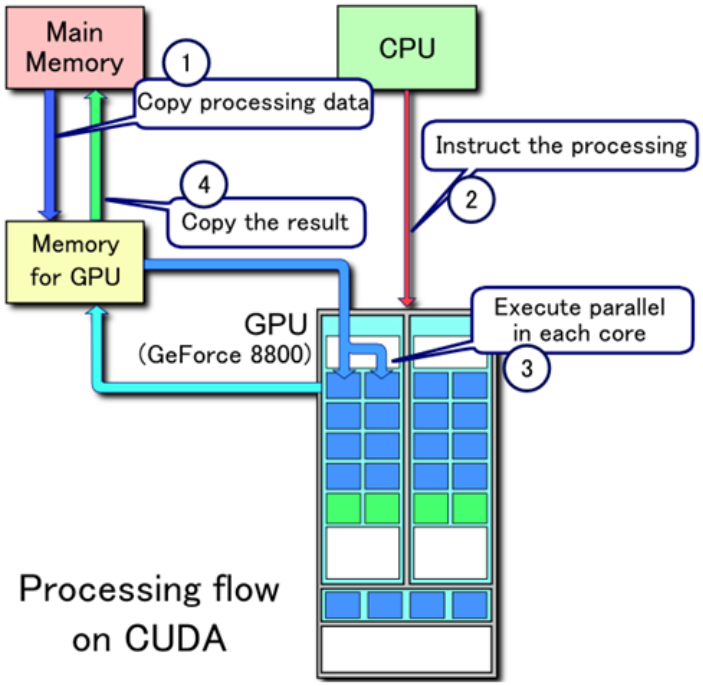
CUDA

What is CUDA

- CUDA is the acronym for Compute Unified Device Architecture.
  - A parallel computing architecture developed by NVIDIA.
  - The computing engine in GPU.
  - CUDA can be accessible to software developers through industry standard programming languages.
- CUDA gives developers access to the instruction set and memory of the parallel computation elements in GPUs.

Processing Flow

- **Processing Flow of CUDA**
  - **Copy data from main mem to GPU mem.**
  - **CPU instructs the process to GPU.**
  - **GPU execute parallel in each core.**
  - **Copy the result from GPU mem to main mem.**

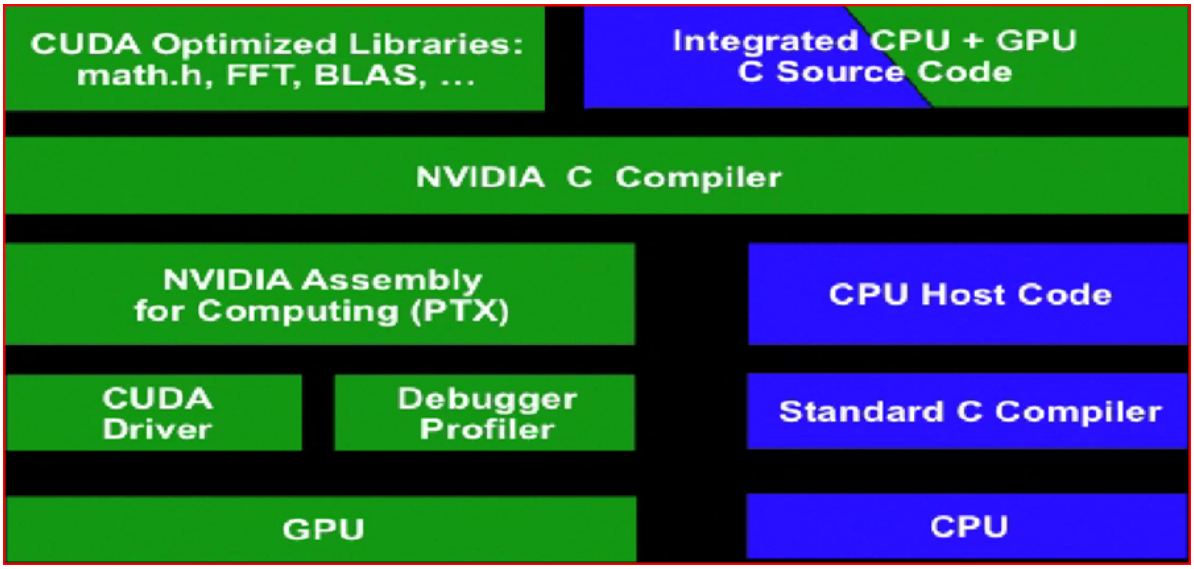


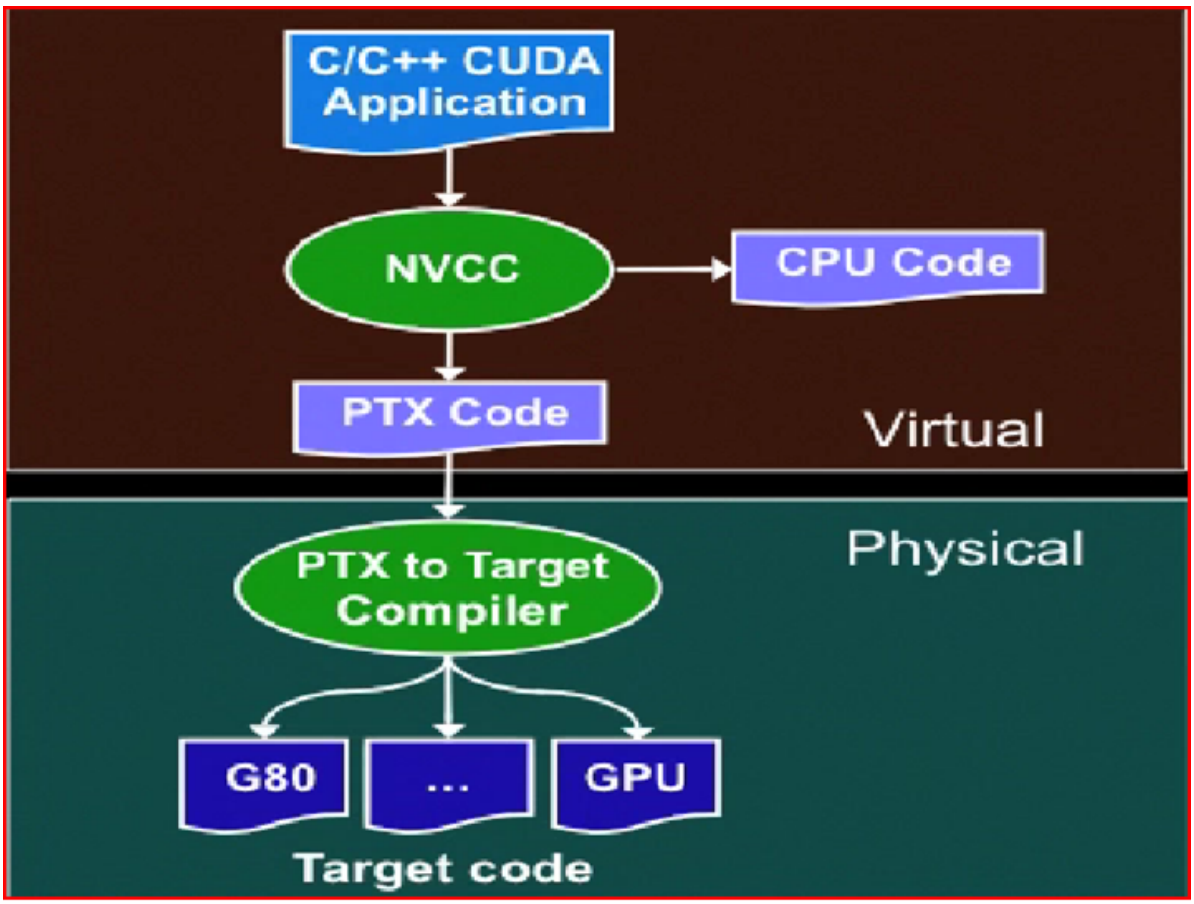
CUDA Environment

Installing CUDA.

