Memory Interfaces

Evaluator7T Memory Map

After reset the BSL code begins running from address 0x0, and then reconfigures the memory map very early in its execution. After the BSL reconfigures the memory map, it is structured as shown in Table 3-1.

Table 3-1 Memory map after remap

<table>
<thead>
<tr>
<th>Address range</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 to 0x0003FFFF</td>
<td>256KB</td>
<td>32 bit SRAM bank, using ROMCON1</td>
</tr>
<tr>
<td>0x00040000 to 0x0007FFFF</td>
<td>256KB</td>
<td>32 bit SRAM bank, using ROMCON2</td>
</tr>
<tr>
<td>0x01800000 to 0x0187FFFF</td>
<td>512KB</td>
<td>16 bit flash bank, using ROMCON0</td>
</tr>
<tr>
<td>0x03FE0000 to 0x03FE1FFF</td>
<td>8KB</td>
<td>32 bit internal SRAM</td>
</tr>
<tr>
<td>0x03FF0000 to 0x03FFFFF</td>
<td>64KB</td>
<td>Microcontroller register space</td>
</tr>
</tbody>
</table>

--- Note ---
The BSL does not enable the cache. When the caches are enabled, you cannot use the 32-bit internal SRAM.
Evaluator7T Memory Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400000</td>
<td>ROM/SRAM/FLASH Bank 0 area (nonaccessible)</td>
</tr>
<tr>
<td>0x200000</td>
<td>ROM/SRAM/FLASH Bank 0 area (accessible)</td>
</tr>
<tr>
<td>0x000000</td>
<td>Undefined area</td>
</tr>
<tr>
<td>0xffffffff</td>
<td>Special function registers</td>
</tr>
</tbody>
</table>

Memory Map under BSL

<table>
<thead>
<tr>
<th>Address range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 to 0x0000003F</td>
<td>Exception vector table and address constants</td>
</tr>
<tr>
<td>0x00000040 to 0x000000FF</td>
<td>Unused</td>
</tr>
<tr>
<td>0x00001000 to 0x00007FFF</td>
<td>Read-write data space for BSL</td>
</tr>
<tr>
<td>0x00008000 to 0x000077FFF</td>
<td>Available as download area for user code and data</td>
</tr>
<tr>
<td>0x00078000 to 0x0007FFFF</td>
<td>System and user stacks</td>
</tr>
</tbody>
</table>
### Memory Map under Angel debug monitor

<table>
<thead>
<tr>
<th>Address range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 to 0x0000003F</td>
<td>Exception vector table and address constants</td>
</tr>
<tr>
<td>0x00000040 to 0x000000FF</td>
<td>Unused</td>
</tr>
<tr>
<td>0x00000100 to 0x000007FF</td>
<td>Read-write data and privileged mode stacks</td>
</tr>
<tr>
<td>0x00008000 to 0x00073FFF</td>
<td>Available as download area for user code and data</td>
</tr>
<tr>
<td>0x00074000 to 0x0007FFFF</td>
<td>Angel code execution region</td>
</tr>
</tbody>
</table>

### Flash Memory Map

<table>
<thead>
<tr>
<th>ADDRESS RANGE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01800000 to 0x01806FFF</td>
<td>Bootstrap loader</td>
</tr>
<tr>
<td>0x01807000 to 0x01807FFF</td>
<td>Production test</td>
</tr>
<tr>
<td>0x01808000 to 0x0180FFFF</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x01810000 to 0x0181FFFF</td>
<td>Angel</td>
</tr>
<tr>
<td>0x01820000 to 0x0187FFFF</td>
<td>Available for your programs and data</td>
</tr>
</tbody>
</table>
SRAM Memory Interface

Load/Store (Memory Access) Instructions
Instruction Class

ARM instructions can be broadly separated into three basic classes:

1. Data Movement
   Memory load/store
   Register Transfers

2. Data Operation
   Arithmetic
   Logical
   Register movement
   Comparison and test

3. Flow Control
   Branch
   Conditional execution

ARM Instructions

- Fixed length of 32 bits
- Commonly take two or three operands
- Process data held in registers
- Access memory with load and store instructions only
- Can be extended to execute conditionally by adding the appropriate suffix
- Affect the CPSR status flags by adding the ‘S’ suffix to the instruction
Load / Store Instructions

• 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
• 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)

Load and Store Instructions

Only two basic instructions are used for data transfer between memory and processor registers.

