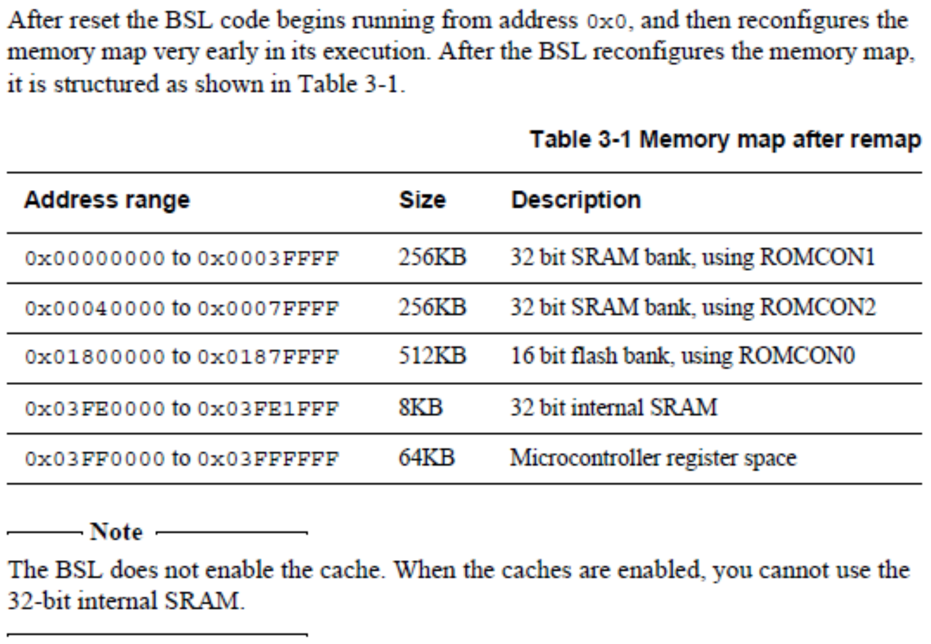
Memory Interfaces

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.

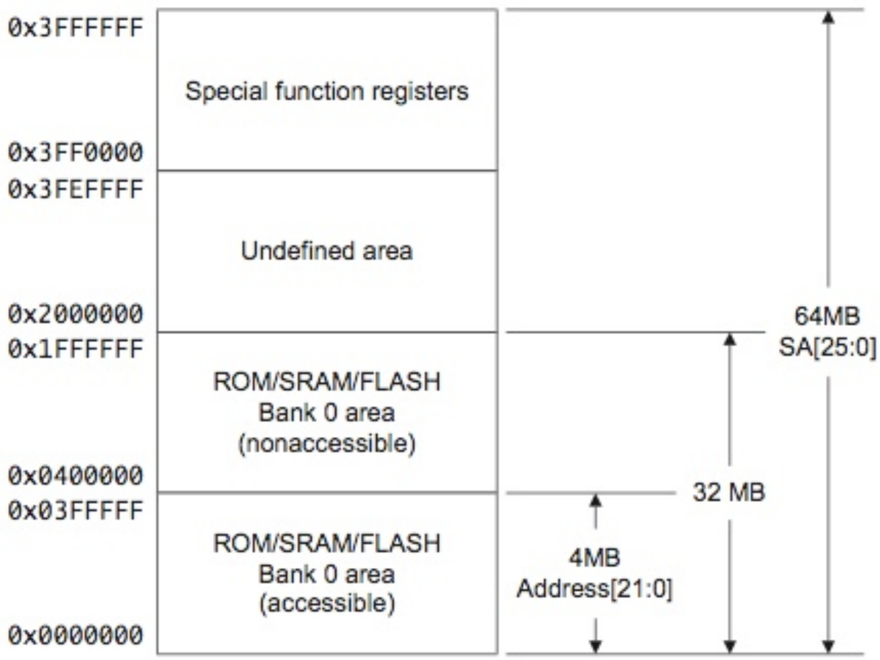
Table 3-1 Memory map after remap

Address range	Size	Description
`0x00000000` to `0x0003FFFF`	256KB	32 bit SRAM bank, using ROMCON1
`0x00040000` to `0x0007FFFF`	256KB	32 bit SRAM bank, using ROMCON2
`0x01800000` to `0x0187FFFF`	512KB	16 bit flash bank, using ROMCON0
`0x03FE0000` to `0x03FE1FFF`	8KB	32 bit internal SRAM
`0x03FF0000` to `0x03FFFFFF`	64KB	Microcontroller register space

