Lec16

μProcessor

Definitions

CPU: central processing unit which does all the calculations in computer

如果中央处理器包含在 one integrated circuit（硅芯片）中，则被称为微处理器。

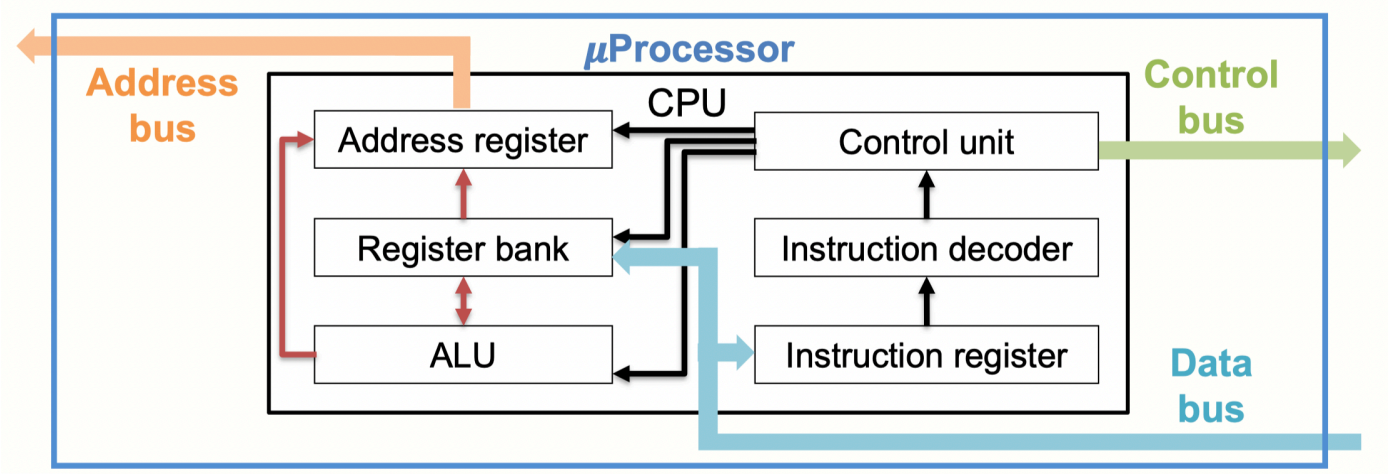
In 1971，Intel 4004 is the first microprocessor

15 November 1971 2,300 transistors 10 μm gate length 4 bit address bus 100 kHz clock

μProcessor vs μController 微处理器与微控制器

Micro-processor:

微处理器是一个通用的中央处理单元（CPU），被设计用来执行广泛的任务。微处理器通常至少有一个支持指令获取、解码和执行的处理器内核，以及各种其他功能，如高速缓冲存储器、浮点单元和多核配置。然而，微处理器需要外部组件的额外支持，如内存、输入/输出（I/O）设备和其他外设，才能正常运行。微处理器通常用于台式电脑和笔记本电脑、服务器以及其他需要高计算能力和灵活性的应用。

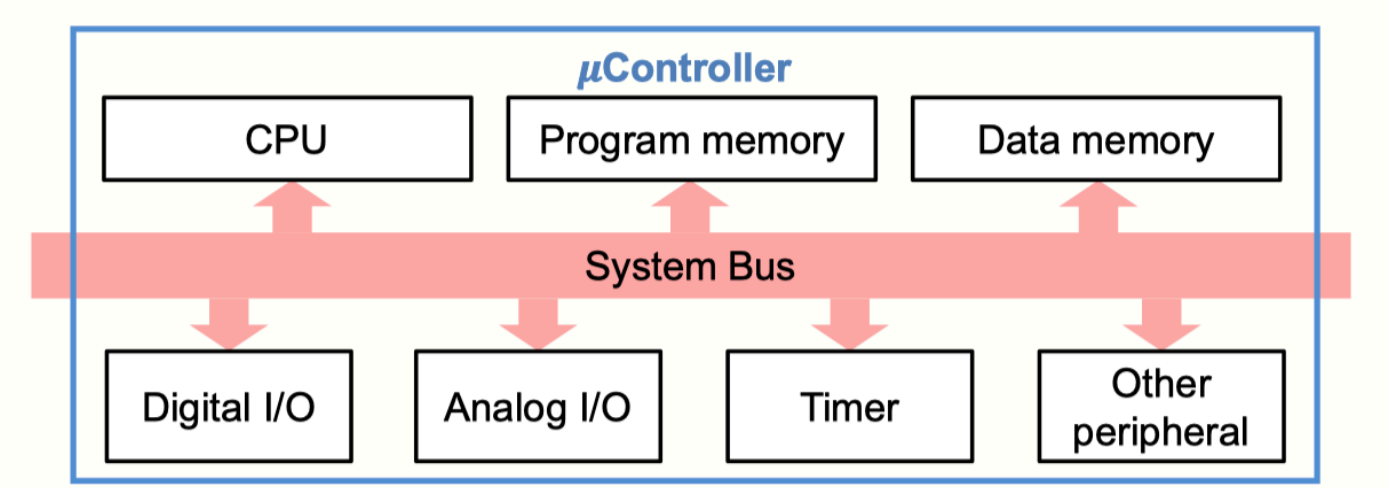


- Address bus（地址总线）：一种计算机系统中的物理总线，用于传输CPU或其他设备发出的内存地址信号。
- CPU（中央处理器）：计算机系统中的中央处理器，执行指令和处理数据。
- Control bus（控制总线）：一种计算机系统中的物理总线，用于传输控制信号，例如时钟脉冲和各种状态信号。
- Address register（地址寄存器）：用于存储CPU指令或数据要访问的内存地址的寄存器。
- Register bank（寄存器组）：计算机系统中的一个组，其中包含多个寄存器，用于存储指令和数据，以及在计算和控制中使用。
- ALU（算术逻辑单元）：CPU中的组成部分，用于执行算术和逻辑运算，例如加减乘除、位移、逻辑运算等。
- Control unit（控制器）：CPU中的组成部分，用于控制CPU的操作，例如从内存中读取指令、解码指令、执行指令等。
- Instruction decoder（指令解码器）：CPU中的组成部分，用于解码指令并确定指令应执行的操作。

- Instruction register（指令寄存器）：用于存储当前正在执行的指令的寄存器。
- Data bus（数据总线）：计算机系统中的一种物理总线，用于传输指令和数据。

Micro-controller:

微控制器是一种集成电路，通常包括一个单一的CPU核心以及内存块、数字输入/输出（I/O）、模拟输入/输出（I/O）和其他基本外设，都在一个芯片上。微控制器通常用于嵌入式应用，其主要任务是控制系统的某些方面，如管理冰箱的温度，控制电机的速度，或监测和调节设备的电源。由于微控制器高度集成并针对特定任务进行了优化，它们通常比通用微处理器更具成本效益，体积更小，更容易设计到系统中。



- System Bus（系统总线）：用于传输指令和数据的物理总线，连接CPU、Program memory、Data memory和各种外设。
- CPU（中央处理器）：计算机系统中的中央处理器，执行指令和处理数据。
- Program memory（程序存储器）：用于存储程序指令的存储器，程序指令被存储在Program memory中，并在CPU执行程序时逐条被读取。
- Data memory（数据存储器）：用于存储程序数据的存储器，数据被存储在Data memory中，在程序执行过程中被CPU读取、处理和更新。
- Digital I/O（数字输入/输出）：一种基本的输入/输出外设，用于读取或控制数字信号，例如开关、传感器、LED等。
- Analog I/O（模拟输入/输出）：一种用于读取或控制模拟信号的外设，例如温度、压力、光线等。
- Timer（定时器）：一种基本的计时器外设，用于计时或生成周期性信号。
-
- Other peripheral（其他外设）：Microcontroller还包括一些其他的外设，如串口、SPI总线、I2C总线等，用于与其他设备进行通信或控制。

二进制 binary hexadecimal

我觉得主要关注形式和格式

$$0011110001000001010100100100110 1_2 \tag{1}$$

$$\mathbf{b}\ 00111100010000010101001001001101 \tag{2}$$

这都表示了二进制的数，

定义：

- 使用2进制=coded in binary=base 2
- in decimal=base 10
- use hexadecimal= base 16

背景：All computers work on information and data coded in binary, that is base 2.所以使用数字0或者1，就算1位。

ARM 微处理器是一种 32 位处理器，它使用 32 位**二进制数字**表示数据和指令。

因为32位二进制数字非常长，我们一般使用十六进制或基数16。

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

base16 形式和格式 hexadecimal number

$$01001110000001100010000111100000_2 \tag{3}$$

$$4E0621E0_{16} \tag{4}$$

$$\mathbf{0x}\ 4E0621E0 \tag{5}$$

0x denotes a hexadecimal number

16进制的加减乘除

2B_16 X 5_16=

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

155÷=9(144).....11

=B9_16

$$2B_{16} * 5_{16} = \tag{6}$$

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。

除法

ASCII

有对应关系查表就好。

Lec17

计算机作为处理器Computer as a processor

计算机与人类 Computer versus human

指令Instructions

通用处理器

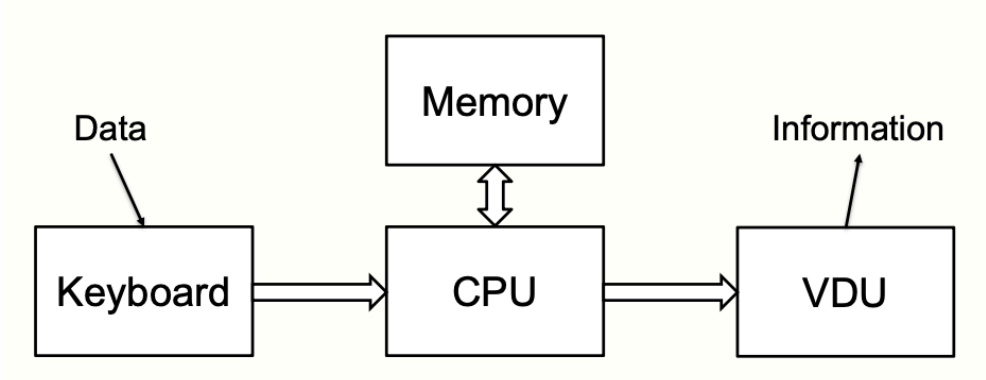
A processor is a device that runs a set of instructions (program)

- **Reads input data**
  - Keyboard, mouse, camera, microphone
- **Processes data**
  - Registers, Control Unit, ALU
- **Stores data**
  - Memory, hard-drive
- **Send output data to different peripherals**
  - Screen, speakers

简单计算机

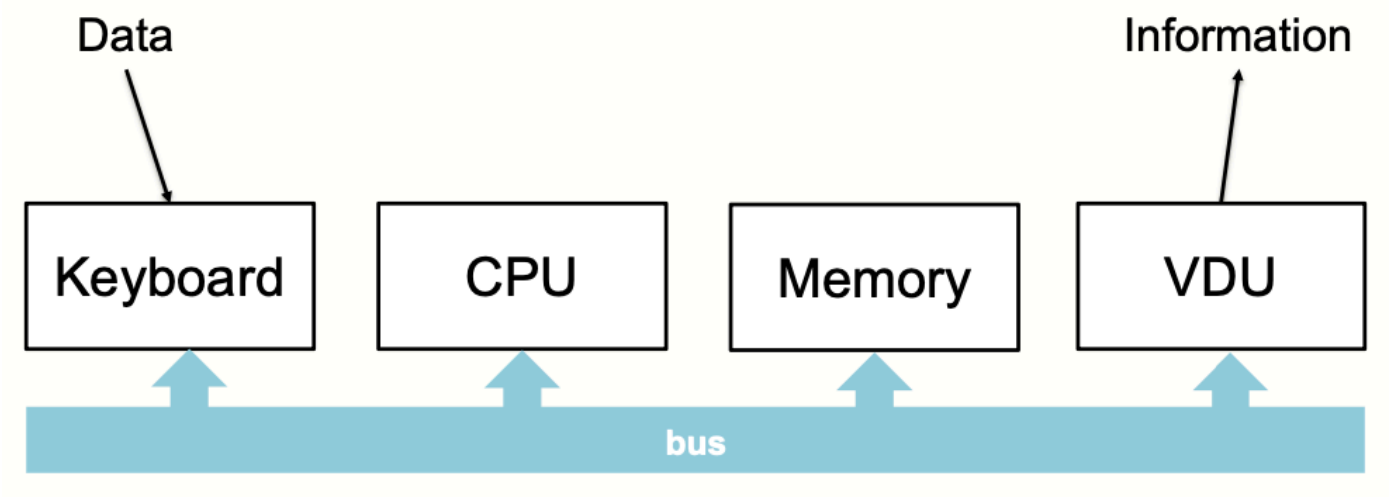
- 简单计算机需要接收数据、处理数据并返回信息。
- 另外，计算机必须能够存储指令。
- 键盘可将数据输入计算机。
- CPU（中央处理器）可以处理数据。
- VDU（视觉显示单元）可以将信息返回给用户。
- 计算机内存可以存储指令。

简单计算机架构



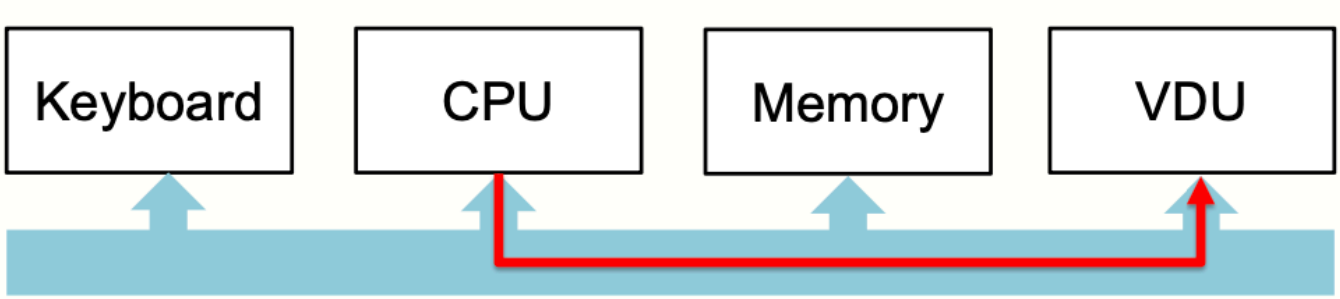
The Bus Architecture

总线架构可以扩展到任意数量的设备。所有设备都只有一个连接到“总线”Bus Architecture。



总线是一组electrical connections **- 通常为8、16、32或64根独立导线**。
32位数据和信息可以同时沿32位总线传输。
例如，4个ASCII字符的编码可以从CPU发送到VDU。

总线是计算机硬件组件之间传输数据和信息的通道。当我们说总线通常包括8、16、32或64根独立导线时，我们是指这些导线是实际的物理连接，它们在计算机内部连接各个组件，如CPU、内存和外部设备。



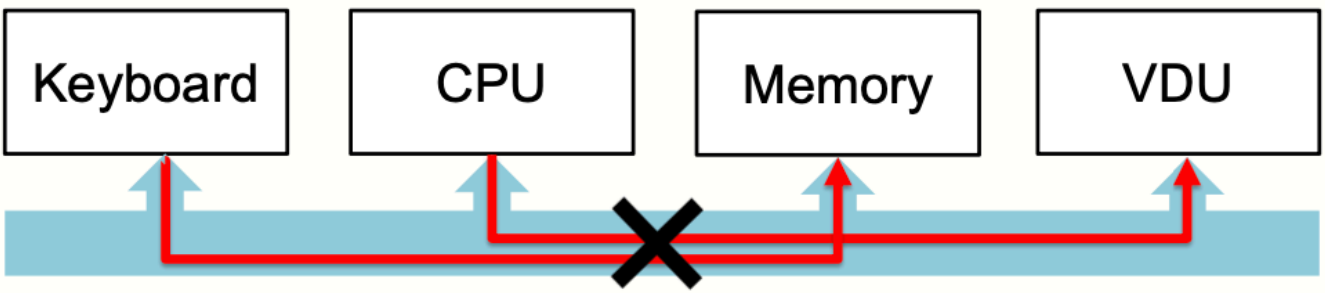
根据总线上的导线数量，可以同时传输不同数量的数据位。例如：

- 8位总线：一次可以传输8位数据（1字节）。
- 16位总线：一次可以传输16位数据（2字节）。
- 32位总线：一次可以传输32位数据（4字节）。
- 64位总线：一次可以传输64位数据（8字节）。

