Simple Model of a Microprocessor System

- **A Processing Unit** 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
- Other peripheral devices
  - e.g. displays, USB ports, network devices, keyboards, mice, ...
Program 7.1 – Upper Case String

- 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

```c
char = first character in string
while (char not past end of string) {
    if (char ≥ ‘a’ AND char ≤ ‘z’) {
        char = char AND 0xFFFFFFDF
    }
    char = next character
}
```
**Load – Store Architecture**

- ARM5vTE 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

<table>
<thead>
<tr>
<th>RISC (e.g. ARM)</th>
<th>CISC (e.g. x86)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple operations</td>
<td>Complex operations</td>
</tr>
<tr>
<td>Small / simple design</td>
<td>Large / complex Design</td>
</tr>
<tr>
<td>More operations per unit time</td>
<td>Fewer operations per unit time</td>
</tr>
</tbody>
</table>

**Trade-off**
Refine pseudo-code solution

\begin{align*}
\text{char} &= \text{first character in string} \\
\text{while} \ (\text{char not past end of string}) \\
&\quad \{ \\
&\quad \quad \text{if} \ (\text{char} \geq \text{‘a’} \ \text{AND} \ \text{char} \leq \text{‘z’}) \\
&\quad \quad \quad \{ \\
&\quad \quad \quad \quad \text{char} = \text{char} \ \text{AND} \ 0xFFFFFFFFDF \\
&\quad \quad \quad \} \\
&\quad \quad \text{char} = \text{next character} \\
&\quad \} \\
\end{align*}

\begin{align*}
\text{address} &= \text{address of first character} \\
\text{char} &= \text{Memory.byte}[\text{address}] \\
\text{while} \ (\text{char not past end of string}) \\
&\quad \{ \\
&\quad \quad \text{if} \ (\text{char} \geq \text{‘a’} \ \text{AND} \ \text{char} \leq \text{‘z’}) \\
&\quad \quad \quad \{ \\
&\quad \quad \quad \quad \text{char} = \text{char} \ \text{AND} \ 0xFFFFFFFFDF \\
&\quad \quad \quad \quad \text{Memory.byte}[\text{address}] = \text{char} \\
&\quad \quad \} \\
&\quad \quad \text{address} = \text{address} + 1 \\
&\quad \quad \text{char} = \text{Memory.byte}[\text{address}] \\
&\quad \} \\
\end{align*}
How do we know when we have reached the end of the string?

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

```c
address = address of first character
char = Memory.byte[address]
while (char ≠ 0) {
    if (char ≥ ‘a’ AND char ≤ ‘z’) {
        char = char AND 0xFFFFFFDF
        Memory.byte[address] = char
    }
    address = address + 1
    char = Memory.byte[address]
}
```
Load / Store Instructions

- Load – Rd = Memory.<size>[<address>]
  - Load a word-, half-word- or byte- <size> value ...
  - ... from Memory at a specified <address> ...
  - ... into a register
  - LDR Rd, <address> load word
  - LDRH Rd, <address> load half-word
  - LDRB Rd, <address> load byte

- Store – Memory.<size>[<address>] = Rd
  - Store a a word-, half-word- or byte- <size> value ...
  - ... from a register ...
  - ... into Memory at a specified <address>
  - STR Rd, <address> store word
  - STRH Rd, <address> store half-word
  - STRB Rd, <address> store byte
Memory Map

- Flash (16 MB): 0x0000_0000 to 0x0100_0000
- RAM (64 MB): 0xA000_0000 to 0xA400_0000
R0=00000000 R1=00000000 R2=00000000 R3=00000000
R4=00000000 R5=00000000 R6=00000000 R7=00000000
R8=00000000 R9=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=00000000
PSR=400000d3 -Z--

> 3 0000 0810A0E3
MOV r1, #0x8
MOV r2, #0x6
LDR r11, =0xA1000400
STRB r1, [r11], #1
STRB r2, [r11], #1

0xA1000400: 00 00 00 00 00 00 00 00 00 00 00
0xA1000408: 00 00 00 00 00 00 00 00 00 00 00
0xA1000410: 00 00 00 00 00 00 00 00 00 00 00
0xA1000418: 00 00 00 00 00 00 00 00 00 00 00
0xA1000420: 00 00 00 00 00 00 00 00 00 00 00
0xA1000428: 00 00 00 00 00 00 00 00 00 00 00
0xA1000430: 00 00 00 00 00 00 00 00 00 00 00
0xA1000438: 00 00 00 00 00 00 00 00 00 00 00

