Overview:

• Part I
  • Machine Codes
  • Branches and Offsets
  • Subroutine
  • Time Delay
32-Bit ARM Vs. 16/32-Bit THUMB2 Assembly Instructions

**ARM Instruction Formats**

<table>
<thead>
<tr>
<th>Cond</th>
<th>0 0 0</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift Amount</th>
<th>Shift 0</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>0 0 1</td>
<td>Opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>rotate</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>0 1 0</td>
<td>P</td>
<td>B</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>0 1 1</td>
<td>P</td>
<td>B</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>Shift Amount</td>
</tr>
<tr>
<td>Cond</td>
<td>1 0 1</td>
<td>L</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>1 0 1</td>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>SWord7</td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>1 0 1</td>
<td>L</td>
<td>1</td>
<td>0</td>
<td>R</td>
<td>Rlist</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>1 1 0</td>
<td>L</td>
<td>Rb</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>1 1 0</td>
<td>L</td>
<td>Rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>1 1 1</td>
<td>L</td>
<td>Value8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **S** = For data processing instructions, updates condition codes
- **S** = For load/store multiple instructions, execution restricted to supervisor mode
- **P, U, W** = distinguish between different types of addressing mode
- **B** = Unsigned byte (B==1) or word (B==0) access
- **L** = For load/store instructions, Load (L==1) or Store (L==0)
- **L** = For branch instructions, is return address stored in link register
### 32-Bit ARM Vs. 16/32-Bit THUMB2 Assembly Instructions

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>Flags</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Z==1, Equal</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Z==0, Not Equal</td>
</tr>
<tr>
<td>0010</td>
<td>HS/CS</td>
<td>C==1, &gt;= (u) / C=1</td>
</tr>
<tr>
<td>0011</td>
<td>LO/CC</td>
<td>C==0, &lt; (u) / C=1</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>N==1, minus(neg)</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>N==0, plus(pos)</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>V==1, V set(ovfl)</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>V==0, V clr</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>C==1&amp;&amp;Z==0, &gt; (u)</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>C==0</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>N==V, &gt;=</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>N!=V, &lt;</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Z==0&amp;&amp;N==V, &gt;</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Z==1</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>always</td>
</tr>
<tr>
<td>1111</td>
<td>NE</td>
<td>never</td>
</tr>
</tbody>
</table>

*(u) = unsigned
ARM & THUMB2
Assembly Instructions
### Conditional Branch – THUMB2 ASSEMBLY

#### Table: THUMB Instruction Set Formats

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800240</td>
<td>101 00 111</td>
<td>LDR R0, [R12, #0]</td>
</tr>
<tr>
<td>0x0800244</td>
<td>101 00 111</td>
<td>LDR R1, [R12, #4]</td>
</tr>
</tbody>
</table>

#### Diagram:

- **Address**: 0x0800256
- **Machine Code (16-32 bit)**: E7FE
- **Assembly Instruction**: B
- **Memory**

---

*Figure 5-1: THUMB Instruction set formats*
Forward Branch – THUMB2 ASSEMBLY

- Instruction 0x8000244 is an 8-bit branch
- Machine code is E005; Opcode = E0, Offset is 0X05=5
- The next instruction will be:
- Offset * 2+Current_Add + 2*2
- Thus
  \((5*2)_{\text{d}}+0x244+(4)_{\text{d}}=0x244+0xE=0x252\)
  - Similarly:
  - Instruction 0x8000256 is an 8-bit branch
  - Machine code is E7FE; Opcode = E7, Offset is FE=-1
  - The next instruction will be:
  - Offset * 2 + Current_Add + 4
  - Thus
    \((-1*4)_{\text{d}}+0x256+(4)_{\text{d}}=0x256+0x0=0x256\) (same location)
Backward Branch – THUMB2 ASSEMBLY

```
1101 0001 <1111 1001>
D 1 < F 9 >
```