**LDR:** **LoaD** words from memory into a **Register**

**STR:** **STore** words from a **Register** into memory

Basic syntax:

<**LDR/STR**>{cond}{type} Rd, [Rn, addressing]

where

- **Rd** = destination (for LDR) & source (for STR)
- **Rn** = Base address register
- **cond** = condition flag
- **type** = byte, halfword, word(default), signed & unsigned
Load Instruction Format - LDR

Syntax

LDR <Rd>, [<Rn>, #<immed_5> * 4]

- `<Rd>` Destination Register for Memory Word
- `<Rn>` Register Containing Base Address
- `<immed_5>` 5-bit value Multiplied by 4 and added to `<Rn>` to Form the Memory Address

RTL

```
address ← Rn + (immed_5 * 4)
if (address[1:0] == 0b00)
data ← Memory[address,4]
else
data ← UNPREDICTABLE
Rd ← data
```

Load Instruction Format - LDR

Syntax

LDR <Rd>, [<Rn>, #<immed_5> * 4]

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | Rd | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | immed_5 | 0  | 0  |

cond code op code reserved
```

32 bit ARM LDR

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>immem_5</td>
<td>Rd</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

op code
```

16 bit Thumb LDR

Nice Ref on Machine Code Format:

http://www.wss.co.uk/pinknoise/ARMinstrs/ARMinstrs.html#Transfer
Store Instruction Format - STR

Syntax

\texttt{STR <Rd>, [<Rn>, #<immed_5> * 4]}

- \texttt{<Rd>} Register containing Word to Write to Memory
- \texttt{<Rn>} Register Containing Base Address
- \texttt{<immed_5>} 5-bit value Multiplied by 4 and added to \texttt{<Rn>} to Form the Memory Address

RTL

\begin{verbatim}
address <- Rn + (immed_5 * 4)
if (address[1:0] == 0b00)
    Memory[address,4] <- Rd
else
    data <- UNPREDICTABLE
\end{verbatim}

Store Instruction Format - STR

Syntax

\texttt{STR <Rd>, [<Rn>, #<immed_5> * 4]}

32 bit ARM STR

\begin{verbatim}
1110 0101 1000 Rn Rd 000000 immed_5 00
\end{verbatim}

16 bit Thumb STR

\begin{verbatim}
0110 0 immed_5 Rn Rd
\end{verbatim}
### Examples

LDR  r0, [r1]
; load r0 with the content of the memory location
; pointed to in r1

STR  r2, [r1]
; store content of r2 to memory location with address
; pointed to in r1

LDRB r0, [r1]  ; load byte size data
STRH r0, [r1]  ; store halfword size data
LDRSB r0, [r1]  ; load signed byte

---

### Load and Store Word or Byte: Base Register

- The memory location to be accessed is held in a base register
  - STR r0, [r1]  ; Store contents of r0 to location pointed to
  ; by contents of r1.
  - LDR r2, [r1]  ; Load r2 with contents of memory location
  ; pointed to by contents of r1.
**Common Load/Store Instructions**

<table>
<thead>
<tr>
<th>Loads</th>
<th>Stores</th>
<th>Size and Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>STR</td>
<td>Word (32 bits)</td>
</tr>
<tr>
<td>LDRB</td>
<td>STRB</td>
<td>Byte (8 bits)</td>
</tr>
<tr>
<td>LDRH</td>
<td>STRH</td>
<td>Halfword (16 bits)</td>
</tr>
<tr>
<td>LDRSB</td>
<td></td>
<td>Signed Byte</td>
</tr>
<tr>
<td>LDRSH</td>
<td></td>
<td>Signed Halfword</td>
</tr>
<tr>
<td>LDM</td>
<td>STM</td>
<td>Multiple Words</td>
</tr>
</tbody>
</table>

**Load Half Word Example**

```assembly
LDRH r11, [r0] ; load a halfword into r11
```

<table>
<thead>
<tr>
<th>r0 content before/after load</th>
<th>Memory Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000800</td>
<td>0x8000</td>
<td>0xEE</td>
</tr>
<tr>
<td>0x12345678</td>
<td>0x8001</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x12345678</td>
<td>0x8002</td>
<td>0x90</td>
</tr>
<tr>
<td>0x12345678</td>
<td>0x8003</td>
<td>0xA7</td>
</tr>
</tbody>
</table>

