CS1021 Introduction to Computing I
5. Memory Again

Rebekah Clarke
clarker7@scss.tcd.ie

Simple Model of a Microprocessor System

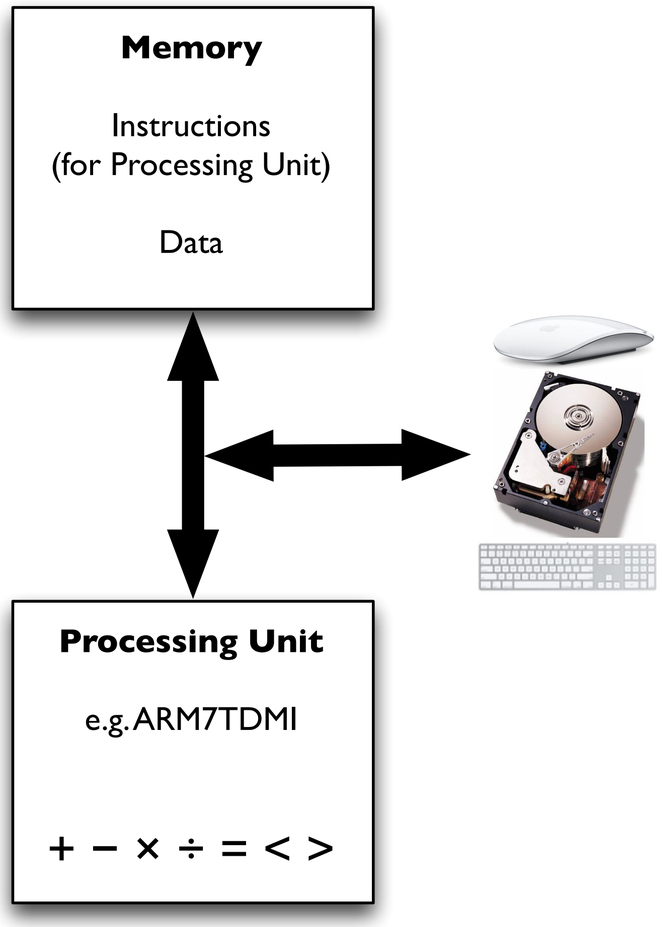
A **processing unit** or **processor** which performs operations on data

Memory, which stores:

Data: representing text, images, videos, sensor readings, π, audio, etc. ...

Instructions: Programs are composed of sequences of instructions that control the actions of the processing unit

So far, all of our data has been stored in registers, internal to the Processing Unit ("processor" or "CPU")