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>Flags</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Z==1 Equal</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Z==0 Not Equal</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code (16-32 bit)</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08000240</td>
<td>F44F727A MOV r2,#0x3E8</td>
<td></td>
</tr>
<tr>
<td>0x08000244</td>
<td>F04F0000 MOV r0,#0x00</td>
<td></td>
</tr>
<tr>
<td>0x08000248</td>
<td>F1000009 ADD r0,r0,#0x09</td>
<td></td>
</tr>
<tr>
<td>0x0800024C</td>
<td>BF00 NOP</td>
<td></td>
</tr>
<tr>
<td>0x08000250</td>
<td>1E52 SUBS r2,r2,#1</td>
<td></td>
</tr>
<tr>
<td>0x08000252</td>
<td>D1F9 BNE 0x08000248</td>
<td></td>
</tr>
<tr>
<td>0x08000254</td>
<td>4604 MOV r4,r0</td>
<td></td>
</tr>
<tr>
<td>0x08000256</td>
<td>E7FE B 0x08000256</td>
<td></td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: 0x08000240</td>
</tr>
<tr>
<td>0x08000240: 727A44F 0000F04F 00095100 BF00BF00 D1F9E52 E7FE4604</td>
</tr>
<tr>
<td>0x08000250: 00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>0x08000270: 00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
```
Backward Branch – THUMB2 ASSEMBLY

• Instruction 0x8000252 is an 8-bit branch
• Machine code is D1F9; Opcode = D1, Offset is F9=-7
• The next instruction will be:
• Offset * 2+Current_Add + 2*2
• Thus
\((-7*2)_{d}+0\times252+(4)_{d}=0\times252-0\timesA=0\times248\)
• Similarly:
• Instruction 0x8000256 is an 8-bit branch
• Machine code is E7FE; Opcode = E7, Offset is FE=-1
• The next instruction will be:
• Offset * 2 + Current_Add + 4
• Thus
\((-1*4)_{d}+0\times256+(4)_{d}=0\times256-0\times0=0\times256\) (same location)
How Long Does It Take to Run the Code?

- Measure the time at the beginning of the code using the Performance Analyzer
- Then, step through until the end of the code
- One **Instruction Cycle** is equivalent to one **Core Clock Cycle** (Clock Period)

![Performance Analyzer Screenshot](image)

![Waveform Diagram](image)
How Long Does It Take to Run the Code?

• Measure the time at the beginning of the code
• Then, step through until the end of the code
• One Instruction Cycle is equivalent to one Core Clock Cycle (Clock Period)

```
23  //EXAMPLE OF DELAY
24  // CHANGE TIME (Alt+F7) TO 10 MHZ
25  // TO SEE THE TIME: VIEW->ANALYSIS WINDOW->LOGIC ANALYZER
26  DELAY MOV R0,#255
27  AGAIN NOP
28  NOP
29  SUBS R0,R0,#1
30  BNE AGAIN
31  MOV PC,LR
```

Total Instruction Cycles (nsec) = \[1+(1+1+3)\times255+1\] \times \text{Core Clock Cycle} = 100 \text{ nsec} = 153,200 \text{ nsec}
Time Delay in ARM

• Several factors can impact the time it take to execute the code
  • Cote clock cycle: **Instruction Cycle = Core clock cycle \times N**
  • Chip architecture design: Pipelining implementation, instruction size, ALU size, etc.

• In RISC one Instruction cycle is equivalent to on Cluck Cycle:
  • **One Instruction cycle is equivalent to on Machine Cycle = Core clock cycle \times N**
  • Machine Cycle is the time it requires for the CPU to execute one instruction

• Generally, MOV, SUB, ADD, NOP etc. take one Instruction Cycle to execute

• Branches take 3 if branch occurs / one Instruction Cycle if no branch occurs

• It take 3 IC to execute an unconditional branch
Subroutines – Branch and Link

• When we call a subroutine

```
AREA CODE, READONLY
...........code........
BL SUBRTN_1
....more code....