**r11 before load**  0x00000800 0x12345678

**r11 after load**  0x00000FFEE

*Is this Little Endian or Big Endian?*
Load Half Word Example

LDRH  r11, [r0]   ;load a halfword into r11

<table>
<thead>
<tr>
<th>r0 content before/after load</th>
<th>0x00008000</th>
</tr>
</thead>
<tbody>
<tr>
<td>r11 before load</td>
<td>0x12345678</td>
</tr>
<tr>
<td>r11 after load</td>
<td>0x0000FFEE</td>
</tr>
</tbody>
</table>

```
Little Endian
```

Signed Byte Load Example

LDRSB  r11, [r0]   ;load a signed byte into r11

<table>
<thead>
<tr>
<th>r0 content before/after load</th>
<th>0x00008000</th>
</tr>
</thead>
<tbody>
<tr>
<td>r11 before load</td>
<td>0x12345678</td>
</tr>
<tr>
<td>r11 after load</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>

```
Little Endian
```

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>0xEE</td>
</tr>
<tr>
<td>0x8001</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x8002</td>
<td>0x90</td>
</tr>
<tr>
<td>0x8003</td>
<td>0xA7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>0xEE</td>
</tr>
<tr>
<td>0x8001</td>
<td>0x8C</td>
</tr>
<tr>
<td>0x8002</td>
<td>0x90</td>
</tr>
<tr>
<td>0x8003</td>
<td>0xA7</td>
</tr>
</tbody>
</table>
Store Example Using Post Increment

STR  r3, [r8], #4  ;Memory write to address 0x8000

r3 content
before/after store
0xFEEDBABE

Memory after Store
Address   Data
0x8000   0xBE
0x8001   0xBA
0x8002   0xED
0x8003   0xFE

r8 before store
0x00008000

r8 after store
0x00008004

Is this Little Endian or Big Endian?

Little Endian
More Examples

LDR r5, [r3] ;load r5 with data from ea <r3>
STRB r0, [r9] ;store data in r0 to ea<r9>
STR r3, [r0, r5, LSL #3] ;store data in r3 ea<r0+r5*8>
LDR r1, [r0, #4]! ;load r1 from ea<r0+4>, r0 <- r0+4 (! is wrt-bak)
STRB r7, [r6, #-1]! ;store byte to ea<r6-1>, r6 <- r6-1 (! is wrt-bak)
LDR r3, [r9], #4 ;load r3 from ea<r9>, r9 <- r9+4
STR r2, [r5], #8 ;store word to ea<r5>, r5 <- r5+8

Addressing Modes

ARM uses a fixed-length instruction, with the lowest 12 bits available to specify immediate address
• not sufficient to cover the full $2^{32}$ address space
• hence do not support direct addressing

ARM only provides indirect addressing modes
1. Register indirect addressing
2. PC-relative addressing
Register Indirect Addressing

An address is available in a register

Example: \textbf{LDR r0, [r1]}

Here, the \textit{r1} content is known as the ‘base address’, and \textit{r1} is called the \textbf{base address register}.

Can be further extended to:

a) Pre-indexed addressing
b) Pre-indexed with write-back addressing (uses “!”)
c) Post-indexed addressing (implicit write-back)

Pre-Indexed

Pre-indexed addressing adds an offset to the base address \textbf{before} executing the load/store.

\textbf{LDR|STR\{<cond>\} <Rd>, [<Rn>, <offset>]\{!\}}

Example: \textbf{LDR r0, [r1, #8]}

This instruction loads \textit{r0} with the content of memory location at (base address + 8).

Optional \textit{!} specifies to write the effective address back into Rn after execution of instruction. Otherwise Rn retains original value.

Useful for addressing an element in a data structure.
For example, access a particular register of a peripheral through the peripheral base address and its offset.
Pre-Indexed with Write-Back

Pre-indexed addressing with write-back automatically updates the base address before executing the load/store.

Example: \texttt{LDR r0, [r1, \#4]!}

This instruction adds 4 to the base register r1, loads r0 with the content of memory location (now is at base address + 4), and increments r1 by 4.

Increment to the base address is done \textit{before} the execution of the load instruction but r1 changes value after execution.

Useful for automatic stepping through a lookup table with a starting address placed in the base address register.

