1. CPU结构的对比

X86：冯诺依曼结构，复杂指令集架构CISC(Complex Instruction Set Computing)

特点：数据和程序存储在一片物理内存。存储器操作指令多、汇编语言程序相对简单、指令结束后响应中断、CPU 电路丰富、面积大功耗大。

ARM:哈佛结构，精简指令集RISC(Reduced Instruction Set Computing)

特点：将程序指令存储和数据存储分开，中央处理器首先到程序指令存储器中读取程序指令。解码后到数据地址，再到相应的数据存储器读取数据，然后执行指令。对存储器操作有限汇编程序占空间大、在适当地方响应中断、 CPU电路较少，体积小、功耗低

2.ARMV8-A的执行状态

- 在AArch64执行状态下，ARMv8-A架构处理器只能使用A64指令集，该指令集的所有指令均为32位等长指令字。
- 在AArch32执行状态下，可以使用两种指令集：A32指令集对应ARMv7架构及其之前的ARM指令集，为32位等长指令字结构；T32指令集则对应ARMv7架构及其之前的Thumb/Thumb-2指令集，使用16位和32位可变长指令字结构

3.ARMV8的寄存器

AArch64状态下的通用寄存器

31 个 64 位的通用寄存器在汇编语言中被标识为X0~X30。由于 X30 被当作过程链接寄存器（Procedure Link Register，PLR），从严格意义上说，过程链接寄存器并不属于通用寄存器，因而也可以说 A64 指令集使用 30 个通用寄存器。

实际上，在AArch64状态下的应用程序可用使用R0~R30共31个通用寄存器，而每个通用寄存器都可以通过 64位和32位两种方式访问：当进行64位访问时，可以使用的通用寄存器名为 X0~X30；而当进行32位访问时，可以使用的通用寄存器名为 W0~W30。

3. 寄存器对比

Low 32-bits	ARM Register	Conventional use	X86 Register	Low 32-bits	Low 16-bits	Low 8-bits
W0	X0	Return value, callee-owned	%rax	%eax	%ax	%al
W0	X0	1st argument, callee-owned	%rdi	%edi	%di	%dl
W1	X1	2nd argument, callee-owned	%rsi	%esi	%si	%sl
W2	X2	3rd argument, callee-owned	%rdx	%edx	%dx	%dl
W3	X3	4th argument, callee-owned	%rcx	%ecx	%cx	%cl
W4	X4	5th argument, callee-owned	%r8	%r8d	%r8w	%r8b
W5	X5	6th argument, callee-owned	%r9	%r9d	%r9w	%r9b
W6	X6	7th argument, callee-owned				
W7	X7	8th argument, callee-owned				
W16	X16	Scratch/temporary, callee-owned	%r10	%r10d	%r10w	%r10b
W17	X17	Scratch/temporary, callee-owned	%r11	%r11d	%r11w	%r11b
WSP	SP	Stack pointer, caller-owned	%rsp	%esp	%sp	%spl
W19	X19	Local variable, caller-owned	%rbx	%ebx	%bx	%bl
W20	X20	Local variable, caller-owned	%rbp	%ebp	%bp	%bpl
W21	X21	Local variable, caller-owned	%r12	%r12d	%r12w	%r12b
W22	X22	Local variable, caller-owned	%r13	%r13d	%r13w	%r13b
W23	X23	Local variable, caller-owned	%r14	%r14d	%r14w	%r14b
W24	X24	Local variable, caller-owned	%r15	%r15d	%r15w	%r15b

3. 寄存器对比

Low 32-bits	ARM Register	Conventional use	X86 Register
W25	X25	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W26	X26	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W27	X27	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W28	X28	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
	PC(不可直接访问)	Instruction pointer	%rip（可用于地址访问）
	%CPSR	Status/condition code bits	%eflags
W9...W18	X9...X18	Temporary registers (临时寄存器，子函数内可使用)	
	LR(r30)	The Link Register.	
	FP(r29)	The Frame Pointer	
WZR	XZR	Zero-register(read only)	
		Code Segment	CS
		Default Data Segment	DS
		Stack Segment	SS
		Extra Data Segment	ES
		Additional Data Segment	FS
		Additional Data Segment	GS

