复习list

- Digital Electronics:
    - ASM Design techniques
    - One Hot vs Encoding method
    - Mealy / Moore machines
    - Quine-McCluskey v Karnaugh Maps. Lec 6a Lec 6b
    - Registers and Serial data transmission
- Microprocessor Systems:
    - Shift-register tut
    - Branch Lec22
    - Flags Lec22
    - Negative numbers
    - PC vs LR Lec26
    - 🌲IEEE754 standard and Lec24
    - 🌲Carry vs Overflow. Lec24

卡诺图

shiftting时候的flag的值

ASM Design techniques

Algorithmic State Machine (ASM)

ASSEMBLY INSTRUCTION		**FLAGS**
Simple instructions		
MOVS rd, rm	从寄存器 rm 中取	N and Z flags updated, C and V flags unchanged

MOVS rd, #imm	值，将一个值移动到目标寄存器 rd 中 从立即数 #imm 中取值，附值给rd **MOVS指令：8 位立即数**	#imm不可以超过 23bit
LDR rd, =0xZZZZZZZZ LDR rd, =label	把固定的 32 位值中取值，附值给目标寄存器 rd 目标寄存器rd的值赋和label **LDR伪指令：32 位立即数**	All flags unchanged
Arithmetic instructions		
ADDS rz, ry, rx	rz=ry+rx	All flags updated
ADDS rz, ry, #imm	rz=ry+立即数	
SUBS rz, ry, rx *SUBS rz, ry, #imm*	rz,=ry- rx *rz= rz- ry*	All flags updated
RSBS rz, ry, #0	ry 中的值取反，结果存目标寄存器 rz Reverse Subtraction， **rz = 0 - ry**	All flags updated
MULS rx, ry, rx	ry 和 rx 中的值相乘，存rx Multiply with Sign Multiplication	N and Z flags updated, C and V flags unchanged 如果结果超过了可以表示的范围，那么结果将被截断，而不会产生进位或溢出。因此，`MULS` 指令不会改变 C 和 V 标志位。
ADCS rx, ry	rx=rx+ry+value of C flag	All flags updated

这个在lec26但是被放入简单算术运算

计算的是超出32bit的数字相加，这两个数字各被放在两个地址上(总共4个地址)，这个就是把低bit上的进位给算入总的高位计算中。

MOVS rd, rm 和 **MOVS rd, #imm**：

指令都是将一个值移动到目标寄存器 rd 中。第一条指令从寄存器 rm 中取值，第二条指令从立即数 imm 中取值。这两条指令会更新 N 和 Z 标志位，但不会改变 C 和 V 标志位。

C for unsigned V for signed（负负出正）

在汇编语言中，标签（Label） 是一个用来标识特定位置的符号或名称。标签通常用于标记程序中的某个位置，如指令的起始位置、数据的存储位置或跳转的目标位置等。

标签可以被视为一个指向内存地址的指针。

ADCS rx, ry计算carry上

是怎么操作的（已经更新）

LOGICAL INSTRUCTIONS		
ANDS ry, rx	拆开来乘	N, Z and C flags updated, V flag unchanged
ORRS ry, rx	加起来看1 or 0	N, Z and C flags updated, V flag unchanged
EORS ry, rx	可以理解成打开来1，0出1或者0，1出1，即不同的值出1	N, Z and C flags updated, V flag unchanged
BICS ry, rx	首先ry是最后的输出的值，ry在上，rx在下，rx等于1堵住不让过	N, Z and C flags updated, V flag unchanged
Shift instructions		
LSLS rx, ry, #imm LSLS ry, ry, rz	Logical shift left逻辑左移， 所有的数字往左移动， 后面使用0填充	N, Z and C flags updated, V flag unchanged
LSRS rx, ry, #imm LSRS ry, ry, rz	logical shift right 逻辑右移 所有数字往右移动 前面数字使用0填充	N, Z and C flags updated, V flag unchanged
ASRS rx, ry, #imm ASRS ry, ry, rz	Aritemetic 右移， 算数的意思的保留第一位的符号位置，比如sign的负数，还是会记录是负数	N, Z and C （因为C是unsigned的） flags updated, V flag unchanged
RORS ry, ry, rz	Rotate循环右移，屁股出尾巴进。 唯一不可以使用立即数进行计算	N, Z and C flags updated, V flag unchanged

LSRS ry, ry, rz 这种都是ry=ry使用 rz的数据shift。

rz 数据是指rz在0x(16 bit)的最后两个的情况下，最后两个数据

r5 0x00008008这个意思是r5这个在Register bank中的位置上，储存的值是0x00008008，这个储存的值是32 bit的。再比如，MOVS r6, r5这个instruction对应的Machine code是16 bit的，在ARM M0中，可以储存在32bit的地址上，也可以储存在32bit的地址的指代的Register bank中的位置上?

