Data Transfer Instructions

CSE 30: Computer Organization and Systems Programming

Diba Mirza
Dept. of Computer Science and Engineering
University of California, San Diego
Assembly Operands: Memory

- **Memory**: Think of as single one-dimensional array where each cell
  - Stores a byte size value
  - Is referred to by a 32 bit address e.g. value at 0x4000 is 0x0a

<table>
<thead>
<tr>
<th>0x0a</th>
<th>0x0b</th>
<th>0x0c</th>
<th>0x0d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000</td>
<td>0x4001</td>
<td>0x4002</td>
<td>0x4003</td>
</tr>
</tbody>
</table>

- Data is stored in memory as: variables, arrays, structures
- But ARM arithmetic instructions only operate on registers, never directly on memory.

- **Data transfer instructions** transfer data between registers and memory:
  - Memory to register or LOAD from memory to register
  - Register to memory or STORE from register to memory
The ARM is a Load/Store Architecture:
- Does not support memory to memory data processing operations.
- Must move data values into registers before using them.

This might sound inefficient, but in practice isn’t:
- Load data values from memory into registers.
- Process data in registers using a number of data processing instructions which are not slowed down by memory access.
- Store results from registers out to memory.
Load/Store Instructions

- The ARM has three sets of instructions which interact with main memory. These are:
  - Single register data transfer (LDR/STR)
  - Block data transfer (LDM/STM)
  - Single Data Swap (SWP)

- The basic load and store instructions are:
  - Load and Store Word or Byte or Halfword
    - LDR / STR / LDRB / STRB / LDRH / STRH
Single register data transfer

- **LDR** STR  Word
- **LDRB** STRB  Byte
- **LDRH** STRH  Halfword
- **LDRSB** Signed byte load
- **LDRSH** Signed halfword load

- Memory system must support all access sizes

- Syntax:
  - **LDR**\{<cond>\}\{<size>\} Rd, <address>
  - **STR**\{<cond>\}\{<size>\} Rd, <address>
  - e.g. LDREQB
Data Transfer: Memory to Register

- To transfer a word of data, we need to specify two things:
  - Register: r0-r15
  - Memory address: more difficult
    - How do we specify the memory address of data to operate on?
    - We will look at different ways of how this is done in ARM

Remember: Load value/data FROM memory
Addressing Modes

- There are many ways in ARM to specify the address; these are called addressing modes.
- Two basic classification
  1. **Base register Addressing**
     - Register holds the 32 bit memory address
     - Also called the base address
  2. **Base Displacement Addressing mode**
     - An *effective address is calculated*:
       Effective address = < Base address + offset>
     - Base address in a register as before
     - Offset can be specified in different ways
Base Register Addressing Modes

- Specify a register which contains the memory address
  - In case of the load instruction (LDR) this is the memory address of the data that we want to retrieve from memory
  - In case of the store instruction (STR), this is the memory address where we want to write the value which is currently in a register

- Example: \([r0]\)
  - specifies the memory address pointed to by the value in \(r0\)
Data Transfer: Memory to Register

- **Load Instruction Syntax:**
  1. operation name
  2. register that will receive value
  3. register containing pointer to memory

- **ARM Instruction Name:**
  - LDR (meaning Load Register, so 32 bits or one word are loaded at a time)
Data Transfer: Memory to Register

- **LDR r2, [r1]**
  
  This instruction will take the address in r1, and then load a 4 byte value from the memory pointed to by it into register r2

- **Note:** r1 is called the base register

![Memory Transfer Diagram]

- Base Register: 0x200
- Destination Register for LDR: 0xddccbbaa

0x200, 0x200, 0x201, 0x202, 0x203

Memory:

- 0xaa
- 0xbb
- 0xcc
- 0xdd

UCSD
STR r2, [r1]

This instruction will take the address in r1, and then store a 4 byte value from the register r2 to the memory pointed to by r1.

