# 2019 北京大学软件与微电子学院嵌入式处理器系统设计期末考试题

一、填空题（每空 1’/40’）

1. 根据指令集的复杂程度，通常将处理器分为（**RISC（精简指令集计算机）**）和（**CISC（复杂指令集计算机）**）两类。ARM 处理器属于其中的（**RISC**）。

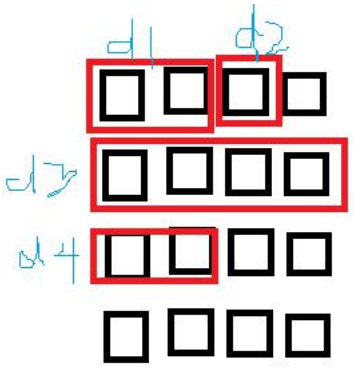
2. 对于处理器系统来说，CPI 是指（**cycles per instruction 平均每条指令的平均时钟周期个数**）。通常情况下，影响因素有（**指令集设计，流水线深度，缓存设计**）和（**系统结构，机器周期**）等。

3. （**必考**）某处理器在 100M 的主频下运行，如果某程序平均指令执行的时间是 25ns，则该处理器运行该程序的平均性能为（**40**）MIPS。

解答：时钟周期 T=1/100MHz=10ns，CPI=2.5，则平均速度为：f/(CPIx$10^6$)=40MIPS

4. 对于 ARM 处理器，结构 struct data{short d1, char d2, unsigned int d3, short d4}编译后占用的实际空间为（**12**）bytes。

解答：char d2 是 1 字节(byte)，short d1, d4 都是 2 字节，unsigned int d3 是 4 字节。




最少都得 12bytes

注意：1byte=8bit

5. 在 C 语言中声明数据类型 double *ptr，则语句“ptr++”执行后 ptr 的数值将增加（**8**）。

解答：
Double *ptr 是指针，8 个字节
Char * 1
(unsigned)Int *4
Short * 2

6. 指令“MOV R0,#(:NOT:0x55)”执行后，R0 内的值为（**0xFFFFFFAA**）。

解答：“MOV R0,#(:NOT:0x55)”等价于“MVN R1,#0x55”

7. 根据 ATPCS 规范，在 C 语言程序中调用函数 function(20,21,22,23)，则调用过程中传入 ARM 寄存器 R0、R1 的值分别是（20）和（21）。

8. （**必考**）在 Keil 平台上，某工程 make 后出现如下信息：“Code:2KB; RO Data: 0.5KB; RW Data: 1KB; ZI Data: 0.5KB”，则该程序需要的最小 ROM 空间为（**3.5**）KB，最小 RAM 空间为（**1.5**）KB。

解答：最小 ROM=Code+ RO Data+ RW Data=2+0.5+1=3.5KB，
最小 RAM=RW Data+ZI Data=1+0.5=1.5KB

9. （**必考**）标出指令的寻址方式：

“MVN R0,#1024”（**立即数寻址**）;R0←NOT(1024)

“LDR R0,=1024”（**直接寻址**）;R0←1024

“ADD R0, R1, R2”（**寄存器寻址**）;R0←R1+R2

“STR R0, [R1]”（**寄存器间接寻址**）;[R1]←R0

解答：立即数寻址：操作数是立即数的寻址方式

直接寻址：操作数**直接给出**而不需要任何变换
寄存器寻址：操作数是寄存器中的数值
寄存器间接寻址：操作数的地址放在寄存器中，而操作数本身放在存储器中

10. （必考）ARM 开发板有 2MB NOR Flash 和 128MB SDRAM，初始地址分别映射在 0x00000000 和 0x31000000。如果要求程序在 NOR Flash 中运行，则 Keil 环境中 Linker 的 target 地址配置选项应该为：RO Base=（0x00000000）；RW Base=（0x31000000）。

Analyse：常识

**11.** （必考）设 ARM 处理器当前寄存器 R0=0x1000，R1=0x300，R2=0x400；存储单元数值如表一所示（假设处理器工作在小端模式，CPSR 初始状态为 T=0，C=0，N=0，Z=0）。请给出相关指令执行后的结果（题中的指令是独立的，执行前状态相同）。