位移可以等效于乘法除法操作

结果就是实际结果往后面取，比如-1.2取-2，-1.8取-2，1.8取1，1.2取1。对于负数结果the result is always rounded up

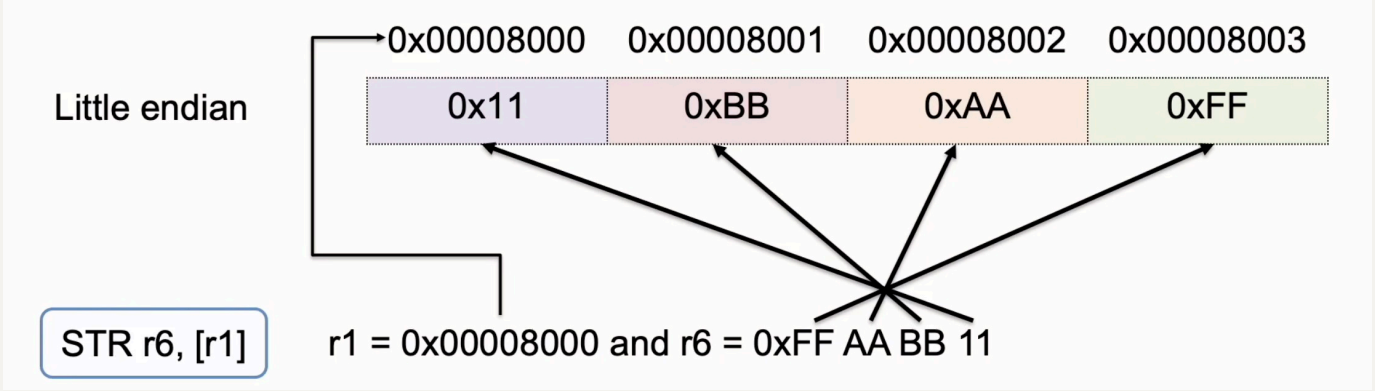
Load, Store and Endianness instructions

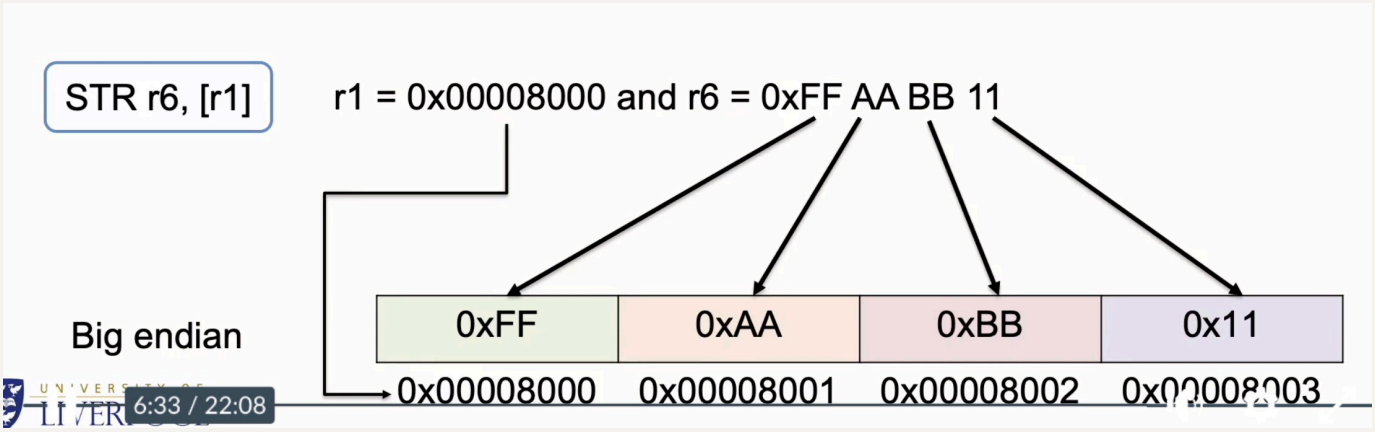
立即数，有限制：**label**，有限制

LOAD, STORE AND ENDIANNESS INSTRUCTIONS		
LDR rd, [rn] LDR rd, [rn, #imm] *LDR rd, [rn, rm]* LDR rd, =0x12345678	**LoaD Register** 寄存器r1中保存的值是0x00004000，而内存地址0x00004000中保存的数据是0xF97D5EC5。那么： 执行指令LDR r3,[r1]，那么将会把数据0xF97D5EC5加载到寄存器r3中。 执行指令MOV r3,r1，那么将会把值0x00004000（即r1中的值）加载到寄存器r3中 数据从内存加载到寄存器中 ***LDR rd, [rn, rm];*** *结果地址是通过将 rm 中保存的值加上 rn 中保存的值来计算的。 没有任何限制，并且使用整个 32 位值，而不仅仅是逻辑移位指令中的 l.s.byte *。 载入 寄存器	All flags unchanged
LDRH rd, [rn] LDRH rd, [rn, #imm] LDRH rd, [rn, rm]	**LoaD Register Half** 要后一半，剩下空的放置0	All flags unchanged

LDRSH rd, [rn, rm]	**LoaD Register Sign Half** 要后一半，有sign	All flags unchanged
LDRB rd, [rn] LDRB rd, [rn, #imm] LDRB rd, [rn, rm]	**LoaD Register Byte** **后8 bit**	All flags unchanged
LDRSB rd, [rn, rm]	**LoaD Register Sign Byte**	All flags unchanged
STR rd, [rn] STR rd, [rn, #imm] STR rd, [rn, rm]	STR就是位置反转，STR r1, [r6]是指R6等于R1的反转后的值 **STore Register** STRH将数据从寄存器存储到内存中 简称返回	All flags unchanged
STRH rd, [rn] STRH rd, [rn, #imm] STRH rd, [rn, rm]	**STore Register Half**	All flags unchanged
STRB rd, [rn] STRB rd, [rn, #imm] STRB rd, [rn, rm]	**STore Register B** **后8 bit**	All flags unchanged
REV ry, rx	换一下rx的顺序写入ry，在r0到r7 low registers only	All flags unchanged

```
1  LDR R1,=0xE0000000 ;R1=0xE000 0000
2  LDR R1,0xE0000000  ;将内存中地址为0xE0000000的内容载入到R1
3  LDR R1, [R0]        ;将R0中的数所指定的地址的内容传输到R1
4  LDR rd, [rn, rm]    ;?
5  STR rd, [rn, #imm]  ;将Rn中的数偏移#imm所指定的地址的内容传输到Rd
```