总线上的导线数量决定了总线的宽度，进而影响计算机的性能。更宽的总线意味着可以在更短的时间内传输更多的数据，从而提高计算机的处理能力。但是，增加总线宽度也会增加硬件复杂性和成本。因此，设计者需要在性能和成本之间找到平衡

控制总线 Controlling the Bus

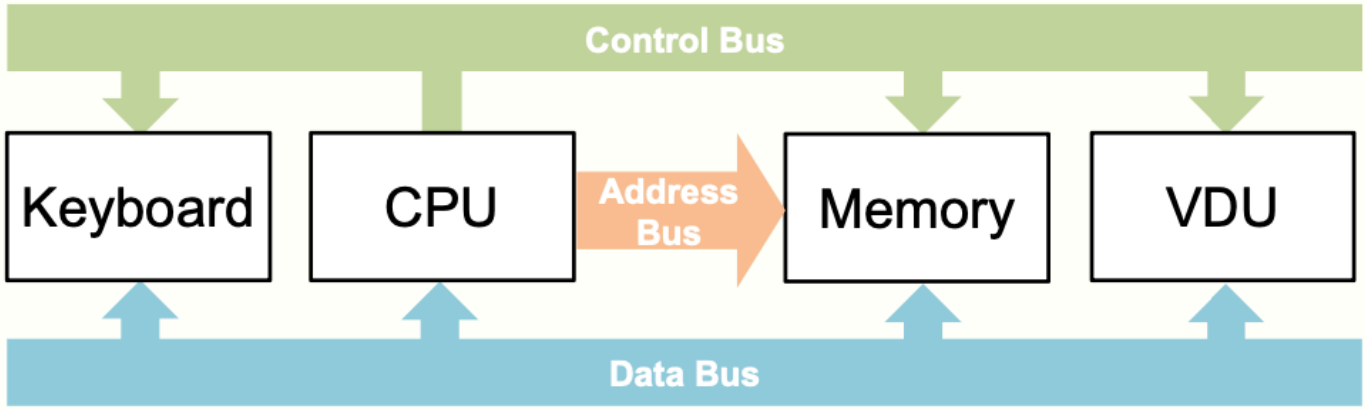
一次只能有一个设备发送数据。



CPU使用特殊导线激活设备并同步发送和接收设备以控制总线上的所有动作。special wires to activate devices and to synchronise the sending and receiving devices.这些特殊连接称为**control bus**，它们与传输数据的总线完全独立。
为了避免混淆，这被称为**data bus**。

第三条总线

除数据总线和控制总线外，还有第三条称为地址总线的总线。地址总线由CPU用于确定在内存中发送或接收数据的位置。



内存Memory

计算机内存（Memory）用于临时存储数据和指令，以便CPU在执行程序时可以快速访问它们。计算机内存有两种主要类型：随机访问存储器（RAM）和只读存储器（ROM）。

内存中存储的内容包括：

- 指令：这些是CPU执行的低级操作，用machine code表示。指令通常包括操作码（表示操作类型）和操作数（表示操作涉及的数据）。
- 数据：这包括程序需要处理的数字和字符。数据以二进制形式存储，字符通常使用ASCII或Unicode编码。

计算机内存的基本组成部分是大量的逻辑门，例如**D触发器（D type flip flop）**它们可以存储一个比特（位）的数据，即0或1。这些逻辑门通常组织成内存单元或存储单元，每个单元包含一定数量的位（通常为8位，即1字节）。每个存储单元都有一个唯一的内存地址，以便CPU可以精确访问所需的数据和指令。

内存中的数据按地址顺序排列，地址范围取决于计算机的地址总线宽度。例如，32位地址总线可以访问2^32（约43亿）个内存地址。

内存组织

ARM Cortex M0

- 在每个内存位置有8 bits of data – 8 bits is known as a byte.
- ARM是一个32位的处理器，地址长度为32位，从0x00000000到0xFFFFFFFF。
- 这意味着可以有多达4,294,967,296（或2^32）个不同的内存位置，都有一个唯一的内存地址。
- 并不是所有的地址都用于内存。

Memory storage capacity

1 byte = 8bits

1024 bytes = 1 kibibyte

1024 kibibytes = 1 mebibyte (MiB)

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value
kilobyte	kB	10^3	kibibyte	KiB	2^10
megabyte	MB	10^6	mebibyte	MiB	2^20
gigabyte	GB	10^9	gibibyte	GiB	2^30
terabyte	TB	10^12	tebibyte	TiB	2^40
petabyte	PB	10^15	pebibyte	PiB	2^50
exabyte	EB	10^18	exbibyte	EiB	2^60
zettabyte	ZB	10^21	zebibyte	ZiB	2^70
yottabyte	YB	10^24	yobibyte	YiB	2^80

这里十进制和二进制要理解一下

definitions for word

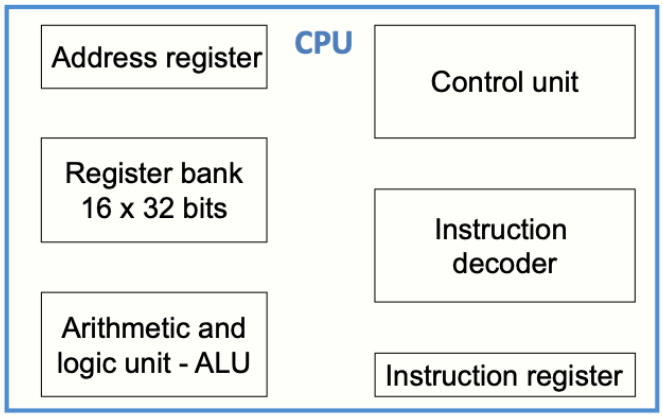
- word”（字）是一个常用的术语，表示由处理器使用的一个特定的位数。对于一个32位的处理器，例如ARM处理器，一个字就等于32 bits或4 bytes。
- 同样，一个“half word”（半字）就是16 bits 或2 bytes。
- 另一种说法是“字长”是32，表示这个处理器的字的位数是32 bits。这意味着每次处理器从内存中读取数据，它将读取32 bits或4 bytes的数据。

Lec 18

CPU The ARM Cortex M0 core

CPU的高效性，其可以被分成多个块（block）

We will concentrate on the **ARM Cortex M0** ‘core’ – the CPU for the Cortex M0 range of microprocessors.



下面分别介绍ARM Cortex M0的各个block的功能和

1 Instruction register

指令存储在内存中

数据总线（data bus）传输到CPU，然后被加载到指令寄存器（Instruction Register，IR）中

指令通常是16 bit，但有些指令可能是32 bit。它们被存储在 instruction register - not part of the main memory.

2 Instruction decoder and control unit

也是接受指令的block

但是这些指令是 "machine code"

Instruction decoder 和 control unit决定CPU的其他部分做什么。

control unit也负责控制总线（bus）。

interpreting each instruction的过程被称为 "**decode** "**cycle (ID)**.

3 Arithmetic and logic unit

The arithmetic and logic unit or ALU 按要求执行数学功能。

Logical：AND, OR, XOR etc. Arithmetic：Addition, subtraction, multiplication

performing each instruction 的过程被称为 **'execution' cycle (IE)**.

4 Register bank

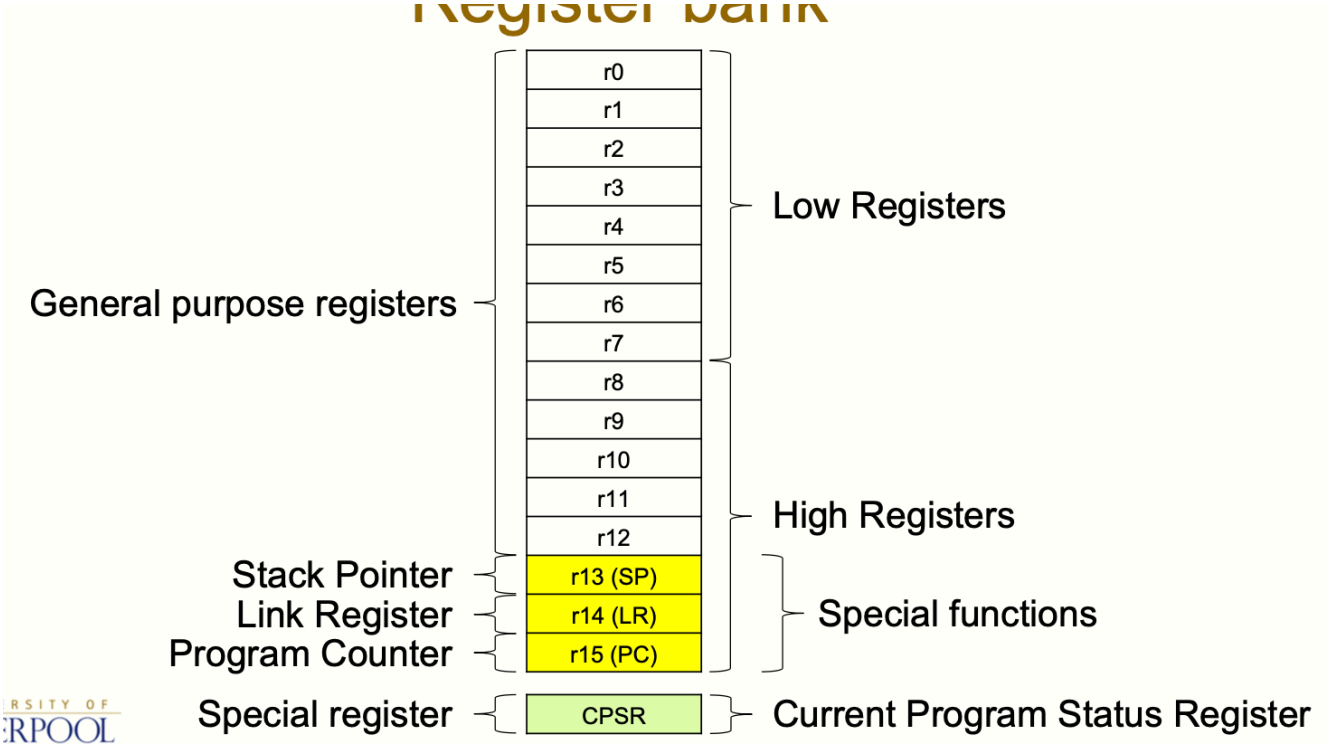
Register bank是CPU的一个 local memory

- It has **16 location s** – each location can hold 32 bits of data

$$16 \times 32 \text{ bits} \tag{7}$$

- The registers are named **r0, r1, r2, r3,** ... etc. up to **r15**.
- 它们被用来保存由ALU(Arithmetic & Logic Unit)处理的数据，同时也保存任何计算的结果。

- 寄存器r13、r14和r15具有特殊功能



5 Address register

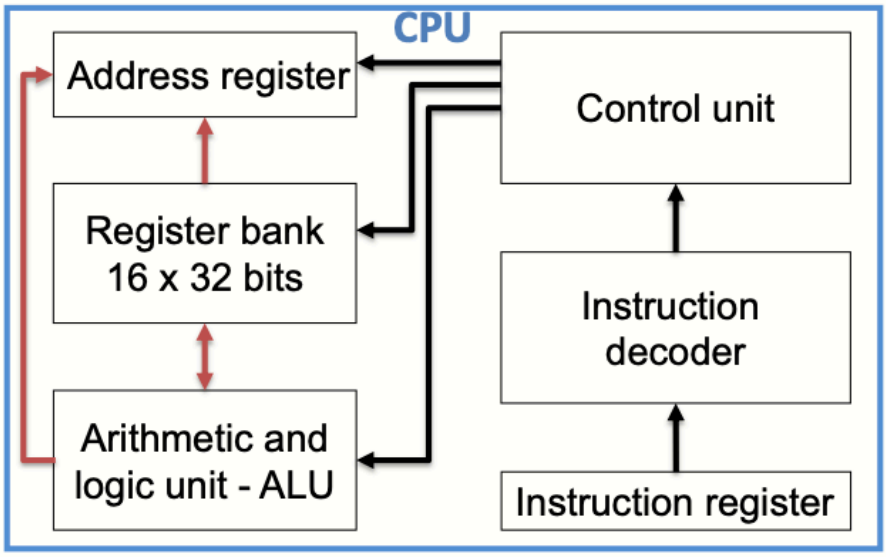
- 地址寄存器是一种**32 bit memory device**，用于存储内存地址的值。在CPU执行不同的操作时，地址寄存器中的值可能代表不同的含义。
- 在取指令（Fetch）周期（IF）中，地址寄存器中存储的地址通常是下一条指令的内存地址，因为CPU需要从内存中读取下一条指令以执行程序。

instruction during the ‘fetch’ cycle (IF).

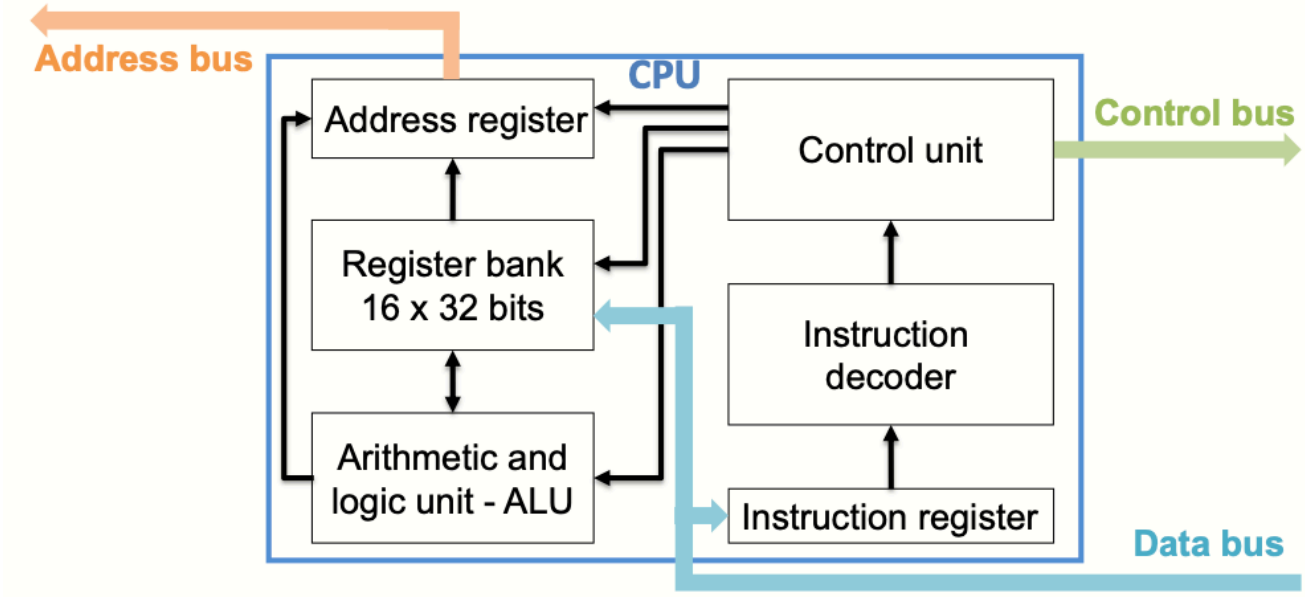
- Or during the ‘execute’ cycle the address is for a memory location either containing data to be loaded into a register or where data from a register is to be stored.

?? ??? ???

6 Internal connections & External connections



- The instruction register is connected to the instruction decoder.
- The instruction decoder is connected to the control unit.
- The control unit is connected to the ALU, the register bank and the address register.
- The ALU and the register bank are connected to each other and the address register.



- The data bus is connected to both the instruction register and the register bank.
- The control bus is connected to the control unit.
- Address bus is connected to the address register.