表一 内存中数据

| 地址 | 0xff0 | 0xff1 | 0xff2 | 0xff3 | 0xff4 | 0xff5 | 0xff6 | 0xff7 |
|---|---|---|---|---|---|---|---|---|
| 数据 | 0x11 | 0x22 | 0x33 | 0x44 | 0x55 | 0x66 | 0x77 | 0x88 |
| 地址 | 0xff8 | 0xff9 | 0xffa | 0xffb | 0xffc | 0xffd | 0xffe | 0xfff |
| 数据 | 0x99 | 0xaa | 0xbb | 0xcc | 0xdd | 0xee | 0xff | 0xf0 |
| 地址 | 0x1000 | 0x1001 | 0x1002 | 0x1003 | 0x1004 | 0x1005 | 0x1006 | 0x1007 |
| 数据 | 0x00 | 0xf1 | 0xf2 | 0xf3 | 0xf4 | 0xf5 | 0xf6 | 0xf7 |
| 地址 | 0x1008 | 0x1009 | 0x100a | 0x100b | 0x100c | 0x100d | 0x100e | 0x100f |
| 数据 | 0xf8 | 0xf9 | 0xfa | 0xfb | 0xfc | 0xfd | 0xfe | 0xff |

（1）“ADD R2, R2, R1” R2=(**0x700**), R1=(**0x300**), C=(**0**), N=(**0**)；
（2）“SUB**S** R2, R1, R2” R2=(**0xFFFFFF00**), R1=(0x300), C=(0), N=(1)；
（3）“CMP R2, R1” R2=(**0x400**), R1=(**0x300**), C=(**1**), N=(**0**)；
（4）“LDMDB R0!, {R1-R4}” R0=(**0xFF0**), R2=(**0xCCBBAA99**), R3=(**0x88776655**)；
（5）“LDRB R1, [R0], #4” R0=(**0x1004**), R1=(**0x00F1F2F3**)；
（6）“STR R1, [R0,#-4]” R0=(0xFFC), [0xFFD]=(0x00), [0xFFC]=(0x00),[0xFFE]=(0x03)；
（7）“BX R0” R15=(**0x1000**)；

Analyse：
注意：大端模式：内存低地址保存数据高字节
**考试时候是小端模式，内存高地址保存数据高字节**

二、(10’)判断
（1）ADD R3, R7, R2（right）
Analyse: 寄存器寻址，result is R3←R7＋R2
（2）SUB R11, R12, R3, LSL #34（error）
Analyse: the range of LSL is **0-31**, the range of LSR is 1-32;
（3）BL R0（error）
BL lable
B lable
BX Rm

BLX lable | Rm

Analyse：**跳转指令**

| 指令 | 语法 | 功能 |
|---|---|---|
| **B(跳转)** | **B{<cond>} label** | **pc=label，无条件跳转** |
| **BL(带返回跳转)** | BL{<cond>} label | **pc=label, lr=BL 后面第一条指令地址** |
| **BX(切换状态跳转)** | BX{<cond>} Rm | **pc=Rm & 0xfffffffe, T=Rm & 1** |
| **BLX** | BLX{<cond>} label\|Rm | **pc= label, T=1**<br>**pc=Rm & 0xfffffffe, T=Rm & 1**<br>**lr=BLX 后面的第一条指令地址** |

**注意：**B 语句只能在**±32MB** 空间跳转

（4）B lable （right）

（5）MOVS R4, R4, RRX #2 （error）

Analyse：RRX 后面不能跟东西，修改为“MOVS R4, R4, RRX”，

或者可以修改为“MOVS R4, R4, LSR/ASR #2”

（6）LDR R8, [R10] （right）

Analyse：寄存器间接寻址，result is R8←[R10]

（7）LDRNE R2, [R5, #960]! （right）

Analyse：基址变址寻址，result : if Z=0, then R2←[R5＋960]、R5←R5＋960

如果标志位 Z=0，存储器地址为 R5＋960 的字数据读入寄存器 R2，并将新地址 R5＋960 写入 R5

（8）LDR R0, localdata （error）

Analyse：lack of “=”, should revise that “LDR R0, =localdata”

**装载指令:主要是 MOV 和 MVN**

**包括伪指令 LDR**

| 指令 | **语法** | 功能 | 说明 |
|---|---|---|---|
| MOV | MOV{<cond>}{S} Rd, #const | 装载立即数 | **8bit,0x0 to 0xFF (0-255)** |
| MVN | MVN{<cond>}{S} Rd, #const | 装载立即数<br># (!const) | 可 **32bit** 位常量 |
| LDR | LDR Rd, =const | 装载常数(伪) | 32bit 位常量 |

例：LDR R0， =_start 得到的是绝对的地址，链接时决定;