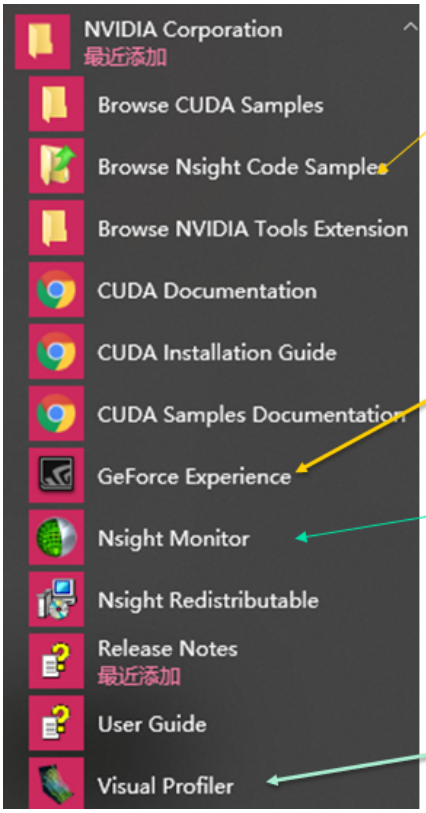
- Windows – VSXXXX
- Linux – gcc

https://developer.nvidia.com/cuda-downloads





PTX： *parallel thread execution* virtual machine



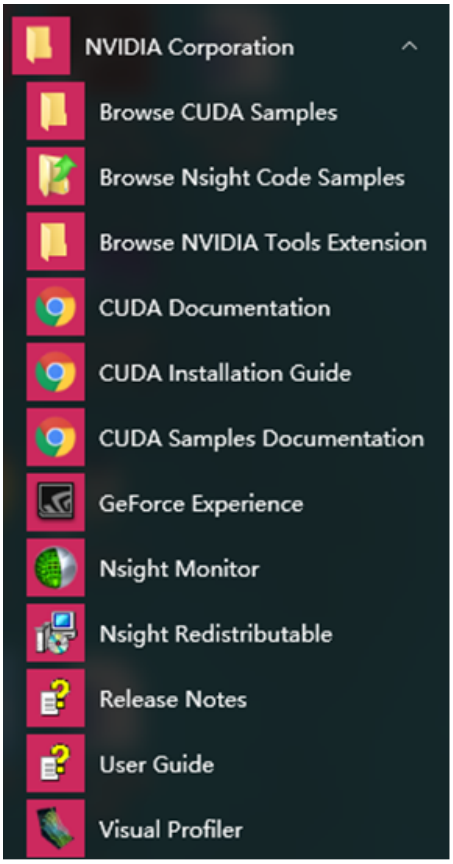
示例

云管理

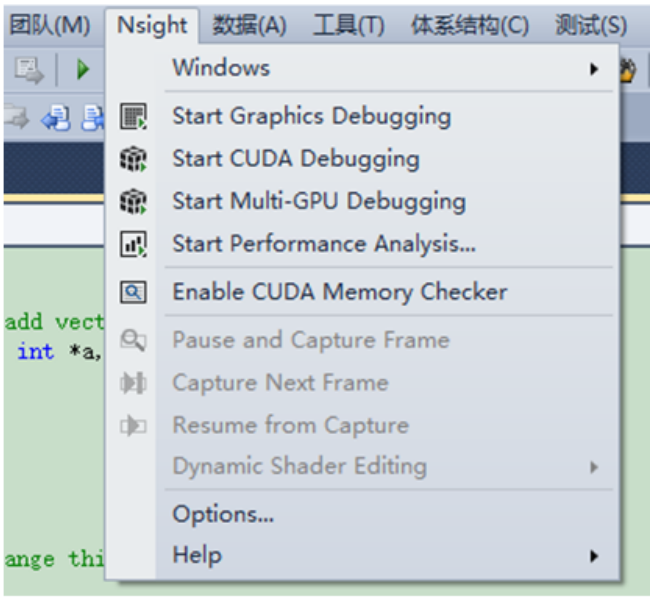
调式工具

本机 **GeForce 840M**，不支持！

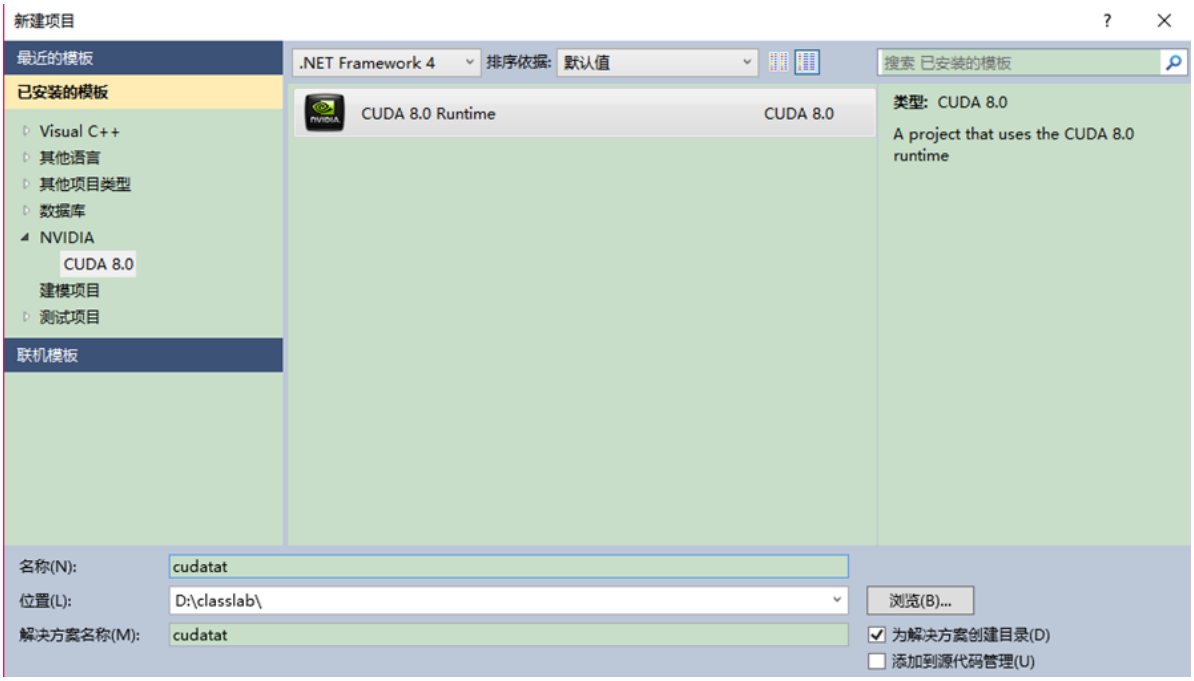
性能分析



← Windows



VS2013



新建例程，运行例程，结果?

