CS1021 After Reading Week

Mid-Semester Test

- NOW Thurs 8th Nov @ 9am in Goldsmith Hall (ALL students to attend at 9am)

Final 2 Labs

- Lab5 2-Nov-18, due 16-Nov-18 (2 weeks duration)
- Lab6 16-Nov-19, due 30-Nov-18 (2 weeks duration)

End of Semester Exam

- written 2 hour exam (week of 12-Dec-18)
- answer 3 out of 4 questions (not 2 out of 3 as in recent CS1021 exams)
- 40 mins per question
- questions similar to previous CS1021 exams (shorter, less parts)

Yet to cover

- reading and writing to memory, stacks and subroutines
ARM Memory System

- ARM system comprises CPU and memory
- instructions and data stored in memory
- CPU can read (LOAD) data from memory into a register
- CPU can write (STORE) data from a register into memory
- called a load / store architecture
- to operate on data in memory, the data must first be **loaded** into register(s), updated in the registers and then **stored** back to memory
Memory Revision

- memory comprises an array of memory locations
- each location stores a byte of data
- each location has a unique 32 bit address
  - 0x00000000 to 0xFFFFFFFF
- the address space, $2^{32}$ bytes (4GB), is the amount of memory that can be physically attached to the CPU

memory as an array of BYTES

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFF</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFFFFFE</td>
<td>0xEE</td>
</tr>
<tr>
<td>0xFFFFFFFFD</td>
<td>0xDD</td>
</tr>
<tr>
<td>0xFFFFFFFFC</td>
<td>0xCC</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x00000000</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00000001</td>
<td>0x01</td>
</tr>
<tr>
<td>0x00000002</td>
<td>0x02</td>
</tr>
<tr>
<td>0x00000003</td>
<td>0x03</td>
</tr>
<tr>
<td>0x00000004</td>
<td>0x04</td>
</tr>
<tr>
<td>0x00000005</td>
<td>0x05</td>
</tr>
</tbody>
</table>
Memory Revision...

- often easier to view memory as an array of WORDs (32 bits) rather than an array of BYTES

- as each WORD location is aligned on a 4 byte boundary, the low order 2 bits of each address is 0

- making a comparison with the previous slide, the byte of data stored at memory location 0 is the least significant byte of the WORD stored in location 0

- this way of storing a WORD is termed LITTLE ENDIAN - the least significant byte is stored at the lowest address (the other way is BIG ENDIAN)

- ARM CPUs can be configured to be LITTLE ENDIAN or BIG ENDIAN (term from *Gulliver’s Travels*)
NXP LPC2468 Memory Map

- address space NOT fully populated with memory devices
- 512K of flash memory at address 0x00000000 to 0x0007FFFF
- 64K RAM at address 0x40000000 to 0x4000FFFF

- flash memory
  - read ONLY (programmed electronically - “flashed”)
  - retains data when power removed

- RAM (random access memory)
  - read write
  - looses its data when power removed

- uVision projects are configured to simulate this memory map
- code placed in flash memory starting at address 0x00000000
Load Instructions - LDR and LDRB

- memory address specified in a register

- load word

  LDR  R1, =0x40000000 ; R1 -> 0x40000000 (in RAM)
  LDR  R0, [R1] ; R0 = MWORD[0x40000000]

- load byte

  LDR  R1, =0x40000003 ; R1 -> 0x40000003 (in RAM)
  LDRB R0, [R1] ; R0 = MBYTE[0x40000003]
LDR and LDRB

• load word
  ▪ reads 4 bytes from memory address into a register
  ▪ address must be even (LS address bit = 0)
  ▪ normally used with an address aligned on a 4 byte boundary (address ends with ...00<sub>2</sub>) BUT ...
  ▪ if address end with ...10<sub>2</sub>, it accesses memory as though the address ended with ...00<sub>2</sub> but swaps the high and low 16 bits

• load byte
  ▪ reads byte from memory address and stores in LS byte of register
  ▪ clears MS bytes of register
LDR and LDRB...