（9）LDMia R0, {R1, R2, R3}（error）

Analyse：一个指令中大小写不能混用，should revise that “LDMIA/ldrmia R0, {R1-R3}”,

And the result is R1←[R0], R2←[R0+4], R3←[R0+8],

Extension:

LDMIB R0!, {R1, R2, R3};R1←[R0+1*4], R2←[R0+2*4], R3←[R0+3*4], R0←R0+3*4

STMDB R0!, {R3, R1, R2};[R0-1*4]←R1, [R0-2*4]←R2, [R0-3*4]←R3, R0]←R0-3*4

（10）CMNS R0, R1 （error?这条指令可以执行，不报错）

Analyse：Because for condition run, do not use the S suffix with **CMP, CMN, TST, or TEQ.** The instruction should revise that “CMN R0, R1”.

三、（10’）在 Keil 环境下开发程序时，Build 和调试过程中经常会出现错误信息提示。请分析产生下列错误信息的原因，并指出是哪一个阶段的提示。

（1）（2’）“Error: A163E: Unknown opcode Mov, expecting opcode or Macro”;

Analyse: opcode is error,ARM 指令不能大小写混用，

因为 arm 汇编器对标志符的大小写敏感，因此书写标志符及指令时，大小写要一致。在 arm 汇编程序中，指令、寄存器名可以全部为大写，也可以全部为小写，但是不能大小写混合使用

Translate 阶段（编译阶段 compile，编译错误）

解决办法：将“Mov”改写成“MOV”或者“mov”

扩展知识点：

Translate 是**编译**当前改动的源文件，在这个过程中检查语法错误。但并不生成可执行文件。

Build 是指**编译**工程中上次**修改**的文件及其它依赖于这些修改过的文件的模块，**同时重新链接**生成可执行文件。如果工程之前没编译链接过，它会直接调用 Rebuild All。另外在技术文档中，Build 实际上是指 increase build，即增量编译。

Rebuild 是不管工程的文件有没有编译过，会对工程中所有文件**重新进行编译生成可执行文件**，因此时间较长。

（2）（ 2’ ）“Error: L6406E: No space in execution regions with. ANY selector matching lab1.o(.text)”;

Analyse: 提示空间不够

Build 阶段，链接阶段错误（链接错误）

原因：有可能**某些段的 size 溢出**了。（**flash 溢出或者是 RAM 溢出**）

解决办法：

①使用微库

打开 Project->Options for target->Target，将 Use MicroLIB 前面的复选框勾上，然后重新编译、链接。

②修改链接脚本

在 Project->Options->Linker 中将 Use Memory Layout from Target Dialog 前面的复选框勾上。

然后在 Project->Options->Target 中**修改存储空间中只读部分和可读写部分的起始和大小**，一般来说加大只读部分大小（该部分存放程序中的指令），而减小可读写部分的大小（该部分存放堆栈、局部变量等）（尝试修改 ROM 空间，或者尝试修改 RAM 空间，将 Read/Write memory 中的 IRAM1 的 size 改大一些，再次 link）

最后，重新编译、链接

③将 keil option 里面的**编译优化**选项，从 level 0(-O0) 调高到 level 2(-O2)，看看经过优化还会不会有溢出的问题。**(依次点击 option for target ‘target1’ -> C/C++ -> optimization, 从 level 0(-O0) 逐步调高到 level 3(-O3)，观察结果，看看经过优化后是否还会溢出)**

④看看有没有 Scatter File,有就删了, Use Memory Layout from Target Dialog 勾上

⑤暴力解决换硬件，直接重新选择下芯片（ROM，RAM 空间不够）！！！！

（3）（2’）“Error: L16281E: undefined symbol main (referred from_rtentry2.o)”;

Analyse：错误提示为“main 这个符号未定义”

错误原因：
①没有将 main.c 文件添加到工程文件中；
②main 单词拼写错误
③启动代码不对
解决办法：
①检查是否将 main.c 添加到工程中
②检查 main 单词是否拼写错误
③去官网下载新的启动代码，把新下载的启动代码添加到工程中