源程序的扩展名?

哪一段代码是在GPU上运行的?

如何调试GPU上运行的程序?

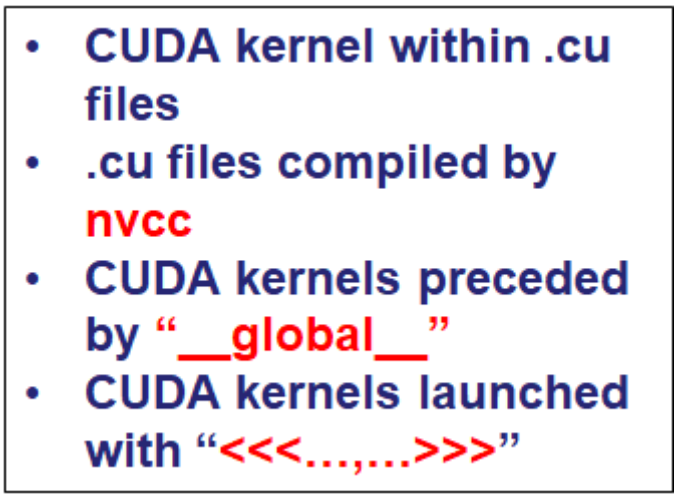
Cuda_test

Testing-CPU

```
// hello_world.c:
#include <stdio.h>

void hello_world_kernel()
{
    printf("Hello World\n");
}

int main()
{
    hello_world_kernel();
}
```

Testing-GPU

```
//hello_world.cu:
#include <stdio.h>

__global__ void hello_world_kernel()
{
    printf("Hello World\n");
}

int main()
{
     hello_world_kernel<<<1,1>>>();
}
```



What's Wrong?

Why?

CUDA 编程

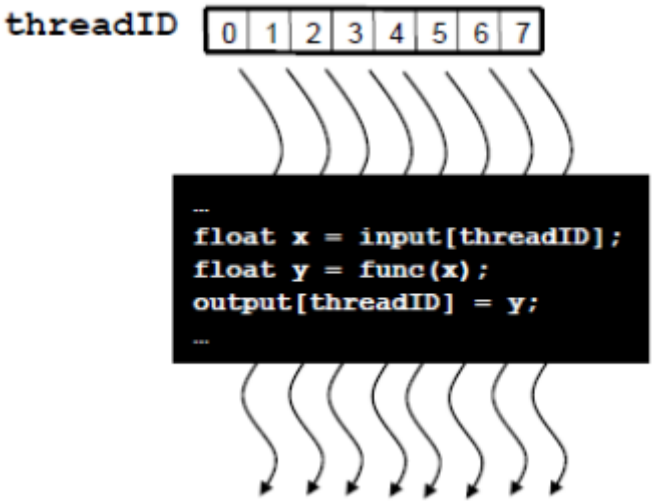
CUDA Kernel program

```
addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
```

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

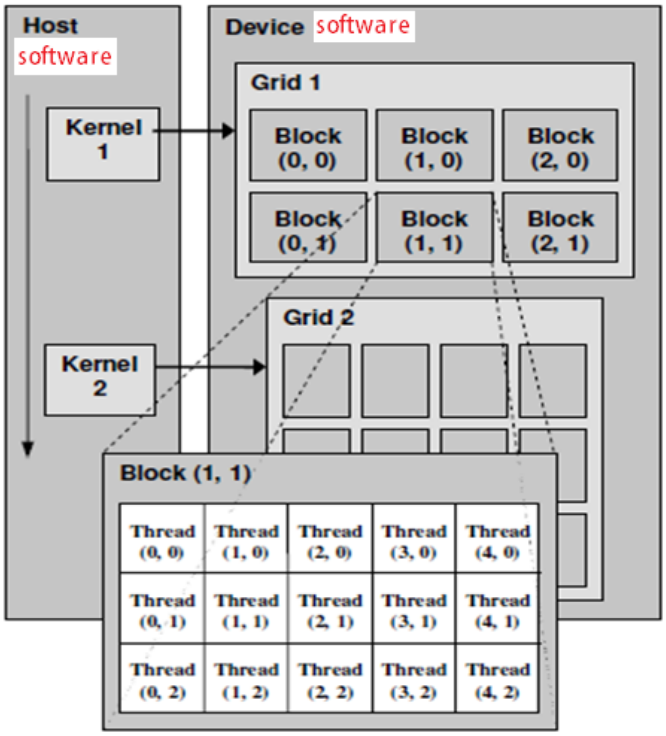
CUDA Programming Mode

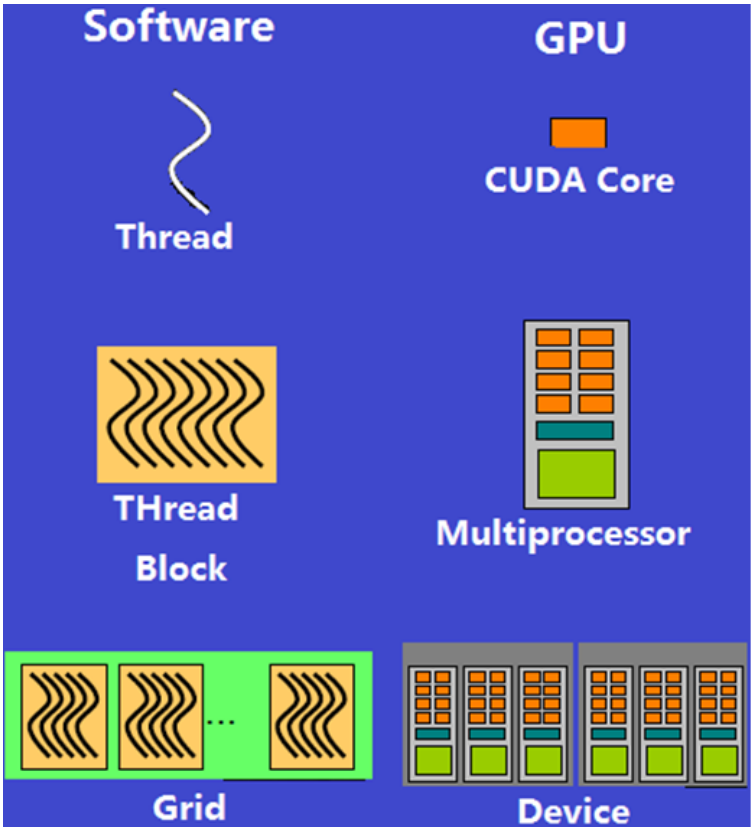
- CUDA C extends C by allowing the programmer to define C functions, called kernels.
- When called, kernels are executed N times in parallel by N different *CUDA* threads.
- A kernel is defined using the __global__ .
- The number of CUDA threads that execute the kernels is specified using <<<...>>> (Threads)
    - cuda_kernel<<<nBlk, nTid>>>(...)
- Each thread that executes the kernel is given a unique thread ID.
- A thread block is a batch of threads
- All threads run the same code
- Each thread has an ID that it uses to compute memory addresses and make control decision
- Threads can cooperate with each other by
- sharing data and synchronizing their execution
- Threads from different blocks cannot cooperate
- There is a limit to the number of threads per block