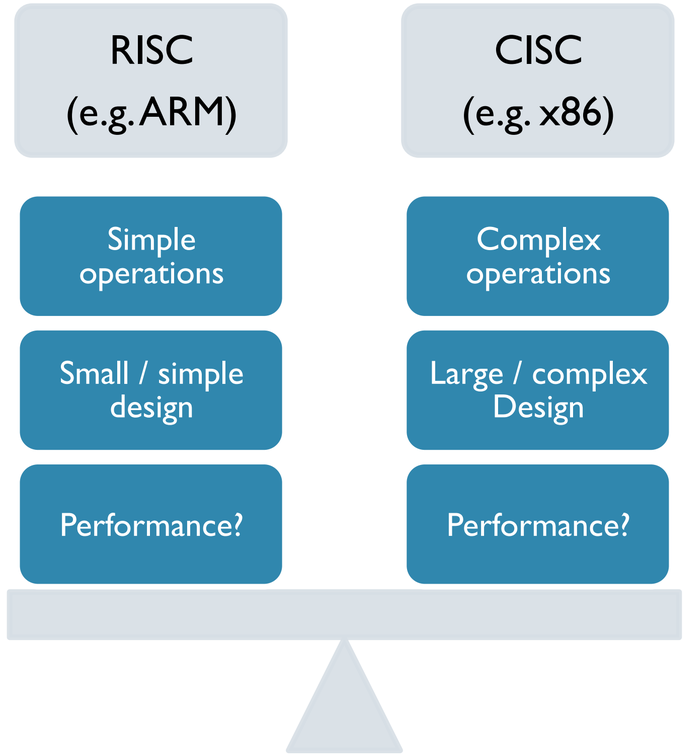
Load – Store Architecture

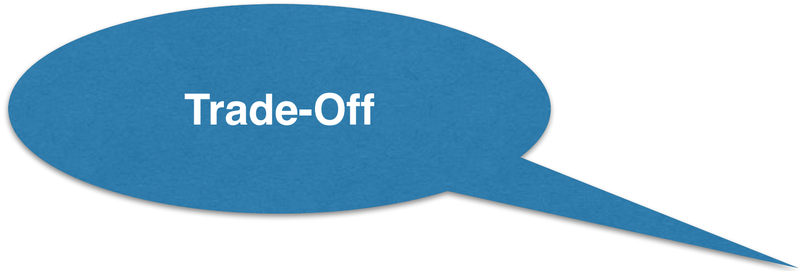
ARM7TDMI is based on a Load – Store Architecture

Cannot directly perform operations (e.g. addition, subtraction, comparison, …) on values in memory

Only way to operate on a value stored in memory is to load it into a register, then operate on the register

Only way to change a value in memory is to store the value from a register into memory





Using Memory: Upper Case String Example

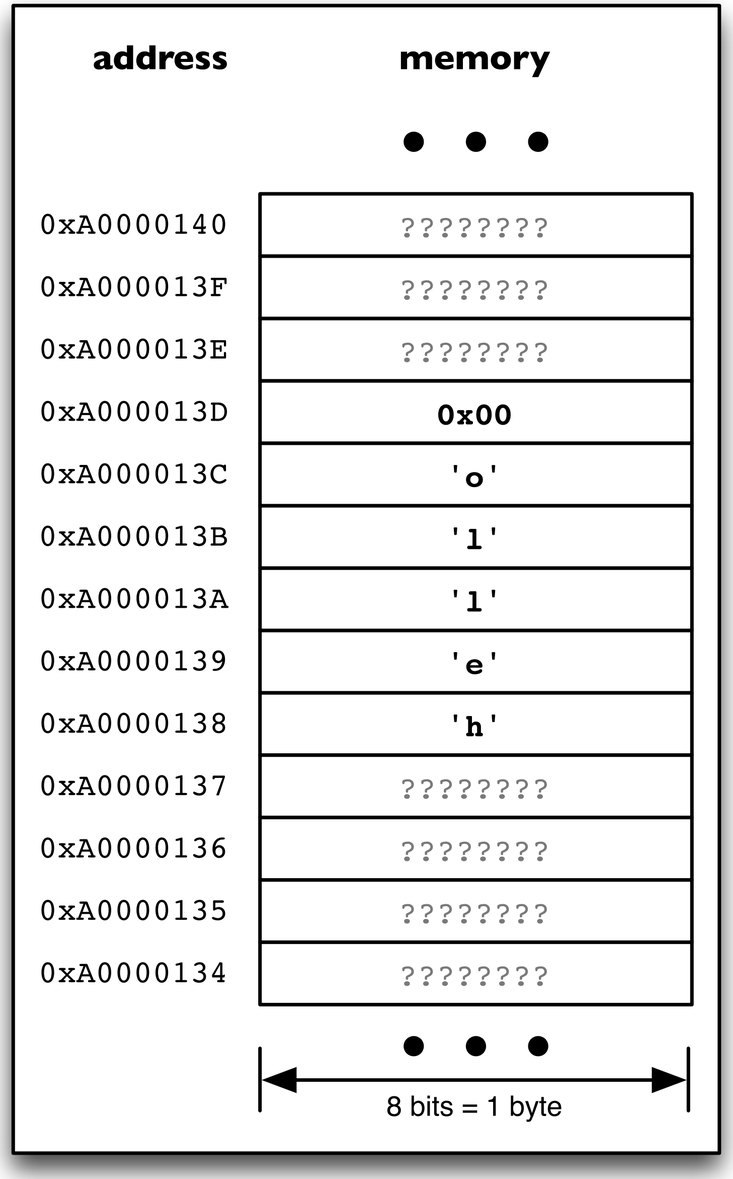
Design and write an assembly language program to convert a string stored in memory to UPPER CASE

String – sequence of ASCII characters stored in consecutive memory locations

```
char = first character in string

while (char not past end of string)
{
        if (char ≥ ‘a’ AND char ≤ ‘z’)
        {
              char = char - 0x20
        }

        char = next character
}
```



Using Memory: Upper Case String Example


```
char = first character in string

while (char not past end of string)
{
        if (char ≥ ‘a’ AND char ≤ ‘z’)
        {
                char = char - 0x20
        }

        char = next character
}
```

refine

```
address = address of first character
char = Memory.byte[address]

while (char not past end of string)
{
        if (char ≥ ‘a’ AND char ≤ ‘z’)
        {
                char = char – 0x20
                Memory.byte[address] = char
        }

        address = address + 1
        char = Memory.byte[address]
}
```

`char = Memory.byte[address]`

Load the byte-size contents of memory at address **`address`** *into the variable* **`char`**

My pseudo-code notation … you are free to use your own!