（4）（2’）“Error: #20: identifier “i” is undefined”;
Analyse：
错误原因：①没有定义标识符“i”；②头文件定义有错误
解决办法：
①正确定义变量“i”，正确定义头文件
②可以用一个.h 文件来写需要在多个文件中使用的结构体
（5）（2’）“Error: Target DLL has been canceled. Debugger aborted”;
Analyse：错误提示为“目标 DLL 已被取消，调试器中止”
错误原因：
①Nor Flash 中的 U-BOOT 抢着启动！！！
②在"option for target"里的"output"里没有选择生成"hex"文件；
③没连接板子情况下，选了硬件仿真
④错误操作，比如热插拔（解决方法：重启板子）
解决办法：
①用 JLINK 将 Nor Flash 中的 U-BOOT 擦掉，就可以解决问题。
②在"option for target"里的"output"里选择生成"hex"文件；
③没连接硬件的话，选中软件仿真 Use Simulator！
在 Options for Target “Target 1”选项-->>Debug 选项里有两类仿真形式可选：Use Simulator 和 Use：Keil Monitor-51 Driver，前一种是纯软件仿真，后一种是带有硬件仿真器的仿真。
④实在不行，重装 Keil 软件。

四、（20’）如图 4-1，用 S3C2410 Port B 端口和段口 1 控制 LED0 和 LED1，GPIO 相关寄存器定义如图 4-2 所示
（1）（10’）编写 C 语言程序实现 LED0 和 LED1 循环交替发光；
Answer：

```
//led.c
int *rGPBCON = (int *) 0x56000010;
int *rGPBDAT = (int *) 0x56000014;
int main(){
    void delay();
    *rGPBCON = 0x05;//0b 0000 0000 0000 0101
    *rGPBDAT = 0x02;//0b 0000 0000 10,LED0 亮，LED1 灭
    delay(1000);//延时
    *rGPBDAT = 0x01;//0b 0000 0000 01,LED1 亮，LED0 灭
    delay(1000);//延时
```

```
}
//延时函数 delay 设计
void delay(int n){
    int i,j;
    for(i=0;i<n;i++)
        for(j=0;j<500;j++)  //执行 500 条空指令
            ;
}
```

（2）（5’）如果实现上电后能够自动执行，需要哪些条件？

Answer：

①Keil 环境中 Linker 的 target 地址配置选项应该为：

RO Base =0x00000000

RW Base = 0x30000000

②Output 中勾选“Create HEX file”

③DRAM 初始化：Initialization by code

Debug Init：需要在代码中加入初始化，上电后才能正常访问 RAM

map 0x48000000, 0x60000000 read write ;

④硬件无问题（电源，复位电路，BOOT 引导程序等）

（3）（5’）如果调试过程中程序工作正常，但写入 ROM 后上电程序不能正确执行，可能的原因是什么？

Answer:

电源有问题；复位电路有问题（缺少 reset 复位程序）；BOOT 启动有问题（启动模式选择错误）；复位以后，中断向量表不在程序起始位 0x00。

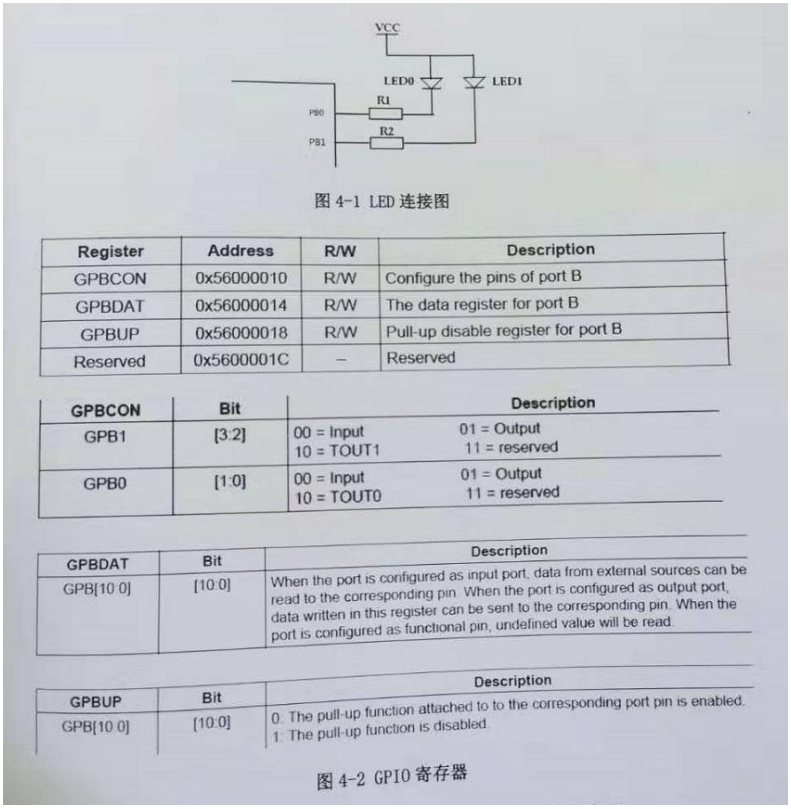



图 4-1 LED 连接图

| Register | Address | R/W | Description |
|---|---|---|---|
| GPBCON | 0x56000010 | R/W | Configure the pins of port B |
| GPBDAT | 0x56000014 | R/W | The data register for port B |
| GPBUP | 0x56000018 | R/W | Pull-up disable register for port B |
| Reserved | 0x5600001C | – | Reserved |