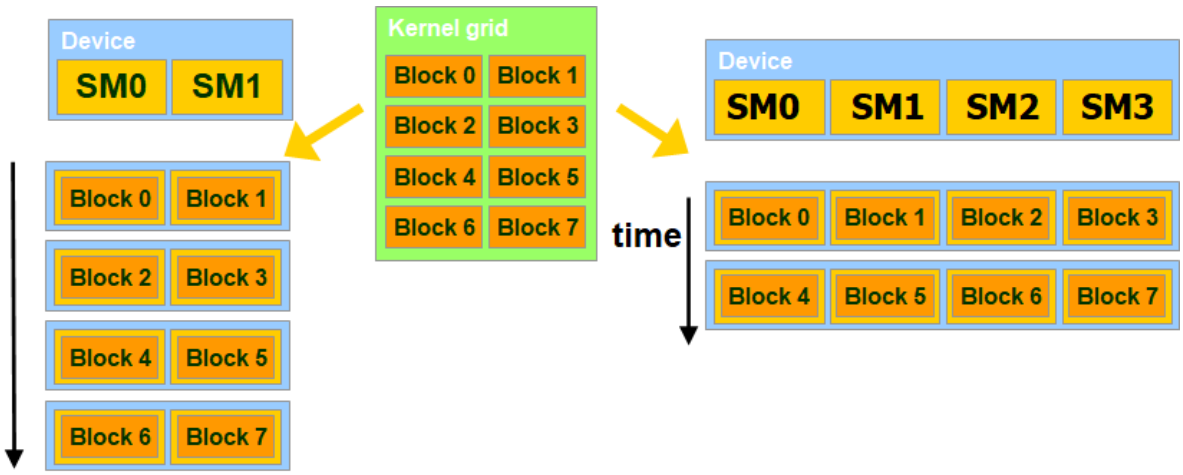
Definitions:

Device = GPU
Host = CPU
Kernel = function that runs on the device





Thread map to Core（SP）
Block map to SM

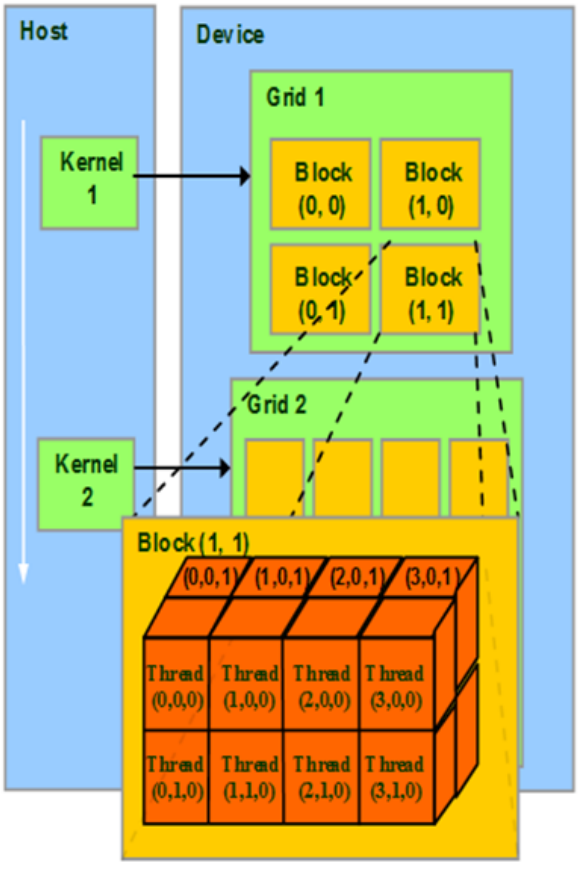


How blocks (program/threads) are scheduled on SMs (Device) ?

Kernel Grid on Device
Each block can execute in any order relative to other blocks.

If the number of SMs is more than that of Blocks, then?

- **Built-in Variables**
  - **gridDim**
    - **x, y**
  - **blockIdx**
    - **x, y**
  - **blockDim**
    - **x, y, z**
  - **threadIdx**
    - **x, y, z**

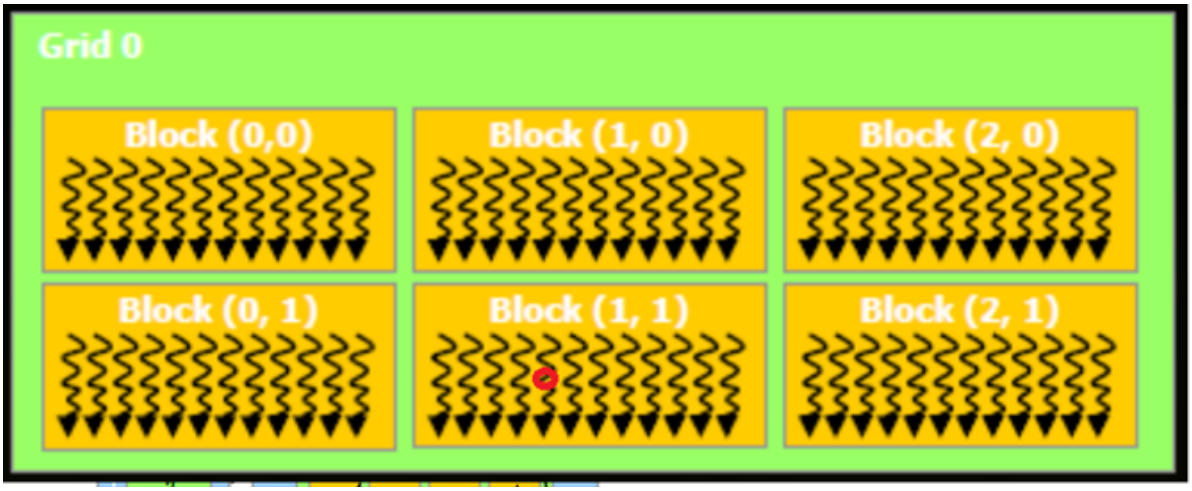


How to calculate a thread index?

Thread ID

```
BlockID= blockIdx.y*gridDim.x+blockIdx.x

ThreadID=BlockID*blockDim.x*blockDim.y*blockDim.z
    +threadIdx.z*blockdIm.x*blockDim.y*
    +threadIdx.y*blockDim.x
    +threadIdx.x
```

How about 2D or 1D blocks?