7 Program counter

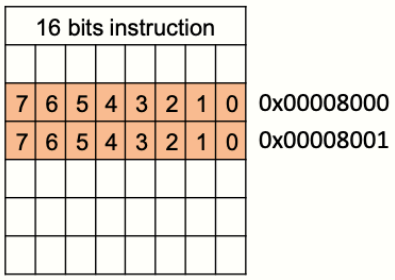
R15的特殊作用：寄存器r15总是保存着下一条指令的内存地址。

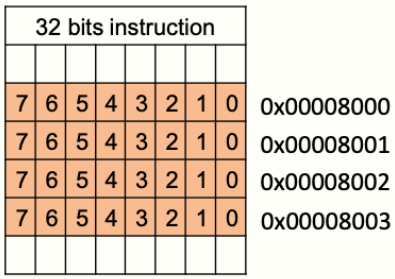
8 Instruction stored in memory

指令的长度为16位（半字）或32位（字）。

- 内存位置的长度为8位，所以一条指令在内存中占据了两个或四个位置

- 例如，一条16位的指令将被存储在地址为0x00008000和0x00008001的两个内存位置。
- 下面的16位指令将被存储在地址0x00008002和0x00008003





一般来说，指令在内存中的位置是连续的

- 假设指令的长度为16位
- 如果正在执行的指令被存储在地址0x00008000，那么下一条要执行的指令将被存储在地址0x00008002，下一条在0x00008004，下一条在0x00008006
- 除非...

除非在执行 "branch "指令时，计算机程序 "branch "到memory的另一部分，the program counter持有一个全新的memory address.

这段描述主要涉及到计算机程序在执行分支指令（branch instruction）时的行为。分支指令用于改变程序执行流程，例如根据条件跳转到其他内存地址的指令。

程序计数器（Program Counter，PC）是一个特殊的寄存器，它用于存储正在执行的指令在内存中的地址。通常情况下，程序计数器会顺序递增，使得计算机程序按顺序执行内存中的指令。但当遇到分支指令时，程序计数器的行为会有所不同。

当执行分支指令时，程序计数器将持有一个全新的内存地址，这个地址通常不是顺序递增的。这意味着程序会“跳转”到内存的另一部分执行指令。这种跳转可以是无条件的，也可以基于某些条件（例如比较两个数值的大小）进行。分支指令在循环、条件语句（如`if-else`）和函数调用等程序结构中都有广泛应用。

因此，上述描述表明，在执行分支指令时，程序计数器将获取一个新的内存地址，从而使程序跳转到内存的另一部分继续执行指令。这有助于实现程序的非线性执行流程。

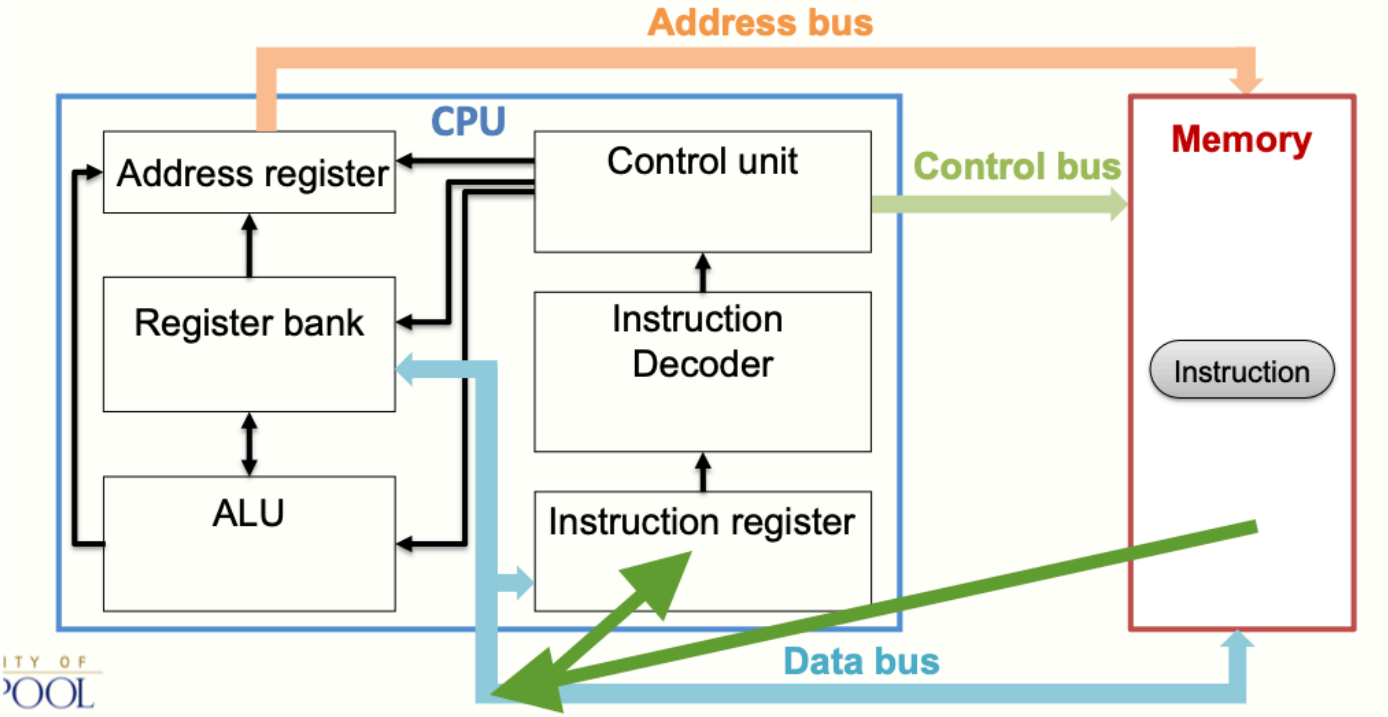
Fetch, decode, execute

• The CPU performs three cycles sequentially.

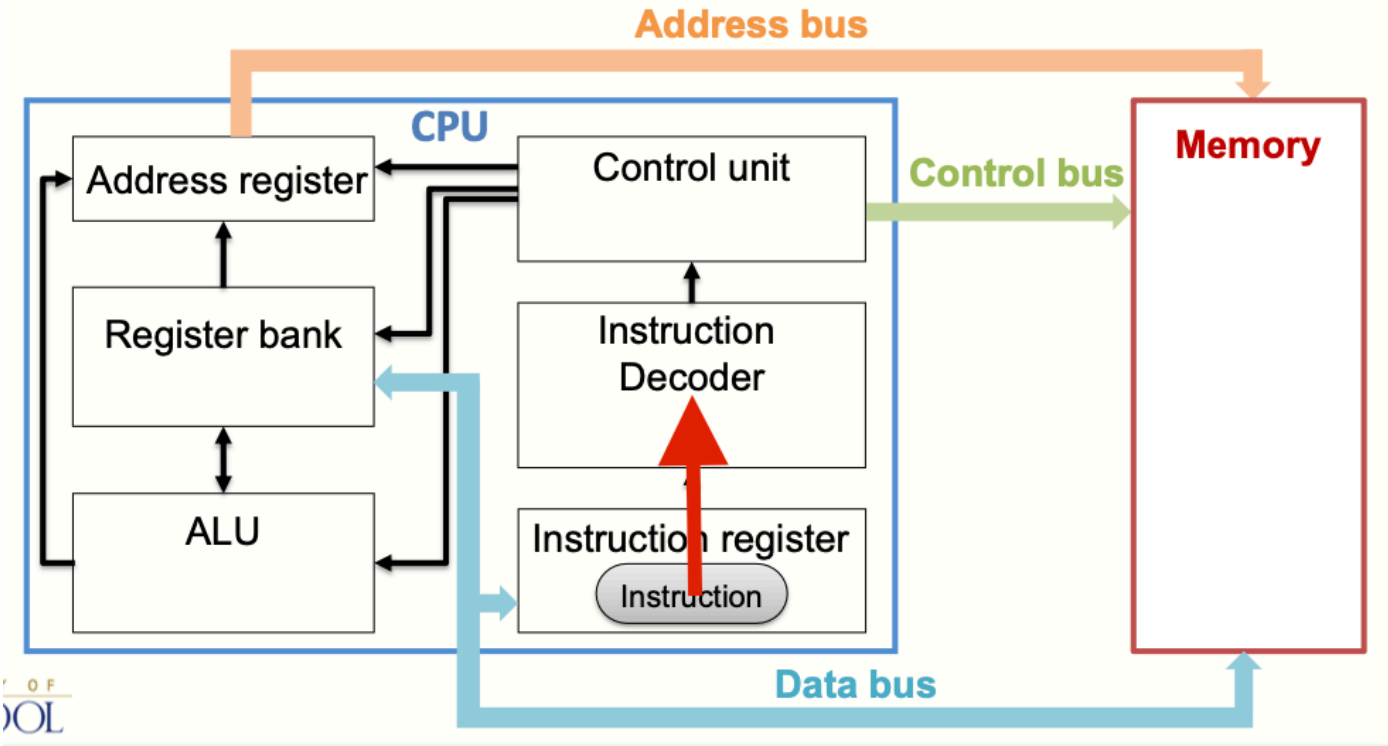
Fetch cycle

取指（Fetch）： 在这个阶段，处理器从程序计数器（Program Counter，PC）指向的内存地址读取instruction。程序计数器（Program Counter，R15）存储了下一条待执行指令在内存中的地址。指令被取出后，程序计数器通常会递增，以指向下一条指令的内存地址。取指阶段结束后，指令被存储在指令寄存器（Instruction Register）中。

During the **FETCH** cycle an instruction in memory is loaded into the instruction register.

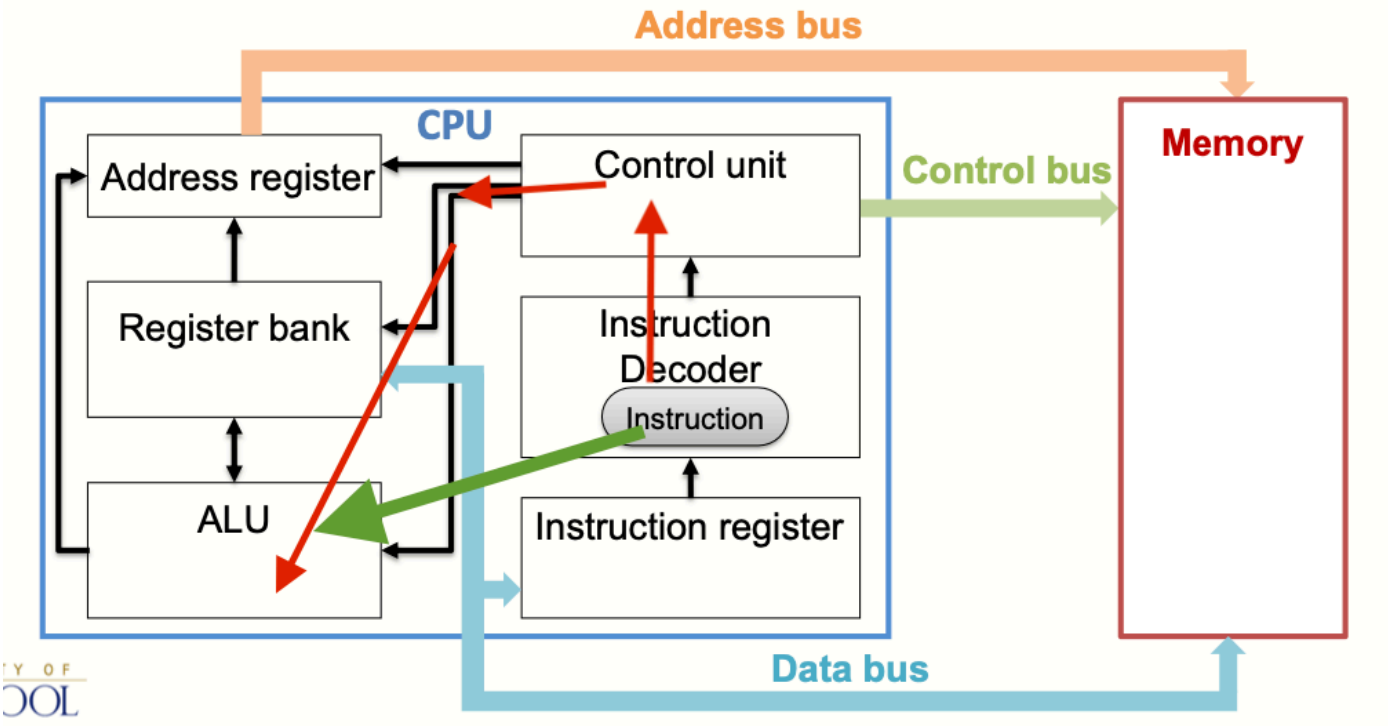


Decode cycle

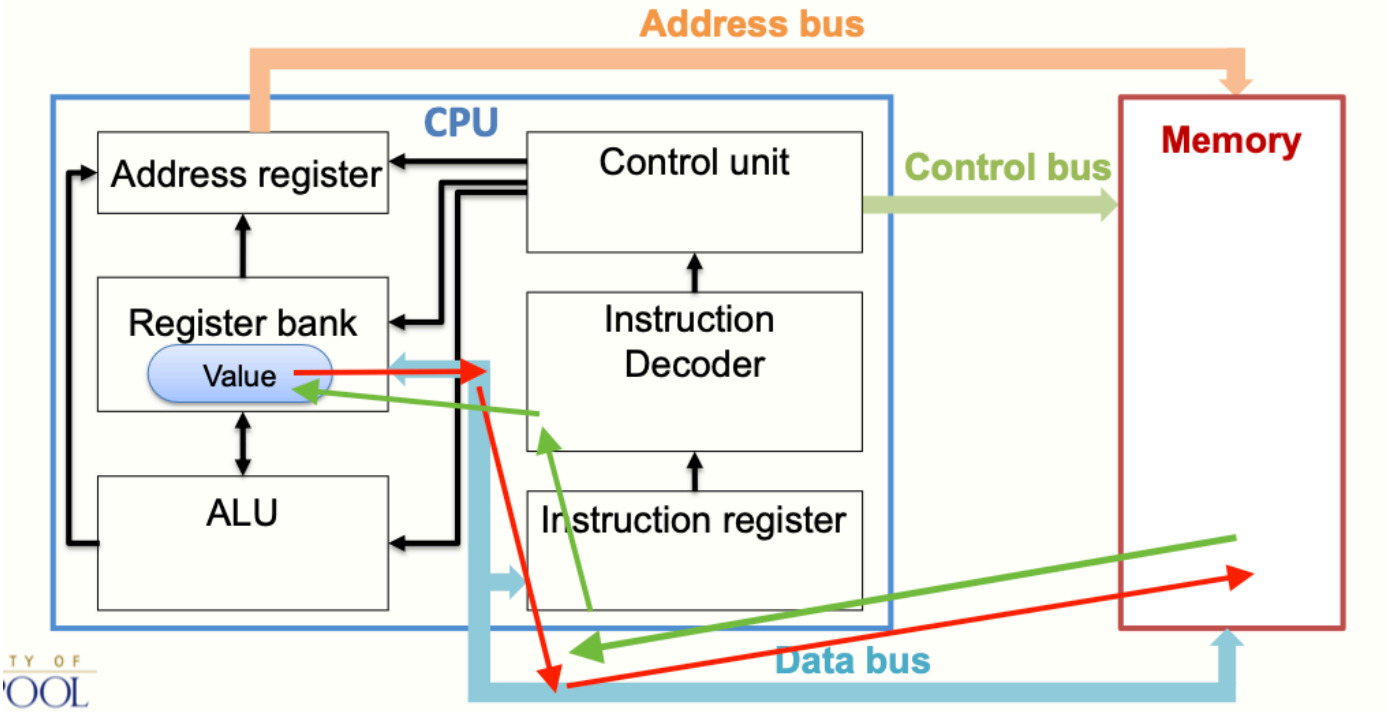


During the **DECODE** cycle the instruction is interpreted by the instruction decoder.

Execute cycle



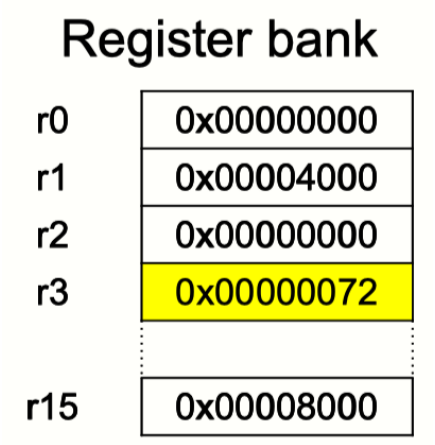
During the **EXECUTE** cycle either the ALU performs a calculation on values held in registers,



or a value in memory is loaded into a register.