---

Pre-Indexed Example

\texttt{LDR|STR\{<cond>\} <Rd>, [<Rn>, <offset>]\{!\}}

\texttt{STR r0, [r1, \#12] ;writes \texttt{0x5} to address \texttt{0x20C}}

- offset 12
- address 0x20C
- memory 0x5
- r0 0x5

r1 retains 0x200 after instruction is executed
STR Pre-indexed Addressing

STR  r0, [r1, #12]  ;writes 0x5 to address 0x20C

• To store to location 0x1f4 instead use
  - STR  r0, [r1, #-12]
• To auto-increment base pointer to 0x20c use
  - STR  r0, [r1, #12]!
• If r2 contains 3, access 0x20c by multiplying this by 4
  - STR  r0, [r1, r2, LSL #2]

Pre-Indexed Example with Writeback

LDR|STR{<cond>} <Rd>, [<Rn>, <offset>]{}!

STR  r0, [r1, #12]!  ;writes 0x5 to address 0x20C

r1 contains 0x20C
after instruction is executed
Pre-Indexed Examples

LDR|STR{<cond>} <Rd>, [<Rn>, <offset>]{}!

;r3 data written to ea(r0+(r5*8))
STR r3, [r0, r5, LSL #3]

;r6 gets data from ea(r0+(r1/64)), after data is read
r0 is updated r0 ← r0+(r1/64)
LDR r6, [r0, r1, ROR #6]!

;r0 gets data from ea(r1-8)
LDR r0, [r1, #8]

;r0 gets data from ea(r1-(r2*4))
LDR r0, [r1, -r2, LSL #2]

Pre-Indexed Examples

LDR|STR{<cond>} <Rd>, [<Rn>, <offset>]{}!

;r5 gets 2 bytes of data from ea(r9) and is sign
;extended to fill the 32 bit r5 register
LDRSH R5, [R9]

;r3 gets 1 byte of data from ea(r8*8) and is sign
;extended to fill the 32 bit r3 register
LDRSB R3, [R8, #3]

;r4 gets 1 byte of data from ea(r10+193) and is sign
;extended to fill the 32 bit r4 register
LDRSB R4, [R10, #0xc1]

;r4 gets 1 byte of data from ea(r10+193) and is sign
;extended to fill the 32 bit r4 register and r10 is
;updated to contain r10 ← r10+193
LDRSB R4, [R10, #0xc1]!
Post-Indexed

Post-indexed addressing automatically updates the base address after executing the load/store.
Example: STR r0, [r1], #4;

This instruction stores the content of r0 into the memory location pointed to in base address register r1, executes the store operation, and increases the base address value by 4. Increment is done after the execution.

Note that ‘!’ is not needed for post-indexed addressing since the update is implicit.

Useful for storing a list of data into a table with a starting address pointed to by the base address value in r1.

Post-Indexed Example

LDR|STR{<cond>} <Rd>, [<Rn>], <offset>

STR r0, [r1], #12 ;writes 0x5 to address 0x200

Updated
r1  offset  address  memory
0x20C  12  \rightarrow 0x20C

Original
r1  \leftarrow 0x200

Source for STR
r0  \leftarrow 0x5
### STR: Post-indexed Addressing

\[
\text{STR } r0, [r1], #12 \quad ;\text{writes 0x5 to address 0x200}
\]

- **To auto-increment the base register to location 0x1f4 instead use:**
  - \text{STR } r0, [r1], #-12

- **If \( r2 \) contains 3, auto-increment base register to 0x20c by multiplying this by 4:**
  - \text{STR } r0, [r1], r2, LSL #2

### Post-Indexed Examples

<table>
<thead>
<tr>
<th>LDR</th>
<th>STR{&lt;cond&gt;}</th>
<th>&lt;Rd&gt;, [&lt;Rn&gt;], &lt;offset&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>;32 bit data in r7 written to ea(r0) and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;r0 is updated to contain ( r0 \leftarrow r0 + 24 ) after write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{STR } r7, [r0], #24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;r3 gets 32 bits of data from address beginning at</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;ea(r0) and r0 is updated to contain ( r0 \leftarrow r0 + (r4/16) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{LDR } r2, [r0], r4, ASR #4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;r3 gets 16 bits of data from address beginning at</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;ea(r9) and r9 is updated to contain ( r9 \leftarrow r9 + 2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{LDRH } r3, [r9], #2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;16 bit data written from r2 to address beginning at</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;ea(r5) and r5 is updated to contain ( r5 \leftarrow r5 + 8 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{STRH } r2, [r5], #8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PC Relative Addressing

Program Counter relative addressing makes up for the unavailability of full direct addressing in ARM instructions.