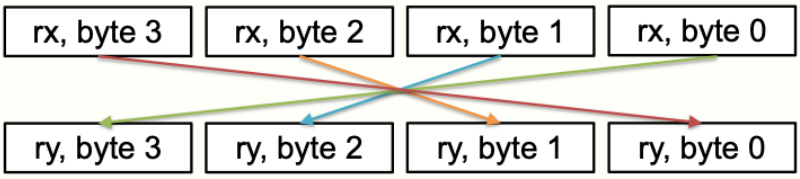


Reversing byte order

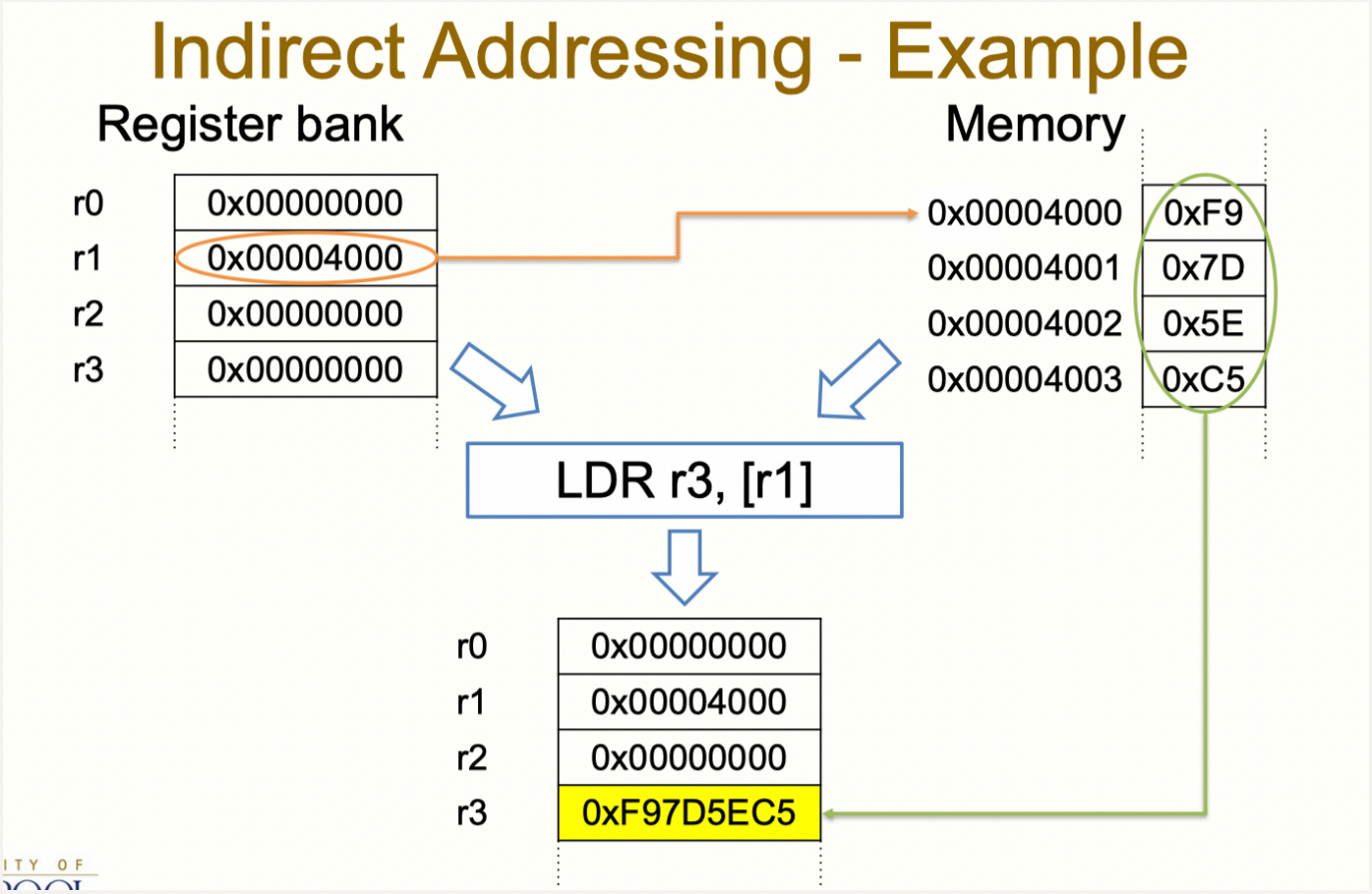
The ARM Cortex M0 processor has an instruction that switches data in registers between big endian and little endian.