SUBRTN_1
...........code........
BX LR ;RETURN TO CALLER

END
```
Example of a Subroutine

LR maintains the address of the caller

Go back to where it was called

Note the subroutine is assembled right after the code:

```
27: AGAIN NOP
28: ;NOP
29: ;SUBS R0, R0, #1
30: ;BNE AGAIN
31: ;MOV PC, LR ;RETURN
32:
33: ;Example of a subroutine
34: LDR R1, =RAM_ADDR
35: AGAIN MOV R0, #X55
36: STRB R0, [R1]
37: BL DELAY
38: MOV R0, #XAA
39: STRB R0, [R1]
40: BL DELAY
41: B AGAIN
42:
43: DELAY LDR R3, =5
44: L1 SUBS R3, R3, #1
45: BNE L1
46: BX LR
```

RETURN to the CALLER

we can use MOV PC, LR

LR maintains the address of the caller
Quiz: Write a program to perform $X = X + 2$ 100 times, assume $X = 2$

• Draw the flow chart and implement the program
  • Let’s assume $X$ is in R2
  • The final results is in R1
  • Assume the counter is in R3
• Show that your code works

• Answer the following questions:
  • How long did it take to do complete the code?
  • What’s your Xstat Freq?
  • How many instruction cycles are you going through for each loop?
  • How much memory (in BYTE) your code consumes?
  • Using the assembled code, how many instructions are THUMB2?
  • PC Value:
    • Which register contains the PC value?
    • What is the PC value when the code starts?
    • What is the PC value by the time the code is completed?
    • By the time the code is completed, how many WORDs the PC has moved?
  • In which SRAM locations the code is stored?
• Decode the following – what does it mean?
  0x0800024C 1E5B      SUBS          r3,r3,#1
Quiz Answer: Write a program to perform \( X = X + 2 \) 100 times, assume \( X = 2 \)

- Let’s assume \( X \) is in R2
- The final results is in R1
- Assume the counter is in R3
- Draw the flow chart and implement the program

```
0x0800024C 1E5B      SUBS          r3,r3,#1
0001 1110 0101 1011
0001 111 001 011 011
```