0x014 FEFFFE
stop: B stop

13 0018 3247336E str: .asciz "2G3niwo"
17 0020 000400A1
Addressing mode – method of specifying the `<address>` to be used in a load / store operation

Address used is called the effective address

Immediate Offset (with a default zero offset)

- Rn is the base register

\[ \text{<address> = Rn} \]

Example: load word-size value from memory at address 0xA1000000 into register r1

- \( r1 = \text{Memory.word[0xA1000000]} \)

```
LDR r0, =0xA1000000@ Initialise base register r0=0xA1000000
LDR r1, [r0] @ r1 = Memory.word[r0]
```
### Addressing Mode \([Rn, \#0]\) or \([Rn]\)

- **Example@** Store word-size value from register R1 into memory at address 0xA1000000
  - Memory[0xA1000000] = r1

  ```
  LDR r0, =0xA1000000@ Initialise base register r0=0xA10000000
  STR r1, [r0] @ Memory.word[r0] = r1
  ```

- **Example@** Load byte-size value from register r1 into memory at address 0xA1000000
  - r1 = Memory.byte[0xA1000000]

  ```
  LDR r0, =0xA1000000@ Initialise base register r0=0xA10000000
  LDRB r1, [r0] @ r1 = Memory.byte[r0]
  ```
while construct evaluates condition at end of loop with an initial branch to the evaluation – more efficient

Use GDB to initialise memory with a test string

Program 7.1a – Upper Case String

```
start:
    LDR    r1, =0xA1000000@ address = 0xA1000000
          @ while ( (char = Memory.byte[address]) != 0 ) {
    B      testwhl1 @
    wh1:
        CMP    r0, #'a' @ if (char >= 'a'
        BCC    endif1 @ AND
        CMP    r0, #'z' @ char <= 'z')
        BHI    endif1 @
        BIC    r0, #0x00000020 @ char = char AND NOT 0x00000020
        STRB   r0, [r1] @ Memory.byte[address] = char
    endif1:
          @}
        ADD    r1, r1, #1 @ address = address + 1
    testwhl1: LDRB  r0, [r1] @}
        CMP    r0, #0 @
        BNE    wh1 @}
    stop:   B      stop
```
Indirect Addressing

- Storing an address in a register and subsequently using the register as an operand in a load/store operation is an example of **indirection**

- Indirection is an important concept in Computing generally, not just assembly language programming

```
LDR r0, =0xA1000000@ Initialise base register r1=0xA1000000
LDRB r1, [r0] @ r1 = Memory.byte[r0]
```

- We can say r0 **"points to"** the data in memory at address 0xA1000000

- In some contexts (e.g. high-level programming languages such as C or C++) r0 could be referred to as a **"pointer"** to some data
Byte, half-word, word
at address 0xA1000044

LDR r0, =0xA1000044
LDRB r1, [r0]

LDR r0, =0xA1000044
LDRH r1, [r0]

LDR r0, =0xA1000044
LDR r1, [r0]
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, =0xA1000044
LDRSB r1, [r0]

LDR r0, =0xA1000044
LDRSH r1, [r0]

LDR r0, =0xA1000046
LDRSB r1, [r0]

LDR r0, =0xA1000046
LDRSH r1, [r0]

Remember@ interpretation!!
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

```
.text

start:

stop:   B stop

TestStr: .asciz "123aBc12"
.end
```
GNU ARM Assembler Quick Reference

A summary of useful commands and expressions for the ARM architecture using the GNU assembler is presented briefly in the concluding portion of this Appendix. Each assembly line has the following format:

```
[<label>:] [<instruction or directive>] @ comment
```

Unlike the ARM assembler, using the GNU assembler does not require you to indent instructions and directives. Labels are recognized by the following colon instead of their position at the start of a line. An example follows showing a simple assembly program defining a function ‘add’ that returns the sum of two input arguments:

```
.section .text, "x"

.global add @ give the symbol add external linkage

add:
    ADD r0, r0, r1 @ add input arguments
    MCV pc, lr @ return from subroutine

@ end of program
```

GNU Assembler Directives for ARM

The follow is an alphabetical listing of the more common GNU assembler directives.