Using Memory: Upper Case String Example

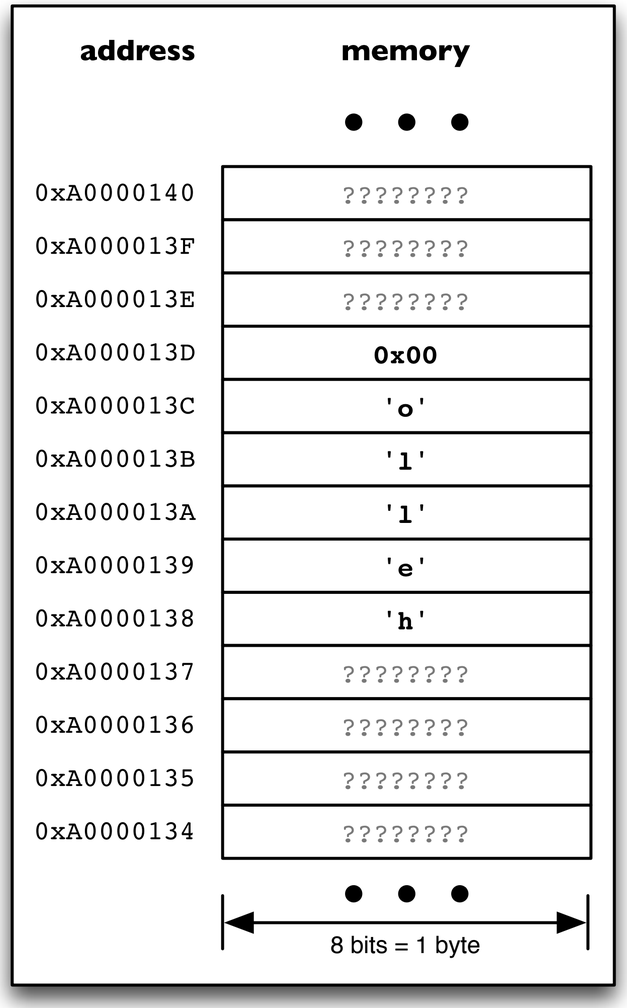
How do we know when we have reached the end of the string?

NULL terminated strings use the code 0 (ASCII NULL character code) to denote the end of a string

```
address = address of first character
char = Memory.byte[address]

while (char ≠ 0)
{
        if (char ≥ ‘a’ AND char ≤ ‘z’)
        {
              char = char - 0x20
              Memory.byte[address] = char
        }

        address = address + 1
        char = Memory.byte[address]
}
```



Using Memory: Upper Case String Example


```
        LDR   r1, =0xA1000138     ; address = 0xA1000138

        LDRB  r0, [r1]            ; char = Memory.byte[address]

whStr   CMP   r0, #0              ; while ( char != 0 )
        BEQ   eWhStr              ; {
        CMP   r0, #'a'            ;  if (char >= 'a'
        BLO   endifLC             ;   AND
        CMP   r0, #'z'            ;   char <= 'z')
        BHI   endifLC             ;  {
        SUB   r0, r0, #0x20       ;   char = char – 0x20
        STRB  r0, [r1]            ;   Memory.byte[address] = char
endifLC                           ;  }
        ADD   r1, r1, #1          ;  address++;
        LDRB  r0, [r1]            ; char = Memory.byte[address]
        B     whStr               ; }
eWhStr
```