Lec19

感觉主要介绍了两个指令给出了Simple instruction，Mnemonics，Assembly language，Assembler，Compiler，Type of instructions，Arithmetic instructions的概念。



比如说我们要在register bank的r3中放入value-**0x72**，注意我们需要使用指令放入。

MOVS1

machine code

$$\mathbf{0x}\ 2372 \tag{8}$$

or

$$0010001101110010_2 \tag{9}$$

在执行指令后，

1. 在r3位置上会保持数值 **0x**00000072（72是十六进制的114_{10}），#114
2. In register, r15中的值, the program counter, 将增加2。

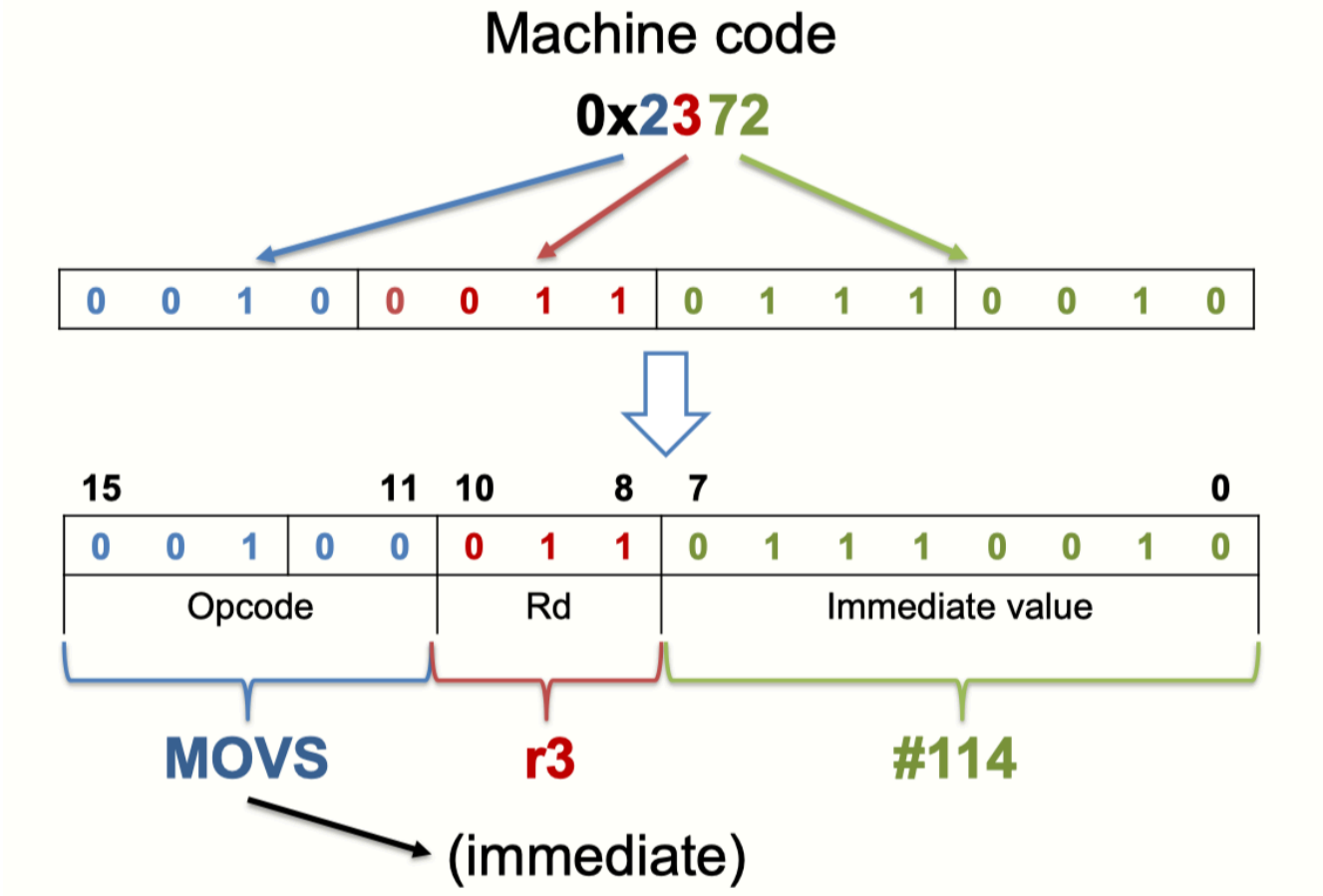
我们将详细解释machine code for **0x**2372

指令：0x2372（0010 0011 0111 0010）

1. 操作码（Opcode）：0x23（0010 0011） 操作码是指令的一部分，用于指示处理器要执行的操作。在这个例子中，操作码是0x23（0010 0011），我们假设这个操作码代表将一个立即数（即，操作数）加载到寄存器中的指令（如，MOVS1）。
2. 寄存器编号：0x3（0011） 在这个指令中，我们需要表示目标寄存器。在这里，我们假设指令的下一部分表示目标寄存器。0x3（0011）表示目标寄存器是寄存器组（register bank）中的r3。
3. 操作数（Operand）：0x72（0111 0010） 操作数是指令的另一部分，提供指令所需的数据。在这个例子中，操作数是**0x**72（0111 0010），即要加载到目标寄存器的值。

综合上述解释，我们得到以下指令：

- 操作：MOVS1
- 目标寄存器：r3
- 操作数：**0x**72



MOVS2

machine code

$$\mathbf{0x}002E \tag{10}$$

or

$$0000000000101110_2 \tag{11}$$

指令执行后，

1. 寄存器r6的值与寄存器r5的值相同。
2. 寄存器r15中的值，即程序计数器，将增加2

我们将详细解释machine code的结构 for **0x**002E

根据您提供的信息，我们可以得知指令格式如下：

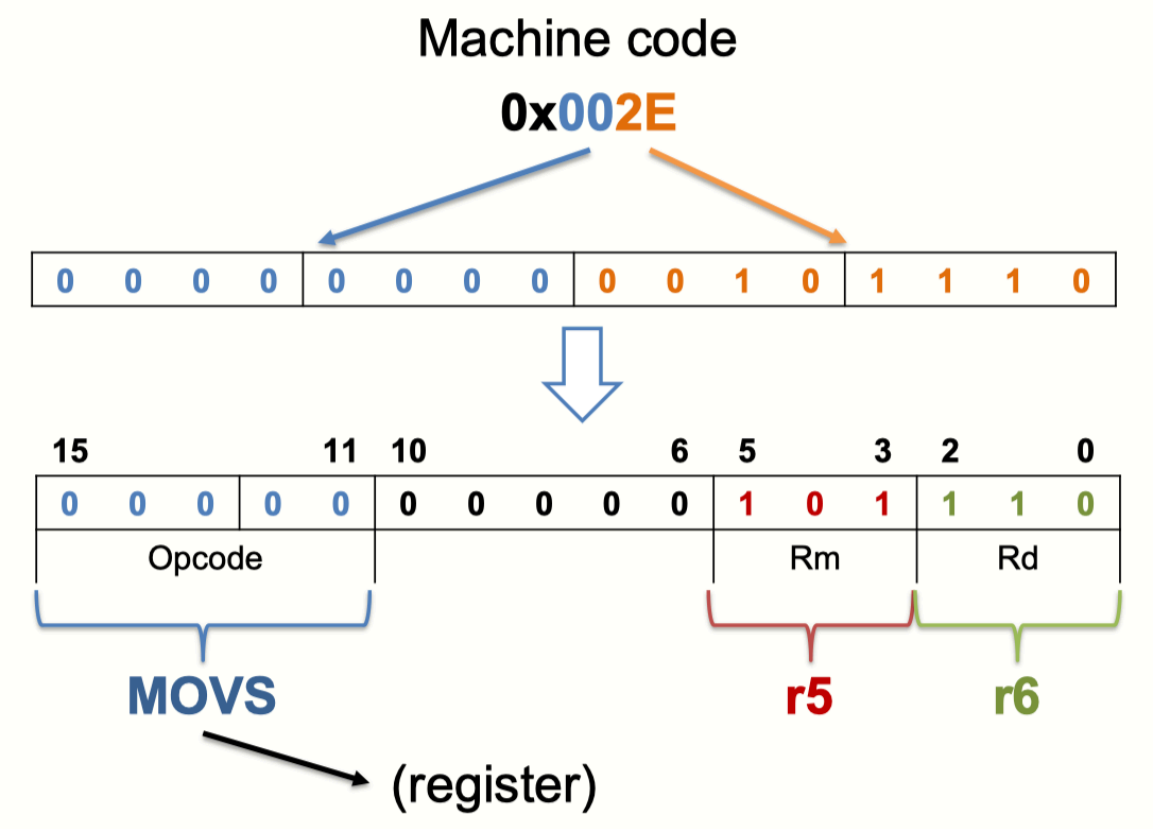
指令：0x002E（0000 0000 0010 1110）

1. 操作码（Opcode）：操作码占据了前5位，即0x00（0000 0000）。我们需要知道0x00对应的操作。在这个例子中，我们假设0x00表示将一个寄存器的内容复制到另一个寄存器中的指令。
2. 寄存器r6：最后3位表示寄存器r6。在这个例子中，这3位为**0x**6（110）
3. 寄存器r5：给定的指令格式为0x002E（*0000 0000* 00**10 1**110），根据ppt的信息，我们知道寄存器r6后的3位表示寄存器r5。在这个例子中，这3位为**0x**5（101）。

现在我们已经知道源寄存器和目标寄存器，我们可以重新解释这个机器码指令的结构和具体含义。

- 操作：MOVS2
- 源寄存器：r5（101）
- 目标寄存器：r6（110）

因此，这个指令的含义是：将寄存器r5的内容复制到寄存器r6中。机器码0x002E（0000 0000 0010 1110）表示了这个操作，处理器会识别并执行这个指令。



Mnemonics

一般来说，没有人记得任何特定处理器的所有机器代码（或实际上是任何）。

相反，我们使用助记符：每条指令都有一个机器码和一个助记词。

Machine Code	Mnemonic
0x2372	MOVS r3 #144
0x21CB	MOVS r1 # **0x**CB
0x002E	MOVS r6, r5
0x0038	MOVS r0, r7

Assembly Language(ASM) 汇编

汇编（ASM）是一种low level的编程语言，它针对architecture (processor)

顺序执行每条instruction，这就是一个汇编语言程序

E.g.

```
MOVS r6, r5
MOVS r1, #0xCB
MOVS r0, r7
MOVS r3, #114
```

Assembler

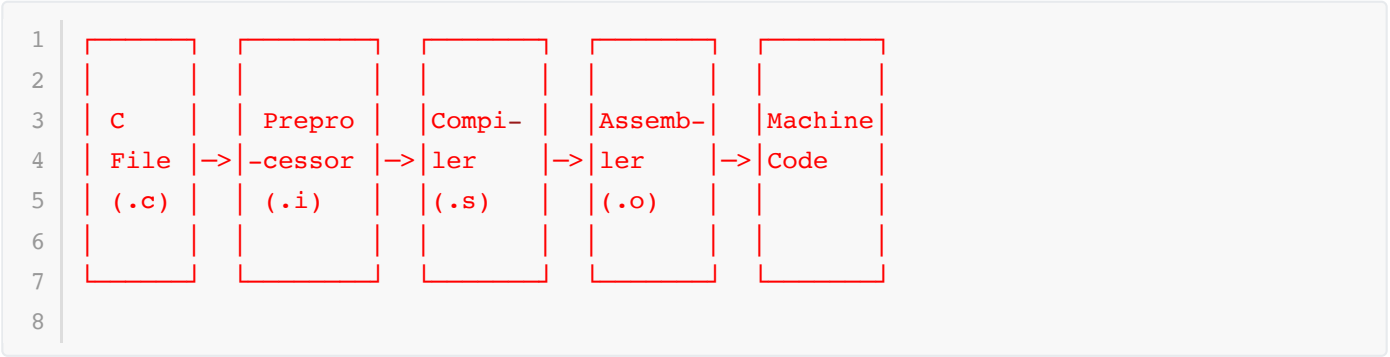
一个称为assembler的computer package将assembly language program转换为machine code program

```
   Mnemonic       >>  Machine Code
 MOVS r3, #114    >>    0x2372
 MOVS r1, #0xCB   >>    0x21CB
 MOVS r6, r5      >>    0x002E
 MOVS r0, r7      >>    0x0038
```

The machine code可以下载到microprocessor memory中；每条instruction占用2个相邻的memory locations

Compiler

一种high level language（如Java、C、C++、Fortran、Pascal等）通过一个称为compiler(编译器)的computer package被转换成mnemonics(助记符)。



C file

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}

```

Assembly code (.s)

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello, world!"
        .text
        .globl  main
        .type   main, @function
main:
        push    {lr}
        ldr     r1, .L3
        mov     r0, r1
        bl      puts
        mov     r0, #0
        pop     {pc}
.L4:
        .align  2
.L3:
        .word   .LC0
        .size   main, .-main
        .ident  "GCC: (GNU) 4.6.3"

```

Machine code

```
00000000: B5F0 4801 4800 4800 BF00 4698 4802 6800  ..H.H.H...F.H.h.
00000010: F7FF FFF8 BD00 023C 696E 633A 2025 640A  .......<inc: %d.
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................

```

Importance of Assembly

- 直接进行硬件操作
- Access to specialised processor instructions and registers
- Speed optimisations
- Use in real-time systems, device drivers, low-level embedded systems

Instructions for Arithmetic

Basic: ARM Cortex M0可以做加减法和乘法（但不能做除法）。

Add

将register **x**中的值与register **y**中的值相加，并将其放在寄存器z中的记忆法是：

```
ADDS rz, ry, rx → rz = ry + rx
```

E.g. register **r1**中的数值与register **r2**中的数值相加，并将总和留在寄存器r3中，其记号是

```
ADDS r3, r2, r1
```

我们不需要知道机器代码，汇编器会为我们生成它。

Subtraction

同样，从ry的值中减去rx的值，并将差值放在rz中的记忆法是：

```
SUBS rz, ry, rx → rz = ry - rx
```

***** 请注意，顺序是很重要的，如果rx中的值大于ry中的值，那么一个负数的结果将被放在rz中。负数是以two's complement format 表示的。

Reverse Subtraction

Code :

```
RSBS rz, ry, #0 → rz = 0 - ry
```

0减去ry的值，并将差值放入rz。感觉其实就是反了一下。

Multiplication

rx中的数值与ry中的数值相乘，并将乘积放在rx中

```
MULS rx, ry, rx → rx = ry × rx
```

*****如果结果超过32bits，目的寄存器rx只保留结果the bottom 32bits，其余部分将丢失。

Integer overflow

当一个算术运算的结果超过了容纳它的数据类型的范围，就会发生整数溢出。

例如，在32位系统中，如果一个运算产生的结果大于2^{31-1}或小于-2^{31}，就会发生溢出。

```
实际上是大于等于 2^31，因为在32位有符号整数的表示中，最高位（即最左边的位）是符号位，它用来表示整数的正负性。如果最高位为0，那么这个整数就是正数；如果最高位为1，那么这个整数就是负数。因此，在32位有符号整数的范围中，最大的正数是2^31-1，最小的负数是-2^31。当一个操作的结果超过这个范围时，就会发生整数溢出。
```

例子：

Two 32-bit signed integers, a = 2147483647 and b = 2. If we add them together, the result will be

$$a + b = 2147483647 + 2 = 2147483649 \tag{12}$$

However, 2147483649 is greater than the maximum value of a 32-bit signed integer (2^31-1), so an overflow will occur. Therefore, the actual value stored in the target register will be

$$2147483649 - 2^{32} = -2147483647 \tag{13}$$

This is an example of a signed integer overflow. Note that in the two's complement representation, the value -2147483647 is represented by the bit pattern 0x80000001, which has a leading 1 in the most significant bit.

Lec 20

the ARM CPU can also do **logic** such as AND and OR

mnemonic for **AND** :

```
1                       ANDS ry,rx → ry=ry.rx
```

rx with value in ry and leave the result in ry

mnemonic for **OR** :

```
1                       ORRS ry,rx → ry=ry + rx
```

rx with value in ry and leave the result in ry

Logical AND。

Logical OR

Exclusive OR (XOR)

Bit clear

function

```
1 ry AND NOT(rx)
```

mnemonic instruction

```
1 BICS ry, rx
```

Mean A logic 1 in rx ‘clears’ a 1 bit in ry.