REV ry, rx

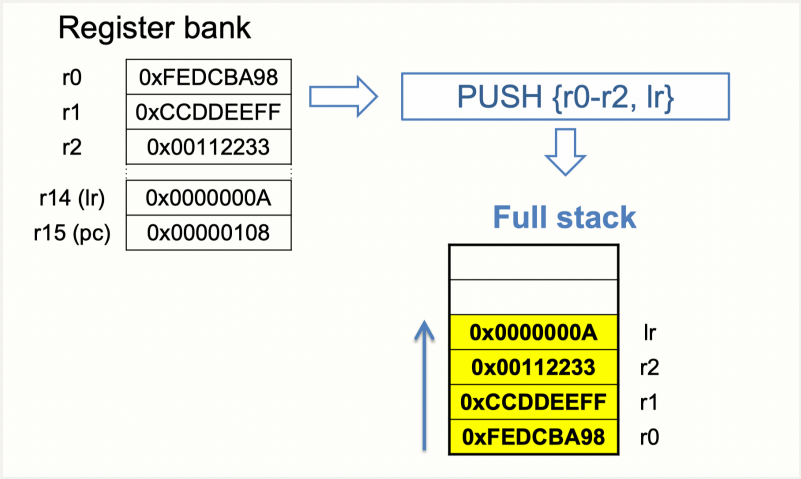
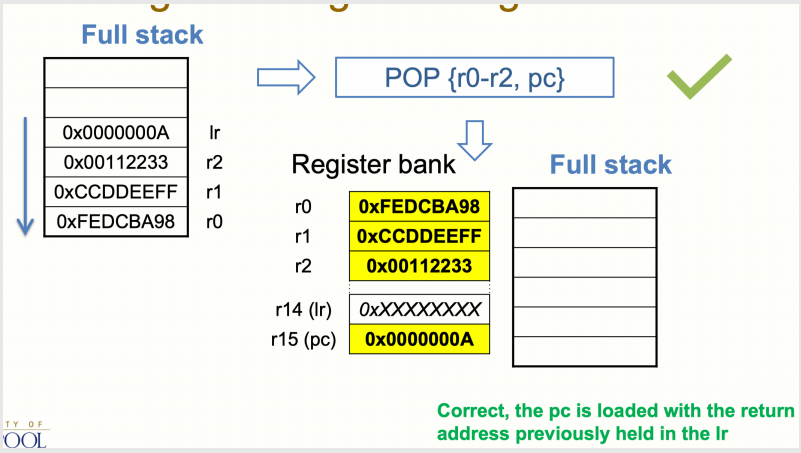
Bytes 0, 1, 2 and 3 of register rx are copied into bytes 3, 2, 1 and 0 of register ry in that order.