Note

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

Evaluator7T Memory Map



0x3FFFFFF

Special function registers

0x3FF0000

0x3FEFFFF

Undefined area

0x2000000

0x1FFFFFF

ROM/SRAM/FLASH
Bank 0 area
(nonaccessible)

0x0400000

0x03FFFFF

ROM/SRAM/FLASH
Bank 0 area
(accessible)

0x0000000

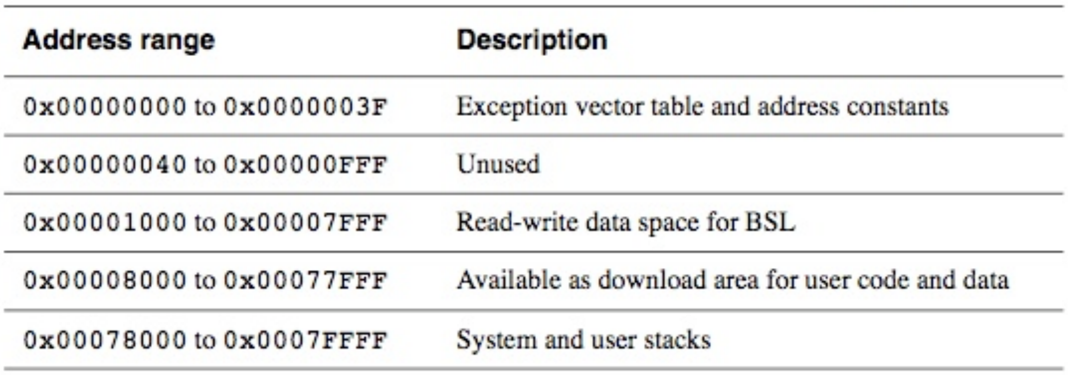
4MB
Address[21:0]

32 MB

64MB
SA[25:0]

Memory Map under BSL

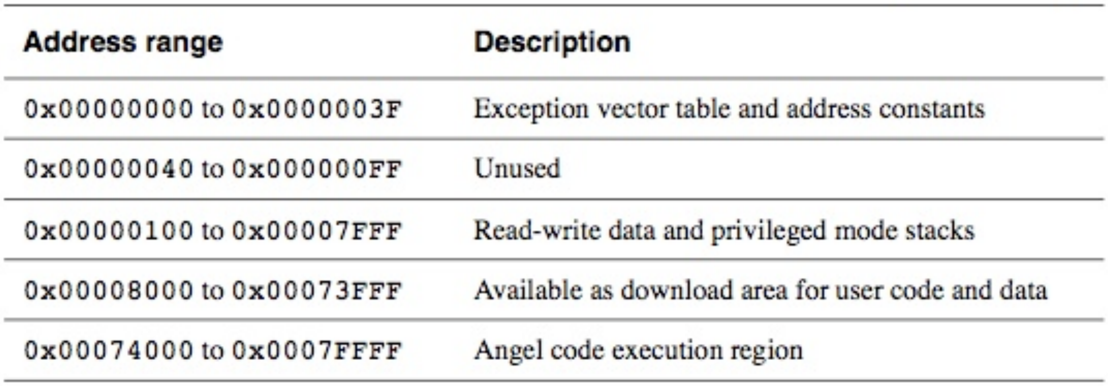
Table 3-2 SRAM usage under BSL



Address range	Description
0x00000000 to 0x0000003F	Exception vector table and address constants
0x00000040 to 0x00000FFF	Unused
0x00001000 to 0x00007FFF	Read-write data space for BSL
0x00008000 to 0x00077FFF	Available as download area for user code and data
0x00078000 to 0x0007FFFF	System and user stacks