- **load word**

  \[
  \text{LDR}\ R1, =0x40000000 \\
  \text{LDR}\ R0, [R1]
  \]

- **load byte**

  \[
  \text{LDR}\ R1, =0x40000003 \\
  \text{LDRB}\ R0, [R1]
  \]

- **load word (address ends with ..10_2)**

  \[
  \text{LDR}\ R1, =0x40000002 \\
  \text{LDR}\ R0, [R1]
  \]

Little endian:
- 0x01 in address 0x40000000
- 0x04 in address 0x40000003

High and low 16 bits swapped:
Store Instructions – STR and STRB

• memory address specified in a register

• store word

LDR  R1, =0x40000000  ; R1 -> 0x40000000 (in RAM)
STR  R0, [R1]  ; MWORD[0x40000000] = R0

• store byte

LDR  R1, =0x40000001  ; R1 -> 0x40000002 (in RAM)
STRB R0, [R1]  ; MBYTE[0x40000002] = R0 (LS byte)
STR and STRB

• store word
  ▪ writes ALL 4 bytes of register to memory address
  ▪ address must be aligned on a 4 byte boundary (address ends with \( ...00_2 \))

• store byte
  ▪ writes LS byte of register to memory address
Example

- a, b and c are 32-bit signed binary integers stored in memory locations 0x40000000, 0x40000004 and 0x40000008 respectively

- compute \( c = a + b \)

```
LDR  R1, =0x40000000  ; R1 -> a
LDR  R0, [R1]        ; R0 = a
LDR  R1, =0x40000004  ; R1 -> b
LDR  R1, [R1]        ; R1 = b
ADD  R0, R0, R1      ; R0 = a + b
LDR  R1, =0x40000008  ; R1 -> c
STR  R0, [R1]        ; c = a + b
```

R1 points to 0x40000000 (where a is stored in RAM)
ASCII strings

• American Standard Code for Information Interchange

• ASCII is a standard used to encode alphanumeric and other characters

• each character is stored as a single byte (8 bits)

• upper and lower case characters have different ASCII codes

• ASCII only uses 7 bits to encode the character, giving 128 possible characters

• MSB may be used as a parity bit
  • ODD or EVEN parity
  • parity bit set so that number of 1 bits in a character is either odd or even
  • used to detect transmit and receive errors

• originally used to transmit characters from a computer to a tele printer (terminal)
### ASCII Table (hex values)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
<td>HT</td>
<td>LF</td>
<td>VT</td>
<td>FF</td>
<td>CR</td>
<td>SO</td>
<td>SI</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>0004</td>
<td>0005</td>
<td>0006</td>
<td>0007</td>
<td>0008</td>
<td>0009</td>
<td>000A</td>
<td>000B</td>
<td>000C</td>
<td>000D</td>
<td>000E</td>
<td>000F</td>
</tr>
<tr>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
<td>DC3</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
<td>0012</td>
<td>0013</td>
<td>0014</td>
<td>0015</td>
<td>0016</td>
<td>0017</td>
<td>0018</td>
<td>0019</td>
<td>001A</td>
<td>001B</td>
<td>001C</td>
<td>001D</td>
<td>001E</td>
<td>001F</td>
</tr>
<tr>
<td>SP</td>
<td>&quot;</td>
<td>'</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
</tr>
<tr>
<td>0020</td>
<td>0021</td>
<td>0022</td>
<td>0023</td>
<td>0024</td>
<td>0025</td>
<td>0026</td>
<td>0027</td>
<td>0028</td>
<td>0029</td>
<td>002A</td>
<td>002B</td>
<td>002C</td>
<td>002D</td>
<td>002E</td>
<td>002F</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
</tr>
<tr>
<td>0030</td>
<td>0031</td>
<td>0032</td>
<td>0033</td>
<td>0034</td>
<td>0035</td>
<td>0036</td>
<td>0037</td>
<td>0038</td>
<td>0039</td>
<td>003A</td>
<td>003B</td>
<td>003C</td>
<td>003D</td>
<td>003E</td>
<td>003F</td>
</tr>
<tr>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>0040</td>
<td>0041</td>
<td>0042</td>
<td>0043</td>
<td>0044</td>
<td>0045</td>
<td>0046</td>
<td>0047</td>
<td>0048</td>
<td>0049</td>
<td>004A</td>
<td>004B</td>
<td>004C</td>
<td>004D</td>
<td>004E</td>
<td>004F</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
</tr>
<tr>
<td>0050</td>
<td>0051</td>
<td>0052</td>
<td>0053</td>
<td>0054</td>
<td>0055</td>
<td>0056</td>
<td>0057</td>
<td>0058</td>
<td>0059</td>
<td>005A</td>
<td>005B</td>
<td>005C</td>
<td>005D</td>
<td>005E</td>
<td>005F</td>
</tr>
<tr>
<td>&quot;</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
</tr>
<tr>
<td>0060</td>
<td>0061</td>
<td>0062</td>
<td>0063</td>
<td>0064</td>
<td>0065</td>
<td>0066</td>
<td>0067</td>
<td>0068</td>
<td>0069</td>
<td>006A</td>
<td>006B</td>
<td>006C</td>
<td>006D</td>
<td>006E</td>
<td>006F</td>
</tr>
<tr>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td>~</td>
<td>DEL</td>
</tr>
<tr>
<td>0070</td>
<td>0071</td>
<td>0072</td>
<td>0073</td>
<td>0074</td>
<td>0075</td>
<td>0076</td>
<td>0077</td>
<td>0078</td>
<td>0079</td>
<td>007A</td>
<td>007B</td>
<td>007C</td>
<td>007D</td>
<td>007E</td>
<td>007F</td>
</tr>
</tbody>
</table>