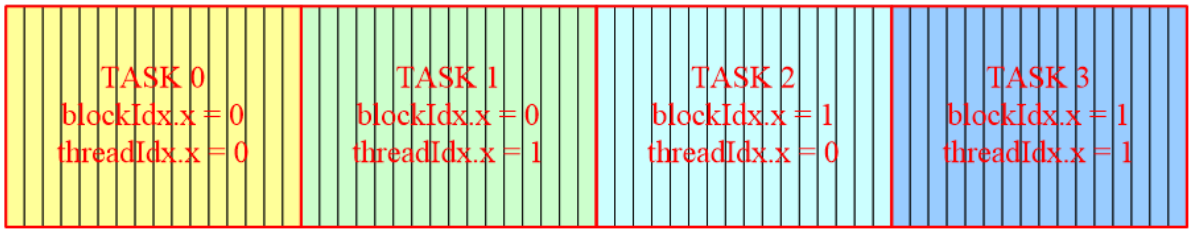



```
dim3 grid(3,2) ; dim3 block(12,1,1);
gridDim.x=3; gridDim.y=2;
blockDim.x=12; blockDim.y=1; blockDim.z=1;
blockIdx.x=1; blockIdx.y=1;
threadIdx.x=4, threadIdx.y=0; threadIdx.z=0;
threadID=(1*3+1)*12+4=52
```

Data Partition-example

cuda_kernel<<<2, 2>>>(...)

When processing 64 integer data:



Example (1D-Block)

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Example (2D-Block)

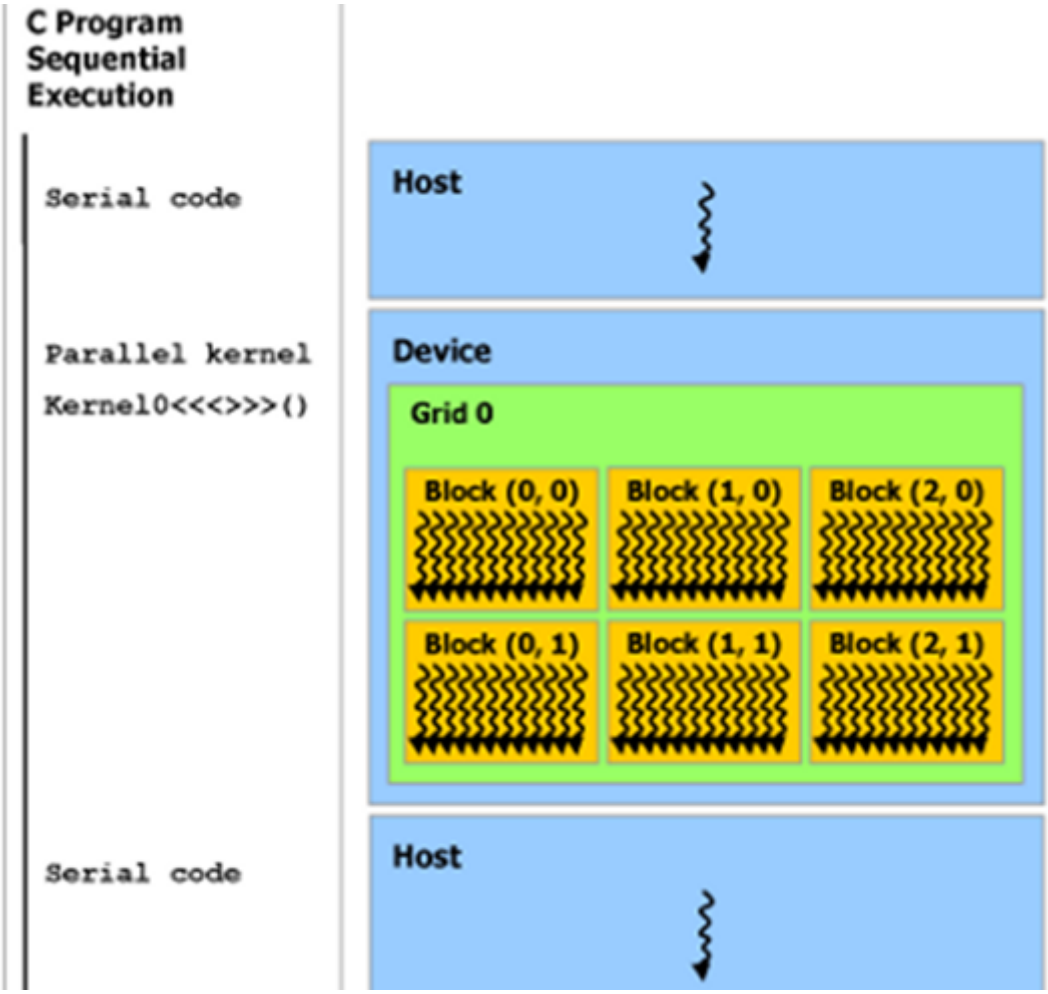
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

- Example (Multi-blocks)

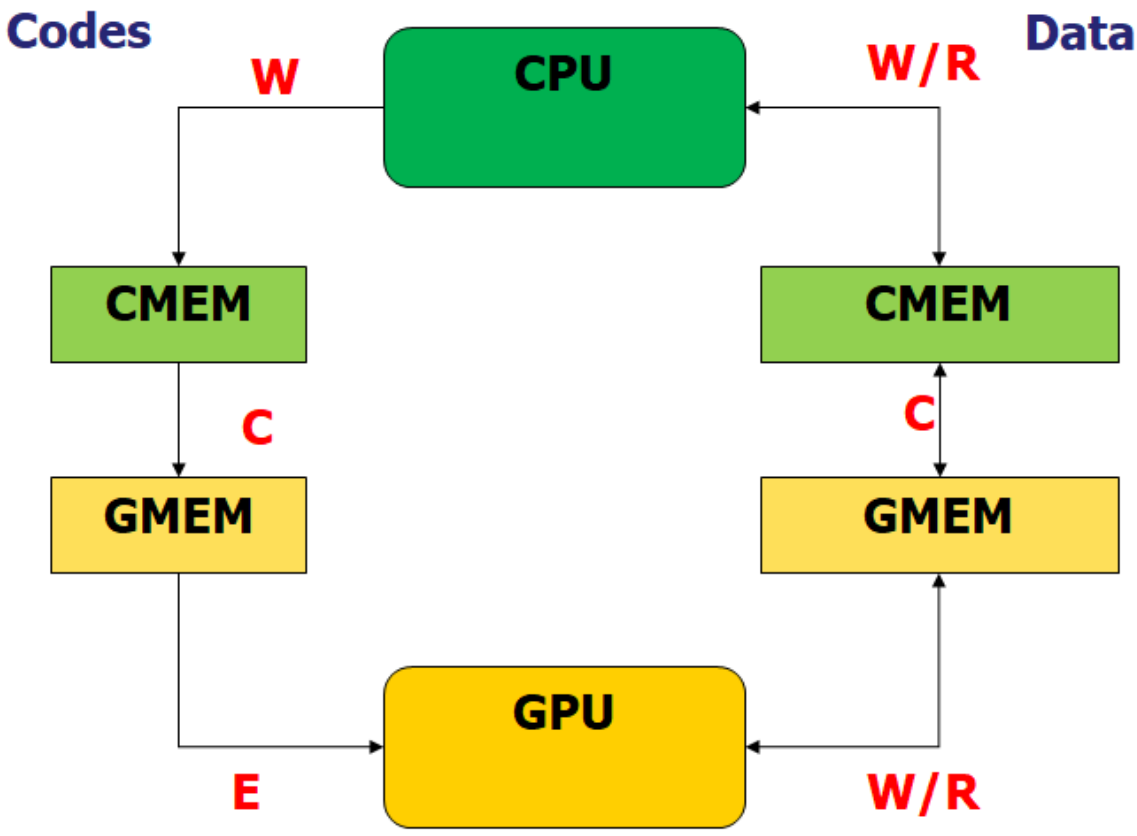
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Programming