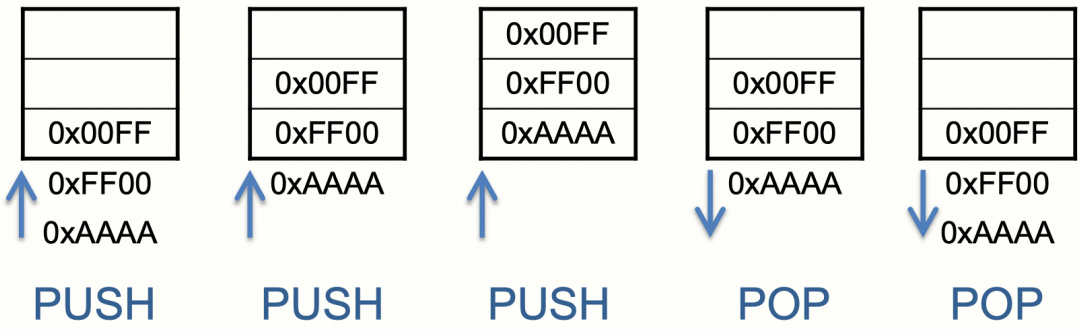
UNIVERSITY OF LIVERPOOL



BRANCH INSTRUCTIONS		
B <target_address|label>		All flags unchanged
B <target_address|label>	**Conditional execution** The common ones are: • EQ: 'equal', it is executed only if the zero flag (Z) is set • NE: 'not equal', it is executed if the zero flag (Z) is clear • CS: 'carry set', it is executed if the carry flag (C) is set • CC: 'carry clear', it is executed if the carry flag (C) is clear • MI: 'minus', it is executed only if the negative flag (N) is set • PL: 'plus' or 'positive', executed only if the negative flag (N) is clear • VS: 'overflow', executed if V is set • VC: 'no overflow', executed if V is clear • AL: 'always', execute always unconditionally (default if no condition field is specified.	All flags unchanged
BL <target_address|label>	可以使用label，但对target_address有限制 a 32 bit instruction（执行完毕以后PC会加4） **branch with link,** which uses a target address or label to know where to branch to	All flags unchanged
BX rz	ranch and exchange	All flags

		unchanged
BLX rz	branch, link and exchange 不可以使用label， 需要使用实际的地址	All flags unchanged
Stack-related instructions		
PUSH {<register(s)>}		All flags unchanged
POP {<register(s)>}		All flags unchanged

- E.g., if the values pushed onto a stack where 0x00FF, 0xFF00, 0xAAAA in that order then they would be popped from the stack in the reverse order



BL BX BLX （还没有解释）

Q: Could you clarify how the BL, BX and BLX instructions differ and how these can be used to create an Assembly program?