<table>
<thead>
<tr>
<th>GNU Assembler Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.asci &quot;&lt;string&gt;&quot;</td>
<td>Inserts the string as data into the assembly (like DCB in armasm).</td>
</tr>
<tr>
<td>.asciz &quot;&lt;string&gt;&quot;</td>
<td>Like .ascii, but follows the string with a zero byte.</td>
</tr>
<tr>
<td>.balign &lt;power_of_2&gt; [,&lt;fill_value&gt; [,&lt;max_padding&gt;]]</td>
<td>Aligns the address to &lt;power_of_2&gt; bytes. The assembler aligns by adding bytes of value &lt;fill_value&gt; or a suitable default. The alignment will not occur if more than &lt;max_padding&gt; fill bytes are required (similar to ALIGN in armasm).</td>
</tr>
<tr>
<td>.byte &lt;byte1&gt; {,&lt;byte2&gt;} ...</td>
<td>Inserts a list of byte values as data into the assembly (like DCB in armasm).</td>
</tr>
<tr>
<td>.code &lt;number_of_bits&gt;</td>
<td>Sets the instruction width in bits. Use 16 for Thumb and 32 for ARM assembly (similar to CODE16 and CODE32 in armasm).</td>
</tr>
<tr>
<td>.byte {&lt;byte1&gt; &lt;byte2&gt;}</td>
<td>Use with .byte (similar to .L0F in armasm).</td>
</tr>
</tbody>
</table>
ARM expects all memory accesses to be **aligned**

**Examples**

<table>
<thead>
<tr>
<th>Alignment</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word-aligned</td>
<td>0x00000000, 0x00001008, 0xA100000C</td>
</tr>
<tr>
<td>Not word-aligned</td>
<td>0x00000001, 0x00001006, 0xA100000F</td>
</tr>
<tr>
<td>Half-word aligned</td>
<td>0x00000000, 0x00001002, 0xA100000A</td>
</tr>
<tr>
<td>Not half-word aligned</td>
<td>0x00000003, 0x00001001, 0xA100000B</td>
</tr>
</tbody>
</table>

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
Unaligned Access Example

- Load word at address 0xA10000DB

- address is rounded down to nearest aligned address

- loaded value is rotated right 3 bytes before being stored in destination register
Addressing Mode [Rn], #+/-<offset>

- Immediate post-indexed

- After calculating the effective address, the immediate value <offset> is added/subtracted to/from the base register Rn
- Convenient way of updating base register to point to address of next value in memory

Example @ load three consecutive half-word values, beginning at address 0xA1001000, into registers r0, r1 and r2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LDR r0, =0xA1001000</code></td>
<td>Initialise base register r1 = 0xA1001000</td>
</tr>
<tr>
<td><code>LDRH r1, [r0], #2</code></td>
<td>r1 = Memory.halfword[r0], r0 = r0 + 2</td>
</tr>
<tr>
<td><code>LDRH r2, [r0], #2</code></td>
<td>r2 = Memory.halfword[r0], r0 = r0 + 2</td>
</tr>
<tr>
<td><code>LDRH r3, [r0], #2</code></td>
<td>r3 = Memory.halfword[r0], r0 = r0 + 2</td>
</tr>
</tbody>
</table>
Program 7.1b – Upper Case String

```
start:   LDR   r1, =teststr     @ address = 0xA1000000
        @ while ( (char = Memory.byte[address])
        B      testwh1     @      != 0 )
wh1:    CMP   r0, #'a'        @   if (char >= 'a'
        BCC   endif1      @   AND
        CMP   r0, #'z'      @   char <= 'z')
        BHI   endif1      @
        BIC   r0, #0x00000020@   char = char AND NOT 0x00000020
        SUB   r2, r1, #1     @   store_address = address - 1
        STRB  r0, [r2]      @   Memory.byte[store_address] = char
endif1: @
testwh1: LDRB  r0, [r1], #1 @
        CMP   r0, #0        @
        BNE   wh1           @
stop:    B     stop

teststr: .asciz  "hello"     @ NULL terminated test string
.end
```
Immediate Offset (with non-zero offset)

- Effective address is calculated by adding <offset> to the value in the base register Rn
- Base register Rn is not changed