3. 标志寄存器对比

AArch64	Flag Name	含义	X86
CF	Carry Flag	进位/借位Carry。如果算术指令的结果的最高有效位产生了进位/借位，则置1；否则置0。这个标志位指示了无符号整数计算的结果是否溢出	CF
	Parity Flag	奇偶校验位Parity。如果指令结果的最低字节含有偶数个比特1，则置1；否则置0。	PF
	Auxiliary Flag	辅助进位/借位Auxiliary Carry。如果算术运算在第3比特位（最低比特位是0）产生了进位/借位，则置1；否则置0。这个标志主要用于二进制编码的十进制数BCD算术运算。	AF
ZF	Zero Flag	零标志位Zero。如果指令结果为0，则置1；否则置0	ZF
NF(negative)	Sign Flag	符号位Sign。设置为指令结果的最高有效比特位，即有符号整数的符号比特位。0表示结果是正数或者是0；1表示负数。	SF
VF	Overflow Flag	溢出位Overflow。如果整数运算结果超出了目标操作数可容纳的值域（正数过大，负数过小，不包括符号位），则置1；否则置0（即结果可信）。这个标志指示了有符号整数算术运算的结果是否溢出。	OF
	DF Direction Flag		

4. 程序框架——GNU ARM程序


```
.text                       ; 定义一个代码段
.global tart1               ;定义一个全局标号，可被其它文件调用
tart1:
        mov x0,#0
        ldr x1,=msg         ;获取msg的地址
        mov x2,len          ;
        mov x8,64
        svc #0              ; 系统调用输出一个字符串

        mov x0,123
        mov x8,93
        svc #0              ; 退出程序返回操作系统

.data                       ; 定义数据段
msg:                        ; 定义数据段的一个变量
        .ascii "Hello World!\n"
len=.-msg                   ; .为该语句当前地址，len为msg长度
```

5. ARM V8寻址方式

寻址就是找到存储数据或指令的地址，寻址方式的方便与快捷是衡量CPU性能的一个重要方面，ARM处理器共有八种寻址方式：

- 立即数寻址
- 寄存器寻址
- 寄存器间接寻址
- 基址寻址
- 多寄存器寻址
- 堆栈寻址
- 相对寻址
- 寄存器移位寻址

5. ARM V8寻址方式

寻址就是找到存储数据或指令的地址，寻址方式的方便与快捷是衡量CPU性能的一个重要方面，ARM处理器共有八种寻址方式：

(1)立即数寻址：

立即数寻址指令中的地址码就是操作数本身，可以立即使用的操作数。其中，#0xFF000和#64都是立即数。如操作数是常量，用#表示常量；0x或&表示16进制数，否则表示十进制数。

例1：

```
MOV   X0，#0xFF000 @将立即数0xFF000(第2操作数)装入X0寄存器
```

例2：

```
SUB X0, X0, #64          @X0减64，结果放入X0
```

5. ARM V8寻址方式

寻址就是找到存储数据或指令的地址，寻址方式的方便与快捷是衡量CPU性能的一个重要方面，ARM处理器共有八种寻址方式：

- (2)寄存器寻址：操作数的值在寄存器中，指令执行时直接取出寄存器值来操作，寄存器寻址是根据寄存器编码获取寄存器内存储的操作数

例1：

```
MOV   X1，X2          @将X2的值存入X1
```

例2：

```
SUB  X0，  X1，  X2     @将X1的值减去X2的值，结果保存到X0
```

5. ARM V8寻址方式

寻址就是找到存储数据或指令的地址，寻址方式的方便与快捷是衡量CPU性能的一个重要方面，ARM处理器共有八种寻址方式：

- (3)寄存器间接寻址：操作数从寄存器所指向的内存中取出，寄存地存储的是内存地址

例1:LDR X1，[X2] @将X2指向的存储单元的数据读出保存在X1中，X2相当于指针变量

例2：STR X1, [X2] @将X1的值写入到X2所指向的内存

例3：SWP X1,X1,[X2] @将寄存器X1的值和X2指定的存储单元的内容交换

[X2]表示寄存器所指向的内存
LDR指令用于读取内存数据
STR 指令用于写入内存数据

5. ARM V8寻址方式

- (4)基址变址寻址：基址寄存器的内容与指令中的偏移量相加，得到操作数的有效地址，然后访问该地址空间，基址变址寻址分为三种：
- 前索引： LDR X0, [X1, #4]
  @X1存的地址+4，访问新地址里面的值，放到X0；

- 自动索引：LDR X0, [X1, #4]!
  @在前索引的基础上，新地址回写进X1；(注：!表示回写地址)