Need to use µVision to initialise memory with a test string

Using Memory: Upper Case String Example

Possible optimisation by moving the LDRB to the top of the while loop … at the expense of less elegant pseudo-code …

```
        LDR   r1, =0xA1000138    ; address = 0xA1000138

whStr   LDRB  r0, [r1]           ; while ( (char = Memory.byte[address])
        CMP   r0, #0             ;               != 0 )
        BEQ   eWhStr             ; {
        CMP   r0, #'a'           ;  if (char >= 'a'
        BLO   endifLC            ;   AND
        CMP   r0, #'z'           ;   char <= 'z')
        BHI   endifLC            ;  {
        SUB   r0, r0, #0x20      ;   char = char – 0x20
        STRB  r0, [r1]           ;   Memory.byte[address] = char
endifLC                          ;  }
        ADD   r1, r1, #1         ;  address++;
        B     whStr              ; }
eWhStr
```

LDR / STR Instructions

Load a word-, half-word- or byte-size value from a specified address into a register

LDR	load word
LDRH	load half-word
LDRB	load byte

Store a word-, half-word- or byte-size value from a register into memory at a specified address

STR	store word
STRH	store half-word
STRB	store byte

Using Memory: Sum Example

Design and write an assembly language program that will calculate the sum of 10 word-size values stored in memory

```
address = address of first word-size value
sum = 0
count = 0;

while (count < 10)
{
      sum = sum + Memory.word[address]
      address = address + 4
      count = count + 1
}
```

Using Memory: Sum Example


```
start
        LDR    R1, =testdata          ; address = address of first word-wize value
        LDR    R0, =0                 ; sum = 0
        LDR    R4, =0                 ; count = 0

whSum   CMP    R4, #10                ; while (count < 10)
        BHS    eWhSum                 ; {
        LDR    R5, [R1]               ;  num = Memory.byte[address]
        ADD    R0, R0, R5             ;  sum = sum + num
        ADD    R1, R1, #4             ;  address = address + 4
        ADD    R4, R4, #1             ;  count = count + 1
        B      whSum                  ; }
eWhSum                                ;

stop    B      stop


AREA    TestData, DATA, READWRITE

        ; sequence of 10 word-size values
testdata
        DCD    56,23,407,298,4,75,84,37,92,43
```

DCD, DCW and DCB Assembler Directives

Use the assembler to initialise contents of memory

Example: instead of manually writing a test string into memory, the string can be included with program machine code by the assembler

```
        AREA   UpperCaseString, CODE, READONLY
        IMPORT main
        EXPORT start

start
        LDR    r1, =teststr             ; address = teststr

        ...    ...                      ...
        <rest of program>
        ...    ...                      ...

        AREA   TestData, DATA, READWRITE
teststr DCB    "hello",0                ; NULL terminated test string

        END
```

DCD, DCW and DCB Assembler Directives

DCD, DCW and DCB are assembler directives. They are not instructions and no machine code is produced.

Other data declaration examples

8 word values

```
mywords
        DCD  0x4D1F4004, 0x10301030, 0x141030D4, 0xE4503003
        DCD  0x4AB345F0, 0x3049FDEA, 0x0400D4F8, 0x34FD303A
```

Lotto numbers as byte values

```
draw  DCB  32, 43, 10, 11, 14, 15
bonus DCB  7
```

2 half-word values

```
values     DCW                 407, -208
```

Other Assembler Directives

AREA directive

- Marks the beginning of a section and specifies attributes for the section
- Sections are indivisible sequences of instructions and/or data
- Attribute examples: CODE, READONLY, DATA, READWRITE
- Attributes define how a section is loaded into memory
- Programs must contain at least one CODE section

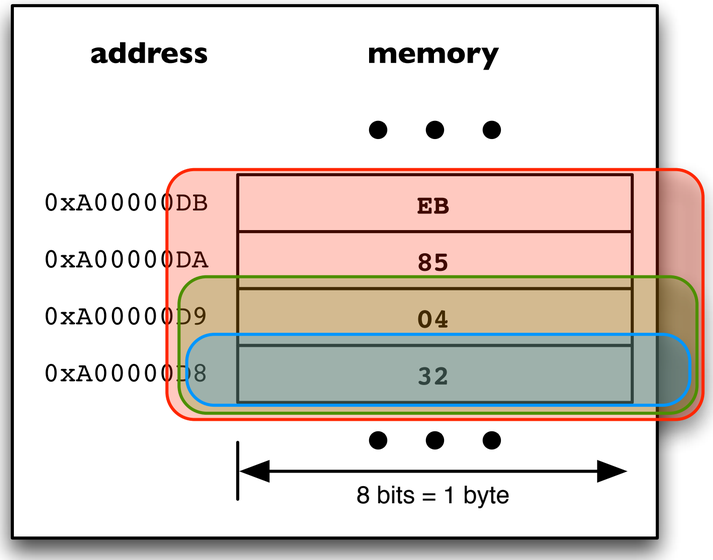
END directive

- Tells the assembler to stop processing the source file

IMPORT / EXPORT directives

- EXPORT directive exports labels for use by other assemblies
- IMPORT directive allows one assembly to use a label exported by another assembly