- **Note:** r1 is called the **base register**
Base Displacement Addressing Mode

- To specify a memory address to copy from, specify two things:
  - A register which contains a pointer to memory
  - A numerical offset (in bytes)
- The effective memory address is the sum of these two values.
- Example: \([r0, \#8]\)
  - specifies the memory address pointed to by the value in \(r0\), plus 8 bytes
Base Displacement Addressing Mode

1. Pre-indexed addressing syntax:
   I. Base register is not updated
   LDR/STR  <dest_reg>[<base_reg>,offset]

Examples:
LDR/STR  r1 [r2, #4];  offset: immediate 4
  ; The effective memory address is calculated as r2+4
LDR/STR  r1 [r2, r3];  offset: value in register r3
  ; The effective memory address is calculated as r2+r3
LDR/STR  r1 [r2, r3, LSL #3];  offset: register value *2^3
  ; The effective memory address is calculated as r2+r3*2^3
Base Displacement Addressing Mode

1. Pre-indexed addressing:
   I. Base register is not updated:
      LDR/STR  <dest_reg>[<base_reg>,offset]
   II. Base register is first updated, the updated address is used
      LDR/STR  <dest_reg>[<base_reg>,offset]!

Examples:

   LDR/STR  r1  [r2,  #4]!;  offset: immediate 4
       ; r2=r2+4

   LDR/STR  r1  [r2, r3]!;  offset: value in register r3
       ; r2=r2+r3

   LDR  r1  [r2, r3, LSL #3]!;  offset: register value *2^3
       ; r2=r2+r3*2^3
**Base Displacement, Pre-Indexed**

- **Example:** `LDR r0, [r1, #12]`
  
  This instruction will take the pointer in `r1`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `r0`

- **Example:** `STR r0, [r1, #-8]`
  
  This instruction will take the pointer in `r0`, subtract 8 bytes from it, and then store the value from register `r0` into the memory address pointed to by the calculated sum

- **Notes:**
  - `r1` is called the **base register**
  - `#constant` is called the **offset**
  - Offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure
Pre indexed addressing

What is the value in r1 after the following instruction is executed?

```
STR r2, [r1, #-4]!
```

A. 0x200
B. 0x1fc
C. 0x196
D. None of the above
1. **Post-indexed addressing:** Base register is updated after load/store

   LDR/STR  <dest_reg>[<base_reg>], offset

   **Examples:**

   LDR/STR  r1 [r2], #4;  offset: immediate 4
   ; Load/Store to/from memory address in r2, update r2=r2+4

   LDR/STR  r1 [r2], r3;  offset: value in register r3
   ; Load/Store to/from memory address in r2, update r2=r2+r3

   LDR  r1 [r2] r3, LSL #3;  offset: register value left shifted
   ; Load/Store to/from memory address in r2, update r2=r2+r3*2^3
Post-indexed Addressing Mode

* Example: `STR r0, [r1], #12`

* If r2 contains 3, auto-increment base register to 0x20c by multiplying this by 4:
  * `STR r0, [r1], r2, LSL #2`

* To auto-increment the base register to location 0x1f4 instead use:
  * `STR r0, [r1], #-12`
* Imagine an array, the first element of which is pointed to by the contents of r0.

* If we want to access a particular element, then we can use pre-indexed addressing:
  - r1 is element we want.
  - LDR r2, [r0, r1, LSL #2]

* If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:
  - r1 is address of current element (initially equal to r0).
  - LDR r2, [r1], #4

Use a further register to store the address of final element, so that the loop can be correctly terminated.
Pointers vs. Values

- **Key Concept**: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), and so on.

- If you write \texttt{ADD r2, r1, r0} then \texttt{r0} and \texttt{r1} better contain values.

- If you write \texttt{LDR r2, [r0]} then \texttt{[r0]} better contain a pointer.