- 后索引：LDR X0 [X1], #4
  @X1存的地址的内容写进X0，X1存的地址+4再写进X1；

5. ARM V8寻址方式

- (5)多寄存器寻址：一条指令完成多个寄存器的传送，最多16个寄存器，也称为块拷贝寻址

例：LDMIA X1!, {X2-X7,X12} @将X1指向的存储单元中的数据读写到X2～X7、X12中，然后X1自加1

例：STMIA X1!,{X2-X7,X12} @将寄存器X2～X7、X12的值保存到X1指向的存储单元中，然后X1自加1

注：基址寄存器不允许为X15，寄存器列表可以为X0～X15 的任意组合。这里X1没有写成[X1]!，是因为这个位不是操作数位，而是寄存器位

LDMIA 和 STMIA 是块拷贝指令，LDMIA是从X1所指向的内存中读数据，STMIA是向R1所指向的内存写入数据

执行这类指令要考虑如下几个问题：

基址寄存器指向原始地址有没有放一个有效值
寄存器列表哪个寄存器被最先传送
存储器地址增长方向
指令执行完成后，基址寄存器有没有指向一个有效值

5. ARM V8寻址方式

- (6)寄存器堆栈寻址：是按特定顺序存取存储区，按后进先出原则，使用专门的寄存器SP（堆栈指针)指向一块存储区

例：LDMIA SP!, {X2-X7,X12} @将栈内的数据，读写到X2～X7、X12中，然后下一个地址成为栈顶

例：STMIA SP!,{X2-X7,X12} @将寄存器X2～X7、X12的值保存到SP指向的栈中，SP指向的是栈顶

5. ARM V8寻址方式

- (7)相对寻址：即读取指令本身在内存中的地址。是相对于PC内指令地址偏移后的地址。由程序计数器PC提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址

```
BL     ROUTE1          @调用到 ROUTE1 子程序
BEQ    LOOP            @条件跳转到 LOOP 标号处
    …
LOOP MOV R2,#2
    …
ROUTE1
  …
```

6. ARM V8指令集

GNU ARM汇编语言语句格式：

[<label>:][<instruction or directive or pseudo-instruction>} @comment

- instruction伪指令
- directive伪操作
- pseudo-instruction伪指令
- <label>：为标号，GNU ARM汇编中，任何以冒号结尾的标识符都被认为是一个标号，而不一定非要在一行的开始
- comment为语句的注释

注意：

- ARM指令，伪指令，伪操作，寄存器名可以全部为大写字母，也可全部为小写字母，但不可大小写混用。
- 如果语句太长，可以将一条语句分几行来书写，在行末用“\”表示换行（即下一行与本行为同一语句），“\”后不能有任何字符，包含空格和制表符（Tab）。

6. ARM V8指令集

GNU ARM汇编语言语法格式：

- 局部变量定义的语法格式：N{routname}
  - N：为0~99之间的数字。
  - routname：当前局部范围的名称（为符号），通常为该变量作用范围的名称（用ROUT伪操作定义的）
- 局部变量引用的语法格式：%{F|B}{A|T}N{routname}
  - %：表示引用操作
  - N：为局部变量的数字号
  - routname：为当前作用范围的名称（用ROUT伪操作定义的）
  - F：指示编译器只向前搜索
  - B：指示编译器只向后搜索
  - A：指示编译器搜索宏的所有嵌套层次
  - T：指示编译器搜索宏的当前层次

6. ARM V8指令集——A64指令特点

- A64指令编码宽度固定32bit
- 31个（X0-X30）个64bit通用用途寄存器（用作32bit时是W0-W30），寄存器名使用5bit编码
- PC指针不能作为数据处理指或load指令的目的寄存器，X30通常用作LR
- 移除了批量加载寄存器指令 LDM/STM, PUSH/POP, 使用STP/LDP 一对加载寄存器指令代替
- 增加支持未对齐的load/store指令立即数偏移寻址，提供非-暂存LDNP/STNP指令，不需要hold数据到cache中
- 没有提供访问CPSR的单一寄存器，但是提供访问PSTATE的状态域寄存器
- 相比A32少了很多条件执行指令，只有条件跳转和少数数据处理这类指令才有条件执行.
- 支持48bit虚拟寻址空间
- 大部分A64指令都有32/64位两种形式
- A64没有协处理器的概念