1. rx 寄存器中的逻辑 1 将清除（设置为 0）“ry”寄存器中的相应位。
2. rx 寄存器中存在逻辑0，'ry'寄存器中的相应位保持不变。

Example

```
    1111 1110 1101 1100 1011 1010 1001 1000 (ry)
BIC 0001 0001 0010 0010 0011 0011 0100 0100 (rx)
-----------------------------------------
=   1110 1110 1101 1100 1000 1000 1001 1000
```

ry: 给出的原始数据，需要被**rx**处理重新得到**ry**的值

rx: **rx**是处理的规则，**rx**是0则通过**ry**的值，**rx**是1则 clear成0

Addressing modes

The ARM Cortex M0 microprocessor supports many different addressing modes 我们将集中讨论四个模式：

• Register addressing
• Immediate addressing
• Indirect addressing
• Base plus offset addressing

Register Addressing

说明machine code 存在 a number (or numbers) 识别 a register (or registers).

比如在之前提到的 MOVS r6, r5(0000 0000 0010 1110) ,在这个机器码中，最后位的101和110分别表示r5和r6的位置。

在下面的代码中，我们都使用了 register addressing

```
SUBS r5, r1, r2
MULS r3, r7, r3
BICS r1, r6
```

Restricted Register Use

并非所有指令都可以访问所有寄存器。带有

- 助记符 MOVS 的指令，例如**MOVS ry, rx** 仅限于“低”寄存器，即 r0 到 r7 范围内的寄存器。
- MOV ，例如**MOV ry, rx** 允许使用所有 16 个寄存器（包括“低”寄存器），例如 MOV r13, r10 有机器码 0x46D5 或 0100 0110 1101 0101，
- MOV r13, r10 has machine code 0x46D5 or 0100 0110 11010101

这里要指出什么是 ‘low’ registers,，即register **r0** 到register **r7** 。

Q about how to definite r5，r6，r10，r13，the rule of it

Immediate Addressing

表示机器代码包含要使用的值。

MOVS r1, #0xCB

0x21CB which is the code for **MOVS r1, #0xCB**

Move into r1 the value **0x**000000CB - the # means 'immediate'.

该指令使用 r1 的Register Addressing和 0xCB (=20310) 的Immediate Addressing

ADDS r2 , r1, #5

mean: adding 5 for Register r1,r2 is at the front of the r1, so the sum will put in r2(我想知道在运行完这个代码后，r1和r2的寄存器的值是不是一样的)

Machine code: 0x1D4A or 0001 1101 01001010$_2$

001 for r1，010 for r2

SUBS r6, r6, #99

mean: Subtract 99_{10} from value in r6 and put the difference in r6

Machine code: 0x3E63 or 0011 1110 0110 0011$_2$

$$99_{10} = 0x63 \tag{14}$$

$$99 = 6 * 16\hat{}1 + 3 * 16\hat{}0 \tag{15}$$

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

```
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

```
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。

```
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。

From register to memory

这个概念是指将数据从寄存器传输到内存位置的过程。

In this case, the 'store' instruction is used, which can be thought of as being the opposite of the 'load' instruction.

instruction：

```
STR rd, [rn]
```

解释：

1. 'rd'是要存储value的寄存器，这个value将被写入rn中给出的内存地址
2. '[rn]'表示我们正在使用寄存器'rn'中包含的值作为内存地址。rn中的值是一个内存地址
3. 该指令意味着我们将寄存器“rd”中的数据存储到内存位置，该内存位置的地址是“rn”中阐明的值。

Load and Store

执行 LOAD instruction 时，数据从memory传输到register，LOAD 是 memory ‘read’
执行 STORE instruction 时，数据从register传输到memory，STORE 是memory ‘write’

LEC 21

Endianness 字节顺序

我们如何在memory中 store bytes.

Little Endian小字节顺序

Least significant bytes first.

Big Endian 大字节顺序

Most significant bytes first

Little and big endian

Little endian和big endian是计算机系统中表示数据字节顺序的两种方法。它们指的是多字节数据（如16位、32位或64位整数）在内存中的存储顺序。

Little endian：在little endian系统中，数据的最低有效字节（Least Significant Byte, LSB）存储在最低的内存地址，而最高有效字节（Most Significant Byte, MSB）存储在最高的内存地址。换句话说，字节顺序从右到左。
例如，假设我们有一个32位整数0x12345678，它的字节顺序在little endian系统中如下：

内存地址：0x00 0x01 0x02 0x03
存储数据：0x78 0x56 0x34 0x12

Big endian：在big endian系统中，数据的最高有效字节（MSB）存储在最低的内存地址，而最低有效字节（LSB）存储在最高的内存地址。换句话说，字节顺序从左到右。
同样，假设我们有一个32位整数0x12345678，它的字节顺序在big endian系统中如下：

内存地址：0x00 0x01 0x02 0x03
存储数据：0x12 0x34 0x56 0x78

不同的计算机系统和架构可能采用不同的字节顺序。例如，Intel x86和x64架构采用little endian，而大部分网络协议（如TCP/IP）采用big endian。在处理跨平台数据交换时，了解这两种字节顺序的区别非常重要，以避免出现数据错误。

example:

```
STR r6, [r1]   r1 = 0x00008000 and r6 = 0xFF AA BB 11
```

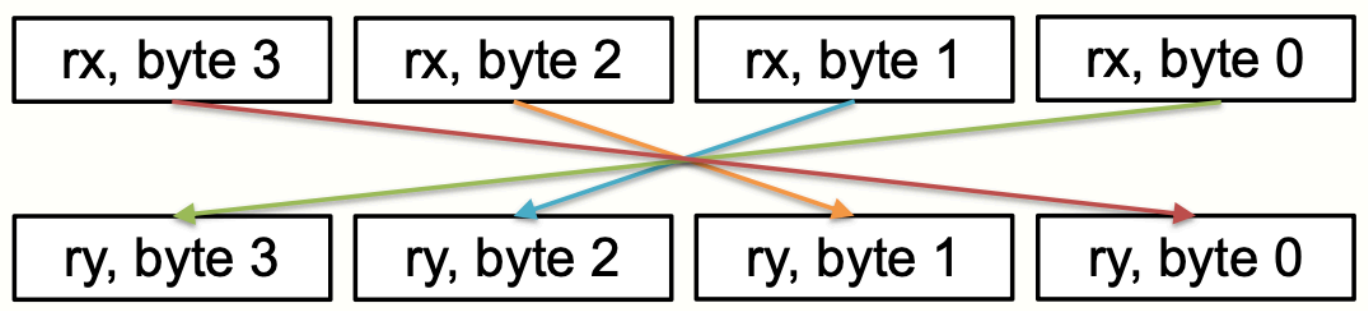
1. STR instruction 该指令将把r6中的值存储到内存地址r1中
2. Firstly r1=0x00008000 in [ ] which means it is a value for addressing memory
3. 读取R6的值为**0x** FF AA BB 11
4. 在 r1给出的对应的在内存的地址中，写入**0x** FF AA BB 11

Reversing byte order

instruction

```
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.



example

instruction

```
REV r0, r5
```

0xF0ACB714 in register r5

put **0x**14B7ACF0 into register r0

Analysis

1. 首先永远记住靠近指令名的位置为最后需要保存的位置，可以是在register bank中本身的位置，可以是在register本身所指向的内存的位置。
2. REV给出的含义是交换 byte order，已知最后要写入r0所代表的位置，所以在实现REV的功能后写入

同一条指令可以将小端转换为大端，将大端转换为小端。
它仅限于“低”寄存器，r0 到 r7。

Loading half words

感觉和kahoot的题有点冲突，主要是理解 half的定义

LDR: LDR 指令用于将一个完整的word（32bits）加载到寄存器中

LDRH: 但有时候，我们可能只需要加载一个half word (16 bits)。这时候，我们可以使用 LDRH 指令。LDRH 指令将内存中的2 bytes (16 bits)加载到寄存器的低的 16 bits，而寄存器的高16位被设置为零。

```
        31              16 15              0
         0000000000000000   ZZZZZZZZZZZZZZZZ
```

LDRSH: LDRSH 指令与 L``DRH 类似，但它会根据半字的符号位 填充寄存器的高16位。这在处理有符号数时非常有用。

符号位: 最高位（也就是第15位）被用作符号位，表示该半字所代表的数值是正数还是负数

```
31              16 15              0
 SSSSSSSSSSSSSSSS   SZZZZZZZZZZZZZZZ
```

Loading bytes

对于字节（8位）的加载，我们可以使用 LDRB 和 LDRSB 指令。这两条指令分别将内存中的一个字节加载到寄存器的低8位。对于 LDRB，寄存器的剩余24位被设置为零；而对于 LDRSB，剩余的24位被设置为符号位。这使得处理有符号字节变得简单。

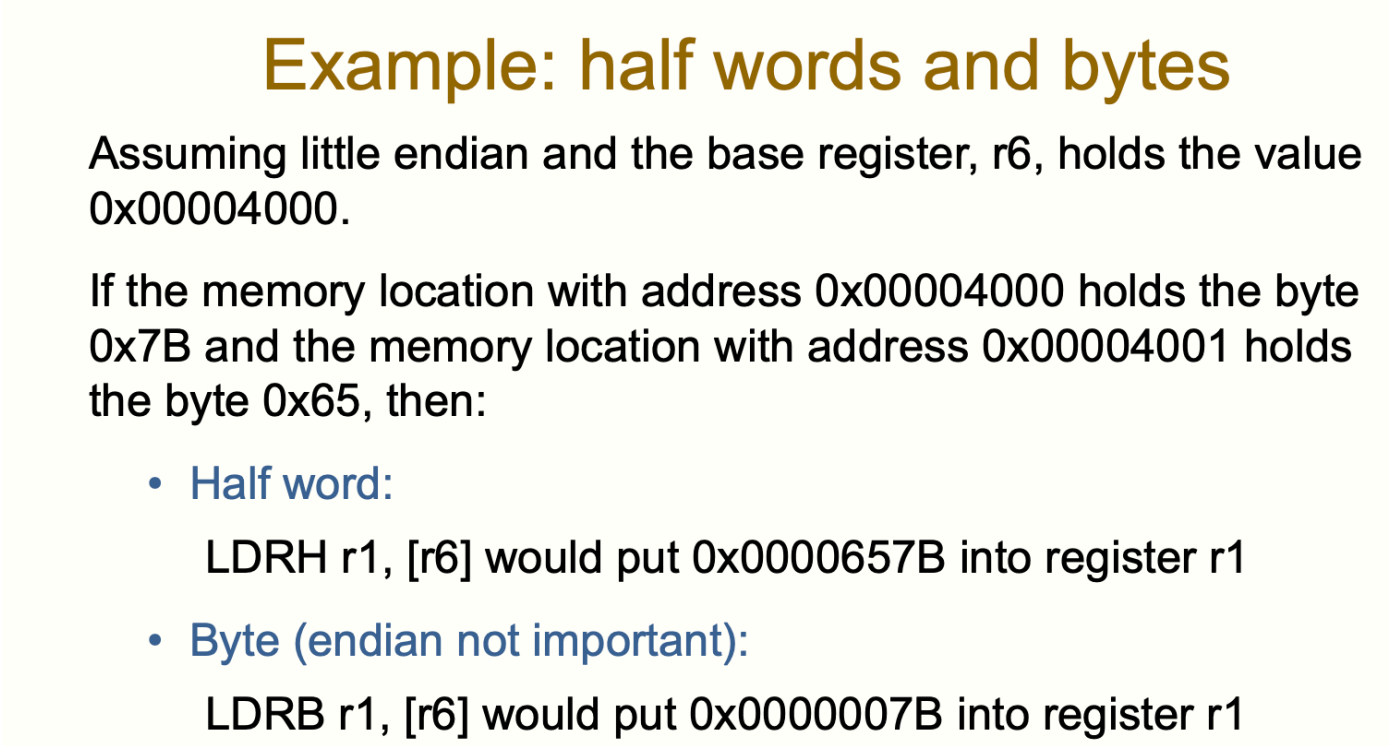
LDRB Load Register Byte

```
         31                       8 7      0
          00000000000000000000000000 ZZZZZZZZ
```

LDRSB Load Register Signed Byte

```
     31                       8 7      0
      SSSSSSSSSSSSSSSSSSSSSSSS SZZZZZZZ
```

Example: half words and bytes



小端字节序（Little Endian）在小端字节序中，数据的低字节（较小的部分）存储在低内存地址中，而高字节（较大的部分）存储在高内存地址中。

背景: 在register bank r6中，r6储存的值是 0x00004000。这指向了一个内存地址。即 **0x**00004000

1. 在**0x**00004000这个地址中存储了一个字节 **0x**7B。
2. 同时，memory地址为 **0x**00004001的地方，存储了一个byte **0x**65。

接下来，两个指令的作用：

1. **LDRH r1, [r6]（加载半字/Load Register Half-word）**：这个指令会从内存地址 **r6的value**（0x00004000）开始，加载两个字节（半字）**到 r1**。在这里加载两个字节是我疑惑是的地方，我不知道为什么在出结果时会出现一个65，观察后发现

   ㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗

   如果一个内存地址是8 bits的话，加载半字，对于**M0**这个32位处理器而言是，16位的！！那对于load来说，会自动抓取内穿地址中的**0x**00004000和**0x**00004001这两个位置的value，再进行把数据的低字节（较小的部分）存储在低内存地址中的排列，得到一个16 bits的答案。

   ㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗㊗

   * 由于我们假设是小端字节序，所以我们需要将 0x7B 和 0x65 按照小端字节序组合起来，得到 0x0000657B。因此，执行这个指令后，寄存器 r1 的值将变为 0x0000657B。

2. **LDRB r1, [r6]（加载字节/Load Register Byte）**：这个指令会从内存地址 r6（0x00004000）开始，加载一个字节（无论大端或小端字节序）到寄存器 r1。在这个例子中，内存地址 0x00004000 存储了一个字节 **0x**7B。因此，执行这个指令后，寄存器 r1 的值将变为 **0x**0000007B。

这个Load Register Byte也使用了 大端或小端字节序 吗

总结一下，当使用小端字节序时，这两个指令的执行结果分别是：

- LDRH r1, [r6] 使寄存器 r1 的值变为 0x0000657B
- LDRB r1, [r6] 使寄存器 r1 的值变为 0x0000007B从内存加载字节到寄存器

Storing half words and bytes

STRH 和 STRB 指令分别用于将register 中的低16位（半字）和低8位（字节）存储到memory中。这两个指令都使用间接寻址方式。

STRH (Store Register Half-word) ！！！字节序规则适用。！！！

STRB(Store Register Byte)从寄存器加载字节到内存

其实就是load和store区别的重复

Base plus offset addressing基址加偏移寻址

基址加偏移寻址方式使用寄存器中的值（基址）加上一个二进制数（偏移量）来确定内存地址。例如，指令 `LDR r6, [r1, #12]` 将从内存地址 r1+12_{10} 处加载数据到寄存器 r6 中。LDR=Load Register

就是value在变换位置时，我们加了值，# means ‘immediate’.在`#12`中，是否代表同样的意思？

Offset using another register基址加偏移寻址使用另外寄存器

偏移量也可以存储在另一个寄存器中。这对于使用 LDRSH 和 LDRSB 指令非常有用。例如，指令 `LDRSH r6, [r1, r4]` 会将内存地址 r1+r4 处的有符号半字加载到寄存器 r6 中。LDRSH=Load Register Signed Half-word

Words, half words and bytes

基址加偏移寻址可用于加载和存储，用于字、半字和字节，包括带符号的半字和带符号的字节。

```
STRH r6, [r1, r4]
LDRB r2, [r3, #17]
LDRSH r5, [r0, r7]
```

- STRH指令的全称是 "Store Register Half-word"，它用于将一个半字（16位数据）存储到存储器中的指定地址。指令中的写操作是从基址寄存器 r1 和偏移量寄存器 r4 中计算出来的。
- LDRB指令的全称是 "Load Register Byte"，它用于从存储器中加载一个字节（即8位数据），并将其存储到指定的寄存器中。指令中的读操作是从基址寄存器 r3 和一个立即数偏移量 #17 中计算出来的。
- LDRSH指令的全称是 "Load Register Signed Half-word"，它用于从存储器中加载一个有符号半字（即16位数据），并将其扩展为32位有符号整数，并将其存储到指定的寄存器中。指令中的读操作是从基址寄存器 r0 和偏移量寄存器 r7 中计算出来的

Words and half words alignment(对齐)

ARM指令集中的一些指令，如LDR和STR，要求地址值是"字对齐"的，即地址值必须是4的倍数。

因为一个字（32位数据）被存储在4个相邻的存储器位置中。例如，地址值为0x00000000、0x00000004、0x00000008等都是字对齐的，而地址值为0x00000001、0x00000003、0x00000005等都不是字对齐的。

可以说这里的对齐是指memory上的地址值。

问&Q

1. 为什么地址值为0x00000001、0x00000003、0x00000005等都不是半字对齐的

首先，在计算机存储器中，每个存储单元的大小通常为一个字节（8位），并且每个存储单元都有一个唯一的地址，通常是从零开始递增的

半字是一个16位的数据类型，因此在存储器中，一个半字被存储在两个相邻的存储单元中，每个存储单元大小为一个字节（8位）。

当地址值为0x00000000时，半字的第一个字节存储在0x00000000地址上，半字的第二个字节存储在0x00000001地址上，因此地址值为0x00000000是半字对齐的。而当地址值为0x00000001时，半字的第一个字节存储在0x00000001地址上，第二个字节存储在0x00000002地址上，因此地址值为0x00000001是不半字对齐的。

2. 当地址值为0x00000001时，半字的第一个字节存储在0x00000001地址上，第二个字节存储在0x00000002地址上，为什么地址值为0x00000001是不半字对齐的?