A: 在回答之前，让我们回忆一下，我们可以使用包含指令跳转位置（memory address containing an instruction）的 branch or a label. 主要的区别是，如果使用内存地址，我们（程序员）必须知道指令存储在内存中的位置，而如果使用标签，汇编程序将自动用指令的内存地址替换标签**memory address**。请记住，标签**label**可以是任何帮助我们记住跳转位置的词，例如循环（loop）、继续（continue）、下一个（next）等。

重要的是要记住，

BL和BLX

用于主程序（或调用子程序）跳转到子程序（或嵌套子程序）。这两个指令都会更新链接寄存器（r14或lr），将返回地址保存在主程序（或调用子程序）中，并重新加载程序计数器（r15）为指示子程序（或嵌套子程序）第一条指令内存位置的值。而BX指令用于子程序（或嵌套子程序）内部返回到主程序（或调用子程序）。

Let's see two practical examples:

- Using BL and BX:

```
1  Address         Main program            |       Address
   Subroutine
2  ------------------------------------------------------------------
   ---------
3  0x00008000      BL sub1                 |       0x00004000  sub1 SUBS
   r0, r4, #1
4  0x00008004      ADDS r3, r2, #2         |       0x00004002        BX lr
```

the BL instruction uses the label **sub1** to identify the memory location of the first instruction in the subroutine (SUBS).
The Assembler will convert **sub1** into **0x00004000** automatically.
BL will update the link register **lr** with the value **0x00008004** and the program counter with the value **0x00004000**, whereas the **BX** instruction uses the value of '**lr**' to branch back to the main program,

使用**BL**开始 时， **PC** `0x00004000` **LR**从`0x00008000` 到 `0x00008004`，这里因为BL是32bit的指令所以加4，正常16bit的指令加2

使用**BX**返回 时,**PC** `0x00008004` 是之前**LR**保存的`0x00008004`

- Using BLX and BX:

```
Address         Main program            |       Address
Subroutine
-------------------------------------------------------------------
---------
0x00009002      LDR r1, =0x00004000     |
0x00009004      BLX r1                  |       0x00004000        SUBS
r0, r4, #1
0x00009006      ADDS r3, r2, #2         |       0x00004002        BX lr
```

As you can see from the example, the BLX instruction uses the value in register **r1** to identify the memory location of the first instruction in the subroutine (SUBS). Please note that I used the LDR pseudo-instruction to load the value in register r1, and we, as programmers, must know this value in advance; the Assembler is using it directly (no labels). Please also note that the BLX instruction will update the link register with the value **0x00009006** and the program counter with the value **0x00004000**, whereas the BX instruction uses the value of 'lr' to branch back to the main program.

使用**BXL**开始 时， **PC** `0x00004000`因为这是subroutine的下一个地址 **LR**从`0x00009004` 到 `0x00009006`，这里因为BXL是16bit的指令加2

使用**BX**返回 时,**PC**变成 `0x00009006` 是之前**LR**保存的`0x00009006`

BX指令将使用链接寄存器（link register）的值（即0x00009006）更新程序计数器（program counter）的值

- Using BL, BLX, and BX:

```
Address        Main program          |      Address     Calling
Subroutine     |     Address          Nested subroutine
-------------------------------------------------------------------
-------------------------------------------------
0x00009002     LDR r1, =0x00004000   |      0x00004000       PUSH
{lr}          |
0x00009004     BLX r1                |      0x00004002       BL
nested         |     0x00002000    nested SUBS r0, r4, #1
0x00009006     ADDS r3, r2, #2       |      0x00004006       POP
{pc}           |     0x00002002           BX lr
```

```
LDR R1,=0xE0000000 ;R1=0xE000 0000
LDR R1, 0xE0000000 ; Load the contents of the memory address
0xE0000000 into R1
LDR R1, [R0] ;Transfer the contents of the address specified by the
number in R0 to R1
LDR rd, [rn, rm] ;  ???
STR rd, [rn, #imm]  ;Transfer the content of the register specified
by the number offset #imm in Rn to Rd
```

16进制的加减乘除

2B_16 X 5_16=

(2x10+11)x5=31x5=155

155÷=9(144).....11

=B9_16

$$2B_{16} * 5_{16} = \tag{1}$$

方法一

计算16进制数2B乘以5，我们需要逐位相乘，然后将结果相加。

B（11）* 5 = 55（十进制），用十六进制表示为37。将7写在结果的个位数，进位3。2 * 5 = A（十进制），用十六进制表示为A。然后加上进位3，得到D。将D写在结果的十位数。
所以，2B x 5 = D7。

