EEE447 Introduction to Microprocessors

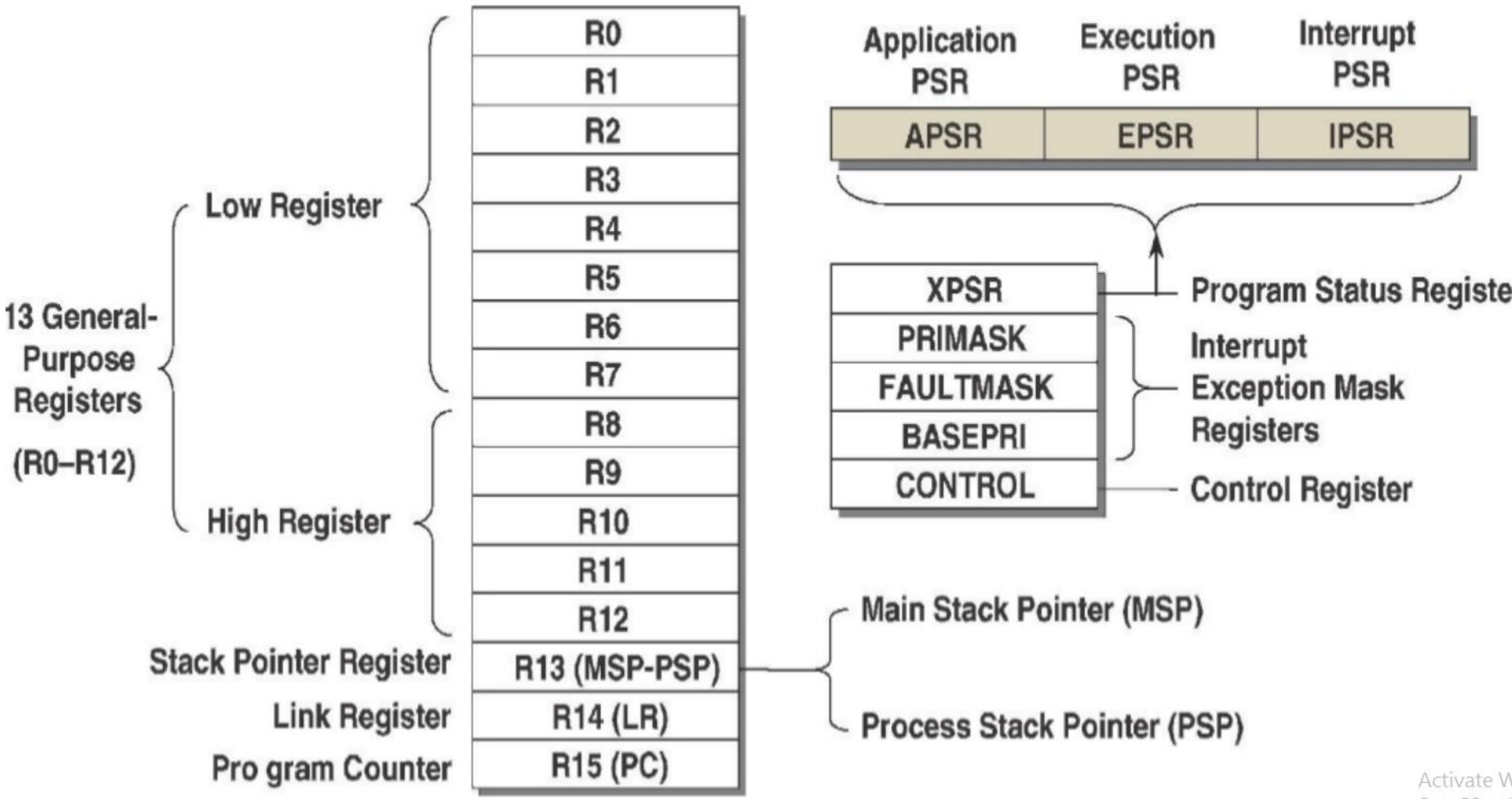
Chapter 2

Programming Model

Programming model

- To be able to program a MCU, assembly language programmer should know in depth the folowing features:
  - CPU registers
  - Instruction set (operations, operands, data sizes and types)
  - Addressing modes
  - Memory organization

ARM Cortex-M3/4 Registers



ARM Cortex-M3/4 Registers

R0-R12 - General purpose registers for data processing

- R0-R7 (Low registers) many 16-bit instructions only access these registers.
- R8-R12 (High registers) can be used with 32-bit instructions.

SP - Stack pointer (Banked R13)

- Can refer to one of two SPs.
  - Main Stack Pointer (MSP)
  - Process Stack Pointer (PSP)
- Uses MSP initially, and whenever in Handler mode.
- In Thread mode, can select either MSP or PSP using CONTROL register.