Heterogeneous Programming

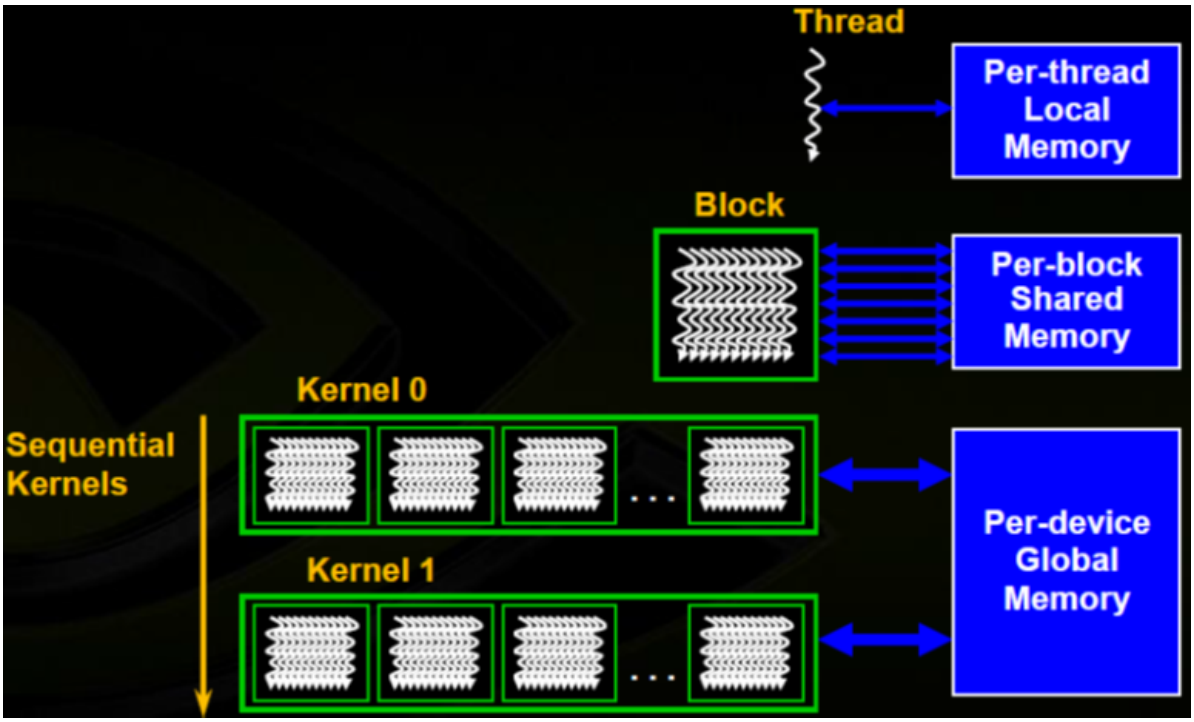


GPU Input and Output



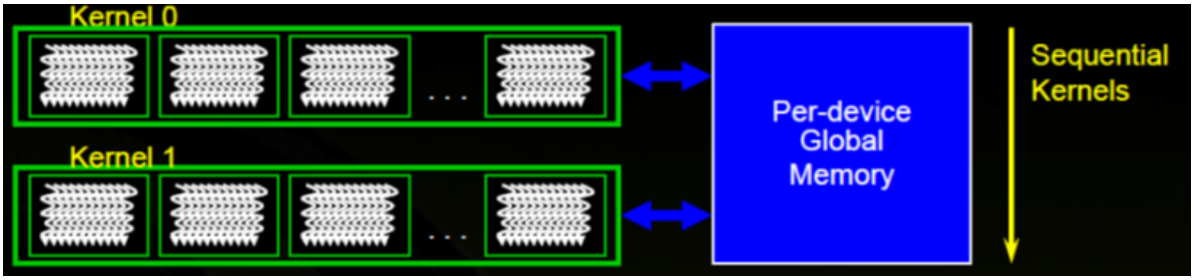
GPU Memory

- **Space**
  - CPU and GPU have separate memory spaces
    - Data is moved across PCIe bus (PC)
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C/Fortran functions
- **Hierarchy**
  - Register
  - Cache
    - L1/L2
    - Shared memory: Block, cached in L1
    - Constant cache: Read-only constant cache
    - Texture cache: Texture memory access
  - DRAM/Global Memory
    - Device, cached in both L1 and L2



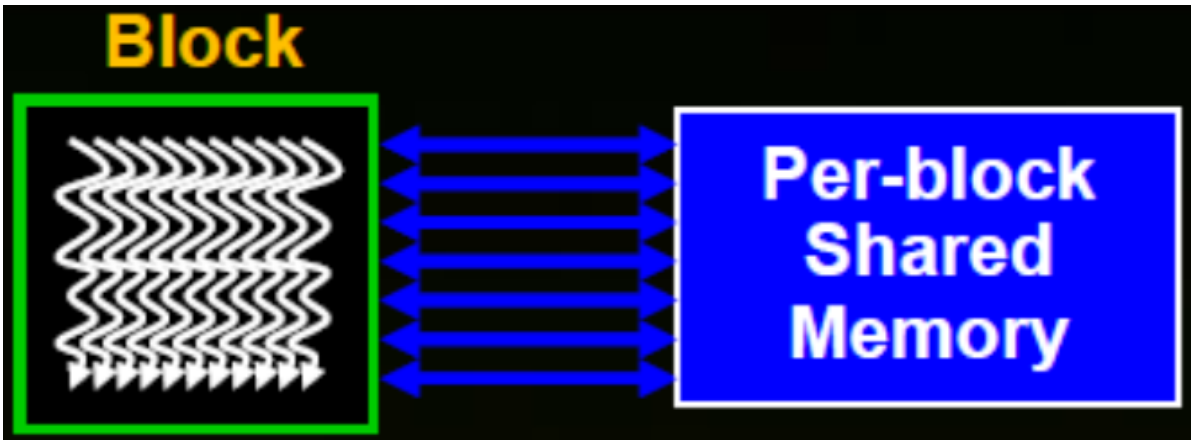
Global Memory

- Accessible by all threads as well as host (CPU)
- Data lifetime = from allocation to deallocation