方法二

将16进制数转换为10进制数，然后进行乘法运算，最后将结果转换回16进制。

将16进制数转换为10进制数：
2B（十六进制）= 2 * 16^1 + 11 * 16^0 = 32 + 11 = 43（十进制）
用10进制数进行乘法运算：
43（十进制）x 5 = 215（十进制）
将结果转换回16进制：
215（十进制）= 13 * 16^1 + 7 * 16^0 = D * 16^1 + 7 * 16^0
所以，215（十进制）= D7（十六进制）
因此，2B x 5 = D7。

Limitation of immediate addressing

The limitation of immediate addressing arises when using a processor like ARM, where the instruction code is 16 bits (sometimes 32 bits) long. This code must include information about the type of instruction (e.g., ADDS, MOVS, etc.), the destination register, and the immediate value. Due to this limitation, a 32-bit value cannot be put into a 32-bit register using immediate addressing and MOVS.

就是不可以直接放下32bit的信息 using immediate addressing and MOVS这些指令，对于一个32位的处理器来说。

为了克服这个限制，我们可以使用 LDR pseudo-instruction(伪指令)，它用 32 位immediate value to load 一个寄存器。 例如，指令 LDR r3, =0x10000000 将 r3 中的值设置为 0x10000000。

The restrictions on immediate values

immediate values被限制为给定的位数，例如 3、5、7 或 8 bits

对于目标寄存器和源寄存器相同的MOVS指令和ADDS/SUBS指令，允许8位值。 8 位值的允许范围是十进制 0 到 255（含）。

E.g. ADDS r6, r6, #99
- Add 99_{10} from value in r6
- Machine code is **0x**3663 or 0011 0110 0110 0011_2

对于目标寄存器与源寄存器不同的 ADDS 和 SUBS 指令，允许使用 3 位立即数。 3 位值的允许范围是十进制 0 到 7（含）。

E.g. SUBS r3, r1, #5
- Subtract 5_{10} from value in r1 and put the difference in r3 .

- Machine code is **0x**1F4B or 0001 1111 01001 011_2

Indirect Addressing 间接寻址

address using a value in a register to identify a memory address

```
1  LDR rd, [rn]
```

- rd 是要加载的寄存器,在加载完成后，值会出现在rd寄存器的位置。
- [rn] 表示正在使用寄存器 rn 中包含的值作为内存地址,指令发生前数据储存的地方，但是rn记录的是数据在memory上的位置，而不是数据直接储存在rn上

example

```
1  LDR r3, [r1]
```

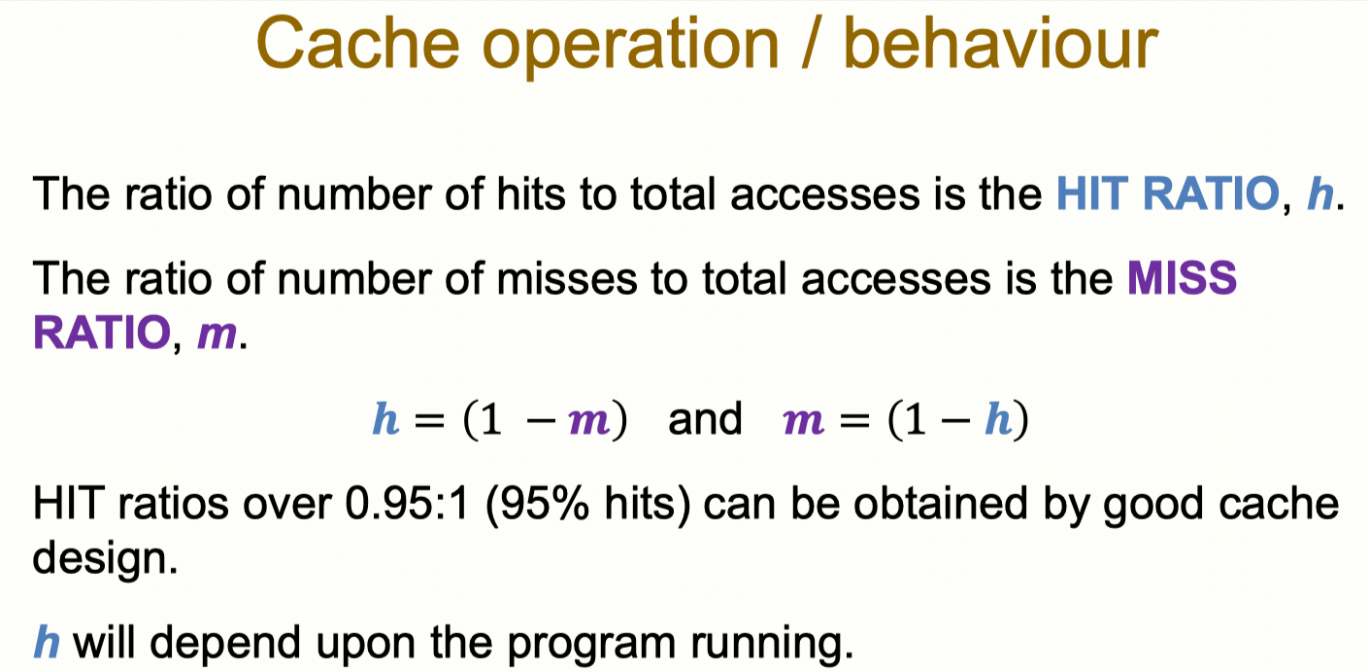
在这种情况下，r1 的值是 0x00004000，这是一个内存地址。 地址为 0x00004000 的内存位置保存值 0xF97D5EC5。 因此，当您执行 LDR 指令时，它将存储在内存地址 (0xF97D5EC5) 的值加载到 r3。