```
0x0800024C 1E5B      SUBS          r3,r3,#1
```

```
0x08000240 04F0202  MOV          r2,#0x02
0x08000244 04F0307  MOV          r3,#7
0x08000248 F1020202 ADD          r0,r2,r3
0x0800024C 1E5B  SUBS          r3,r3,#1
0x08000250 F04F0110 MOV          r1,#0x10
```
Remember: Memory Map

**AREA Defined for DATA - READWRITE**
- SRAM (96 KB aliased by bit-banding): 0x2000 0000 - 0x2001 7FFF

**AREA Defined for CODE - READONLY**
- Flash memory: 0x0800 0000 - 0x0807 FFFF
Overview – PART 2

• How data is organized in memory?
  • Big Endian vs Little Endian

• How data is addressed?
  • Pre-index
  • Post-index
  • Pre-index with update
Logic View of Memory

• By grouping bits together we can store more values
  • 8 bits = 1 byte
  • 16 bits = 2 bytes = 1 halfword
  • 32 bits = 4 bytes = 1 word

• From software perspective, memory is an addressable array of bytes.
  • The byte stored at the memory address 0x20000004 is \(0b10000100\)

\[
\begin{align*}
0b10000100 & \rightarrow 0x84 \rightarrow 132 \\
\text{Binary} & \quad \text{Hexadecimal} & \quad \text{Decimal}
\end{align*}
\]
Logic View of Memory

- When we refer to memory locations by address, we can only do so in units of bytes, halfwords or words.

- Words
  - 32 bits = 4 bytes = 1 word = 2 halfwords
  - In the right diagram, we have two words at addresses:
    - 0x20000000
    - 0x20000004
  - Can you store a word anywhere? NO.
  - A word can only be stored at an address that's divisible by 4.
  - Memory address of a word is the lowest address of all four bytes in that word.

Word-address mod 4 = 0
What are the memory address of these four words?

<table>
<thead>
<tr>
<th>Word 0</th>
<th>Addr = ??</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Word 1</td>
<td>Addr = ??</td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Word 2</td>
<td>Addr = ??</td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td>Word 3</td>
<td>Addr = ??</td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0014</td>
</tr>
</tbody>
</table>
What are the memory address of these four words?

<table>
<thead>
<tr>
<th>Words</th>
<th>32-bit Address</th>
<th>8-bit Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word 3</td>
<td>0x0012</td>
<td>0015</td>
</tr>
<tr>
<td>Word 2</td>
<td>0x0008</td>
<td>0014</td>
</tr>
<tr>
<td>Word 1</td>
<td>0x0004</td>
<td>0013</td>
</tr>
<tr>
<td>Word 0</td>
<td>0x0000</td>
<td>0012</td>
</tr>
</tbody>
</table>

Quiz (Answer)
Endianess

ARM is *Little Endian by default*. However it can be made Big Endian by configuration.
Example

If big endian is used

The word stored at address 0x20008000 is

0xEE8C90A7

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008003</td>
<td>0xA7</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x90</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x8C</td>
</tr>
<tr>
<td>0x20008000</td>
<td>0xEE</td>
</tr>
</tbody>
</table>
Example

If little endian is used

The word stored at address 0x20008000 is 0xA708CEE

Endian only specifies byte order, not bit order in a byte!

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008000</td>
<td>0xEE</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x8C</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x90</td>
</tr>
<tr>
<td>0x20008003</td>
<td>0xA7</td>
</tr>
</tbody>
</table>
Load-Modify-Store

C statement

\[
x = x + 1;
\]

Memory Address | Memory Data
--- | ---
0x20008003 | X

; Assume the memory address of x is stored in r1

LDR r0, [r1] ; load value of x from memory
ADD r0, r0, #1 ; x = x + 1
STR r0, [r1] ; store x into memory
Load Instructions

- **LDR rt, [rs]**
  - fetch data from memory into register rt.
  - The memory address is specified in register rs.
  - For Example:

; Assume r0 = 0x08200004
; Load a word:
LDR r1, [r0] ; r1 = Memory.word[0x08200004]
Store Instructions

**STR rt, [rs]:**
- save data in register rt into memory
- The memory address is specified in a base register rs.
- For Example:

```assembly
; Assume r0 = 0x08200004
; Store a word
STR r1, [r0]        ; Memory.word[0x08200004] = r1
```
# Quick Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load Word</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Byte</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load Halfword</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load Signed Byte</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Load Signed Halfword</td>
</tr>
<tr>
<td>STR</td>
<td>Store Word</td>
</tr>
<tr>
<td>STRB</td>
<td>Store Lower Byte</td>
</tr>
<tr>
<td>STRH</td>
<td>Store Lower Halfword</td>
</tr>
</tbody>
</table>

Pre-Index: LDR r1, [r0, #4]; R0+4, LOAD, R0
Post-Index: LDR r1, [r0], #4; R, LOAD, R+4
Pre-Index with Update: LDR r1, [r0, #4]!; → R+4, LOAD, R+4
### Single register data transfer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load Word</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Byte</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load Halfword</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load Signed Byte</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Load Signed Halfword</td>
</tr>
</tbody>
</table>

Pre-Index: \( \text{LDR } r1, [r0, #4] \); \( R0+4 \), LOAD, \( R0 \)

Post-Index: \( \text{LDR } r1, [r0], #4 \); \( R \), LOAD, \( R+4 \)

Pre-Index with Update: \( \text{LDR } r1, [r0, #4]! \); \( \rightarrow R+4 \), LOAD, \( R+4 \)
Load a Byte, Half-word, Word

Load a Byte
LDRB r1, [r0]

0x00 0x00 0x00 0xE1

| 31 | 0 |

Load a Halfword
LDRH r1, [r0]

0x00 0x00 0xE3 0xE1

| 31 | 0 |

Load a Word
LDR r1, [r0]

0x87 0x65 0xE3 0xE1

| 31 | 0 |

Assume r0 = 0x02000000

Little Endian
Sign Extension

Load a Signed Byte
LDRSB r1, [r0]

Load a Signed Halfword
LDRSH r1, [r0]

Facilitate subsequent 32-bit signed arithmetic!

Assume r0 = 0x02000000

Little Endian
Address

• Address accessed by LDR/STR is specified by a base register plus an offset

• For word and unsigned byte accesses, offset can be
  • An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes).
    LDR r0,[r1,#8]
  • A register, optionally shifted by an immediate value
    LDR r0,[r1,r2]
    LDR r0,[r1,r2,LSL#2]

• This can be either added or subtracted from the base register:
  LDR r0,[r1,#-8]
  LDR r0,[r1,-r2]
  LDR r0,[r1,-r2,LSL#2]

• For halfword and signed halfword / byte, offset can be:
  • An unsigned 8 bit immediate value (i.e. 0-255 bytes).
  • A register (unshifted).

• Choice of pre-indexed or post-indexed addressing
Pre-index

Pre-Index: LDR r1, [r0, #4]

<table>
<thead>
<tr>
<th>Base memory address</th>
<th>Memory address</th>
<th>Memory data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x200000000</td>
<td>0x88</td>
<td></td>
</tr>
<tr>
<td>0x200000006</td>
<td>0x79</td>
<td></td>
</tr>
<tr>
<td>0x200000005</td>
<td>0x6A</td>
<td></td>
</tr>
<tr>
<td>0x200000004</td>
<td>0x5B</td>
<td></td>
</tr>
<tr>
<td>0x200000003</td>
<td>0x4C</td>
<td></td>
</tr>
<tr>
<td>0x200000002</td>
<td>0x3D</td>
<td></td>
</tr>
<tr>
<td>0x200000001</td>
<td>0x2E</td>
<td></td>
</tr>
<tr>
<td>0x200000000</td>
<td>0x1F</td>
<td></td>
</tr>
</tbody>
</table>

Offset range is -255 to +255
Pre-index

Pre-Index: LDR r1, [r0, #4]

Offset range is -255 to +255
Post-index

**Post-Index: LDR r1, [r0], #4**

<table>
<thead>
<tr>
<th>Base memory address</th>
<th>Memory address</th>
<th>Memory data</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0 = 0x20008000</td>
<td>0x200080007</td>
<td>0x88</td>
</tr>
<tr>
<td></td>
<td>0x20008006</td>
<td>0x79</td>
</tr>
<tr>
<td></td>
<td>0x20008005</td>
<td>0x6A</td>
</tr>
<tr>
<td></td>
<td>0x20008004</td>
<td>0x5B</td>
</tr>
<tr>
<td></td>
<td>0x20008003</td>
<td>0x4C</td>
</tr>
<tr>
<td></td>
<td>0x20008002</td>
<td>0x3D</td>
</tr>
<tr>
<td></td>
<td>0x20008001</td>
<td>0x2E</td>
</tr>
<tr>
<td></td>
<td>0x20008000</td>
<td>0x1F</td>
</tr>
</tbody>
</table>

Offset range is -255 to +255
Post-index

Post-Index: LDR r1, [r0], #4

Offset range is -255 to +255
Pre-index with Updates

**Pre-Index with Update: LDR r1, [r0, #4]!**

<table>
<thead>
<tr>
<th>r0</th>
<th>0x20008000</th>
<th>0x20008001</th>
<th>0x20008002</th>
<th>0x20008003</th>
<th>0x20008004</th>
<th>0x20008005</th>
<th>0x20008006</th>
<th>0x20008007</th>
<th>0x20008008</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset = 4</td>
<td></td>
<td>0x88</td>
<td>0x79</td>
<td>0x6A</td>
<td>0x5B</td>
<td>0x4C</td>
<td>0x3D</td>
<td>0x2E</td>
<td>0x1F</td>
</tr>
<tr>
<td>r0 = 0x20008000</td>
<td>Memory address</td>
<td>Memory address</td>
<td>Memory address</td>
<td>Memory address</td>
<td>Memory address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Base memory address

Offset range is -255 to +255
Pre-index with Updates

Pre-Index: LDR r1, [r0, #4]

Offset range is -255 to +255
## Summary of Pre-index and Post-index

<table>
<thead>
<tr>
<th>Index Format</th>
<th>Example</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-index</td>
<td>LDR r1, [r0, #4]</td>
<td>r1 ← memory[r0 + 4], r0 is unchanged</td>
</tr>
<tr>
<td>Pre-index with update</td>
<td>LDR r1, [r0, #4]!</td>
<td>r1 ← memory[r0 + 4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r0 ← r0 + 4</td>
</tr>
<tr>
<td>Post-index</td>
<td>LDR r1, [r0], #4</td>
<td>r1 ← memory[r0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r0 ← r0 + 4</td>
</tr>
</tbody>
</table>

Offset range is -255 to +255
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
Example

LDSB 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
0xFFFFFFFFFEF

THE REST IS ALL-ONE
Example

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

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008000</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008003</td>
<td>0x00</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
Example

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

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

r0 before store

0x20008000

r0 after store

0x20008004
Example

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

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008000</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20008003</td>
<td>0x00</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 the store

0x20008000

r0 after the store

---
Example