对于半字，它的存储方式是将16位数据拆分成两个8位的字节，然后分别存储在两个相邻的存储单元中。因此，为了保证半字的存储是正确的，**半字的地址必须是偶数。** 当地址值为奇数时，半字的第一个字节存储在奇数地址上，而第二个字节则存储在下一个偶数地址上。这意味着半字的两个字节不是在相邻的地址上，而是跨越了两个地址，因此不符合半字对齐的要求。

ppt:

通过基数加偏移量寻址计算的地址值必须是：

- 对LDR和STR来说是 "字对齐 "的。"字对齐 "的地址是可以被4整除的，字被存储在4个相邻的内存位置。

LDR为什么是字? 看前面register储存着对应地址的值，是在内存上的，是字。

- 对LDRH、LDRSH和STRH来说是 "半字对齐"。半字对齐 "地址可被2整除，半字存储在两个相邻的内存位置。

Offset immediates偏移即时值的限制

基准加偏移寻址中使用的偏移值都是5 bits immediate.
对于字节访问（LDRB和STRB），偏移量的范围是0-31。(不包括LDRSB)
对于半字访问（LDRH和STRH），偏移量是0-62范围内的偶数。(不包括LDRSH)
对于字访问（LDR和STR），偏移量是一个在0-124范围内可被4整除的数字。

问：为什么对于 半字访问和字访问会乘2和乘4

```
在 ARM 汇编中，对于半字（half word）和字（word）访问，分别乘以 2 和 4 的原因与内存对齐有关。这里的关键概念是，我们希望确保数据在内存中的地址是其大小的整数倍，从而提高内存访问性能。

半字访问：半字是 2 字节（16 位）的数据。为了保证半字数据在内存中以 2 字节边界对齐，我们需要以 2 字节为单位移动内存地址。所以，当我们在基址寄存器的值上添加偏移量时，需要将原始的偏移值乘以 2，以确保结果地址是 2 字节对齐的。

字访问：字是 4 字节（32 位）的数据。为了保证字数据在内存中以 4 字节边界对齐，我们需要以 4 字节为单位移动内存地址。所以，当我们在基址寄存器的值上添加偏移量时，需要将原始的偏移值乘以 4，以确保结果地址是 4 字节对齐的。
举个例子，如果我们要访问一个字（4 字节）数据，基址寄存器的值为 0x1000，偏移值为 3。在计算实际地址时，我们需要将偏移值乘以 4（因为一个字是 4 字节），所以实际的偏移值是 3 * 4 = 12。那么，我们要访问的内存地址就是 0x1000 + 12 = 0x100C。这个地址是 4 字节对齐的，因为它可以被 4 整除。
简而言之，乘以相应的倍数（2 或 4）是为了确保内存访问操作符合内存对齐要求。
```

问：什么是偏移值，是内存上结果的实际值，还是register上对应的内存位置

偏移值（offset）是一个用于计算实际内存地址的数值。它不是内存上的实际值，也不是寄存器上的内存位置。偏移值通常与基址寄存器的值相加，以确定要访问的内存地址。
在基址寄存器加偏移寻址（base plus offset addressing）方式中，我们使用一个基址寄存器（例如 r0）和一个偏移值（例如 4）来计算内存地址。这个内存地址将用于加载或存储数据。具体计算方式是将基址寄存器的值与偏移值相加，得到实际要访问的内存地址。
以一个简单的例子来说明：
假设我们有一个基址寄存器 r0，其值为 0x1000，我们有一个偏移值 4。要计算实际访问的内存地址，我们需要将基址寄存器的值（0x1000）与偏移值（4）相加，得到 0x1000 + 4 = 0x1004。所以，我们要访问的内存地址是 0x1004。
在这个例子中，偏移值是 4，它用于确定内存地址的具体位置。通过将偏移值与基址寄存器的值相加，我们得到了实际访问的内存地址。这种寻址方式可以灵活地访问内存，特别是在处理数组和结构体等数据结构时非常有用。

问：So in base plus offset addressing, what is changed is the starting point of the addressed memory address, not the actual word or half-word stored in memory, and these stored locations are recorded in registers on register?

Offset value = 5 bits immediate => 2^5 = 32 portions of memory (0 to 31), called chunks.

Addressing modes: a summary

Register addressing:

Immediate addressing:

Indirect addressing:

Base plus offset addressing:

Lec 22 week 8

Flags

Flag可以用来发出signal to reflect某些事情已经发生或尚未发生。
Flag本质上是1 bit memory设备。
微处理器使用flags来表示已经发生了什么。
不同的微处理器有不同的标志

在ARM cortex M0中，这四个标志被存储在一个叫做Current Program Status Register（CPSR）的寄存器中。

⚠️⚠️⚠️⚠️⚠️⚠️CPSR 4bit ⚠️

N,V,C,Z

The ARM Cortex M0 processor has **four** flags that are commonly

Z ero flag (**Z**), **N**egative flag (**N**), **C**arry flag (**C**) and O**v**erflow flag (**V**)

- Zero flag (Z)：表示最近一次算术或逻辑操作的结果是否为零。如果结果为零，则Z标志位被设置为1，否则为0。
- Negative flag (N)：表示最近一次算术或逻辑操作的结果是否为负数。如果结果为负数，则N标志位被设置为1，否则为0。
- Carry flag (C)：表示最近一次算术操作的进位或借位情况。例如，在执行无符号数相加时，如果结果大于指定的位数，则C标志位被设置为1，否则为0。在执行有符号数相减时，如果结果小于零，则C标志位被设置为1，否则为0。
- Overflow flag (V)：表示最近一次算术操作的溢出情况。例如，在执行有符号数相加时，如果结果超出了所能表示的范围，则V标志位被设置为1，否则为0。

当ARM执行大多数指令时，会自动产生4个信号（N、Z、V、C）。N、Z、V、C只有在程序员特意指示时才会被复制到相应的CPSR位上。

除了分支指令（B）外，所有指令都包括可选的项目{S}。

这个可选项S控制 "set flags "的动作

For example, the following instructions set or clear the flags as appropriate:

MOV**S** , ADD**S** , SUB**S** , RSB**S** , MUL**S** , AND**S** , EOR**S** , ORR**S** , BIC**S**

Status Register

我们需要先了解ARM M0处理器中的状态寄存器（Status Register）。状态寄存器（简称xPSR）是一个32位寄存器，其中包含一些用于表示处理器状态的标志位。其中之一便是进位标志（C标志位），它位于xPSR的第29位。

carry flag进位标志

进位标志（Carry Flag）主要用于表示无sign integer算术运算（如加法、减法、位移等）中是否发生了进位或借位。这个标志位可以帮助我们检测数值溢出的情况，以便在进行比较和分支操作时作出正确的判断。

example

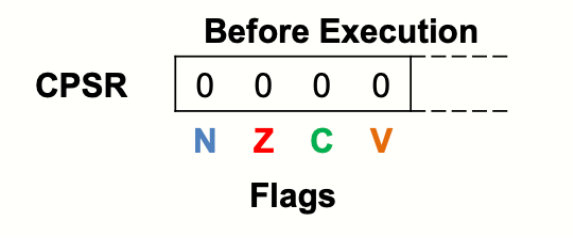
假设我们要将两个8位无符号数相加：A = 10101010（十进制数：170），B = 00111001（十进制数：57）

```
ADD R1, R2, R3 ; R1 = R2 + R3
```

相加就会发生溢出，所以进位标志C会被置为1。我们可以通过检查C标志位来判断是否发生了溢出：

```
mathematicaCopy code
bcs overflow_handler ; 如果进位标志C被置位，跳转到溢出处理程序
```

在这个例子中，进位标志C为1，程序将跳转到overflow_handler进行相应的溢出处理。变化从图一到图二



我们首先执行了一个导致C标志位被置为1的加法操作，然后执行了另一个加法操作，使C标志位从1变为0。在ppt中，在carry位置为1的情况下，可以变回0。

The zero flag

If an instruction produces a result which is 0x00000000 then the zero flag would be set to 1.

The subtraction of a value from itself will always be 0. 零标志在比较、测试和分支指令中非常有用，因为它可以帮助我们确定操作数之间的关系，例如是否相等。

感觉就是value出0，那么z value就会被置1.

Flags作用

1. 条件执行：标志位主要用于判断条件分支指令是否应该执行。处理器执行某些算术或逻辑操作后，会根据操作结果来设置相应的标志位，如零标志（Zero Flag，Z）、进位标志（Carry Flag，C）、溢出标志（Overflow Flag，V）和负数标志（Negative Flag，N）。然后，在条件分支指令（如BNE、BEQ、BGT等）中，处理器会根据这些标志位来决定是否跳转到指定的程序地址。这种基于标志位的决策机制使得程序能够根据数据值或操作结果选择不同的执行路径。
2. 作为额外数值参与算术运算：除了用于条件执行外，进位标志（Carry Flag，C）在某些情况下还可以充当算术运算的额外输入。例如，在处理器执行带进位加法（如ADCS指令）或带借位减法（如SBCS指令）时，进位标志可以作为操作数之间的额外进位或借位参与运算。这种用法在多字长整数运算和加密算法等场景中非常有用，可以帮助实现更高精度和更安全的计算。

Branches

1. 无条件分支：无条件分支总是执行，并且会用一个新值更新程序计数器（Program Counter，PC）(R15)。换句话说，Unconditional Branche会导致程序执行流程跳转到一个新的地址，而不考虑任何条件。在ARM处理器中，无条件跳转指令是B（Branch）。例如，使用`B label`指令可以使程序跳转到标签“label”所对应的地址处继续执行。无条件分支通常用于循环、子程序调用（使用BL指令）和返回（使用BX指令）等场景。
2. 条件分支：条件分支的执行取决于特定条件，即处理器中的标志位状态（例如零标志、进位标志等）。根据条件是否满足，处理器会选择跳转到新地址（执行分支），或者继续正常执行下一条指令（不分支）。ARM处理器提供了多种基于不同条件的分支指令，如BEQ（等于）、BNE（不等于）、BGT（大于）、BLT（小于）等。例如，当零标志（Z）为1时，`BEQ label`指令会导致程序跳转到标签“label”所对应的地址处继续执行；

当零标志为0时，程序将顺序执行下一条指令。条件分支在程序中广泛应用于判断和循环控制结构。

我记得条件储存在 r4

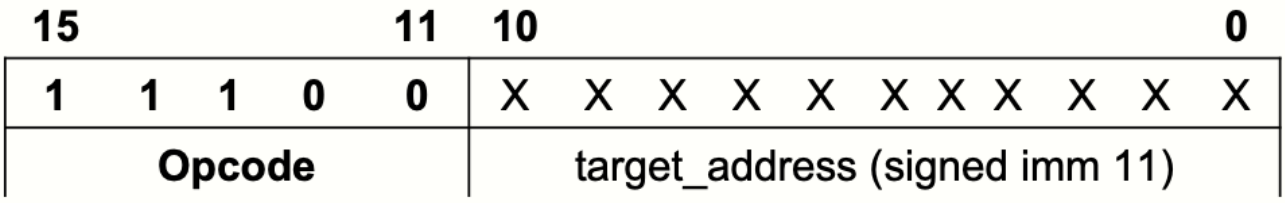
Unconditional Branches – range 无条件指向

B <target_address>

- The lower 11 bits of the machine code are used to determine the destination address as an offset from the current program counter value.应该就是0-10位的地方，是目标地址作为当前程序计数器的偏移量
- The difference between the destination address and the memory address of the unconditional branch instruction must be an even number within the range +2046 to -2048.无条件的branch指令是一个在+2046到-2048点偶数
- *无条件跳转指令的目标地址与指令本身的地址之差（即跳转偏移量）必须是一个偶数，并且在+2046到-2048的范围内。这是因为ARM指令集的指令长度为2个字(16位)，因此跳转目标地址必须是2字节对齐的，即偶数地址。而跳转偏移量的范围限制则是由于指令编码格式的限制所导致的。*
- 在这里Opcode是干什么的
- opcode（operation code）是指一条指令的操作码，它指定了计算机执行的特定操作。例如，以下是一个ARM指令的示例：

```
ADD r1, r2, r3
```

这条指令的操作码是“ADD”，它指示CPU将r2和r3中的值相加，并将结果存储在r1中。



Conditional execution

有条件执行，这个条件在于对于Flag的判断

条件执行是指分支指令根据flag的状态来判断是否执行的一种执行方式。在计算机中，标志位是一组二进制标记，用于记录先前操作的结果或状态。

B <target_address>

在ARM指令集中，分支指令的助记符(mnemonic)是B，通过在助记符后添加两个字母来表示条件执行。例如，BCS表示仅在进位标志位(Carry flag)被设置时执行分支指令。

标志位通常由计算机处理器设置或清除，这些标志位可以表示操作是否成功、是否溢出或是否需要进行进一步操作等信息。在条件执行中，分支指令将根据标志位的值来判断是否跳转到指定的目标地址。如果满足条件，则执行分支指令，否则不执行，继续执行后续指令。

在ARM指令集中，可以将15种不同的条件附加到(almost)任何助记符中。这些条件包括EQ(等于)、NE(不等于)、CS/HS(无符号大于或等于)、CC/LO(无符号小于)、MI(负数)、PL(正数)、VS(溢出)、VC(未溢出)、HI(无符号高位)、*LS(无符号低位)*、*GE(有符号大于或等于)*、*LT(有符号小于)*、*GT(有符号大于)*、*LE(有符号小于或等于)*、AL(总是)。

通过使用条件执行，程序可以根据标志位的状态来决定程序的执行路径，从而提高程序执行的效率和可靠性。

Conditional execution 2

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.

