ARM Cortex-M4 Programming Model
Memory Addressing Instructions

References:
Textbook Chapter 4, Sections 4.1-4.5
Chapter 5, Sections 5.1-5.4
“ARM Cortex-M Users Manual”, Chapter 3
CPU instruction types

- **Data movement operations**
  - register-to-register
  - constant-to-register (or to memory in some CPUs)
  - memory-to-register and register-to-memory
    - includes different memory “addressing” options
    - “memory” includes registers in peripheral functions

- **Arithmetic operations**
  - add/subtract/multiply/divide
  - multi-precision operations (more than 32 bits)

- **Logical operations**
  - and/or/exclusive-or/complement (between operand bits)
  - shift/rotate
  - bit test/set/reset

- **Flow control operations**
  - branch to a location (conditionally or unconditionally)
  - branch to a subroutine/function
  - return from a subroutine/function
ARM ALU instruction operands

- **Register format:**
  \[ \text{ADD } r0, r1, r2 \]
  - Computes \( r1 + r2 \), stores in \( r0 \).

- **Immediate operand:**
  \[ \text{ADD } r0, r1, \#2 \]
  - Computes \( r1 + 2 \), stores in \( r0 \).

- **Set status condition flags, based on the result:**
  \[ \text{ADDS } r0, r1, r2 \]
  - Set status flags: \( Z, N, C, V \)
  - Flags unchanged if \( S \) not specified.
  - “\( S \)” suffix can be used for most ALU instructions.

- **Shortcut for “two-operand format”:**
  \[ \text{ADD } r1, r2 \] converted by assembler to: \[ \text{ADD } r1, r1, r2 \]
  (cannot use this form for MUL instruction)
Flexible second operand \(<op2>\)

General format: \(\text{ADD } Rd, Rs, \ <op2>\)

\[
\begin{align*}
\text{ADD } R0, R1, R2 &; \quad R0 = R0 + R2 \\
\text{ADD } R0, R1, \#4 &; \quad R0 = R0 + 4 \\
\text{ADD } R0, R1, R1, \text{LSL } \#4 &; \quad R0 = R0 + (R1 \times 16)
\end{align*}
\]

Shift operators:
- LSL, LSR
- ASR
- ROR
- RRX
- Shift by 1-32 bits
ARM “move” instruction
(copy register or constant to a register)

**MOV**  
r0, r1  ; copy r1 to r0

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFF8</td>
<td>0x00000040</td>
</tr>
</tbody>
</table>

**MOV**  
r0, #64  ; copy 64 (0x40) to r0

Constant embedded in instruction code.  
# limited to 16-bit unsigned value.  
zero-extended to upper bits of register

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFF8</td>
<td>0x01234FFF8</td>
</tr>
</tbody>
</table>

**MOVT**  
r0, #0x1234 ; # -> r0[31:16] (move to Top)

MOV followed by MOVT to set all 32 bits of a register to any number.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFF8</td>
<td>0x1234FFF8</td>
</tr>
</tbody>
</table>

**MVN**  
r0, r1  ; copy r1 to r0 (move “NOT”)

Example:

mov r0, #−20 will assemble to: mvn r0, #0x13
Since: \( \text{NOT}(0x00000013) = 0xFFFFFFFFEC = -20\)
Data Addressing Modes

- **Immediate** addressing
  - Data is contained within the instruction
    (immediately available as instruction is decoded)

```
MOV   R0,#100      ; R0=100, immediate addressing
```
ARM load/store instruction
(memory to/from register)

• Load operand from memory into target register
  • LDR – load 32 bits
  • LDRH – load halfword (16 bit unsigned #) / zero-extend to 32 bits
  • LDRSH – load signed halfword / sign-extend to 32 bits
  • LDRB – load byte (8 bit unsigned #) / zero-extend to 32 bits
  • LDRSB – load signed byte / sign-extend to 32 bits

• Store operand from register into memory*
  • STR – store 32-bit word
  • STRH – store 16-bit halfword (right-most 16 bits of register)
  • STRB – store 8-bit byte (right-most 8 bits of register)

* Signed/Unsigned not specified for STR instruction, since it simply stores the low N bits of the register into N bits of memory (N=8, 16, or 32)
Addressing Example

- Memory Operand: **Register-Indexed** Addressing
  - A register contains the memory address of *(points to)* the data
  - *Address* equivalent to an *index* into the array of memory bytes

```
LDR   R0, [R1] ; R0 = value pointed to by R1
```

![Addressing Example Diagram](https://via.placeholder.com/150)