LR - Link Register (R14)

- Holds return address when called with Branch & Link instruction (BL).

PC - program counter (R15)

ARM Cortex-M4 Program Status Register

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

Program Status Register (PSR) is three views of same register:

- Application PSR (APSR)
  - Condition code flag bits Negative, Zero, Carry, Overflow, DSP overflow and saturation, Great-Than or Equal
- Interrupt PSR (IPSR)
  - Holds exception number of currently executing ISR
- Execution PSR (EPSR)
  - ICI/IT, Interrupt-Continuable Instruction, IF-THEN instruction
  - Thumb state, always 1

APSR

Bits	Name	Function
[31]	N	Negative flag
[30]	Z	Zero flag
[29]	C	Carry or borrow flag
[28]	V	Overflow flag
[27]	Q	DSP overflow and saturation flag
[26:20]	-	Reserved
[19:16]	GE[3:0]	Greater than or Equal flags.
[15:0]	-	Reserved

IPSR

Bits	Name	Function
[31:9]	-	Reserved
[8:0]	ISR_NUMBER	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4 = MemManage 5 = BusFault 6 = UsageFault 7-10 = Reserved 11 = SVCall 12 = Reserved for Debug 13 = Reserved 14 = PendSV 15 = SysTick 16 = IRQ0. . . .

Interrupt Exception Mask Registers

PRIMASK, FAULTMASK, and BASEPRI

- Used to mask exceptions based on priority levels.
- Only accessed in the privileged access level.
- By default, all zero, which means the masking (disabling of exception/interrupt) is not active.

PRIMASK

- 1-bit wide interrupt mask register.
- When set, it blocks all exceptions (including interrupts) apart from the Non-Maskable Interrupt (NMI) and the HardFault exception.
- The most common usage for PRIMASK is to disable all interrupts for a time critical process.

- <u>FAULTMASK</u> register
  - Very similar to PRIMASK, but it also blocks the HardFault exception.
- <u>BASPRI</u> register
  - Defines the priority of the executing software.
  - It prevents interrupts with lower or equal priority.

Control Register

- Controls the stack used and the privilege level for software execution when the processor is in Thread mode.
- If implemented, indicates whether the FPU state is active.

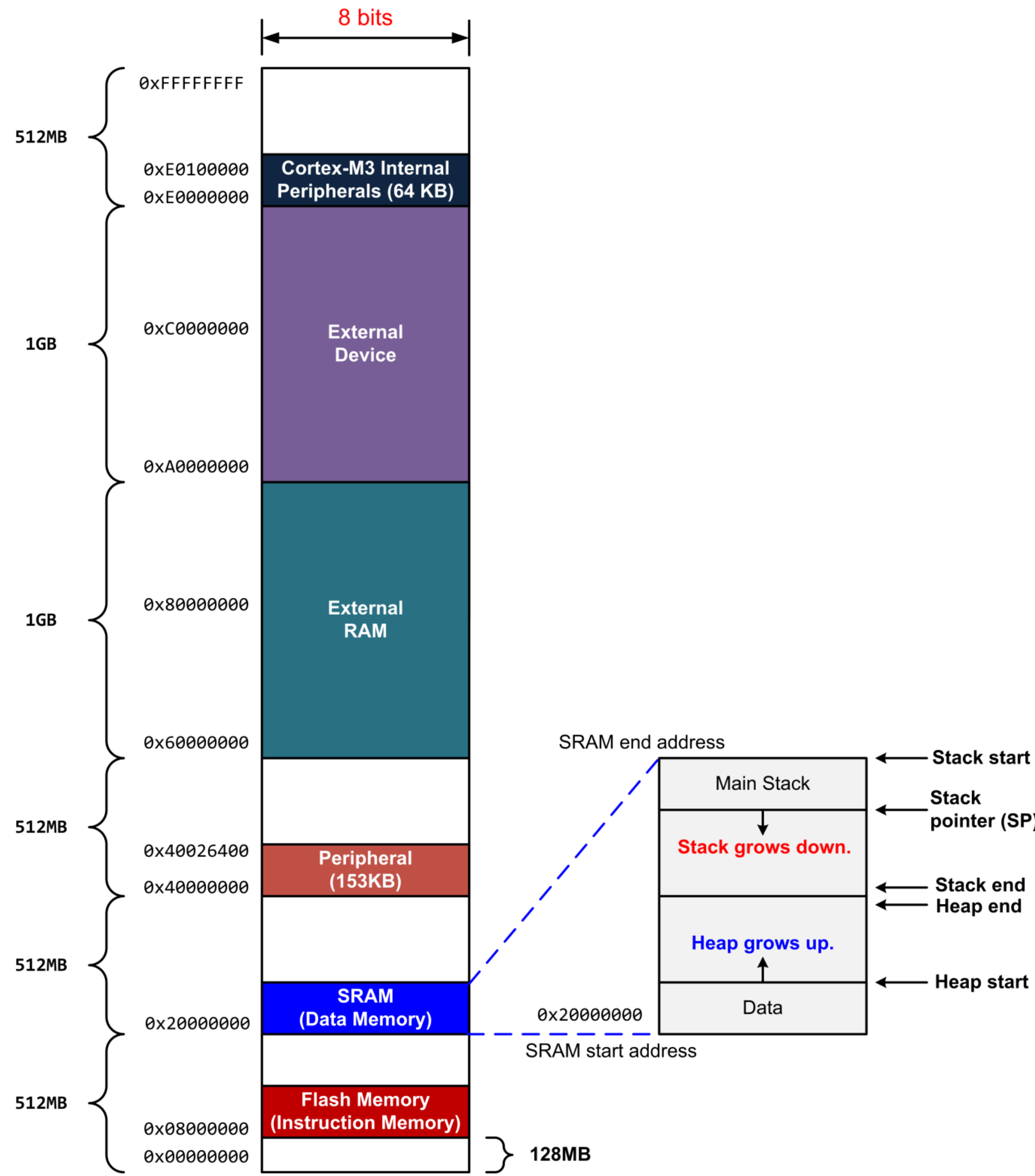
Bits	Name	Function
[31:3]	-	Reserved.
[2]	FPCA	When floating-point is implemented this bit indicates whether context floating-point is currently active: 0 = no floating-point context active 1 = floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
[1]	SPSEL	Defines the currently active stack pointer: In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return: 0 = MSP is the current stack pointer 1 = PSP is the current stack pointer.
[0]	nPRIV	Defines the Thread mode privilege level: 0 = privileged 1 = unprivileged.