Memory Map under Angel debug monitor

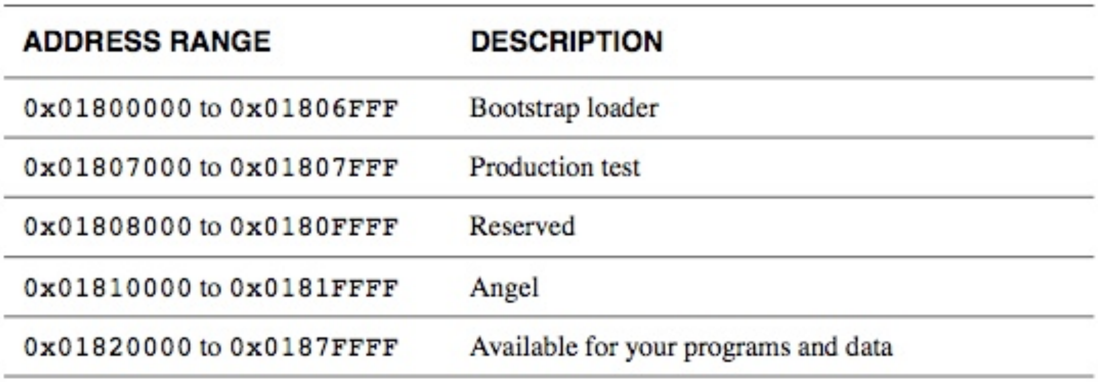
Table 3-3 SRAM usage under Angel



Address range	Description
0x00000000 to 0x0000003F	Exception vector table and address constants
0x00000040 to 0x000000FF	Unused
0x00000100 to 0x00007FFF	Read-write data and privileged mode stacks
0x00008000 to 0x00073FFF	Available as download area for user code and data
0x00074000 to 0x0007FFFF	Angel code execution region

Flash Memory Map

Table 3-4 Flash memory usage



ADDRESS RANGE	DESCRIPTION
0x01800000 to 0x01806FFF	Bootstrap loader
0x01807000 to 0x01807FFF	Production test
0x01808000 to 0x0180FFFF	Reserved
0x01810000 to 0x0181FFFF	Angel
0x01820000 to 0x0187FFFF	Available for your programs and data

SRAM Memory Interface

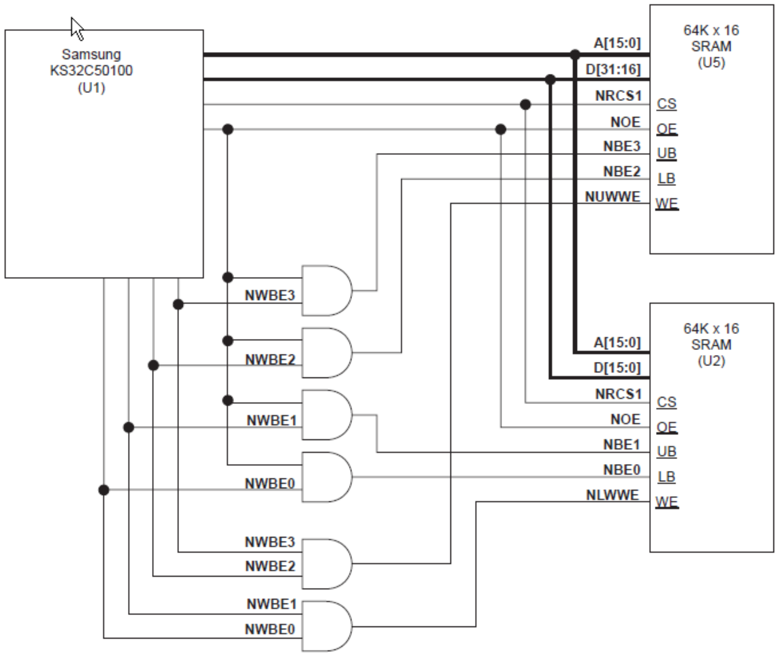


Figure 2-3 SRAM memory array

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 **R**egister

`STR`: **ST**ore words from a **R**egister 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	0	0	1	0	1	1	0	0	1	Rn				Rd				0	0	0	0	0	immed_5					0	0

cond code *op code* 32 bit ARM LDR *reserved*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	immed_5					Rn			Rd		

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 **`STR <Rd>, [<Rn>, #<immed_5> * 4]`**

