Simple Model of a Microprocessor System

- **A Processing Unit** which performs operations on data
- **Memory**, which stores:
  - **Data**: representing text, images, videos, sensor readings, \( \pi \), 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, ...

![Diagram of a microprocessor system with a processing unit, memory, and peripheral devices connected via a bus.]
Program 7.1 – Upper Case String

- Design and write an assembly language program to convert a string stored in memory to UPPERCASE

- **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
}
```
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

RISC
- Simple operations
- Small / simple design
- More operations per unit time

CISC
- Complex operations
- Large / complex Design
- Fewer operations per unit time

Trade-off
Program 7.1 – Upper Case String

- Refine pseudo-code solution

```c
char = first character in string
while (char not past end of string)
{
    if (char ≥ 'a' AND char ≤ 'z')
    {
        char = char AND 0xFFFFFFFDF
    }
    char = next character
}

address = address of first character
char = Memory.byte[address]
while (char not past end of string)
{
    if (char ≥ 'a' AND char ≤ 'z')
    {
        char = char AND 0xFFFFFFFDF
        Memory.byte[address] = char
    }
    address = address + 1
    char = Memory.byte[address]
}
```
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.

Program 7.1 – Upper Case 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** – \( R_d = \text{Memory.}<\text{size}>[<\text{address}>] \)
  - Load a word-, half-word- or byte- <size> value ...
  - ... from Memory at a specified <address> ...
  - ... into a register
  - \( \text{LDR} \ R_d, <\text{address}> \) load word
  - \( \text{LDRH} \ R_d, <\text{address}> \) load half-word
  - \( \text{LDRB} \ R_d, <\text{address}> \) load byte

- **Store** – \( \text{Memory.}<\text{size}>[<\text{address}>] = R_d \)
  - Store a a word-, half-word- or byte- <size> value ...
  - ... from a register ...
  - ... into Memory at a specified <address>
  - \( \text{STR} \ R_d, <\text{address}> \) store word
  - \( \text{STRH} \ R_d, <\text{address}> \) store half-word
  - \( \text{STRB} \ R_d, <\text{address}> \) store byte
Addressing Mode [Rn, #0] or [Rn]

- 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**

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

- Example: load word-size value from memory at address 0xA1000000 into register r1
  - \( r1 = \text{Memory.word}[0xA1000000] \)

```assembly
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

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

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

  ```assembly
  LDR r0, =0xA1000000 ; Initialise base register r0 = 0xA1000000
  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 µVision to initialise memory with a test string
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
Endianness

- **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
Load / Store Bytes, Half-words and Words

**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
LDRSH r1, [r0]

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

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

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

Remember: interpretation!!
D3D1 / Microprocessor Systems I

Memory

- 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 = 0xA1000000

<rest of program>

AREA TestData, DATA, READWRITE

teststr DCB "hello",0 ; NULL terminated test string

END
```
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 DCD 32, 43, 10, 11, 14, 15
bonus DCD 7
```

- 2 half-word values

```
values DCW 407, -208
```
Other Assembler Directive

- **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
**Alignment**

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word-aligned</td>
<td>0x000000000, 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 \([Rn], \#+/\!<\text{offset}>\)

- **Immediate post-indexed**

  \[
  \text{<address>} = Rn \\
  Rn = Rn +/- \langle 12\text{-bit offset} \rangle
  \]

  - After calculating the effective address, the immediate value \(<\text{offset}>\) 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>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR r0, =0xA1001000</td>
<td></td>
<td>Initialise base register r1 = 0xA1001000</td>
</tr>
<tr>
<td>LDRH r1, [r0], #2</td>
<td>0xA1001002</td>
<td>r1 = Memory.halfword[r0], r0 = r0 + 2</td>
</tr>
<tr>
<td>LDRH r2, [r0], #2</td>
<td>0xA1001004</td>
<td>r2 = Memory.halfword[r0], r0 = r0 + 2</td>
</tr>
<tr>
<td>LDRH r3, [r0], #2</td>
<td>0xA1001006</td>
<td>r3 = Memory.halfword[r0], r0 = r0 + 2</td>
</tr>
</tbody>
</table>
Program 7.1b – Upper Case String

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

start
  LDR r1, =teststr ; address = 0xA1000000
                  ; while ( (char = Memory.byte[address])
                  ;       != 0 ) {
    B testwh1

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

AREA    TestData, DATA, READWRITE

teststr  DCB "hello",0 ; NULL terminated test string

END
```
### Addressing Mode \([Rn, \#+/-<offset>]\)