EEPROM

```plaintext
0x12345678
```

RAM

```plaintext
0x20000004
```

2^{32} bytes of memory
Load a Byte

LDRB r1, [r0]

```
0x00 0x00 0x00 0xE1
```

Load a Halfword

LDRH r1, [r0]

```
0x00 0x00 0xE3 0xE1
```

Load a Word

LDR r1, [r0]

```
0x87 0x65 0xE3 0xE1
```

Little Endian

Assume

r0 = 0x20000000
Example

LDRH r1, [r0]
; r0 = 0x20008000

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008003</td>
<td>0x89</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0xCD</td>
</tr>
<tr>
<td>0x20008000</td>
<td>0xEF</td>
</tr>
</tbody>
</table>

r1 before load
0x12345678

r1 after load
0x0000CDEF
## Load with Sign Extension

### Load a Signed Byte

\[
\text{LDRSB } r1, \ [r0]
\]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0xFF | 0xFF | 0xFF | 0xE1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### Load a Signed Halfword

\[
\text{LDRSH } r1, \ [r0]
\]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0xFF | 0xFF | 0xE3 | 0xE1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

---

Little Endian

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0x20000003 | 0x87 | 0x20000002 | 0x65 | 0x20000001 | 0xE3 | 0x20000000 | 0xE1 |

Assume

\[ r0 = 0x20000000 \]

Facilitate subsequent 32-bit signed arithmetic!
Example

LDSB r1, [r0]
; r0 = 0x20008000

r1 before load
0x12345678

r1 after load
0xFFFFFFFFFEF

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008003</td>
<td>0x89</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0xCD</td>
</tr>
<tr>
<td>0x20008000</td>
<td>0xEF</td>
</tr>
</tbody>
</table>
Memory addressing modes

- **DIRECT** – operand address contained within the instruction (used by many CPUs)

  **Example** (Intel x86):

  ```
  MOV AX, Bob ; address of Bob contained in the instruction code
  
  Bob DW 1523 ; defined in a data area
  ```

  **ARM does not support direct addressing!**

  (32-bit address can’t be embedded in a 32-bit instruction)
Memory addressing modes

- **INDIRECT** – instruction tells CPU how to determine operand address
  - Address can be in a register (the register acts as a pointer)
  - Address can be computed as a base address plus an offset

**Example** - Read array value ARY[k], where ARY is an array of characters

<table>
<thead>
<tr>
<th>ARY</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

LDR  r2,=ARY ;r2 = base address of ARY
LDR  r3,=k ;r3 = address of variable k
LDR  r4,[r3] ;r4 = value of k
LDR  r5,[r2,r4] ;r5 = ARY[k] / address = r2+r3 = ARY+k
Create address pointer with ARM load “pseudo-op”

- Load a 32-bit constant (data, address, etc.) into a register
  - Cannot embed 32-bit value in a 32-bit instruction
  - Use a pseudo-operation (pseudo-op), which is translated by the assembler to one or more actual ARM instructions

- LDR r3,=constant
  - Assembler translates this to a MOV instruction, if an appropriate immediate constant can be found
  - Examples:

<table>
<thead>
<tr>
<th>Source Program</th>
<th>Debug Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR r3,=0x55</td>
<td>=&gt; MOV r3,#0x55</td>
</tr>
<tr>
<td>LDR r3,=0x55000000</td>
<td>=&gt; MOV r3,#0x55000000</td>
</tr>
<tr>
<td></td>
<td>(0x55 shifted left 24 bits)</td>
</tr>
</tbody>
</table>
Create address pointer with ARM load “pseudo-op”

- LDR  r3,=0x55555555
- Constant too large for mov instruction
- 32-bit constant is placed in the “literal pool”
  \textit{Literal pool} = set of constants stored after program in code area
- Constant loaded from literal pool address \([PC, \#offset]\)
  \begin{align*}
  \text{LDR } & \text{ r3,}\[PC, \#immediate12] \\
  \text{DCD } & \text{ 0x55555555 ; in literal pool following code}
  \end{align*}

- This pseudo-op requires \textbf{two 32-bit words}:
  \begin{itemize}
    \item One word for the LDR instruction
    \item One word for the 32-bit constant in the literal pool
  \end{itemize}
Address pointer example

```assembly
AREA  C1, CODE
LDR  r3, =Bob ; Load address of Bob into r3
LDR r2, [r3] ; Load value of variable Bob into r2
...
AREA  D1, DATA ; assume D1 starts at 0x20000000
Bob  DCD  0

- Assembler stores address of Bob (constant 0x20000000) in the “literal pool” in code area C1
- Constant loaded from literal pool address [PC, #offset]
  LDR  r3, =Bob
    LDR  r3, [PC, #offset] ; access value from literal pool
    ....
  DCD  0x20000000 ; in literal pool following code
```

Assembler translation

PC points to next instruction

PC + offset points to literal pool

address of Bob
Create address pointer with ARM ADR “pseudo-op”