- `'H' = 0x48`
- `'e' = 0x65`
- `'l' = 0x6C`
- `'o' = 0x6F`
ASCII ...

- the string “Hello”, if at address 0x1000, stored as follows

<table>
<thead>
<tr>
<th>H</th>
<th>e</th>
<th>l</th>
<th>l</th>
<th>o</th>
<th>NUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x48</td>
<td>0x65</td>
<td>0x6c</td>
<td>0x6c</td>
<td>0x6f</td>
<td>0x00</td>
</tr>
<tr>
<td>0x1000</td>
<td>0x1001</td>
<td>0x1002</td>
<td>0x1003</td>
<td>0x1004</td>
<td>0x1005</td>
</tr>
</tbody>
</table>

- ASCII strings early always NUL terminated

- only 96 ASCII characters are printable, the remainder are control codes

- example control codes

| 0x0A | LF | line feed |
| 0x0D | CR | carriage return |
| 0x08 | BS | backspace |
| 0x09 | HT | horizontal tab |
| 0x1B | ESC | escape |
| 0x00 | NUL | NUL |
Example

- copy a NULL terminated ASCII string from 0x1000 (in read-only flash memory) to 0x40000000 (RAM)

```assembly
LDR R1, =0x1000 ; R1 -> src
LDR R2, =0x40000000 ; R2 -> dst
LDRB R0, [R1] ; get char from src string
STRB R0, [R2] ; store char in dst string
ADD R1, R1, #1 ; move to next src char
ADD R2, R2, #1 ; move to next dst char
CMP R0, #0 ; char == 0 ?
BNE L ; next character if not finished
```

R1 points to src string in RAM
R2 points to dst string in RAM
CS1021 Mid-Semester Test

• Thurs 8th Nov @ 9am in Goldsmith Hall

• ALL students to attend at 9am (no Tutorial @ 13.00)

• NO calculators, phones, laptop etc.

• 20 Questions (like Tutorial questions)

• ALL questions carry equal marks (some are easier than others)

• Remember to fill in exam number, student ID and name on answer booklet
uVision Console and Memory Windows

- Console window (UART #1)
  - View
  - Serial Windows
  - UART #1

- Memory window
  - View
  - Memory Windows
  - Memory 1

- Memory address
- Memory data
  - ASCII bytes for “0123456” + 0x0D (CR)
DCB, DCW and DCW Assembler Directives

- can use assembler to initialise the contents of flash memory with constant data
- if RAM needs to be initialised, it is normally initialised at start-up by copying data from flash memory (very microcontroller centric)

C0  DCB  “Hello World!”, 0  ; NUL terminated ASCII string

C1  DCD  1, 2, 3, 4, 5, 6, 7, 8  ; 8 x 32-bit (word) constants (why DCD for words?)