Load / Store Bytes, Half-words and Words



Byte, **half-word** and **word** at address 0xA00000D8

```
LDR  r0, =0xA00000D8
LDRB r1, [r0]
```


```
LDR  r0, =0xA00000D8
LDRH r1, [r0]
```


```
LDR  r0, =0xA00000D8
LDR  r1, [r0]
```

Alignment

ARM7TDMI expects all memory accesses to be **aligned**

Examples

Word aligned	`0x00000000, 0x00001008, 0xA100000C`
Not word aligned	`0x00000001, 0x00001006, 0xA100000F`
Half-word aligned	`0x00000000, 0x00001002, 0xA100000A`
Not half-word aligned	`0x00000003, 0x00001001, 0xA100000B`

See ARM Architecture Reference Manual Section A2.8

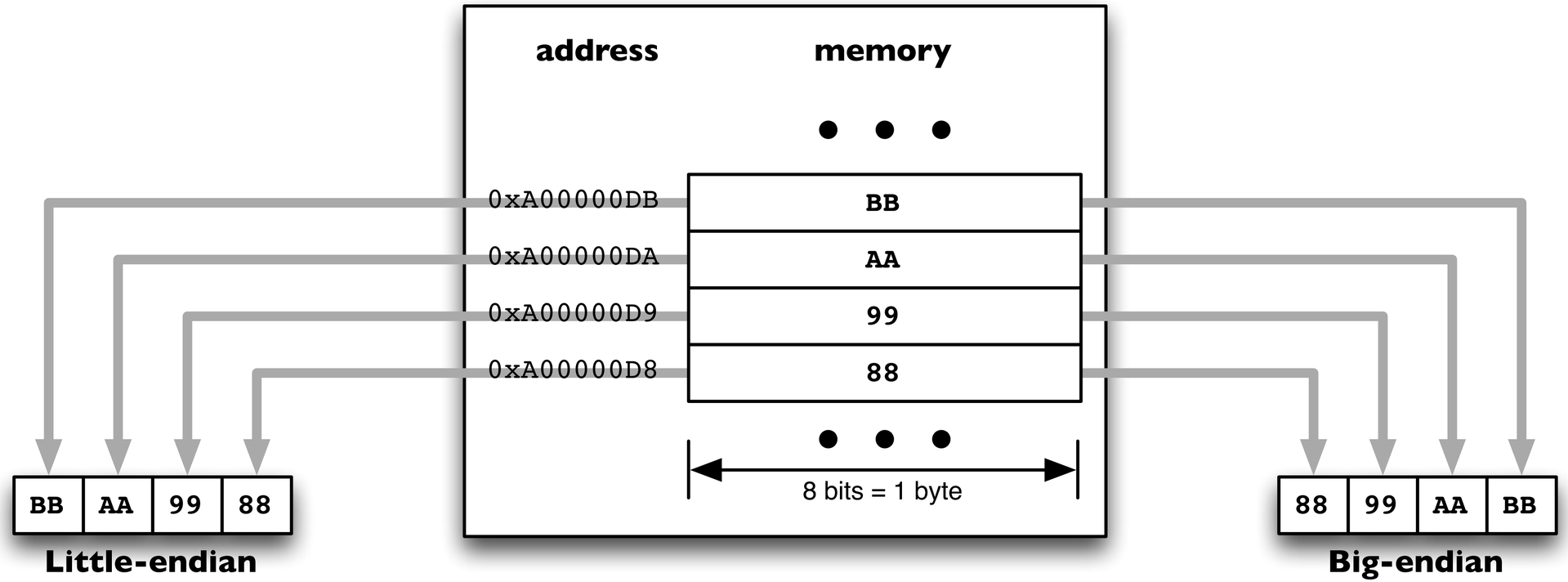
Unaligned accesses are permitted but the result is unlikely to be what was intended