**Only valid for labels defined in a CODE AREA**

ADR r1, LABEL ; load r1 with address of LABEL

- Assembler translates ADR pseudo-op to ARM instruction(s) that will result in address of a LABEL being placed in a register
- Use LDR rd,=LABEL for label in DATA AREA
- Can also use LDR rd,=LABEL for label in CODE AREA

```
AREA   C1, CODE
Main   ADR   r0,Prompt      ; r0 = address of Prompt  (PC + 16)
       LDRB  r1,[r0]        ; r1 = 1st character of Prompt
       LDR    r2,=Bob       ; r2 = address of Bob
       STRB  r1,[r2]        ; store r1 byte in variable Bob
       B      .              ; Assembled as: Here  B  Here
Prompt DCB  “Enter data:”,0  ;constant in code area

AREA   D1,Data      ; start of data area
Bob    DCB  0
```
ARM load/store addressing modes

- **Addressing modes:** base address + offset (optional)
  - register indirect: \( \text{LDR r0, [r1]} \)
  - with constant: \( \text{LDR r0, [r1, #4]} \)  
  \( \text{LDR r0, [r1, #-4]} \)
  - with second register: \( \text{LDR r0, [r1, r2]} \)  
  \( \text{LDR r0, [r1, -r2]} \)
  - pre-indexed: \( \text{LDR r0, [r1, #4]}! \)
  - post-indexed: \( \text{LDR r0, [r1], #8} \)
  - scaled index: \( \text{LDR r1, [r2, r3, LSL #2]} \)

Immediate \#offset = 12 bits (2’s complement)
Addressing examples

- `ldr r1,[r2]` ; address = (r2)
- `ldr r1,[r2,#4]` ; address = (r2)+4
- `ldr r1,[r2,#-4]` ; address = (r2)-4
- `ldr r1,[r2,r3]` ; address = (r2)+(r3)
- `ldr r1,[r2,-r3]` ; address = (r2)-(r3)
- `ldr r1,[r2,r3,LSL #2]` ; address=(r2)+(r3 x 4)

Base register r2 is not altered in any of these instructions
Addressing examples

- **Base-plus-offset addressing:**
  - LDR r0,[r1,#4] ; positive offset
    - Loads from location [r1+4]
  
  LDR r0,[r1,#-4] ; negative offset
  - Loads from location [r1-4]
  
  LDR r0,[r1,r2] ; add variable offset
  - Loads from location [r1+r2]
  
  LDR r0,[r1,-r2] ; sub variable offset
  - Loads from location [r1-r2]
Example

STR r1, [r0, #4]
; r0 = 0x20008000,  r1=0x76543210

r0 before store

0x20008000

r0 after store

0x20008000

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008007</td>
<td>0x76</td>
</tr>
<tr>
<td>0x20008006</td>
<td>0x54</td>
</tr>
<tr>
<td>0x20008005</td>
<td>0x32</td>
</tr>
<tr>
<td>0x20008004</td>
<td>0x10</td>
</tr>
<tr>
<td>0x20008003</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008000</td>
<td>0x00</td>
</tr>
</tbody>
</table>
Example

LDR r2, [r0, r1]
; r0 = 0x20008000, r1 = 0x00000004, r2 = 0x00000000

r2 before load
0x00000000

r2 after load
0x76543210

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008007</td>
<td>0x76</td>
</tr>
<tr>
<td>0x20008006</td>
<td>0x54</td>
</tr>
<tr>
<td>0x20008005</td>
<td>0x32</td>
</tr>
<tr>
<td>0x20008004</td>
<td>0x10</td>
</tr>
<tr>
<td>0x20008003</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008000</td>
<td>0x00</td>
</tr>
</tbody>
</table>
Addressing examples

• Base + scaled offset addressing

• Shift offset left by N bits to scale by factor $2^N$
  
  $$\text{LDR } r0,[r1,r2,lsl \ #2] \ ; \text{scale offset by } 4$$

• Loads from location $[r1+(4\times r2)]$

• Useful for array access
  
  • Scale index by data size
    
    (# bytes in a datum)

  Addr of $A[k] = \text{Addr of } A + k \times \text{datasize}$

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[0]$</td>
<td>$A[0]$</td>
</tr>
</tbody>
</table>

Offset = index * 1  Offset = index * 4
Example = read Bob[1]

LDR r0, [r1, r2, lsl #2]

; r1 = 0x20008000, r2=0x00000001

Base address of Bob

Index into array Bob

Offset = 0x00000001 x 4
= 0x00000004

Address = 0x20008000
+0x00000004
= 0x20008004

r0 after load
0x76543210

int Bob[2];
Addressing examples

- **Auto-indexing increments base register:**
  - LDR r0, [r1, #4]!
  - Loads from location [r1+4], then sets r1 = r1 + 4
  - Called “pre-indexing” since index added before memory access