Shared Memory

- Accessible by all threads in the block
- Data lifetime = the longest thread lifetime



Register & Local memory

- Automatic variables (scalar/array) inside kernels
- Spills to local memory
- Data lifetime = thread lifetime

Memory Allocation

- Host (CPU) manages device (GPU) memory
  - cudaMalloc(void ** pointer, size_tnbytes)
  - cudaMemset(void * pointer, intvalue, size_tcount)
  - cudaFree(void* pointer)
- Examples

```
int nbytes= 1024*sizeof(int);
int* d_a= 0;
cudaMalloc( (void**)&d_a, nbytes);
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Memory Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enumcudaMemcpyKind);
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn't start copying until previous CUDA calls complete
- enumcudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- Non-blocking memcopies are provided

示例

Arrays Plus

```
#include "cuda_runtime.h"

#include <stdio.h>

cudaError_t addWithCuda(int *c, const int *a,
            const int *b, unsigned int size);

__global__ void addKernel(int *c, const int *a,
                const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) { return 1; }

    printf("{1,2,3,4,5} + {10,20,30,40,50} =
             {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);

   cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {return 1; }
```

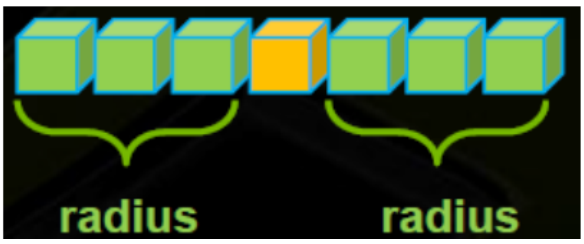
```
    return 0;
}


cudaError_t addWithCuda(int *c, const int *a, const int *b,
                unsigned int size)
{
    int *dev_a = 0; int *dev_b = 0; int *dev_c = 0;
    cudaError_t cudaStatus;
    // Allocate GPU buffers for three vectors (two input, one output)    .
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) { goto Error;}

    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {goto Error;}

    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {goto Error;}

    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int),
                cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {goto Error;}

     cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int),
                cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {goto Error;}
    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<1, size>>>(dev_c, dev_a, dev_b);

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {goto Error;}

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {goto Error;}

    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int),
                cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {goto Error;}
Error:
    cudaFree(dev_c); cudaFree(dev_a); cudaFree(dev_b);
    return cudaStatus;
}
```

Run the code !

一维信号均值滤波

Case

- **Applying a 1D stencil to a 1D array of elements:**
  - **Each output element is the sum of all elements within a radius**
- **For example, for radius = 3, each output element is the sum of 7 input elements:**



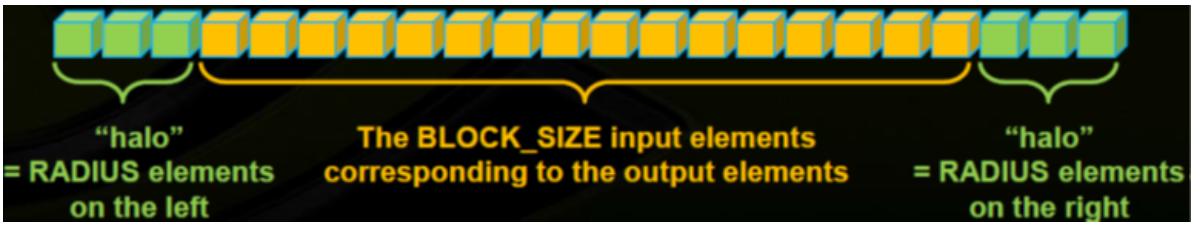
Where is the case?

- Using global memory
  - One element per thread

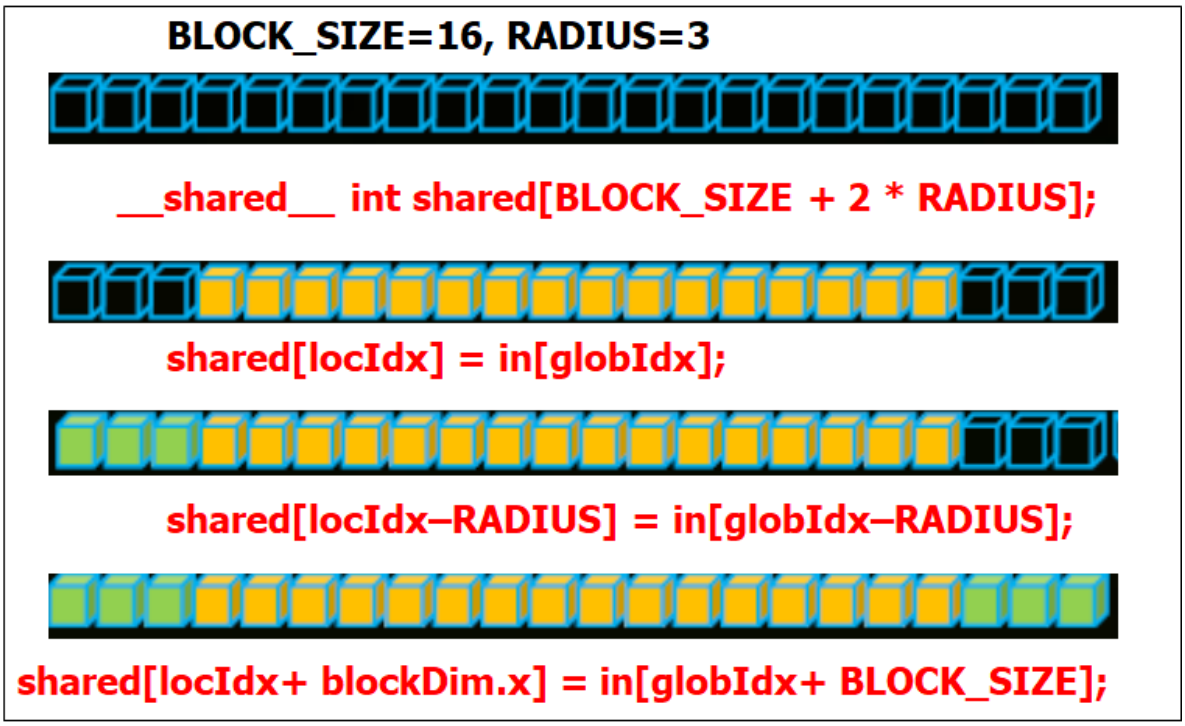
```
__global__ void stencil(int* in, int* out)
 {
    int globIdx = blockIdx.x * blockDim.x +
            threadIdx.x;

    int value = 0;