Memory Map

Memory is needed as

- Code memory (normally read-only memory)
  - Program instructions
  - Constant data
- Data memory (normally read/write memory – RAM)
  - Variable data/operands
- Stack (located in data memory)
  - Special Last-In/First-Out (LIFO) data structure
    - Save information temporarily and retrieve it later
    - Save return addresses for subroutines and interrupt/exception handlers
    - Data to be passed to/from a subroutine/function can be saved
  - Stack Pointer register (r13/sp) points to last item placed on the stack
- Peripheral addresses
  - Used to access registers in “peripheral functions” (timers, ADCs, communication modules, etc.) outside the CPU

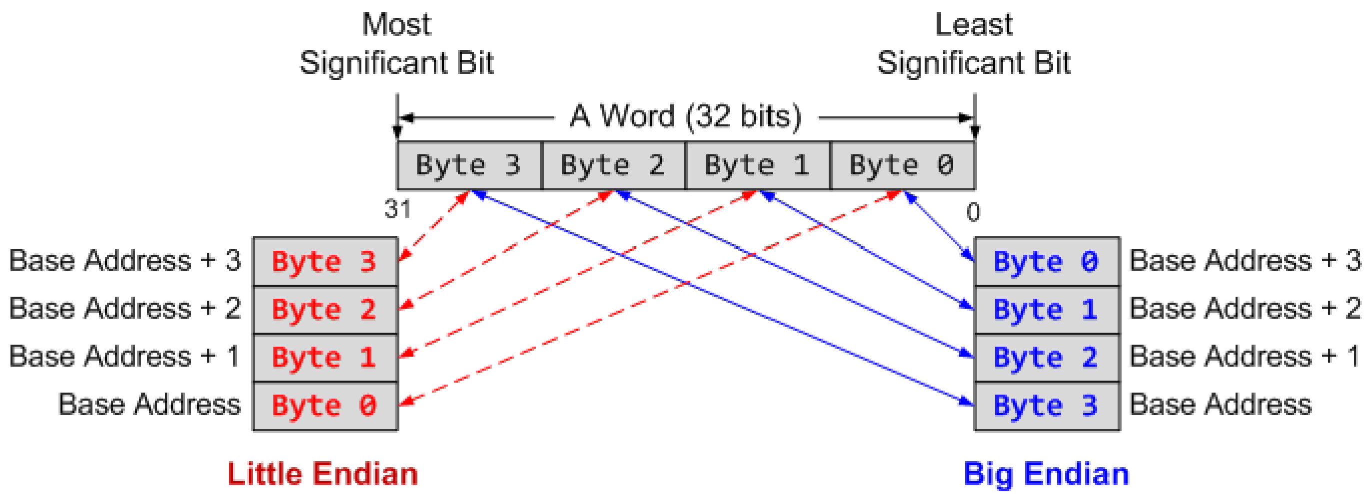
- The Cortex-M3 and M4 processors themselves do not include memories (i.e.,they do not have program memory, SRAM, or cache).
- Instead, they come with a generic on-chip bus interface, so microcontroller vendors can add their own memory system to their design.
- In this way, different microcontroller products can have different memory configurations, different memory sizes and types, and different peripherals.