| GPBCON | Bit | Description |
|---|---|---|
| GPB1 | [3:2] | 00 = Input<br>10 = TOUT1<br>01 = Output<br>11 = reserved |
| GPB0 | [1:0] | 00 = Input<br>10 = TOUT0<br>01 = Output<br>11 = reserved |

| GPBDAT | Bit | Description |
|---|---|---|
| GPB[10:0] | [10:0] | When the port is configured as input port, data from external sources can be read to the corresponding pin. When the port is configured as output port, data written in this register can be sent to the corresponding pin. When the port is configured as functional pin, undefined value will be read. |

| GPBUP | Bit | Description |
|---|---|---|
| GPB[10:0] | [10:0] | 0: The pull-up function attached to to the corresponding port pin is enabled.<br>1: The pull-up function is disabled. |

图 4-2 GPIO 寄存器

五、（20’）给出单处理器上一维 N 个元素的数组求和的 C 函数，

```
int ArrayAdd(){
    int i;
    for(i=0;i<N;i++)
        c[i]=a[i]+b[i];
}
```

（1）（10’）请用 OpenMP 将该函数并行化，并给出主程序调用该函数的形式；

Answer：

①

```
int ArrayAdd(int N, int*a, int *b, int* c){
    int i;
    #pragma omp parallel for
    for(i=0;i<N;i++)
        c[i]=a[i]+b[i];
}
```

②

```
int main() {
    int i;
    int a[10] = { 0 };
    int b[10] = { 1,2,3,4,5,6,7,8,9,10 };
    int c[10];
    ArrayAdd(10,a,b,c);
    return 0;
}
```

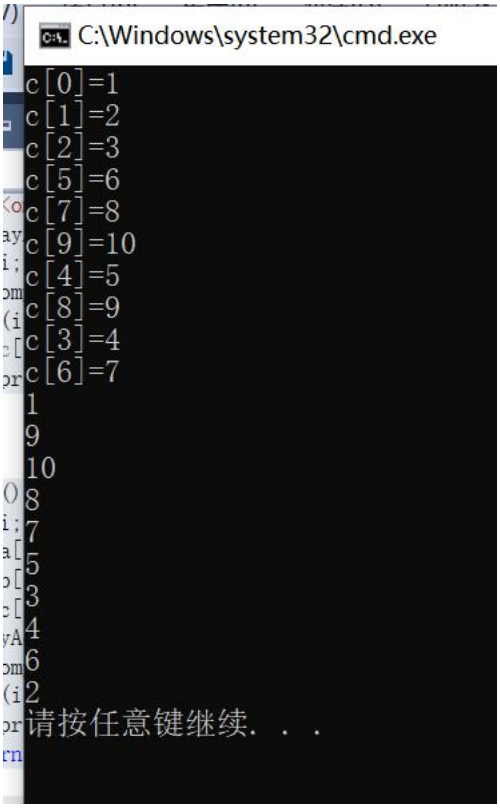
完整测试程序如下：

```
#include<stdio.h>
#include<omp.h>
void ArrayAdd(int N, int*a, int *b, int* c) {
    int i;
#pragma omp parallel for
    for (i = 0; i < N; i++) {
        c[i]=a[i]+b[i];
        printf("c[%d]=%d\n",i,c[i]);
    }
}

int main() {
    int i;
    int a[10] = { 0 };
    int b[10] = { 1,2,3,4,5,6,7,8,9,10 };
    int c[10];
    ArrayAdd(10,a,b,c);
#pragma omp parallel for
```

```
    for (i = 0; i < 10; i++)
        printf("%d\n",c[i]);
    return 0;
}
```

程序运行结果如下：




（2）（10’）请用 Cuda 改写成 GPU 上运行的函数，并给出 Host 函数调用该函数的形式。

Answer：

①

```
    //kernel definition
    __global__ void ArrayAdd(float* a, float* b, float* c){
        int i=threadIdx.x;
        c[i]=a[i]+b[i];
    }
```

②

```
ArrayAdd<<<1,N>>>(a, b, c); //kernel invocation with N thread
```