Conditional Branches举例子

无条件分支用处很小，但条件分支有很多用处。任何条件字段都可以与分支一起使用，例如

```
1 | BNE 0x08C00000
```

意味着如果不相等，则分支到0x08C00000：

当遇到一个表示“如果不相等（Not Equal）则分支至地址0x08C00000”的指令时，处理器将根据零标志（Z flag）的状态来判断是否执行分支。

Zero Flag 是状态寄存器中的一个bit，用于表示比较或算术运算的结果是否为零。当零标志清零（clear）时，表示两个数不相等；当零标志置位（set）时，表示两个数相等。

在这个例子中，有两种情况：

1. 如果零标志清零（Z flag is clear），则处理器会跳转至地址0x08C00000并执行该地址处的指令。这意味着两个数不相等，条件满足，所以执行分支。
2. 如果零标志置位（Z flag is set），则处理器会继续执行分支指令后面的下一条指令。这意味着两个数相等，条件不满足，所以不执行分支，而是继续顺序执行。
   通过这样的条件分支指令，程序可以根据不同的运行时情况选择执行不同的代码路径，实现灵活的控制流程。

Branches – mnemonics

mnemonics指的是助记符，比如 SUB ADDS

一般来说，编写汇编语言程序时不知道相应机器代码的内存地址。
因此，分支的记忆法使用 "标签 "而不是实际的内存地址，汇编程序决定了实际使用的数值。

E.g.

```
        BNE continue
        SUB r4, r7, r7
continue ADDS r4, r4, #76
```

Use of the zero flag

零标志（Zero flag）是计算机程序中常用的一个状态标志位，它通常用于检查程序的“循环”是否已经执行了特定的次数。

例如，假设我们需要将内存中的10个数相加，这可以通过执行一个重复10次的循环来实现。在这种情况下，我们需要选择一个寄存器作为“循环计数器”（loop counter）。在每次循环过程中，计数器的值将减1，当计数器的值减少到0时，循环结束。

现在我将用汇编语言提供一个例子，并对每一步进行解释：

```
; 假设我们要将内存地址0x1000开始的10个数相加，将结果存储在寄存器R0中

MOV R0, #0      ; 将寄存器R0清零，用于存储累加结果
MOV R1, #10     ; 将寄存器R1设置为10，用作循环计数器
LDR R2, =0x1000 ; 将寄存器R2设置为内存地址0x1000，用于访问内存中的数据

LOOP:
    LDR R3, [R2]    ; 将内存地址R2处的值加载到寄存器R3中
    ADD R0, R0, R3  ; 将寄存器R0与R3中的值相加，并将结果存储回R0中
    ADD R2, R2, #4  ; 将寄存器R2的值加4，以指向下一个内存地址
    SUBS R1, R1, #1 ; 将寄存器R1的值减1，并更新零标志
    BNE LOOP        ; 如果零标志未置位（不相等），则跳转回LOOP标签继续执行循环

```

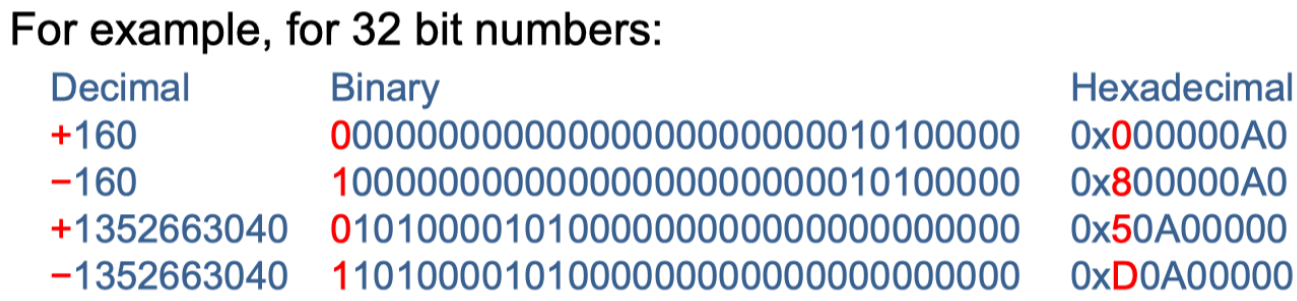
课上的kahoot 习题1，没理解

Lec 23

Negative numbers

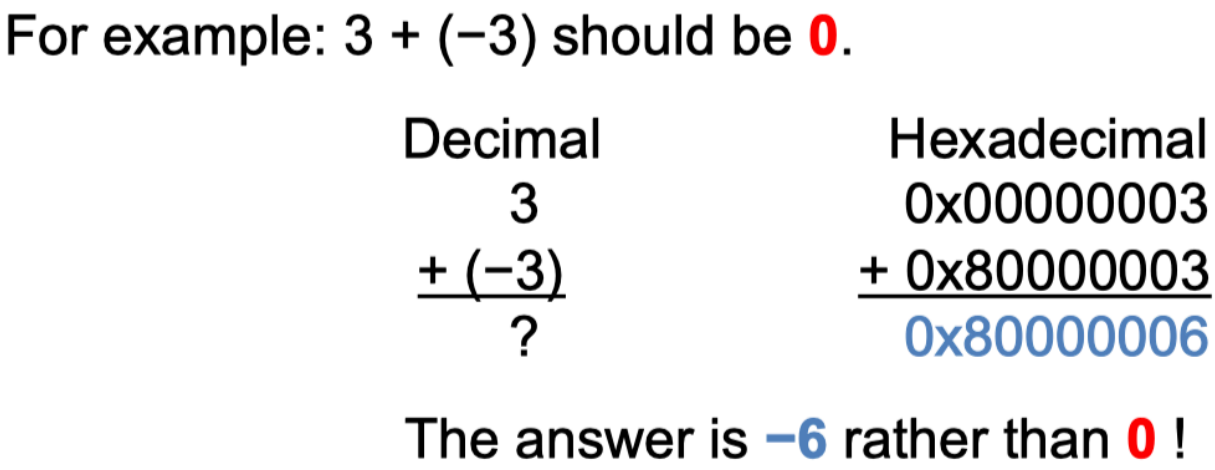
对于负数的表示主要有两种方法 in microprocessors.

1. Sign magnitude符号位位于一个二进制数的最高位。如果符号位为 1，则该数为负数；如果符号位为 0，则该数为正数。



使用一个二进制数的最高位作为符号位。其余位表示该数的绝对值。例如：

- 正数 5 的二进制表示为：0101
- 负数 -5 的二进制表示为：1101
- 请注意，这种表示法存在正零和负零的情况。



2. Two's complement

补码表示法是计算机中最常用的表示有符号整数的方法。在这种表示法中，仍然使用最高位作为符号位。对于正数，其补码表示与原码（即符号大小表示法）相同；对于负数，其补码是对该数的绝对值求原码，然后对原码进行按位取反（即将 0 变为 1，将 1 变为 0），最后加 1。例如：

- 正数 5 的补码表示为：0101
- 负数 -5 的补码表示为：1011
- 补码表示法的一个优点是它只有一个零表示，没有正零和负零的问题。

课件说法On PPT

最高位（most significant bit，MSB）

如果一个二进制数的m.s.b.是1，那么这个数就是负数
如果一个二进制数字的m.s.b.是0，那么这个数字就是正数。

如果一个十六进制数的最重要数字是8，9，A，B、C、D、E或F，那么它就是一个负数。
如果一个十六进制数字的最重要的数字是0，1，2，3、4、5、6或7，那么它就是一个正数。

2's complement

Unlike sign magnitude, arithmetic is simple in two's complement.

For example (32 bit number): 3 + (−3) should be **0**.

Decimal	Binary
3	0000 0000 0000 0000 0000 0000 0000 0011
+ (−3)	+ 1111 1111 1111 1111 1111 1111 1111 1101
0	1 0000 0000 0000 0000 0000 0000 0000 0000

Note that, if the **carry bit** (33rd bit) is ignored, the answer is **0** which is correct.

在tut中有练习，看一下为什么 N,V,C,Z都变化了

Negative numbers in 4 bits

Hex	Binary	Sign magnitude	Two's complement
8	1000	0	-8
9	1001	-1	-7
A	1010	-2	-6
B	1011	-3	-5
C	1100	-4	-4
D	1101	-5	-3
E	1110	-6	-2
F	1111	-7	-1

所以对于占用位置的不同，也导致了Unsigned integer，Sign magnitude和Two's complement所能表示的范围出现不同

Allowed ranges

	4 bits	16 bits	32 bits
Unsigned integer	0 to 15_{10} 0000_2 to 1111_2	0 to 65535_{10} 0000_{16} to $FFFF_{16}$	0 to $(2^{32} - 1)$ 00000000_{16} to $FFFFFFFF_{16}$
Sign magnitude	-7_{10} to 7_{10} 1111_2 to 0111_2	-32767_{10} to 32767_{10} $FFFF_{16}$ to $7FFF_{16}$	$-(2^{31}-1)$ to $(2^{31}-1)$ $FFFFFFFF_{16}$ to $7FFFFFFF_{16}$
Two's complement	-8_{10} to 7_{10} 1000_2 to 0111_2	-32768_{10} to 32767_{10} 8000_{16} to $7FFF_{16}$	-2^{31} to $(2^{31}-1)$ 80000000_{16} to $7FFFFFFF_{16}$

2's complement: sign extension 符号扩展

在计算机中，有时我们需要将一个较小位数的补码数（如 4 位或 16 位）转换为更高位数的格式（如 8 位或 32 位）。为了正确地执行这种转换，我们使用一种称为"符号扩展"（sign extension）的过程。

符号扩展的主要目的是在保持有符号整数值不变的同时，将其从一个位数转换为另一个更大的位数。为了实现这一点，我们需要将最高位（即符号位）复制到更高位数格式的额外位上。这样，我们可以保持数字的正负性质，同时确保其值保持不变。

例如，假设我们需要将一个 4 位的补码数 1101 转换为 8 位的格式。我们可以按照以下步骤进行：

1. 首先，观察 4 位数的最高位（符号位），在这里是 1。
2. 然后，将该符号位复制到额外的高位上，得到 8 位数：1111 1101。

通过这种符号扩展方法，我们可以确保数值在转换过程中保持不变。在这个例子中，4 位补码数 1101（-3，以补码形式）被成功地转换为 8 位补码数 1111 1101，其值仍为 -3。

Decimal	4 bits	32 bits
7	0111	0000 0000 0000 0000 0000 0000 0000 0111
−6	1010	1111 1111 1111 1111 1111 1111 1111 1010

The extra 28 bits , extending the bits from 4 to 32 take the value of the sign bit shown in red . Likewise in hexadecimal:

Decimal	16 bits	32 bits
20165	0x4EC5	0x00004EC5
−32501	0x810B	0xFFFF810B

2’s complement: arithmetic

很大，就会超

使用 Setting the N flag的方法

如果使用 ADDS 指令，负标志 (N) 将被设置，但进位标志 (C) 将被清除，因为总和仍然是 32 位结果。

任何设置或清除标志的指令，例如 如果in the destination register的值的符号位等于 1，MOVS 将设置负标志。

what? 没看懂

Overflow

-2^{31} to $\left(2^{31}-1\right)$or 0x80000000 to 0x7FFFFFFF.

就是说这样这样减去了以后，在sign bit的部分出现了本身是计算方面的值，所以对于这个结果0x9AF8DA00来说，他超出了

Setting the V flag

对于这个操作来说，

Adding negative numbers

Using two’s complement we can do sums, x + (−y)

如果我们将$1,500,000,00_{10}$ 加到$-1,100,000,000_{10}$上

1. 找到-1,100,000,000的2进制补码，正值是：$1,100,000,000_{10}$或0x4190AB00
2. 反转所有的位数0x4190AB00 - OxBE6F54FF
3. 在结果中加入1：0xBE6F54FF + 1 = 0xBE6F5500
4. 将OxBE6F5500加到0x59682F00（1,500,000,00010）。

Adding negative numbers

```
   0x  59 68 2F 00
+  0x  BE 6F 55 00
  0x1 17 D7 84 00
```

Because we are working in two's complement, we can ignore the carry 1 and the answer in the lowest 32 bits is 0x17D78400 or $400{,}000{,}000_{10}$ which is correct.

There is a 'carry' indicating that the result is too big for a 32 bit unsigned integer

There is **no overflow** in two's complement

The **signed number** is positive because the m.s.b. is 0 (most sig. hex digit is 0x1 = 0001_2).

Adding negative numbers

```
1  1111 00…   ← carry from previous column
   0101 1001 0110 1000 0010 1111 0000 0000
+  1011 1110 0110 1111 0101 0101 0000 0000
   0001 01…   ← result (top 6 bits)
```

The **sign bit** of the result is 0 indicating a positive result.

A **carry out** (1) occurs so:

- The result is incorrect if considered as an unsigned integer
- The 32 bit result is correct if consider as a two's complement number.

Lec 24

TuT 18th

add 不会改变值

7 能不能加一再反

8 -1所以，用1，1=0000 0001，所以-1= 1反+1=FFFF FFFE+1=1

IEEE 754

转成 decimal to binary

Studyeaq

Lec 25

jhpooh

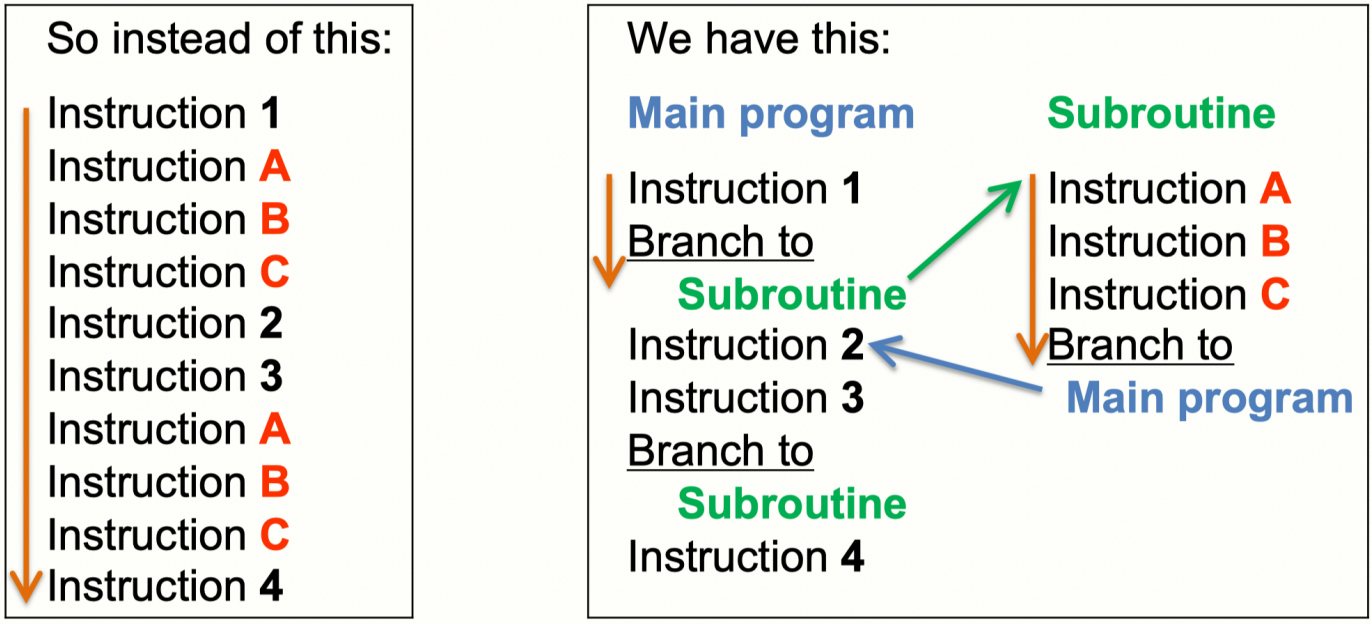
ohgtgoijn 突然功能

vertong

Lec 26

Subroutine

sub，子的意思，没啥好理解的就是，从main会跳到子程序



Link Register

在结束后，subroutine，会要返回到Main的函数值，在ARM Cortex M0微处理器中，链接寄存器是寄存器 'r14'

lr 是助记符，mnemonics，Link Register

Branch with link

The mnemonic助记符 for 'branch with link' is **BL**

```
BL <address>

```

其中，`<address>`表示要跳转到的程序地址。在执行BL指令时，处理器将当前指令的地址（也就是返回地址）保存在LR寄存器中，并跳转到指定的地址。

BL指令的目标范围是±16 MiB（兆字节，即16 * 1024 * 1024字节），相对于当前程序计数器（PC）的值。程序计数器是一个特殊的寄存器，用于存储CPU正在执行的指令的内存地址。因此，BL指令可以跳转到当前PC值前后16 MiB范围内的任何地址。

BL指令使用一个24位的立即数，这意味着它可以表示的地址偏移范围是从-2^23 （-8,388,608）到2^23-1（8,388,607）。由于BL指令采用PC相对寻址方式，因此这个范围是相对于当前PC值的。

此外，BL指令本身是一个32位（4字节）的机器代码，这是因为ARM处理器采用的是32位体系结构。**这个32位机器代码中的24位用于表示立即数（地址偏移）**，剩下的**8位用于表示其他指令信息，如操作码** 等。

总结

BL指令是ARM体系结构中的一种跳转指令，用于调用子程序或函数。它使用24位立即数表示地址偏移，可以在当前程序计数器值前后16 MiB范围内跳转，其机器代码长度为32位（4字节）。