Unaligned accesses are supported by later ARM architecture versions

Endian-ness

Little-endian byte ordering – least-significant byte of word or half-word stored at lower address in memory

Big-endian byte ordering – most-significant byte of word or half-word stored at lower address in memory



Loading Signed Bytes and Half-words

Sign extension performed when loading signed bytes or half-words to facilitate correct subsequent 32-bit signed arithmetic

```
LDR     r0, =0xA00000D8
LDRSB   r1, [r0]
```

00	00	00	32

```
LDR    r0, =0xA00000D8
LDRSH r1, [r0]
```

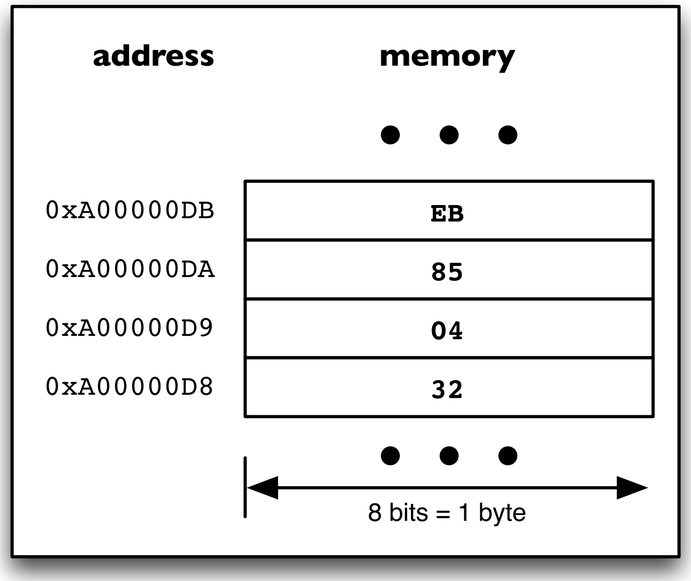
00	00	04	32

```
LDR     r0, =0xA00000DA
LDRSB   r1, [r0]
```

FF	FF	FF	85

```
LDR    r0, =0xA00000DA
LDRSH r1, [r0]
```

FF	FF	EB	85



Example: String Comparison

Design and write an ARM Assembly Language program to compare two strings stored in memory. The program should store a 0 in R0 if the strings are the same

Suggested approach

- Consider some examples to explore the problem
- Develop with a pseudo-code solution
- Translate the pseudo-code solution to ARM Assembly Language

Example: String Comparison


```
ch1 = Memory.byte[adr1];
ch2 = Memory.byte[adr2];

while(ch1 != 0 && ch1 == ch2)
{
   adr1++;
   adr2++;
   ch1 = Memory.byte[adr1];
   ch2 = Memory.byte[adr2];
}

result = ch1 - ch2;
```

Example: String Comparison


```
start
        LDR   R4, =str1              ; adr1 = start address of str1
        LDR   R5, =str2              ; adr2 = start address of str2
        LDRB  R6, [R4]               ; ch1 = Memory.byte[adr1]
        LDRB  R7, [R5]               ; ch2 = Memory.byte[adr2]

whCmp   CMP   R6, #0                 ; while (ch1 != NULL
        BEQ   ewhCmp                 ;         &&
        CMP   R6, R7                 ;         ch1 != ch2)
        BNE   ewhCmp                 ; {
        ADD   R4, R4, #1             ;  adr1++
        ADD   R5, R5, #1             ;  adr2++
        LDRB  R6, [R4]               ;  ch1 = Memory.byte[adr1]
        LDRB  R7, [R5]               ;  ch2 = Memory.byte[adr2]
        B     whCmp                  ; }
ewhCmp
        SUB   R0, R6, R7             ; result = ch1 - ch2

stop    B     stop

        AREA  Strings, DATA, READWRITE
str1    DCB   "Beets",0
str2    DCB   "Bests",0
```

Example: Subset (mathematics, set theory)

Design and write an ARM Assembly Language program to determine if a set, A, is a subset of another set, B. Sets A and B are stored in memory as unordered sequences of unique word-size values, along with the size of each set.

```
        AREA  Sets, DATA, READWRITE

Asize   DCD   3
Aelems  DCD   6, 10, 8
Bsize   DCD   8
Belems  DCD   3, 14, 8, 6, 7, 10, 12, 14
```