#### 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 \(r3\)

<table>
  <tr>
    <td>LDR</td>
    <td>r0, =0xA1001000 ; Initialise base register \(r1 = 0xA1001000\)</td>
  </tr>
  <tr>
    <td>LDR</td>
    <td>r1, [r0] ; \(r1 = \text{Memory.}\text{.word}[r0 + 0]\) (default = 0)</td>
  </tr>
  <tr>
    <td>LDR</td>
    <td>r2, [r0, #4] ; \(r2 = \text{Memory.}\text{.word}[r0 + 4]\)</td>
  </tr>
  <tr>
    <td>LDR</td>
    <td>r3, [r0, #8] ; \(r3 = \text{Memory.}\text{.word}[r0 + 8]\)</td>
  </tr>
</table>
Program 7.1c – Upper Case String

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

start LDR r1, =teststr ; address = teststr
       ; while ( (char = Memory.byte[address])
       B testwh1
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

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

END
```
### Register offset

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

**Example:** load three consecutive half-word values, beginning at address 0xA1001000, into registers \( r1 \), \( r2 \) and \( r3 \)

```
LDR         r0, =0xA1001000   ; Initialise base register \( r1 = 0xA1000000 \)
LDR         r4, =0            ; Initialise offset register \( r4 = 0 \)
LDRH        r1, [r0, r4]     ; \( r1 = \text{Memory.halfword}[r0 + r4] \)
ADD         r4, r4, #2       ; \( r4 = r4 + 2 \)
LDRH        r2, [r0, r4]     ; \( r2 = \text{Memory.halfword}[r0 + r4] \)
ADD         r4, r4, #2       ; \( r4 = r4 + 2 \)
LDRH        r3, [r0, r4]     ; \( r3 = \text{Memory.halfword}[r0 + r4] \)
```
Program 7.1d – Upper Case String

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

start
LDR r1, =teststr ; address = teststr
LDR r2, =0 ; offset = 0

; 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)

sum = 0
count = 100

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

Option 1
Use offset for loop condition

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

AREA Sum, CODE, READONLY
IMPORT main
EXPORT start

MAX EQU 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
    BCS do1 ; } while (count > 0)

stop B stop

AREA TestData, DATA, READWRITE
nums DCD 1,1,1,1,1,1,1,1,1,1
... ...
END
- **Scaled register offset**

  \[
  \text{<address>} = \text{Rn +/- Rm shift count}
  \]

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

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

```
LDR r0, =0xA1001000 ; Initialise base register r1 = 0xA1000000
LDR r4, =0          ; Initialise offset register r4 = 0
LDR r1, [r0, r4, LSL #2] ; r1 = Memory.word[r0 + (r4 * 4)]
ADD r4, r4, #1      ; r4 = r4 + 1
LDR r2, [r0, r4, LSL #2] ; r2 = Memory.word[r0 + (r4 * 4)]
ADD r4, r4, #1      ; r4 = r4 + 1
LDR r3, [r0, r4, LSL #2] ; r3 = Memory.word[r0 + (r4 * 4)]
```
Program 7.2b – Sum

AREA Sum, CODE, READONLY
IMPORT main
EXPORT start

MAX EQU 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>Register Pre-Indexed</td>
<td>[&lt;Rn&gt;, +/-&lt;Rm&gt;]!</td>
<td>✔</td>
<td>✔</td>
<td>Rn ← Rn +/- Rm</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>Immediate Post-Indexed</td>
<td>[&lt;Rn&gt;], #+/-&lt;offset&gt;</td>
<td>✔</td>
<td>✔</td>
<td>address ← Rn</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>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>
</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
Program 7.3 – ASCII string to value

- 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