Branch and exchange

（分支和交换）指令在ARM体系结构中用于在子程序或函数调用之后返回到调用点。在使用BL指令调用子程序时，返回地址（即调用点的下一条指令的地址）被存储在Link Register （连接寄存器，通常表示为LR或r14） 中。子程序执行完成后，需要将Link Register中的值传递回Program Counter（程序计数器，PC）以实现返回。

instruction

```
main:
    // Some code here
    BL subroutine
    // Continue executing code here after returning from subroutine

subroutine:
    // Code for the subroutine
    BX LR // Return to the main function

```

我们使用BL指令跳出main，但是使用BX指令滚回main的代码。

具体的操作其实是改变R15的值，即PC值。

我们使用BX指令将Link Register（LR）中的值移动到程序计数器（PC），从而实现从子程序返回到主程序。

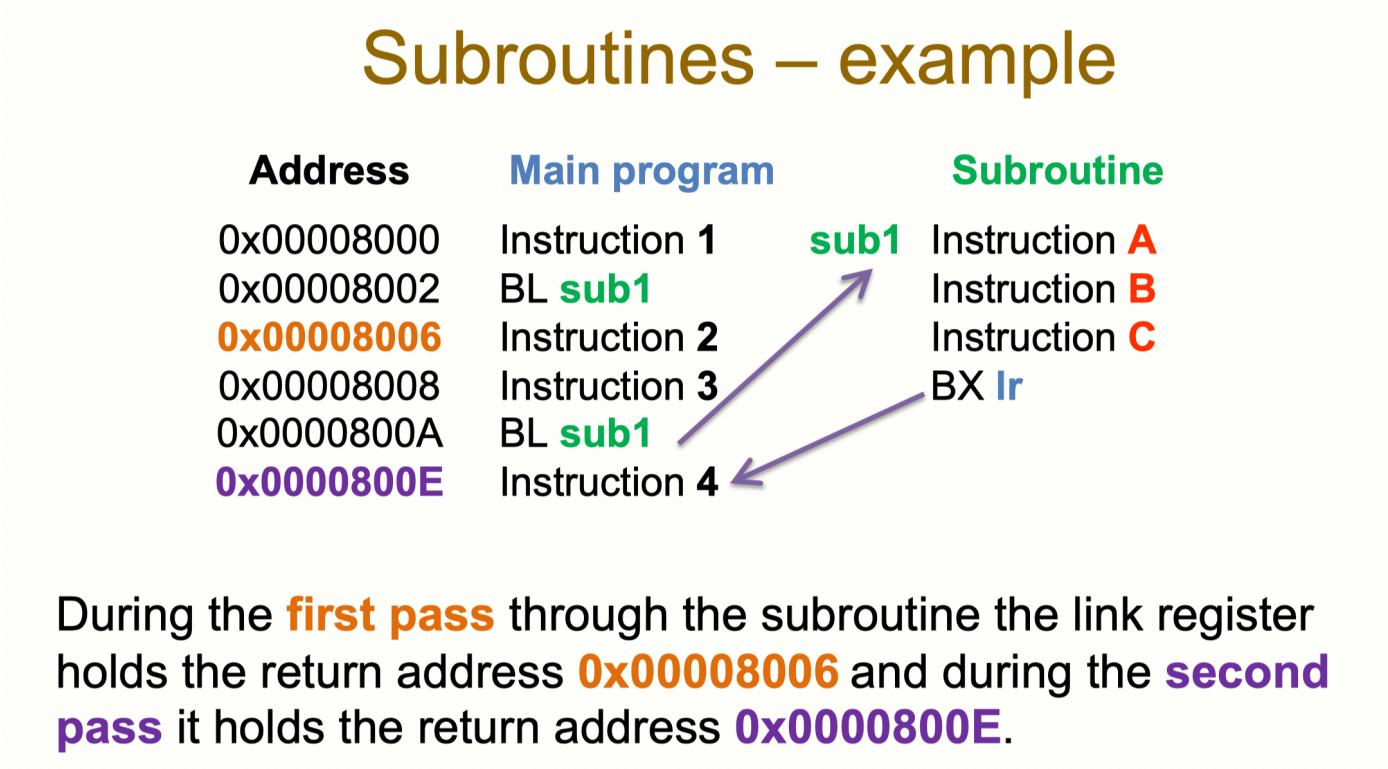
Branch with Link and Exchange 又是一个新指令

instruction 名字就叫Branch with Link and Exchange

```
main:
    // Some code here
    BLX subroutine
    // Continue executing code here after returning from subroutine

subroutine:
    // Code for the subroutine
    BX LR // Return to the main function

```



在ARM状态下，BLX指令有两种格式：

1. BLX： 使用指定寄存器中的地址进行跳转。在这种情况下，目标地址由寄存器提供，并在跳转前检查其最低有效位。
2. BLX： 使用立即数表示的目标地址进行跳转。在这种情况下，跳转目标是一个标签，但指令格式仍然允许在跳转前检查目标地址的最低有效位。就是最后一位?
3. 将返回地址保存在链接寄存器（Link Register，LR或r14）中。

BLX& BL& BX

BLX（Branch with Link and Exchange）、BL（Branch with Link）和BX（Branch and Exchange）

BL和BLX指令用于实现子程序调用，将返回地址保存在链接寄存器中。

BX和BLX　There is no restriction on the destination of the BLX or BX branch

X

BLX 将返回地址保存在链接寄存器（Link Register，LR或r14）中。BX将链接寄存器（LR或r14）中的返回地址移动到程序计数器（PC或r15）中。

未完待续。。。感觉没有搞清楚

栈（Stack）

1. LIFO原则：栈遵循后进先出的原则，这意味着最后添加到栈中的数据（“入栈”或“push”操作）会最先被从栈中取出（“出栈”或“pop”操作）。
2. 举例说明：根据您提供的例子，如果按照以下顺序将值推入栈：0x00FF、0xFF00、0xAAAA。那么它们将按照相反的顺序从栈中弹出：0xAAAA、0xFF00、0x00FF。
3. 栈指针：通常，栈使用一个称为栈指针（Stack Pointer，通常表示为SP或r13）的特殊寄存器来追踪栈顶的位置。当数据入栈时，栈指针会相应地移动，指向新的栈顶；当数据出栈时，栈指针也会相应地移动，回到上一个栈顶的位置。

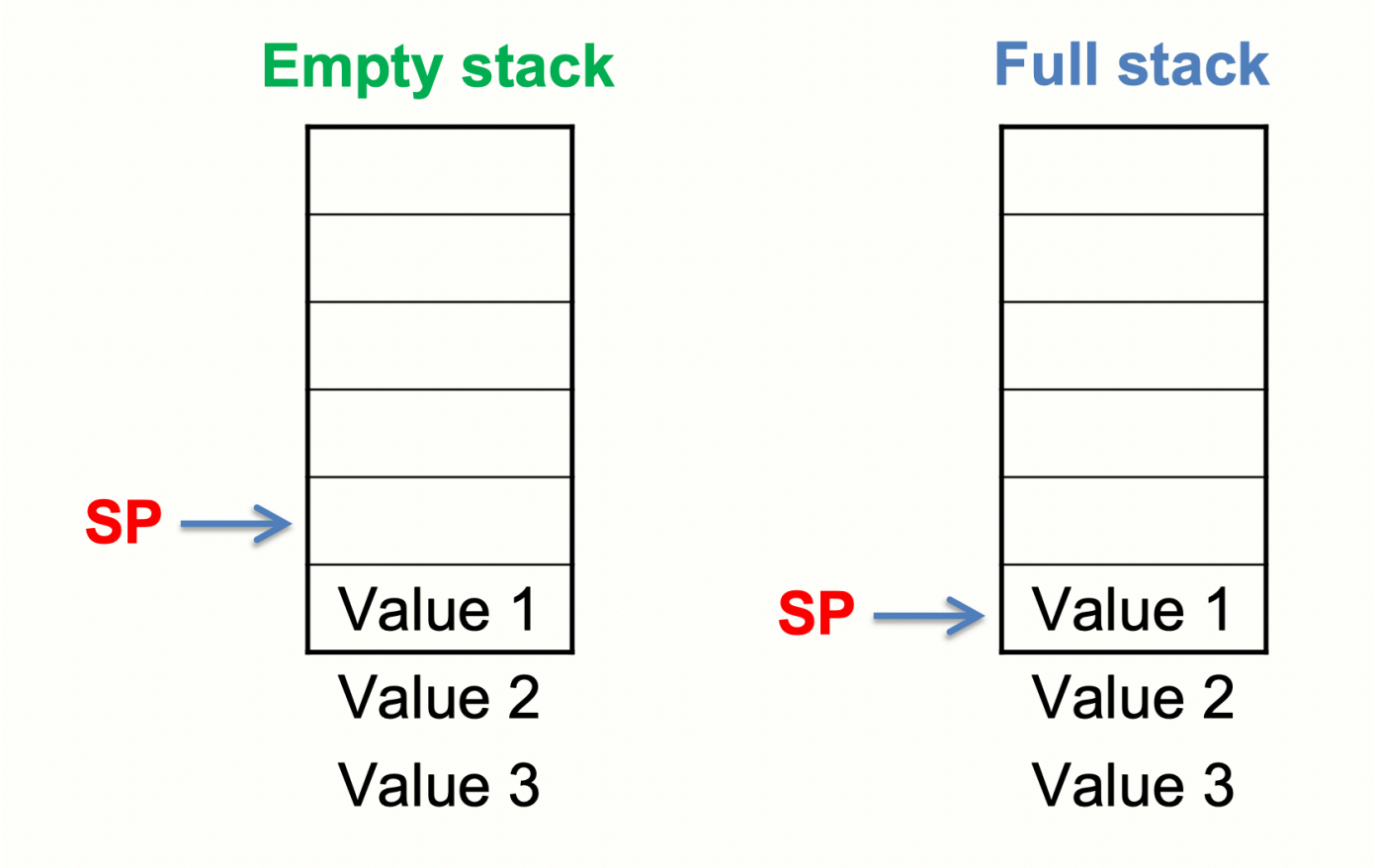
The stack pointer holds an address in memory that identifies the **top** of the stack.

Empty stack and Full stack

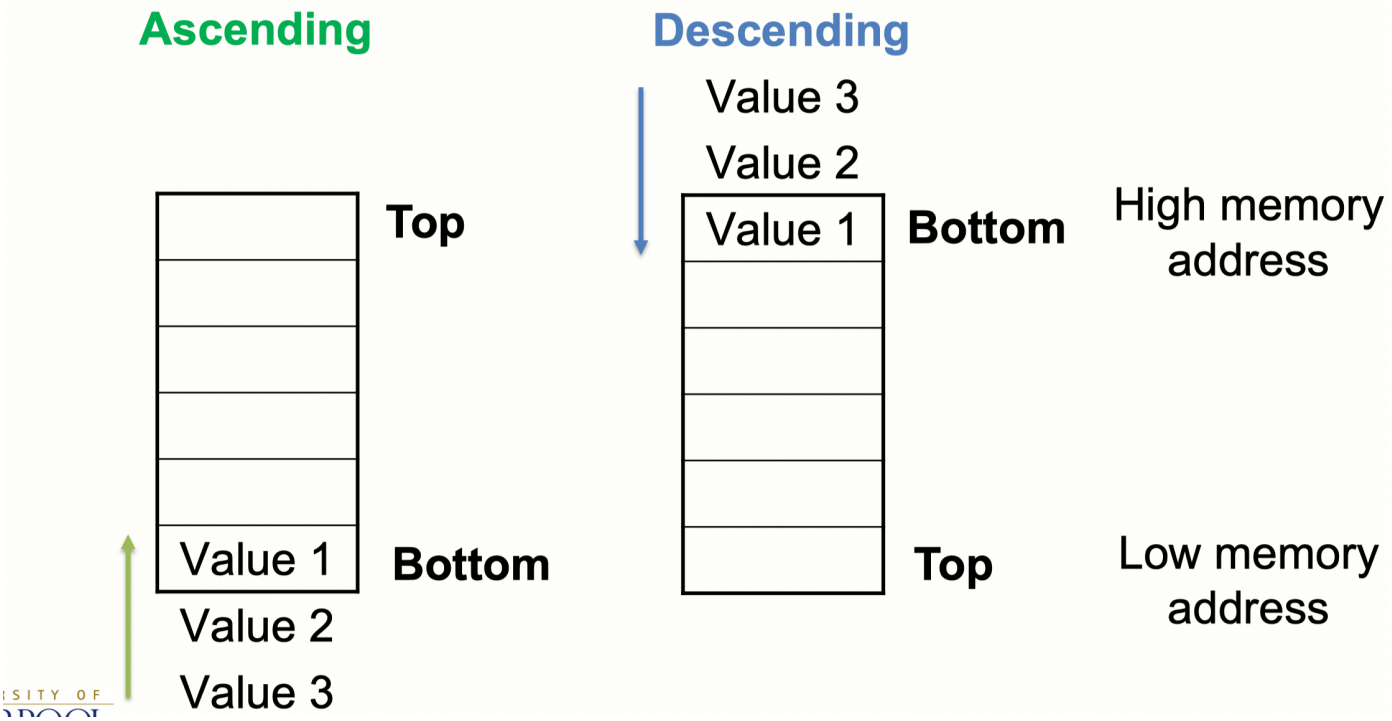
堆栈指针持有内存中的一个地址，用于识别堆栈的top。

Empty stack 下一个可以放置下一个数据的空槽的位置

Full stack 该地址要么是最后一个被推入堆栈的数据的位置



Ascending and descending stacks



top，descending，在

The ARM Cortex M0 uses a descending stack.

Stacks and the ARM Cortex M0

1. 递减栈：ARM Cortex-M0使用递减栈（Descending Stack），这意味着栈从高地址向低地址增长。每次入栈操作时，栈指针（Stack Pointer，通常表示为r13或SP）向低地址移动；每次出栈操作时，栈指针向高地址移动。
2. 栈指针寄存器：在ARM Cortex-M0中，寄存器**r13**被用作栈指针，用于追踪栈顶的位置。
3. 入栈操作（Push）：要将链接寄存器（Link Register，LR或r14）的值推入递减栈，使用的指令助记符是`PUSH {Ir}`，其中`Ir`代表要入栈的寄存器列表。例如，如果要将链接寄存器（LR）和寄存器r0的值推入栈，可以使用以下指令：`PUSH {r0, lr}`。这个指令会先将链接寄存器的值入栈，然后将r0的值入栈。
4. 出栈操作（Pop）：要将值从递减栈弹出到链接寄存器，使用的指令助记符是`POP {Ir}`，其中`Ir`代表要出栈到的寄存器列表。例如，如果要将栈中的值弹出到链接寄存器（LR）和寄存器r0，可以使用以下指令：`POP {r0, lr}`。这个指令会先将栈顶的值弹出到r0，然后将下一个值弹出到链接寄存器。

总之，ARM Cortex-M0处理器使用递减栈来存储临时数据。栈指针寄存器r13用于追踪栈顶的位置。入栈操作使用`PUSH {Ir}`指令，将链接寄存器等寄存器的值推入栈；出栈操作使用`POP {Ir}`指令，将值从栈弹出到链接寄存器等寄存器。这些指令在函数调用和返回时，以及在保存和恢复寄存器值时，起到重要作用。

Trumb

BXL BL BX的区别

Lec 28

Memory
Memory can be defined as any device that can store information.
In the context of microprocessors, memory is a device that can store information in a binary format.

Silicon IC memory can be classified as either read-only or read-write.
Read-only memory (or ROM) is used for:

- BIOS/UEFI on a desktop computer
- Operating system in an embedded system
- Store of cryptographic data to improve security

Read-write memory is used for:

- Operating system(s) and application(s) on a desktop computer

- Application(s) in an embedded system
- Temporary data

Types of ROM
There are several generations of silicon ROM:

- P rogrammable ROM (PROM) could be electrically programmed once only and then the contents were fixed.
- Erasable PROM (EPROM) could also have its contents erased by exposure to UV light and could then be reprogrammed.
- Electrically erasable PROM (EEPROM) could lhave its contents erased by an electrical signal.

In contrast to ROM, the contents of RAM memory will be lost almost
as soon as power is switched off - RAM is volatile.
The acronym RAM stands for Random Access Memory
The time taken to read or write data from or to the
memory (the 'access time") does not depend upon the
order in which the data is accessed.
d
h
h
The memory locations can be accessed randomly or
B
h
are r
sequentially in the same time. This is not true for hard
This Photo by Unknown Author
disks for example.
is licensed under CC BY-NC
There are two types of RAM: Static and Dynamic RAM