8 bits
0xFFFFFFFF
512MB
0xE0100000
Cortex-M3 Internal Peripherals (64 KB)
0xE0000000
1GB
0xC0000000
External Device
0xA0000000
1GB
0x80000000
External RAM
0x60000000
512MB
0x40026400
Peripheral (153KB)
0x40000000
512MB
SRAM (Data Memory)
0x20000000
512MB
Flash Memory (Instruction Memory)
0x08000000
0x00000000
128MB
SRAM end address
Main Stack
Stack start
Stack pointer (SP)
Stack grows down.
Stack end
Heap end
Heap grows up.
Heap start
Data
0x20000000
SRAM start address

ARM Cortex-M4 Memory Formats (Endian)

- Memory address is always in terms of bytes.
- How data is organized in memory?



ARM Cortex-M3/4 Memory Formats (Endian)

- Default memory format for ARM CPUs: <u>LITTLE ENDIAN</u>
- Bytes 0-3 hold the first stored word
- Bytes 4-7 hold the second stored word
- Processor contains a configuration pin BIGEND
  - Enables hardware system developer to select format:
    - Little Endian
    - Big Endian (BE-8)
  - Pin is sampled on reset
  - Cannot change endianness when out of reset

Byte addresses

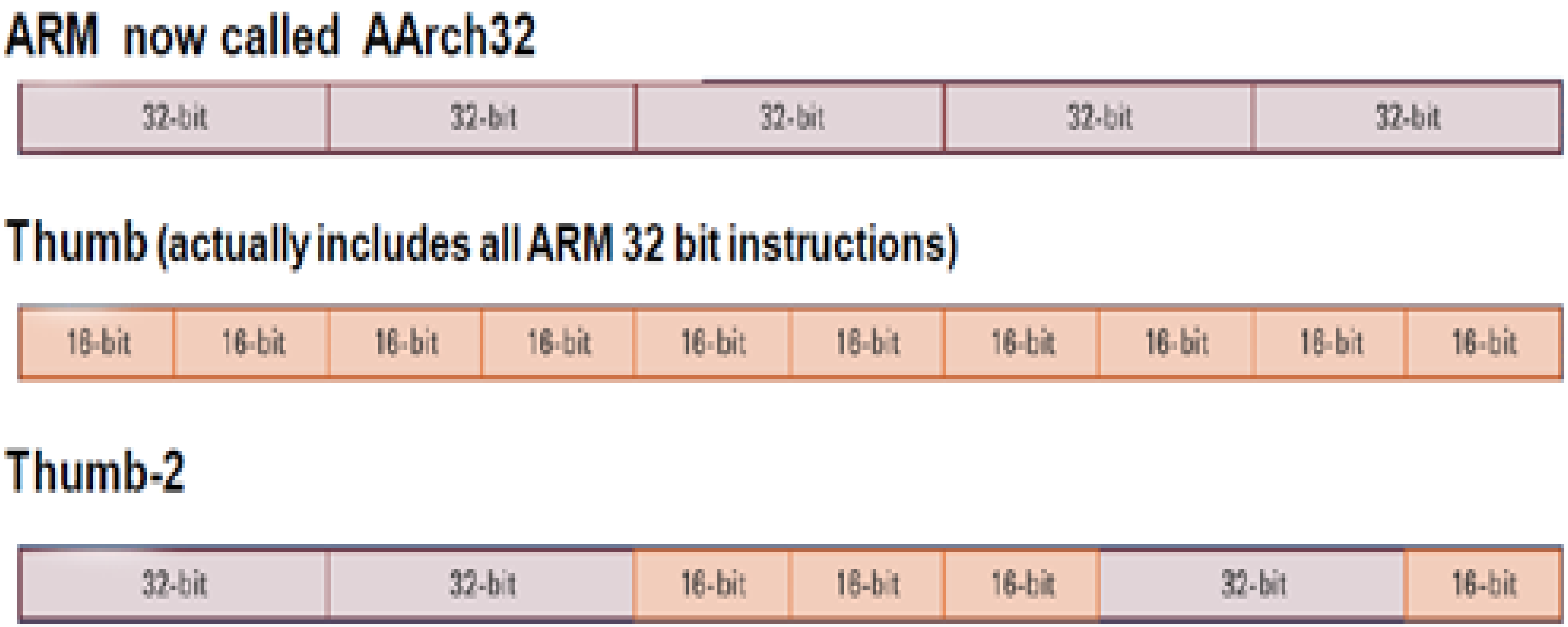
Byte 3	Byte 2	Byte 1	Byte 0	
103	102	101	100	Word 100
107	106	105	104	Word 104
10B	10A	109	108	Word 108
10F	10E	10D	10C	Word 10C

EE447

!!!ATTENTION!!!