Example:

```
LDR r1, label1
```

`label1: ..`

The instruction is to load `r1` with a 32-bit value which is the address of the label named `label1`.

But this will give an error because the 32-bit value cannot fit into an instruction that is itself 32-bit long.
(In fact, for ARM, only the lowest 12 bits are available to store an immediate value)

Pseudo-Instruction ADR

The solution is to store the 32-bit address somewhere nearby the load instruction.

The 32-bit value can then be retrieved by accessing through a relative offset from the PC register value of the instruction.

This PC relative addressing operation is represented using pseudo-instructions `ADR` & `ADRL`.

Example:

```
ADR r1, label1 ; range < 255 Bytes
ADRL r2, label2 ; range < 64K
```

*Who decides where to store the 32-bit value of label1?*

*Ans: The assembler*
Example - Memory String Copy

SRAM_BASE EQU 0x04000000 ; address to store string
AREA StrCopy, CODE
ENTRY ; mark first instruction
Main adr r1, srcstr ; pointer to source string
    ldr r0, =SRAM_BASE ; pointer to destination string
strcopy
    ldrb r2, [r1], #1 ; load byte, update address
    strb r2, [r0], #1 ; store byte, update address
    cmp r2, #0 ; check for zero terminator
    bne strcopy ; loop if terminator not reached
stop b stop ; halt processing, loop forever
srcstr DCB "This is my (source) string",0 ; end of file marker

Endian

• Term arises from paper D. Cohen [1981]
• ARM supports both conventions
• Only arises with systems that have smaller-sized memory storage than wordsize
• Should storage be from “left-to-right” or “roght-to-left”?
• Intel x86 uses “right-to-left”, Motorola 68X (now FreeScale) uses “left-to-right”
• ARM allows for either
• Core has input pin BIGEND, when asserted, results in Big Endian
• Assembler Directive used for this
Effect of endianess

- The ARM can be set up to access its data in either little or big endian format.
- Little endian:
  - Least significant byte of a word is stored in *bits 0-7* of an addressed word.
- Big endian:
  - Least significant byte of a word is stored in *bits 24-31* of an addressed word.
- This has no real relevance unless data is stored as words and then accessed in smaller sized quantities (halfwords or bytes).
  - Which byte / halfword is accessed will depend on the endianess of the system involved.

---

Endianess Example

```
r0 = 0x11223344
```

Little-endian

```
r1 = 0x100
```

```
r2 = 0x44
```

```
STR r0, [r1]
```

```
LDRB r2, [r1]
```

Big-endian

```
r1 = 0x100
```

```
r2 = 0x11
```

```
r1 = 0x100
```

```
r2 = 0x44
```

```
```

```
```
Multiple-Register Load-Store

The multiple-register load-store instructions support the transfer of a block of data in one instruction, through the use of Multiple registers.

Basic instructions: **LDM** and **STM**

Usually used with a suffix: **IA, IB, DA, DB**

- **IA**: increment after
- **IB**: increment before
- **DA**: decrement after
- **DB**: decrement before

Can also be used in conjunction with the ‘!’ for write-back.

---

Multiple-Register Load-Store (cont’d)

**Example**: **STMIA**  
\r0, {r1-r3, lr};

This single instruction stores four registers (r1, r2, r3, and lr – i.e., r14) into the 16 memory locations (4 words) starting at the base address in r0.

**Example**: **LDMDA**  
\r0!, {r1-r3, lr};

This instruction performs the loading into r1-r3, lr in a decrement fashion, and updates the base register r0 itself after performing the four word (16 byte) transfers.
Multiple-Register Load-Store (cont’d)

The execution of single multiple-register transfer instruction will take multiple sequential memory access cycles to complete.

• still much faster than transferring using multiple instructions which incurs more access latency – uses the burst read/write mode of ARM.

But these instructions affect maximum interrupt latency.

• interrupt that occurs during the \texttt{LDM} and \texttt{STM} instruction execution will be pending until the execution is fully completed.