6. ARM V8指令集——A64指令特点

跳转指令：条件跳转

指令	说明
B.cond	cond为真跳转
CBNZ	CBNZ X1，label　//如果X1!= 0则跳转到label
CBZ	CBZ X1，label　//如果X1== 0则跳转到label
TBNZ	TBNZ X1，#3 label　//若X1[3]!=0,则跳转到label
TBZ	TBZ X1，#3 label　//若X1[3]==0,则跳转到label

跳转指令：绝对跳转

指令	说明
B	绝对跳转
BL	绝对跳转 #imm，返回地址保存到LR（X30）
BLR	绝对跳转reg，返回地址保存到LR（X30）
BR	跳转到reg内容地址,
RET	子程序返回指令，返回地址默认保存在LR（X30）

6. ARM V8指令集——A64指令特点

算术运算指令

指令	说明
ADDS	加法指令，若S存在，则更新条件位flag
ADCS	带进位的加法，若S存在，则更新条件位flag
SUBS	减法指令，若S存在，则更新条件位flag
SBC	将操作数 1　　减去操作数 2，再减去 标志位C的取反值，结果送到目的寄存器Xt/Wt
RSB	逆向减法，操作数 2 –操作数 1，结果 Rd
RSC	带借位的逆向减法指令，将操作数 2 减去操作数 1，再减去 标志位C的取反值，结果送目标寄存器Xt/Wt
CMP	比较相等指令
CMN	比较不等指令
NEG	取负数运算，NEG X1，X2 // X1 = X2按位取反+1（负数=正数补码+1）
MADD	乘加运算
MSUB	乘减运算
MUL	乘法运算
SMADDL	有符号乘加运算
SDIV	有符号除法运算
UDIV	无符号除法运算

6. ARM V8指令集——A64指令特点

逻辑运算指令

指令	说明
ANDS	按位与运算，如果S存在，则更新条件位标记
EOR	按位异或运算
ORR	按位或运算
TST	例如：TST W0，#0X40 //指令用来测试W0[3]是否为1,相当于： ANDS WZR,W0，#0X40

数据传输指令

指令	说明
MOV	赋值运算指令
MOVZ	赋值#uimm16到目标寄存器Xd
MOVN	赋值#uimm16到目标寄存器Xd，再取反
MOVK	赋值#uimm16到目标寄存器Xd，保存其它bit不变

移位运算指令

指令	说明
ASR	算术右移 >> （结果带符号）
LSL	逻辑左移 <<
LSR	逻辑右移 >>
ROR	循环右移：头尾相连
SXTB	字节、半字、字符号/0扩展移位运算 关于SXTB #imm和UXTB #imm 的用法可以使用以下图解描述：
SXTH	
SXTW	
UXTB	
UXTH	

6. ARM V8指令集——伪指令

数据定义（Data Definition）伪操作

- .byte 单字节定义　　.byte 0x12,’a’,23
- .short 定义2字节数据　　.short　　0x1234,65535
- .long /.word 定义4字节数据　　.word 0x12345678
- .quad　定义8字节　　.quad 0x1234567812345678
- .float 定义浮点数　　.float　　0f3.2
- .string/.asciz/.ascii　　定义字符串
- .ascii　“abcd\0”,

6. ARM V8指令集——伪指令

汇编控制伪操作

- .if .else .endif ---- 类似c语言里的条件编译，汇编控制伪操作用于控制汇编程序的执行流程 .if、.else、.endif伪操作能根据条件的成立与否决定是否执行某个指令序列。当.if后面的逻辑表达式为真，则执行.if后的指令序列，否则执行.else后的指令序列；.if、.else、.endif伪指令可以嵌套使用。
- .macro，.endm --- 类似c语言里的宏函数 .macro伪操作可以将一段代码定义为一个整体，称为宏指令。然后就可以在程序中通过宏指令多次调用该段代码。其中，$标号在宏指令被展开时，标号会被替换为用户定义的符号。宏操作可以使用一个或多个参数，当宏操作被展开时，这些参数被相应的值替换。

7. ARM V8编译调试工具

使用gcc编译器编译c文件流程

- 预编译处理，生成main.i文件
- 编译处理，生成main.s文件
- 汇编处理，生成main.o文件
- 链接处理，生成main.exe文件

7. ARM V8编译调试工具

- 保存hello.s文件，然后通过运行以下命令将其编译为二进制文件：

```
as hello.s -o hello.o
```

使用以下命令进行链接，输出可执行文件：

```
ld hello.o -o hello
```

使用以下命令执行hello程序

```
./hello
```