Those who have failed to receive "Time Table Request" e-mail please send an e-mail to ee447coordination@gmail.com to request your "Time Table Form" to be filled out by October 19th, 2018.

Instruction Set



Label mnemonic {Rd}, {Rn}, {operand2} ; Comments

(opcode)

```
Loc   ADD    R3,    R2,    R1     ;R2+R1->R3
      ADD    R3,    R2,    #5     ;R2+5->R3
      LDR    R3,   [R2]           ;R3 = value pointed by R2
```

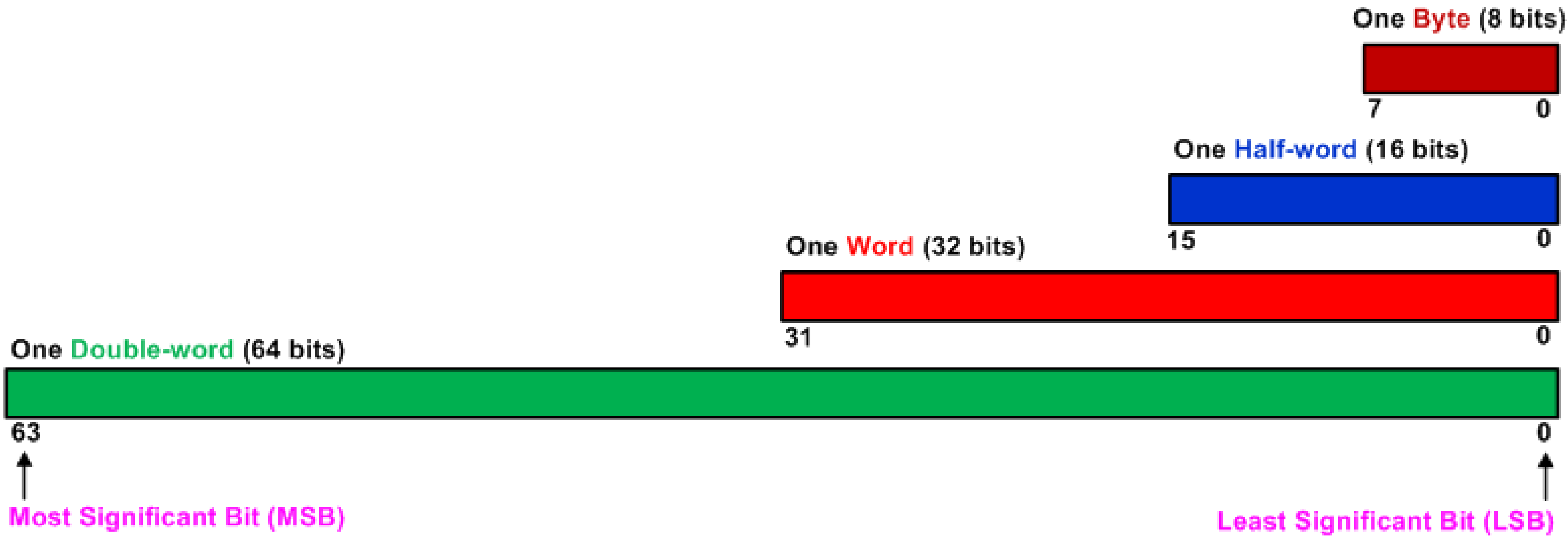
If exists, Rd is typically the destination register

If exists, Rn is typically the source register.

operand2 is the flexible second operand that can be either a register (Rm), shifted register or a constant (immed_8r)

Data types supported

- Integer ALU operations are performed only on 32-bit data
  - Signed or unsigned integers
- Data sizes in memory:
  - Byte (8-bit), Half-word (16-bit), Word (32-bit), Double Word (64-bit)



Data types supported

- Bytes/half-words are converted to 32-bits when moved into a register
  - Signed numbers –extend sign bit to upper bits of a 32-bit register
  - Unsigned numbers –fill upper bits of a 32-bit register with 0's
  - Examples:
    - 255 (unsigned byte) 0xFF=>0x000000FF (fill upper 24 bits with 0)
    - -1 (signed byte) 0xFF=>0xFFFFFFFF (fill upper 24 bits with sign bit 1)
    - +1 (signed byte) 0x01=>0x00000001 (fill upper 24 bits with sign bit 0)
    - -32768 (signed half-word) 0x8000=>0xFFFF8000 (sign bit = 1)
    - 32768 (unsigned half-word) 0x8000=>0x00008000
    - +32767 (signed half-word) 0x7FFF=>0x00007FFF (sign bit = 0)
- Cortex-M4F also supports single and double-precision IEEE floating-point data