- **`<Rd>`** Register containing Word to Write to Memory
- **`<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
   Memory[address,4] <- Rd
else
   data <- UNPREDICTABLE
```

Store Instruction Format - STR

Syntax

`STR <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	0	0	1	0	1	1	0	0	0	Rn				Rd				0	0	0	0	0	immed_5					0	0

32 bit ARM STR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	immed_5					Rn			Rd		

16 bit Thumb STR

Examples

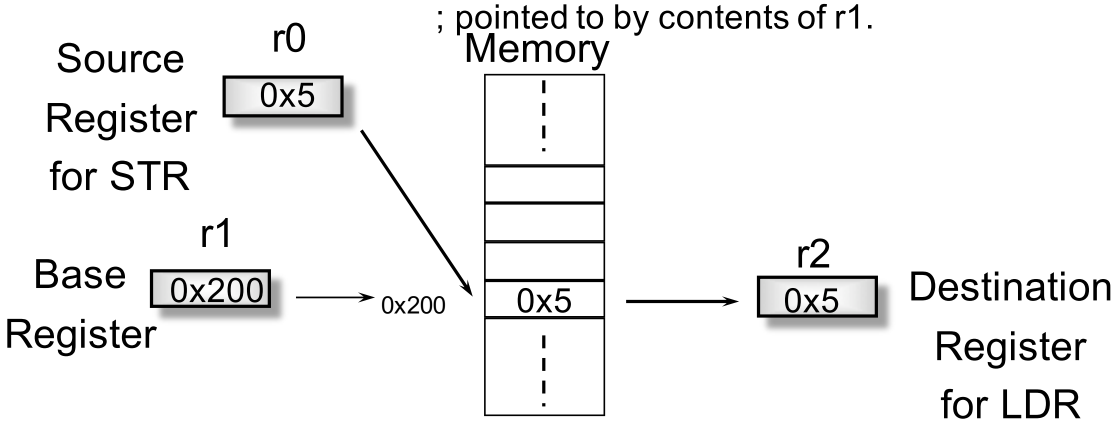
```
 LDR     r0, [r1]
;  load r0 with the content of the memory location
;  pointed to in r1

 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.



Source Register for STR

r0

0x5

Base Register

r1

0x200

0x200

Memory

0x5

r2

0x5

Destination Register for LDR

Common Load/Store Instructions

Loads	Stores	Size and Type
LDR	**STR**	Word (32 bits)
LDRB	**STRB**	Byte (8 bits)
LDRH	**STRH**	Halfword (16 bits)
LDRSB		Signed Byte
LDRSH		Signed Halfword
LDM	**STM**	Multiple Words

Load Half Word Example

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

r0 content
before/after load

`0x00008000`

r11 before load

`0x12345678`

r11 after load

`0x0000FFEE`

Memory

Address	Data
0x8000	0xEE
0x8001	0xFF
0x8002	0x90
0x8003	0xA7

Is this Little Endian or Big Endian ?

Load Half Word Example

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

r0 content before/after load

`0x00008000`

r11 before load

`0x12345678`

r11 after load

`0x0000FFEE`

Memory

Address	Data
0x8000	0xEE
0x8001	0xFF
0x8002	0x90
0x8003	0xA7

Little Endian

Signed Byte Load Example

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

r0 content before/after load

`0x00008000`

r11 before load

`0x12345678`

r11 after load

`0xFFFFFFEE`

Memory

Address	Data
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7

Store Example Using Post Increment

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

inc. by 4

r3 content before/after store

`0xFEEDBABE`

r8 before store

`0x00008000`

r8 after store

`0x00008004`

Memory after Store

Address	Data
0x8000	0xBE
0x8001	0xBA
0x8002	0xED
0x8003	0xFE

Is this Little Endian or Big Endian ?

Store Example Using Post Increment

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

r3 content before/after store

`0xFEEDBABE`

r8 before store

`0x00008000`

r8 after store

`0x00008004`

Memory after Store

Address	Data
0x8000	0xBE
0x8001	0xBA
0x8002	0xED
0x8003	0xFE

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: `LDR r0, [r1]`

Here, the `r1` content is known as the ‘base address’, and `r1` is called the **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 **before** executing the load/store.

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

Example: `LDR r0, [r1, #8]`

This instruction loads `r0` with the content of memory location at (base address + 8).

Optional `!` 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: `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.