- Don’t mix these up!
Compilation with Memory

- What offset in LDR to select A[8] in C?
- 4x8=32 to select A[8]: byte vs word
- Compile by hand using registers:
  \[ g = h + A[8]; \]
  - \( g: r1, h: r2, r3: \) base address of A
- 1st transfer from memory to register:
  \[ \text{LDR } r0, [r3, \#32] \] ; \( r0 \) gets A[8]
  - Add 32 to \( r3 \) to select A[8], put into \( r0 \)
- Next add it to \( h \) and place in \( g \)
  \[ \text{ADD } r1, r2, r0 \] ; \( r1 = h + A[8] \]
Logical Shifts, Addressing modes in ARM Arithmetic
Data Transfer Instructions
Shifts and Rotates

- **LSL** – logical shift by n bits – multiplication by \(2^n\)
  
  ![Diagram of LSL](image)

- **LSR** – logical shift by n bits – unsigned division by \(2^n\)
  
  ![Diagram of LSR](image)

- **ASR** – arithmetic shift by n bits – signed division by \(2^n\)
  
  ![Diagram of ASR](image)

- **ROR** – logical rotate by n bits – 32 bit rotate
  
  ![Diagram of ROR](image)
Compute $01101001 \ll 2$.

A. 00011010

B. 00101001

C. 01101001

D. 10100100
A new instruction HEXSHIFTRIGHT shifts hex numbers over by a digit to the right.

**HEXSHIFTRIGHT $i$ times is equivalent to**

A. Dividing by $i$

B. Dividing by $2^i$

C. Dividing by $16^i$

D. Multiplying by $16^i$
A new instruction HEXSHIFTRIGHT shifts hex numbers over by a digit to the right.

HEXSHIFTRIGHT \(i\) times is equivalent to

A. Dividing by \(i\)

B. Dividing by \(2^i\)

C. Dividing by \(16^i\)

D. Multiplying by \(16^i\)
Ways of specifying operand 2

- **Opcode**  Destination, **Operand_1**, **Operand_2**
  - **Register Direct:**  \( \text{ADD } r0, r1, r2; \)
  - **With shift/rotate:**
    1) Shift value: 5 bit immediate (unsigned integer)
       \( \text{ADD } r0, r1, r2, \text{LSL } #2; \quad r0=r1+r2<<2; \quad r0=r1+4*r2 \)
    2) Shift value: Lower Byte of register:
       \( \text{ADD } r0, r1, r2, \text{LSL } r3; \quad r0=r1+r2<<r3; \quad r0=r1+(2^{r3})*r2 \)
  - **Immediate:**  \( \text{ADD } r0, r1, #0xFF \)
    - **With rotate-right**  \( \text{ADD } r0,r1, #0xFF, 28 \)
      Rotate value must be even: \#0xFF ROR 28 generates: \( 0XFF00000000 \)
Ways of specifying operand 2

- **Opcode Destination, Operand_1, Operand_2**
  - **Register Direct:** ADD r0, r1, r2;
  - **With shift/rotate:**
    1) Shift value: 5 bit immediate (unsigned integer)
       ADD r0, r1, r2, LSL #2;  r0=r1+r2<<2; r0=r1+4*r2
    2) Shift value: Lower Byte of register:
       ADD r0, r1, r2, LSL r3;  r0=r1+r2<<r3; r0=r1+(2^r3)*r2
  - **Immediate addressing:** ADD r0, r1, #0xFFF
    - 8 bit immediate value
    - **With rotate-right**
      - Rotate value must be even
      - #0xFFF ROR 8 generates: 0xFFF000000
      - Maximum rotate value is 30
The data processing instruction format has 12 bits available for operand2.

4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2.

Rule to remember is “8-bits rotated right by an even number of bit positions”
## Generating Constants using immediates

<table>
<thead>
<tr>
<th>Rotate Value</th>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000000000000000000000xxxxxxxxxxxxxx</td>
<td>0-255</td>
<td>0-0xFF</td>
</tr>
<tr>
<td>Right, 30 bits</td>
<td>0000000000000000000000000xxxxxxxxxxx00</td>
<td>4-1020</td>
<td>0x4-0x3FC</td>
</tr>
<tr>
<td>Right, 28 bits</td>
<td>0000000000000000000000000xxxxxxxxxxxx0000</td>
<td>16-4080</td>
<td>0x10-0xFF00</td>
</tr>
<tr>
<td>Right, 26 bits</td>
<td>00000000000000000000000000xxxxxxxxxxxx0000</td>
<td>128-16320</td>
<td>0x40-0x3FC0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Right, 8 bits</td>
<td>xxxxxxxx0000000000000000000000000000000000</td>
<td>16777216-255x2^24</td>
<td>0x1000000-0xFF000000</td>
</tr>
<tr>
<td>Right, 6 bits</td>
<td>xxxxxxxx0000000000000000000000000000000xxxxx</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Right, 4 bits</td>
<td>xxxx000000000000000000000000000000xxxxxxx</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Right, 2 bits</td>
<td>xx00000000000000000000000000000000xxxxxxx</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- This scheme can generate a lot, but not all, constants.
- Others must be done using literal pools (more on that later)
1. Register, optionally with shift operation
   ✤ Shift value can either be:
     ✤ 5 bit unsigned integer
     ✤ Specified in bottom byte of another register.
   ✤ Used for multiplication by constant