执行该指令后，r3 将包含值 0xF97D5EC5。

```
1  MOV r3, r1
```

这条指令的意思是“将寄存器 1 中的值移动到寄存器 3 中”。
在这种情况下，r1 的值是 0x00004000。 当你执行MOV指令时，它会将r1中存储的值（0x00004000）直接传送到r3中。

执行该指令后，r3 将包含值 0x00004000。



Mean access time

Simple approach (ignoring cache controller overheads) analysis suggests for many accesses with a cache access time of t_c, and a main memory access time of t_m the **mean access time** will be

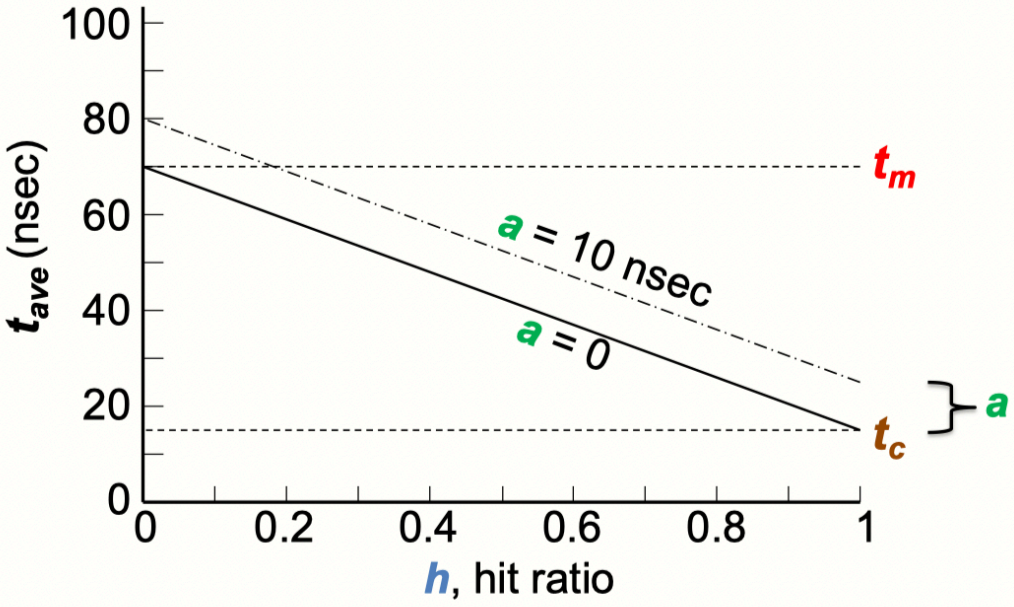
$$t_{ave} = h \times t_c + m \times t_m = h \times t_c + (1 - h) \times t_m$$

In practice the cache control and routing circuits will add an extra time delay of a.

$$t_{ave} = h \times t_c + m \times t_m + a =$$

$$h \times t_c + (1 - h) \times t_m + a$$

Mean access time



t_c = 15 nsec, t_m = 70 nsec