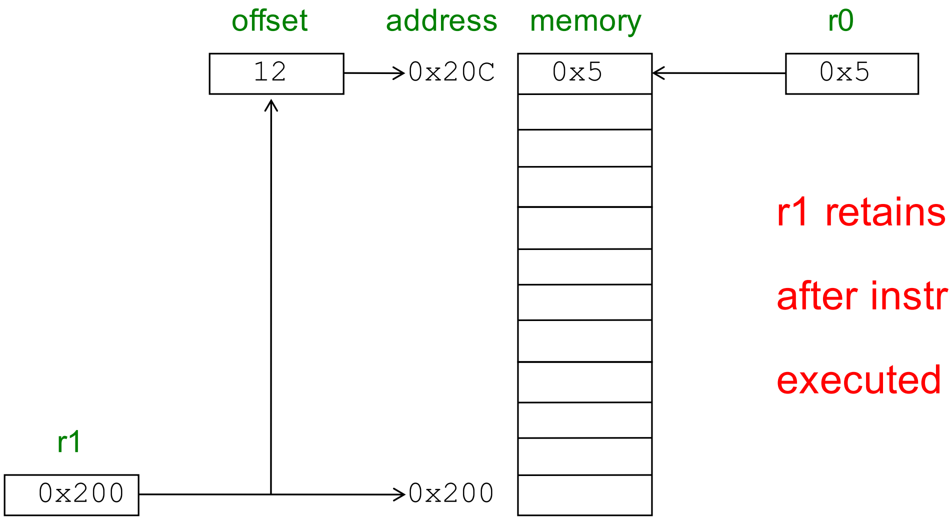
Increment to the base address is done <u>before</u> the execution of the load instruction but `r1` changes value after execution.

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

Pre-Indexed Example

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

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



r1 retains 0x200

after instruction is

executed

STR Pre-indexed Addressing

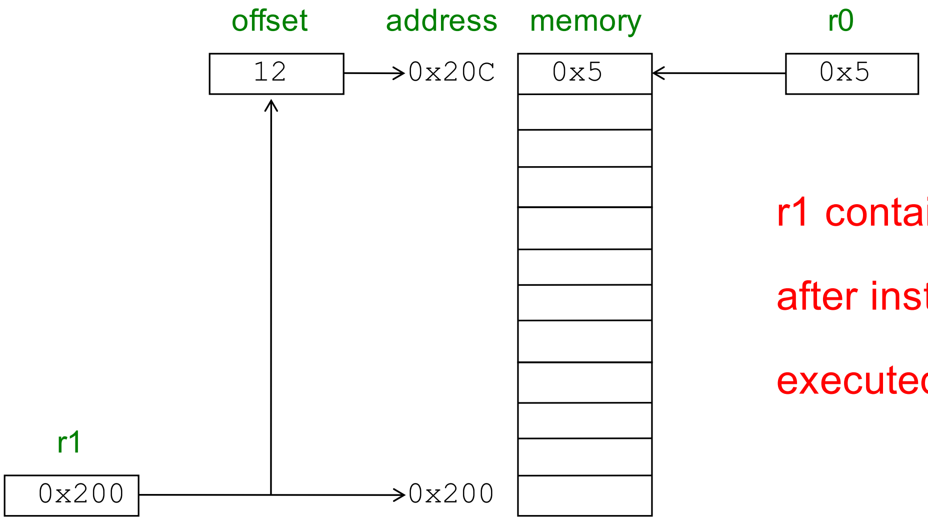
```
STR   r0, [r1, #12]     ;writes 0x5 to address 0x20C
```

- 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.

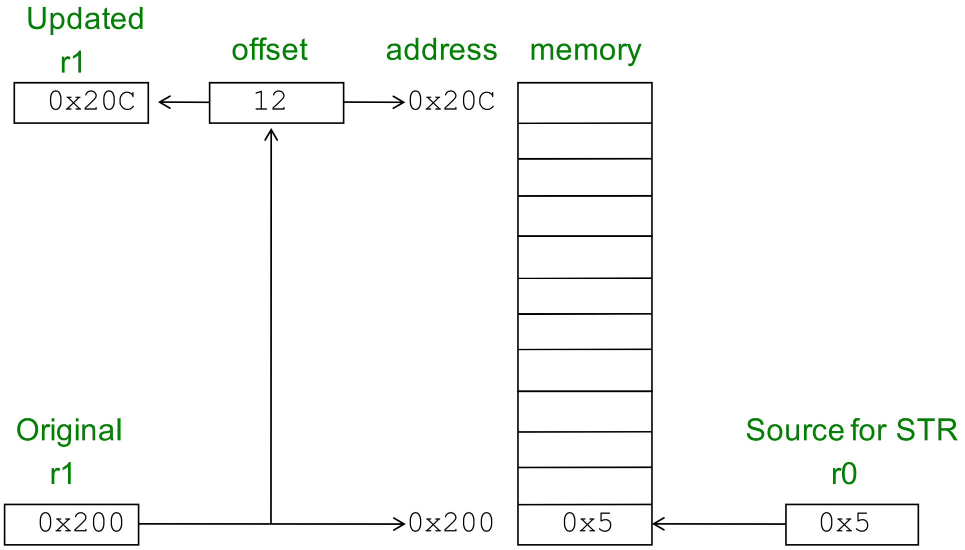
Note that ' !' is not needed for post-indexed addressing since the update is implicit.