```
STR r1, [r0, #4]
; 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>
Example

\textbf{STR r1, [r0, #4]!}

; r0 = 0x20008000, r1=0x76543210

\begin{tabular}{|c|c|}
\hline
Memory Address & Memory Data \\
\hline
0x20008007 & 0x00 \\
0x20008006 & 0x00 \\
0x20008005 & 0x00 \\
0x20008004 & 0x00 \\
0x20008003 & 0x00 \\
0x20008002 & 0x00 \\
0x20008001 & 0x00 \\
0x20008000 & 0x00 \\
\hline
\end{tabular}

r0 before store
0x20008000

r0 after store

\textbf{Memory Address}  \textbf{Memory Data}
Example

```assembly
STR r1, [r0, #4]!
; 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
Example

If big endianess is used

LDR r11, [r0]
; r0 = 0x20008000

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20008003</td>
<td>0xEE</td>
</tr>
<tr>
<td>0x20008002</td>
<td>0x8C</td>
</tr>
<tr>
<td>0x20008001</td>
<td>0x90</td>
</tr>
<tr>
<td>0x20008000</td>
<td>0xA7</td>
</tr>
</tbody>
</table>

r11 before load
0x12345678
r11 after load
0xA7908CEE
Load/Store Multiple Registers

STMxx rn{!}, {register_list}
LDMxx rn{!}, {register_list}

- xx = IA, IB, DA, or DB

<table>
<thead>
<tr>
<th>Addressing Modes</th>
<th>Description</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>Increment After</td>
<td>STMIA, LDMIA</td>
</tr>
<tr>
<td>IB</td>
<td>Increment Before</td>
<td>STMIB, LDMIB</td>
</tr>
<tr>
<td>DA</td>
<td>Decrement After</td>
<td>STMDA, LDMDA</td>
</tr>
<tr>
<td>DB</td>
<td>Decrement Before</td>
<td>STMDB, LDMDB</td>
</tr>
</tbody>
</table>

- **IA**: address is incremented by 4 after a word is loaded or stored.
- **IB**: address is incremented by 4 before a word is loaded or stored.
- **DA**: address is decremented by 4 after a word is loaded or stored.
- **DB**: address is decremented by 4 before a word is loaded or stored.
Load/Store Multiple Registers

- The following are synonyms.
  - STM = STMIA (Increment After) = STMEA (Empty Ascending)
  - LDM = LDMIA (Increment After) = LDMFD (Full Descending)

- The order in which registers are listed does not matter
  - For STM/LDM, the lowest-numbered register is stored/loaded at the lowest memory address.
Cortex-M3 & Cortex-M4 Memory Map

- 32-bit Memory Address
- $2^{32}$ bytes of memory space (4 GB)
- Harvard architecture: physically separated instruction memory and data memory
Cortex-M4 Fixed Memory Map

0xE0100000
0xE00FF000
0xE0042000
0xE0041000
0xE0040000
0xE000F000
0xE000E000
0xE0003000
0xE0002000
0xE0001000
0xE0000000

ROM Table
External PPB
ETM
TPIU
Reserved
SCS
Reserved
FPB
DWT
ITM

System
Private peripheral bus - External
Private peripheral bus - Internal
External device 1.0GB
External RAM 1.0GB
Peripheral 0.5GB
SRAM 0.5GB
Code 0.5GB

0xE0100000
0xE0040000
0xE0000000
0xA0000000
0x60000000
0x40000000
0x40000000
0x40000000
0x24000000
0x22000000
0x20100000
0x20000000

32MB Bit band alias
31MB
1MB Bit band region

32MB Bit band alias
31MB
1MB Bit band region
Practice: What happens to R0 in each case?