Addressing modes

- The addressing mode is the format the instruction uses to specify the memory location to read or write data.
  - Generic types of addressing modes are immediate, direct, indirect or indexed, relative.
  - ARM does not use direct addressing mode since 32-bit address can not be included in a 32-bit inctruction.

Immediate addressing mode

The data itself is contained in the instruction. Once the instruction is fetched no additional memory access cycles are needed to get the data. It is only used to get, load or read data.

Opcode Rd, #constant

Ex:

```
MOV   R0, #100     : R0=100
ADD   R0,#0x64     ; R0=R0+100
```

Indexed addressing mode

- The data is in memory and a register will contain a pointer to the data. Once the instruction is fetched, one or more additional memory access cycles are required to read or write the data.
  - Can include an offset from the index address
  - Can include updating index register with offset (pre- or post- access)

Opcode R1,[R2, optional offset, optional shift], optional offset

Several forms of indexed addressing

```
LDR{type}    Rd,[Rn]        ;load memory at [Rn] to Rd
STR{type}    Rt,[Rn]        ;store Rt to memory at [Rn]
LDR{type}    Rd,[Rn, #n]    ;load memory at [Rn+n] to Rd,
                            ;Rn unchanged
```

PC-relative addressing mode

It is indexed addressing using PC as the pointer. It is represented in the instruction as the PC value plus or minus a numeric offset. The assembler calculates the required offset from the label and the address of the current instruction. If the offset is too big, the assembler produces an error.

This addressing mode is used for branching, calling functions, etc.

Ex:

```
0x00001000              ADR R5, label ;  R5=PC+$offset where offset
                                          is from PC to data=$0C
                .
                .
0x0000100C      label   LDR     r0,[r1]
```

Pseudo instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM or Thumb instructions at assembly time.

In loading data

- Load a 32-bit constant (data, address, etc .) into a register. Cannot embed 32-bit value in a 32-bit instruction
- Use a “pseudo-op” to load the constant: Either LDR with relative addressing or MOV
- A pseudo-operation(pseudo-op) is translated by the assembler to one or more actual ARM instructions

Example:

Addres	Source Program	Debug Disassembly
	MOV**32** r0,#0x12345678	MOVW r0,#0x5678
		MOVT r0,#0x1234

0x0000050C LDR r3,=0x12345678 LDR r3,[PC,#offset]

.....

0x0000051C dcw 0x5678 ;in literal pool following the code
0x0000051E dcw 0x1234

****

Destination address = Pc+offset ; Offset = 16

0x0000051C = 0x0000050C + 16

Assembly

```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR   r1, =srcStr       ; Retrieve address of first string
        LDR   r0, =dstStr       ; Retrieve address of second string
        LDRB  r2, [r1], #1      ; Load a byte & increase src address pointer
        STRB  r2, [r0], #1      ; Store a byte & increase dst address pointer
        CMP   r2, #0            ; Check for the null terminator
        BNE   strcpy            ; Cope the next byte if string is not ended
stop    B     stop              ; Dead loop. Embedded program never exits.

        ENDP

        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB   "The source string.",0       ; Strings are null terminated
dstStr  DCB   "The destination string.",0  ; dststr has more space than srcstr

        END
```

Assembly

Code Area

```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR   r1, =srcStr        ; Retrieve address of the source string
        LDR   r0, =dstStr        ; Retrieve address of the destination string
loop    LDRB  r2, [r1], #1       ; Load a byte & increase src address pointer
        STRB  r2, [r0], #1       ; Store a byte & increase dst address pointer
        CMP   r2, #0             ; Check for the null terminator
        BNE   loop               ; Copy the next byte if string is not ended
stop    B     stop               ; Dead loop. Embedded program never exits.

        ENDP
```

Data Area

```
        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB   "The source string.",0       ; Strings are null terminated
dstStr  DCB   "The destination string.",0  ; dststr has more space than srcstr

        END
```

Assembly

Code Area

```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR   r1, =srcStr        ; Retrieve address of the source string
        LDR   r0, =dstStr        ; Retrieve address of the destination string
loop    LDRB  r2, [r1], #1       ; Load a byte & increase src address pointer
        STRB  r2, [r0], #1       ; Store a byte & increase dst address pointer
        CMP   r2, #0             ; Check for the null terminator
        BNE   loop               ; Copy the next byte if string is not ended
stop    B     stop               ; Dead loop. Embedded program never exits.

        ENDP

        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB   "The source string.",0        ; Strings are null terminated
dstStr  DCB   "The destination string.",0   ; dststr has more space than srcstr

        END
```