Example: Load three consecutive word-size values, beginning at address 0xA1001000, into registers r1, r2 and r3.

```
LDR r0, =0xA1001000@ Initialise base register r1 = 0xA1001000
LDR r1, [r0]       @ r1 = Memory.word[r0 + 0] (default = 0)
LDR r2, [r0, #4]   @ r2 = Memory.word[r0 + 4]
LDR r3, [r0, #8]   @ r3 = Memory.word[r0 + 8]
```
Program 7.1c – Upper Case String

```
.start:       LDR    r1, =teststr    @ address = teststr
             @ while ( (char = Memory.byte[address])
             B      testwh1    @        != 0 ) {
.wh1:        CMP    r0, #'a'    @        if (char >= 'a'
             BCC    endif1    @        AND
             CMP    r0, #'z'    @        char <= 'z')
             BHI    endif1    @        
             BIC    r0, #0x00000020    @        char = char AND NOT 0x00000020
             STRB   r0, [r1, #-1]    @        Memory.byte[address - 1] = char
.endif1:     @        
.testwh1:    LDRB   r0, [r1], #1    @        }
             CMP    r0, #0    @        }
             BNE    wh1    @        stop:
             B      stop

.stop:       B      stop

.teststr    .asciz   "hello"    @ NULL terminated test string
.end
```
Register offset

- Effective address is calculated by adding offset register \( R_m \) to base register \( R_n \)
- \( R_n \) and \( R_m \) are not changed

Example: Load three consecutive half-word values, beginning at address 0xA1001000, into registers \( r_1 \), \( r_2 \) and \( r_3 \)

```
LDR r0, =0xA1001000
LDR r4, =0
LDRH r1, [r0, r4]
ADD r4, r4, #2
LDRH r2, [r0, r4]
ADD r4, r4, #2
LDRH r3, [r0, r4]
```
Program 7.1d – Upper Case String

start:
  LDR r1, =teststr @ address = teststr
  LDR r2, =0 @ offset = 0

B testwh1 @ while ( (char = Memory.byte[address + offset]) != 0 ) {
wh1:
  CMP r0, #'a' @ if (char >= 'a'
  BCC endif1 @ AND
  CMP r0, #'z' @ char <= 'z')
  BHI endif1 @ {
  BIC r0, #0x00000020 @ char = char AND NOT 0x00000020
  STRB r0, [r1, r2] @ Memory.byte[address + offset] = char
  endif1:
    ADD r2, r2, #1 @ offset = offset + 1
  testwh1:
    LDRB r0, [r1, r2] @ }
  CMP r0, #0 @
  BNE wh1 @

stop:  B stop
Design and write an assembly language program that will calculate the sum of 100 word-size values stored in memory.

Program 7.2a – Sum

```
sum = 0
do {
    sum = sum + Memory.word[address + offset]
    offset = offset + 4
} while (offset < 400)
```

Option 1
Use offset for loop condition

```
sum = 0
count = 100
do {
    sum = sum + Memory.word[address + offset]
    offset = offset + 4
    count = count - 1
} while (count > 0)
```

Option 2
Use separate counter for loop condition
Program 7.2a – Sum

.equ MAX, 100

start:
  MOV R0, #0 @ sum = 0
  LDR R1, =nums @ address = nums
  MOV R2, #0 @ offset = 0
  LDR R3, =MAX @ count = MAX
  @ do {
  do1:  LDR R4, [R1, R2] @ tmp = Memory.word[address + offset]
        ADD R0, R0, R4 @ sum = sum + tmp
        ADD R2, R2, #4 @ offset = offset + 4
        SUBS R3, R3, #1 @ count = count - 1
        BHI do1 @ } while (count > 0)

stop:  B stop

nums:
  .4byte 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
... ...
Scaled register offset

- Effective address is calculated by adding (offset register \( R_m \) shifted by count bits) to base register \( R_n \)
- Shift operation can be LSL, LSR, ASR, ROR, RRX
- \( R_n \) and \( R_m \) are not changed

Example@ load three consecutive word values, beginning at address 0xA1001000, into registers \( r_1, r_2 \) and \( r_3 \)

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>[( R_n, +/-R_m, \text{shift #count} ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>(&lt;\text{address}&gt; = R_n +/- R_m \text{ shift count})</td>
</tr>
</tbody>
</table>

LDR \( r_0, =0xA1001000\)@ Initialise base register \( r_1 = 0xA1000000 \)
LDR \( r_4, =0 \) @ Initialise offset register \( r_4 = 0 \)
LDR \( r_1, [r_0, r_4, \text{LSL \#2}] \) @ \( r_0 = \text{Memory.word}[r_1 + (r_4 \times 4)] \)
ADD \( r_4, r_4, \#1 \) @ \( r_4 = r_4 + 1 \)
LDR \( r_2, [r_0, r_4, \text{LSL \#2}] \) @ \( r_0 = \text{Memory.word}[r_1 + (r_4 \times 4)] \)
ADD \( r_4, r_4, \#1 \) @ \( r_4 = r_4 + 1 \)
LDR \( r_3, [r_0, r_4, \text{LSL \#2}] \) @ \( r_0 = \text{Memory.word}[r_1 + (r_4 \times 4)] \)
Program 7.2b – Sum