```
MOV    R1, #0x10
MOV    R2, #0x20
MOV    R3, #0x30
MOV    R4, #0x40
LDR    R0, =0x2000000
; stores the values in [R0], 16 bytes
STMIA  R0, {R1, R2, R3, R4}
; Increment R0 as it stores the values in [R0]
STMIA  R0!, {R1, R2, R3, R4}
LDR    R0, =0x20000000 ; reset R0 pointer
; loads the values pointed by [R0] into R5-R8
LDMIA  R0, {R5, R6, R7, R8}
```

STM = STMIA (Increment After)

0000 0010, 0000 0020, 0000 0030, 0000 0040
R0=20000000

0000 0010, 0000 0020, 0000 0030, 0000 0040
R0=20000010 (updated)
Practice: How the code is assembled

```
24  MOV  R1, #0x10
25  MOV  R2, #0x20
26  MOV  R3, #0x30
27  MOV  R4, #0x40
28  LDR  R0, =0x20000000
29  ; stores the values in [R0], 16 bytes
30  STMIA  R0, [R1, R2, R3, R4]
31  ; increment R0 as it stores the values in [R0]
32  STMIA  R0!, [R1, R2, R3, R4]
33  LDR  R0, =0x20000000 ; reset R0 pointer
34  ; loads the values pointed by [R0] into R5-R8
35  LDMIA  R0, [R5, R6, R7, R8]
36  ; data is stored starting at 0x20000000+20d
37  LDR  R0, array_B ; R0=0x20000000+20d
38  LDR  R1, [R0]
39  LDR  R4, [R0, #4] ; in this case R0 does not change
40  LDR  R4, [R0, #8]
41  LDR  R0, array_B ; R0 is changing by 4
42  LDR  R4, [R0, #4]
43  LDR  R4, [R0, #4]
44  MOV  R1, #0xAA
45  LDR  R0, =0x20000070
46  STR  R1, [R0] ; store into memory
47  HERE:  B  HERE
48  WHEN READONLY we do not override the previous values
49  ALIGN  myDATA, DATA, READONLY
50  ALIGN  array_B DCD 0x11, 0x22, 0x33, 0x44, 0x55 ; save as LONG
51  ; array_W allocates 5 HALF-WORDS starting at 0x20000000+20d
52  ; array_W DCD 0x66, 0x77, 0x88, 0x99, 0xAA ; save as half WORD
```

DCD: One word at a time
Practice: What happens to R0,R1,R4

DCD: One word at a time

DCW: One half-word at a time

Change to READWRTE?
What happens?

What happens if we have array_B?
Practice: What happens to R0, R1, R4

```
; data is stored starting at 0x20000000 + 20d
LDR R0, =array_w ; R0 = 0x20000000 + 20d
LDR R1, [R0]
LDR R4, [R0, #4]; in this case R0 does not change
LDR R4, [R0, #8]
LDR R0, =array_w
LDR R4, [R0], #4 ; R0 is changing by 4
LDR R4, [R0], #4
MOV R1, #0xAA
LDR R0, =0x20000070
STR R1, [R0]; store into memory
```

Change to READWRITE? What happens?

Store 0xAA