C2  DCW  1, 2, 3, 4, 5, 6, 7, 8  ; 8 x 16-bit (halfword) constants

- load address of constant string using

  LDR  R0, =C0  ; R0 -> “Hello World!”

- need to place constants where they will not be mistakenly executed as code
DCB, DCW and DCW Assembler Directives ...

constant data stored in memory @ address 0x50

C0 @ address = 0x50 (padded to be a multiple of 4 bytes)

C1 @ address = 0x60

C2 @ address = 0x80
Additional features of LDR / STR instructions

- additional features can be used to reduce the number of instructions that have to be written and the number of instructions executed at runtime

- `LDR R0, [R1] ; R1 used as an address register as R1 contains an address`

- `STR R0, [R1] ; R1 used as an address register as R1 contains an address`
Some Advanced features of LDR / STR instructions

- best understood by examining LDR / STR machine code fields

<table>
<thead>
<tr>
<th>condition</th>
<th>0</th>
<th>1</th>
<th>I</th>
<th>P</th>
<th>U</th>
<th>B</th>
<th>W</th>
<th>L</th>
<th>Rn</th>
<th>Rd</th>
<th>offset</th>
</tr>
</thead>
</table>

- **I** - immediate bit
  - if 0, offset field specifies a 12 bit offset (0 .. 4095)
  - if 1, offset field specifies a register + shift operation
- **P** - pre or post indexing
  - if 0 (post indexing), add offset to base register after transfer
  - if 1 (pre indexing), add offset to base register before transfer
- **U** (up/down) - 0 for subtract and 1 to add offset to base register
- **B** - 0 for word and 1 for byte transfer
- **W** - 1 for write back of effective address into base register
- **L** - 0 for store and 1 for load
Examples

- **LDR** R0, [R1, #4] ; with immediate offset
  
  \[ R0 = \text{MEM}[R1 + 4] \]

- **LDR** R0, [R1], #4 ; post-indexed
  
  \[ R0 = \text{MEM}[R1] \]
  \[ R1 = R1 + 4 \] ; post increment address register

- **LDR** R0, [R1, #-4]! ; pre-indexed
  
  \[ R1 = R1 - 4 \] ; pre increment address register
  \[ R0 = \text{MEM}[R1] \]

- **LDR** can read memory and increment address register in a single instruction
Examples...

- **LDR** R0, [R1, R4] ; register offset
  
  \[ R0 = \text{MEM}[R1 + R4] \]

- **LDR** R0, [R1], R4 ; register offset post-indexed
  
  \[ R0 = \text{MEM}[R1] \]
  \[ R1 = R1 + R4 \] ; post increment address register

  - can specify a +ve or -ve offset

- **LDR** R0, [R1, -R4]! ; pre-indexed register offset
  
  \[ R1 = R1 - R4 \] ; pre increment address register
  \[ R0 = \text{MEM}[R1] \]

- **LDR** can read memory and increment address register in a single instruction
Examples...

- **LDR** `R0, [R1, R4, LSL #2]` ; scaled register offset
  
  \[
  R0 = \text{MEM}[R1 + R4 \times 4]
  \]

  ![](image)

- **LDR** `R0, [R1], R4, LSL #2` ; post-indexed scaled register offset
  
  \[
  R0 = \text{MEM}[R1] \quad \text{R1} = R1 + R4 \times 4
  \]

  ![](image)

  - can specify a +ve or -ve offset

- **LDR** `R0, [R1, -R4, LSL #2]`! ; pre-indexed scaled register offset
  
  \[
  R1 = R1 - R4 \times 4 \quad R0 = \text{MEM}[R1]
  \]

  ![](image)

- **LDR** can read memory and increment address register in a single instruction