- **Post-indexing fetches, then does offset:**
  - LDR r0, [r1], #4
  - Loads r0 from [r1], then sets r1 = r1 + 4
  - Called “post-indexing” since index added after memory access
ARM load/store examples
(base register updated by auto-indexing)

- `ldr r1,[r2,#4]!` ; use address = (r2)+4
  ; then r2<=(r2)+4  (pre-index)
- `ldr r1,[r2,r3]!` ; use address = (r2)+(r3)
  ; then r2<=(r2)+(r3)  (pre-index)
- `ldr r1,[r2],#4` ; use address = (r2)
  ; then r2<=(r2)+4  (post-index)
- `ldr r1,[r2],[r3]` ; use address = (r2)
  ; then r2<=(r2)+(r3)  (post-index)
Example

```
STR r1, [r0], #4 ; post-increment
; r0 = 0x20008000, r1=0x76543210
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008000</td>
<td>0x10</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x32</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x54</td>
</tr>
<tr>
<td>0x20008003</td>
<td>0x76</td>
</tr>
<tr>
<td>0x20008004</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008005</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008006</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008007</td>
<td>0x00</td>
</tr>
</tbody>
</table>

r0 before store

0x20008000

r0 after store

0x20008004
Example

```
STR r1, [r0, #4]! ; pre-increment
; r0 = 0x20008000, r1=0x76543210
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008007</td>
<td>0x76</td>
</tr>
<tr>
<td>0x20008006</td>
<td>0x54</td>
</tr>
<tr>
<td>0x20008005</td>
<td>0x32</td>
</tr>
<tr>
<td>0x20008004</td>
<td>0x10</td>
</tr>
<tr>
<td>0x20008003</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008000</td>
<td>0x00</td>
</tr>
</tbody>
</table>

r0 before store

0x20008000

r0 after store

0x20008004
Accessing variables in C

int p, k; //signed integer (32-bit) variables*
int w[10]; //array of 10 (32-bit) integers
p = k;   //copy k to p
  ldr r0,=k ; r0 = address of variable k
  ldr r1,[r0] ; read value of k from memory and put -> r1
  ldr r0,=p ; r0 = address of p
  str r1,[r0] ; write value in r1 to variable at memory address p
p = w[k]; //copy kth element of array w to p
  ldr r0,=k ; address of variable k
  ldr r1,[r0] ; load value of k
  ldr r0,=w ; base address of array w
  ldr r2,[r0,r1,lsl #2] ; value of w[k] (scale index k×4 for 32-bit words)
  ldr r0,=p ; address of variable p
  str r2,[r0] ; write to variable p

* unsigned int k; would be more appropriate for an array index (k ≥ 0)
C array example

int total, m;       //integer variables
int ary[10];        //10-integer array
for (total=0, m=0; m<10; m++) {
    total = total + ary[m];           //add up the ary values
}

;equivalent assembly language program
    ldr   r2,=ary                  ;point to array
    mov   r0,#0                   ;initial value of total
    mov   r1,#0                   ;initial value of m
Loop   cmp   r1,#10              ;is m < 4x10 (index x 4)
    bge   Exit                   ;exit loop if not
    ldr   r3,[r2],#4             ;load ary[m], point to ary[m+1]
    add   r0,r3                  ;add ary[m] to total
    add   r1,#1                  ;increment m
    b     Loop                    ;repeat the loop
Exit   ldr   r1,=total           ;point to variable total
    str   r0,[r1]               ;store total in memory
Pointers in C

```
int k,m;       //32-bit signed integers
int *ps,*pm;   //32-bit pointers

pm = &m;     //pm = address of (pointer to) variable m
ldr r0,=m;    ;address of m -> r0
ldr r1,=pm;   ;address of pm -> r1
str r0,[r1];  ;store address of m into pointer variable pm
k = *pm;      //*pm = value of the variable pointed to by pm (value of m)
ldr r2,=k;    ;address of k -> r2
ldr r3,[r0];  ;value of variable m -> r3 (r0 = address of m)
str r3,[r2];  ;store value of variable m -> variable k
ps = pm;      //save a copy of pointer pm (NOT the data pointed to by pm)
ldr r4,=ps;   ; address of ps -> r4
ldr r1,[r4];  ; pm (address of m, still in r1) stored in ps
```
PC-relative addressing Example

- **PC-Relative Addressing:**
  
  \[ \text{LDR \ r1,=Count \ translates to \ LDR \ r1, [pc,#offset]} \]

- Assume \textbf{Count} corresponds to address 0x20000000 in RAM

- Constant 0x20000000 stored in, and accessed from, literal pool in ROM

\[ \text{First,} \]
\[ \text{LDR \ R1,=Count} \]

\[ \text{Second,} \]
\[ \text{LDR \ R0, [R1]} \]