.equ MAX, 100

start:
MOV R0, #0 @ sum = 0
LDR R1, =nums @ address = nums
MOV R2, #0 @ offset = 0
@ do {

do1:
LDR R3, [R1, R2, LSL #2] @ tmp = Memory.word[address + offset * 4]
ADD R0, R0, R3 @ sum = sum + tmp
ADD R2, R2, #1 @ offset = offset + 1
CMP R2, #MAX @ } while (offset < MAX)
BCC do1 @

stop:
B stop

... ...

.end
### Summary of LDR/STR Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Syntax</th>
<th>W, B</th>
<th>H, SH, SB</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Offset</td>
<td>[&lt;Rn&gt;, #+/--&lt;offset&gt;]</td>
<td>✓</td>
<td>✓</td>
<td>address ← Rn +/- offset</td>
</tr>
<tr>
<td>Register Offset</td>
<td>[&lt;Rn&gt;, +/-&lt;Rm&gt;]</td>
<td>✓</td>
<td>✓</td>
<td>address ← Rn +/- Rm</td>
</tr>
<tr>
<td>Scaled Register Offset</td>
<td>[&lt;Rn, +/-&lt;Rm&gt;, &lt;shift&gt; #&lt;count&gt;]</td>
<td>✓</td>
<td></td>
<td>address ← Rn +/- (Rm &lt;shift&gt; &lt;count&gt;)</td>
</tr>
<tr>
<td>Immediate Pre-Indexed</td>
<td>[&lt;Rn&gt;, #+/--&lt;offset&gt;]!</td>
<td>✓</td>
<td>✓</td>
<td>Rn ← Rn +/- offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address ← Rn</td>
</tr>
<tr>
<td>Register Pre-Indexed</td>
<td>[&lt;Rn&gt;, +/-&lt;Rm&gt;]!</td>
<td>✓</td>
<td>✓</td>
<td>Rn ← Rn +/- Rm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address ← Rn</td>
</tr>
<tr>
<td>Scaled Register Pre-Indexed</td>
<td>[&lt;Rn, +/-&lt;Rm&gt;, &lt;shift&gt; #&lt;count&gt;]!</td>
<td>✓</td>
<td></td>
<td>Rn ← Rn +/- (Rm &lt;shift&gt; &lt;count&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address ← Rn</td>
</tr>
<tr>
<td>Immediate Post-Indexed</td>
<td>[&lt;Rn&gt;], #+/--&lt;offset&gt;</td>
<td>✓</td>
<td>✓</td>
<td>address ← Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn ← Rn +/- offset</td>
</tr>
<tr>
<td>Register Post-Indexed</td>
<td>[&lt;Rn&gt;], +/-&lt;Rm&gt;</td>
<td>✓</td>
<td>✓</td>
<td>address ← Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn ← Rn +/- Rm</td>
</tr>
<tr>
<td>Scaled Register Post-Indexed</td>
<td>[&lt;Rn], +/-&lt;Rm&gt;, &lt;shift&gt; #&lt;count&gt;</td>
<td>✓</td>
<td></td>
<td>address ← Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn ← Rn +/- (Rm &lt;shift&gt; &lt;count&gt;)</td>
</tr>
</tbody>
</table>

- All modes are available for loads/stored of words and unsigned bytes
- A subset of the modes are available for loads/stores of unsigned half-words, signed half-words and signed bytes
- Note 12-bit offsets for W, B and 8-bit offsets for H, SH & SB
Design and write an assembly language program that will convert an ASCII representation of a hexadecimal value to a value. The string should be stored as a NULL-terminated string in memory and the converted value should be stored in register r0.
Program 7.4 – Value to ASCII String

- Design and write an assembly language program to convert a word-size unsigned value stored in memory to its hexadecimal ASCII string representation.
Design and write a program that will determine the cardinality of a set of word values stored in memory. The result (cardinality) should be stored in $r0$. 

e.g. if the values stored in memory are ...

4, 9, 3, 4, 7, 9, 12, 10, 4, 7, 3, 12, 5, 5, 7

then the program should store 7 in $r0$