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
```



STR: Post-indexed Addressing

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

- To auto-increment the base register to location `0x1f4` instead use:
  - `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`

Post-Indexed Examples

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

;32 bit data in r7 written to ea(r0) and
;r0 is updated to contain r0 <- r0+24 after write
STR   r7, [r0], #24

;r3 gets 32 bits of data from address beginning at
;ea(r0) and r0 is updated to contain r0 <- r0+(r4/16)
LDR   r2, [r0], r4, ASR #4

;r3 gets 16 bits of data from address beginning at
;ea(r9)and r9 is updated to contain r9 <- r9+2
LDRH  r3, [r9], #2

;16 bit data written from r2 to address beginning at
;ea(r5) and r5 is updated to contain r5 <- r5+8
STRH  r2, [r5], #8
```

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)

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	0	0	1	0	1	1	0	0	1	Rn				Rd				0	0	0	0	0	immed_5					0	0

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                         ;end of file marker
```

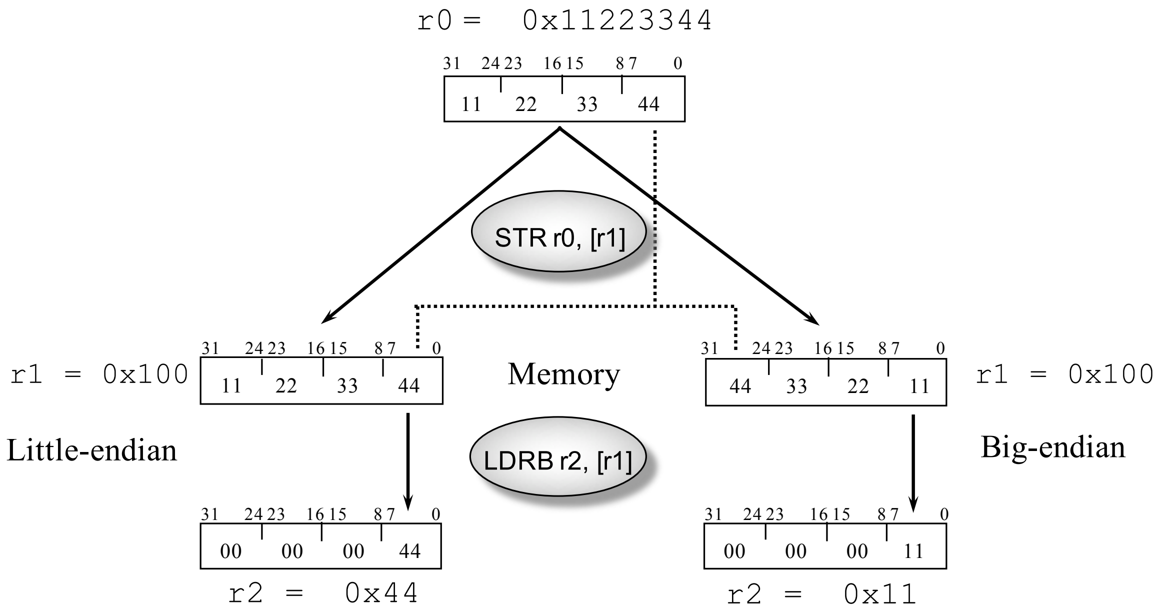
Endian

- 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

31 24 23 16 15 8 7 0
11 22 33 44

STR r0, [r1]

r1 = 0x100

31 24 23 16 15 8 7 0
11 22 33 44

Little-endian

Memory

31 24 23 16 15 8 7 0
44 33 22 11

r1 = 0x100

Big-endian

LDRB r2, [r1]

31 24 23 16 15 8 7 0
00 00 00 44

r2 = 0x44

31 24 23 16 15 8 7 0
00 00 00 11

r2 = 0x11

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 **M**ultiple 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 `LDM` and `STM` instruction execution will be pending until the execution is fully completed.