    for (offset = -RADIUS; offset <= RADIUS;
                offset++)
        value += in[globIdx + offset];
    out[globIdx] = value;
}
```

A lot of redundant read in neighboring threads.

- Using shared memory
  - One element per thread
  - Read (BLOCK_SIZE + 2 * RADIUS) elements from global memory to shared memory
  - Compute BLOCK_SIZE output elements in shared memory
  - Write BLOCK_SIZE output elements to global memory

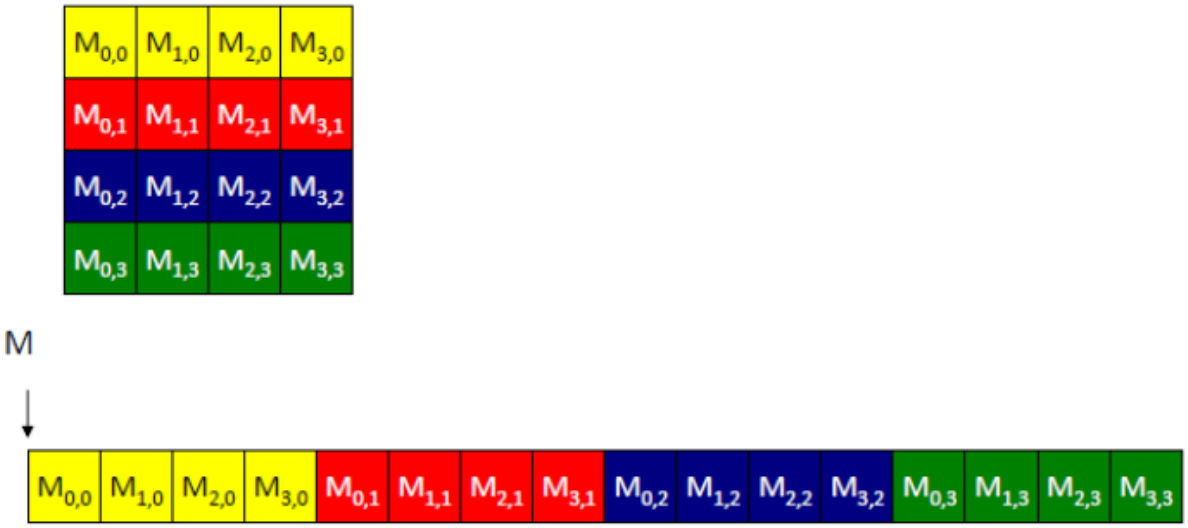


Shared Memory



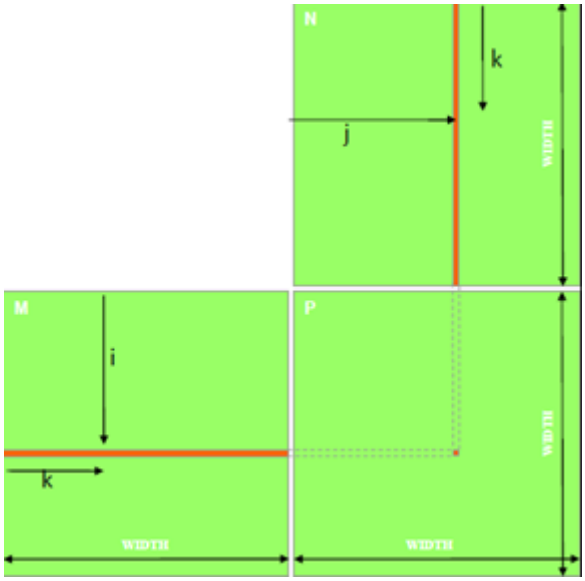

```
__global__ void stencil(int* in, int* out)
 {
    __shared__ int shared[BLOCK_SIZE + 2 * RADIUS];
    int globIdx= blockIdx.x* blockDim.x+ threadIdx.x;
    int locIdx= threadIdx.x+ RADIUS;

    shared[locIdx] = in[globIdx];
    if (threadIdx.x< RADIUS)
    {
        shared[locIdx-RADIUS] = in[globIdx-RADIUS];
        shared[locIdx+ blockDim.x] = in[globIdx+
                BLOCK_SIZE];
    }
    __syncthreads();
    int value = 0;
    for (offset = -RADIUS; offset <= RADIUS; offset++)
        value += shared[locIdx+ offset];
    out[globIdx] = value;
}
```

Matrix Multiplication




```
__int main(void)
{
     //allocate and initialize the matrices M, N, P
      //IO to read the input matrices M, N
   ……
      //M * N on the device
      MatrixMultiplication(M, N, P, Width);
    ……
      // IO to write the Output Matrices P
      // Free M, N, P
      return 0;

}
```

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
    {
        for (int j = 0; j < Width; ++j)
       {
    double sum = 0;
               for (int k = 0; k < Width; ++k)
     {
       double a = M[i * width + k];
       double b = N[k * width + j];
       sum += a * b;
    }
              P[i * Width + j] = sum;
            }
   }
}
```

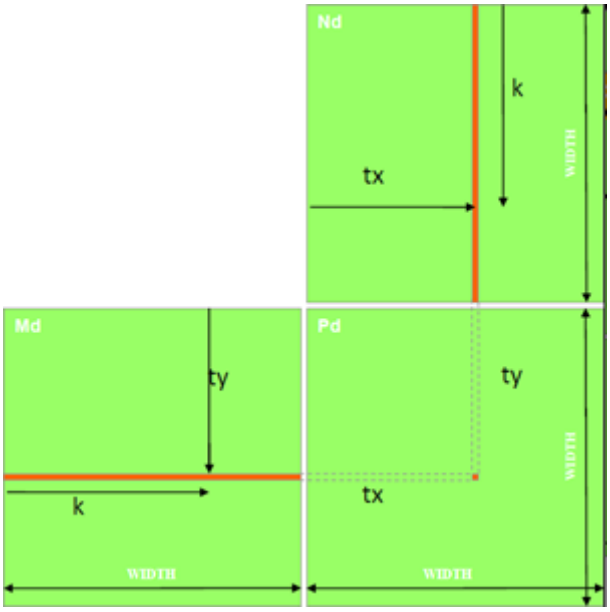


Allocating Memory on GPU

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{ int size=Width*Width*sizeof(float);
      float * Md, Nd, Pd;
      dim3 grid(1,1,1); dim3 block(Width,Width,1);
    //Allocate device memory
      cudaMalloc((void)**&Md, size);
      cudaMalloc((void)**&Nd, size);
       cudaMalloc((void)**&Pd, size);
    //transfer M,N from host to device
       cudaMemcpy(Md, M,size, HostToDevice);
       cudaMemcpy(Nd, N,size, HostToDevice);
    //kernel innovation code
     MatrixMulOnKernel<<<grid, block >>>(Md, Nd, Pd, Width);
     //transfer p from device to host
      cudaMemcpy(P, Pd,size, DeviceToHost);
      cudaFree(Md);cudaFree(Nd);cudaFree(Pd);
}
```

Kernel on GPU

```
__global__void MatrixMulOnKernel(float* Md, float* Nd, float* Pd, int width)
{
    //2D threadID
    int tx=threadIdx.x;
    int ty=threadIdx.y;
   // pvalue stores the pdelement
    for(int k=0;k<width;++k)
    {
         float Mdelement=Md[ty*width+k];
         float Ndelement=Nd[k*width+tx];
         pvalue+=Mdelement*Ndelement;
    }
   //write pvalue to device memory
    Pd[ty*width+tx]=pvalue;
}
```



写个程序计算5x5矩阵相乘，并运行！