2. Immediate value
   ✤ 8 bit number, with a range of 0-255.
     ✤ Rotated right through even number of positions
   ✤ Allows increased range of 32-bit constants to be loaded directly into registers
Shifts and Rotates

- **Shifting in Assembly**
  
  Examples:
  
  MOV   r4, r6, LSL #4 ; r4 = r6 << 4
  MOV   r4, r6, LSR #8 ; r4 = r6 >> 8

- **Rotating in Assembly**
  
  Examples:
  
  MOV   r4, r6, ROR #12
  ; r4 = r6 rotated right 12 bits
  ; r4 = r6 rotated left by 20 bits (32 - 12)

  Therefore no need for rotate left.
Variable Shifts and Rotates

- Also possible to shift by the value of a register

Examples:

```
MOV    r4, r6, LSL r3
; r4 = r6 << value specified in r3
MOV    r4, r6, LSR #8 ; r4 = r6 >> 8
```

- Rotating in Assembly

Examples:

```
MOV    r4, r6, ROR r3
; r4 = r6 rotated right by value specified in r3
```
Constant Multiplication

- Constant multiplication is often faster using shifts and additions
  
  \[ \text{MUL r0, r2, #8} ; r0 = r2 \times 8 \]

  Is the same as:
  
  \[ \text{MOV r0, r2, LSL #3} ; r0 = r2 \times 8 \]

- Constant division
  
  \[ \text{MOV r1, r3, ASR #7} ; r1 = r3 / 128 \]
  
  Treats the register value like signed values (shifts in MSB).

  Vs.
  
  \[ \text{MOV r1, r3, LSR #7} ; r1 = r3 / 128 \]
  
  Treats register value like unsigned values (shifts in 0)
Constant Multiplication

- Constant multiplication with subtractions

\[
\text{MUL } r0, r2, \#7 ; r0 = r2 \times 7
\]

Is the same as:

\[
\text{RSB } r0, r2, r2, \text{LSL } \#3 ; r0 = r2 \times 7
\]
\[
; r0 = -r2 + 8 \times r2 = 7 \times r2
\]

\[
\text{RSB } r0, r1, r2 \text{ is the same as}
\]

\[
\text{SUB } r0, r2, r1 ; r0 = r1 - r2
\]

Multiply by 35:

\[
\text{ADD } r9, r8, r8, \text{LSL } \#2 ; r9=r8 \times 5
\]
\[
\text{RSB } r10, r9, r9, \text{LSL } \#3 ; r10=r9 \times 7
\]

Why have RSB? B/C only the second source operand can be shifted.
Conclusion

- Instructions so far:
  - Previously:
    ADD, SUB, MUL, MLA, [U|S]MULL, [U|S]MLAL
  - New instructions:
    RSB
    AND, ORR, EOR, BIC
    MOV, MVN
    LSL, LSR, ASR, ROR

- Shifting can only be done on the second source operand
- Constant multiplications possible using shifts and addition/subtractions
Comments in Assembly

- Another way to make your code more readable: comments!
- Semicolon (;) is used for ARM comments
  - anything from semicolon to end of line is a comment and will be ignored
- Note: Different from C
  - C comments have format /* comment */, so they can span many lines
Conclusion

- In ARM Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster

- Instructions so far:
  - ADD, SUB, MUL, MULA, [U|S]MULL, [U|S]MLAL

- Registers:
  - Places for general variables: r0–r12