Data Area

Program Comments

Program Comments

Assembly

Labels

Code Area

```
           AREA string_copy, CODE, READONLY
           EXPORT   __main
           ALIGN
           ENTRY
__main     PROC

strcpy     LDR    r1, =srcStr              ; Retrieve address of the source string
           LDR    r0, =dstStr              ; Retrieve address of the destination string
loop       LDRB   r2, [r1], #1             ; Load a byte & increase src address pointer
           STRB   r2, [r0], #1             ; Store a byte & increase dst address pointer
           CMP    r2, #0                   ; Check for the null terminator
           BNE    loop                     ; Copy the next byte if string is not ended
stop       B      stop                     ; Dead loop. Embedded program never exits.

           ENDP

           AREA myData, DATA, READWRITE
           ALIGN

srcStr     DCB    "The source string.",0       ; Strings are null terminated
dstStr     DCB    "The destination string.",0  ; dststr has more space than srcstr

           END
```

Directives

Program Comments

Assembly Instructions

Directives

Data Area

Data

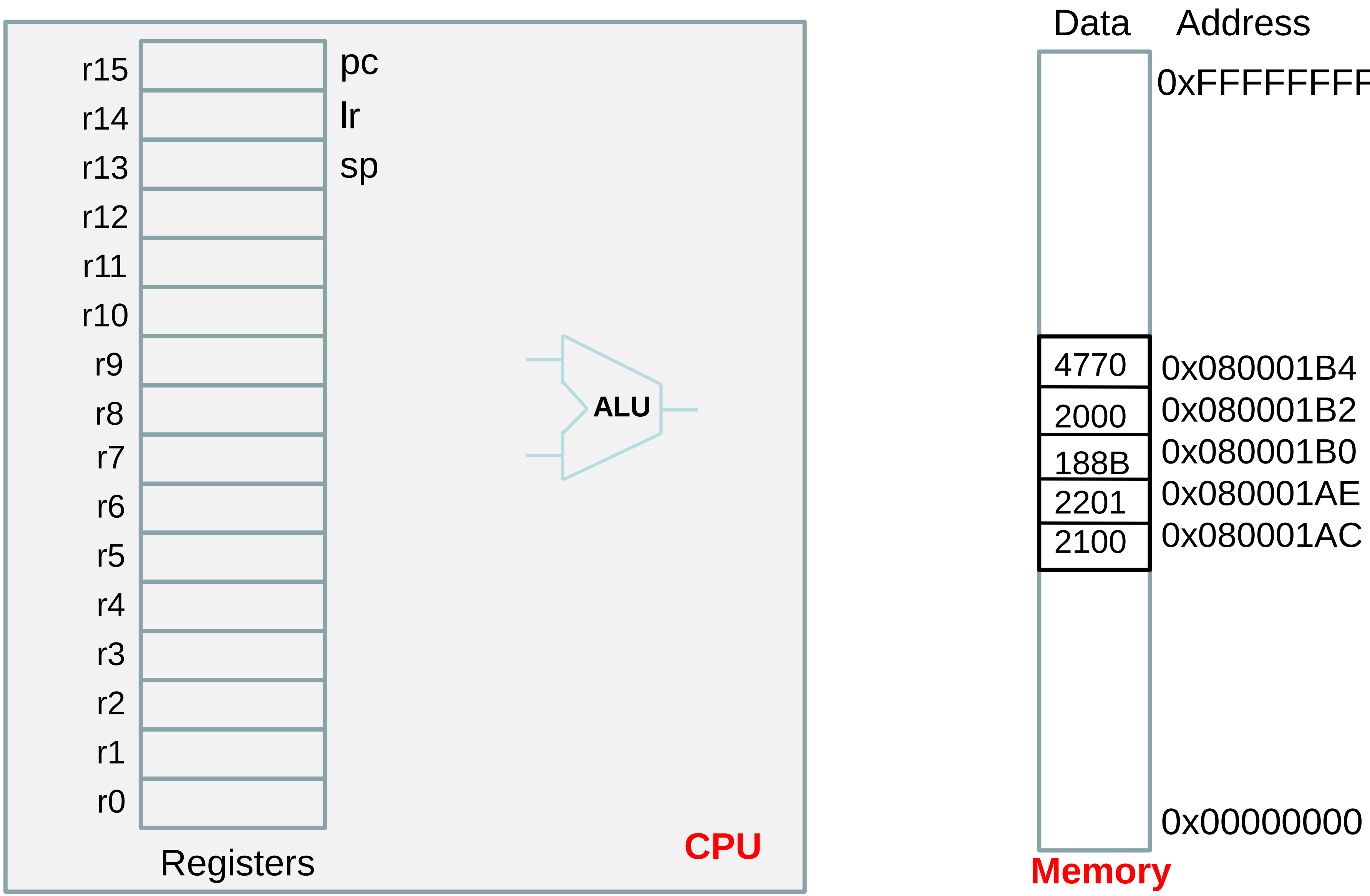
Program Comments

Assembly Directives

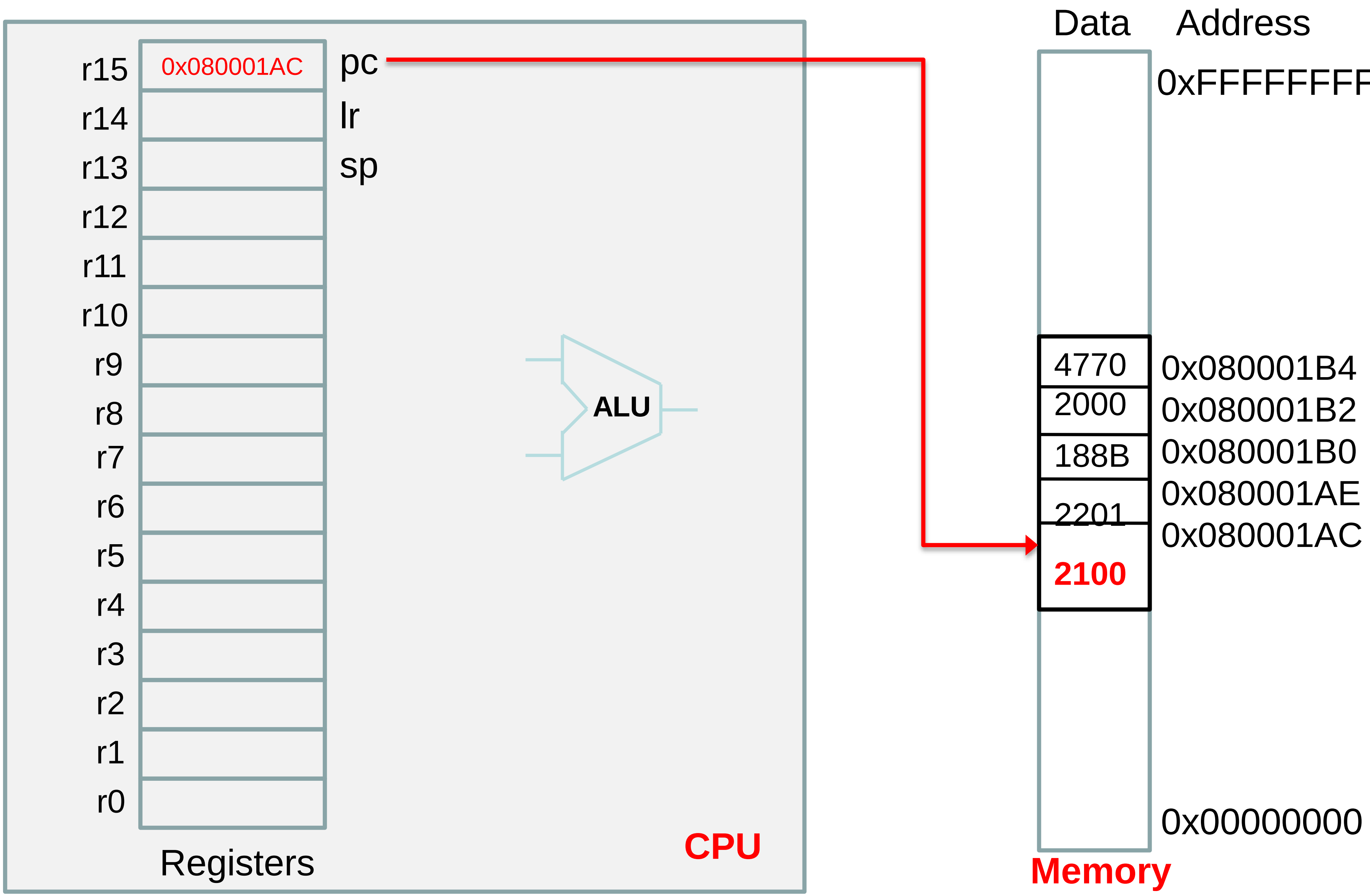
- Directives are **NOT** instruction. Instead, they are used to provide key information for assembly.

AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value.
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

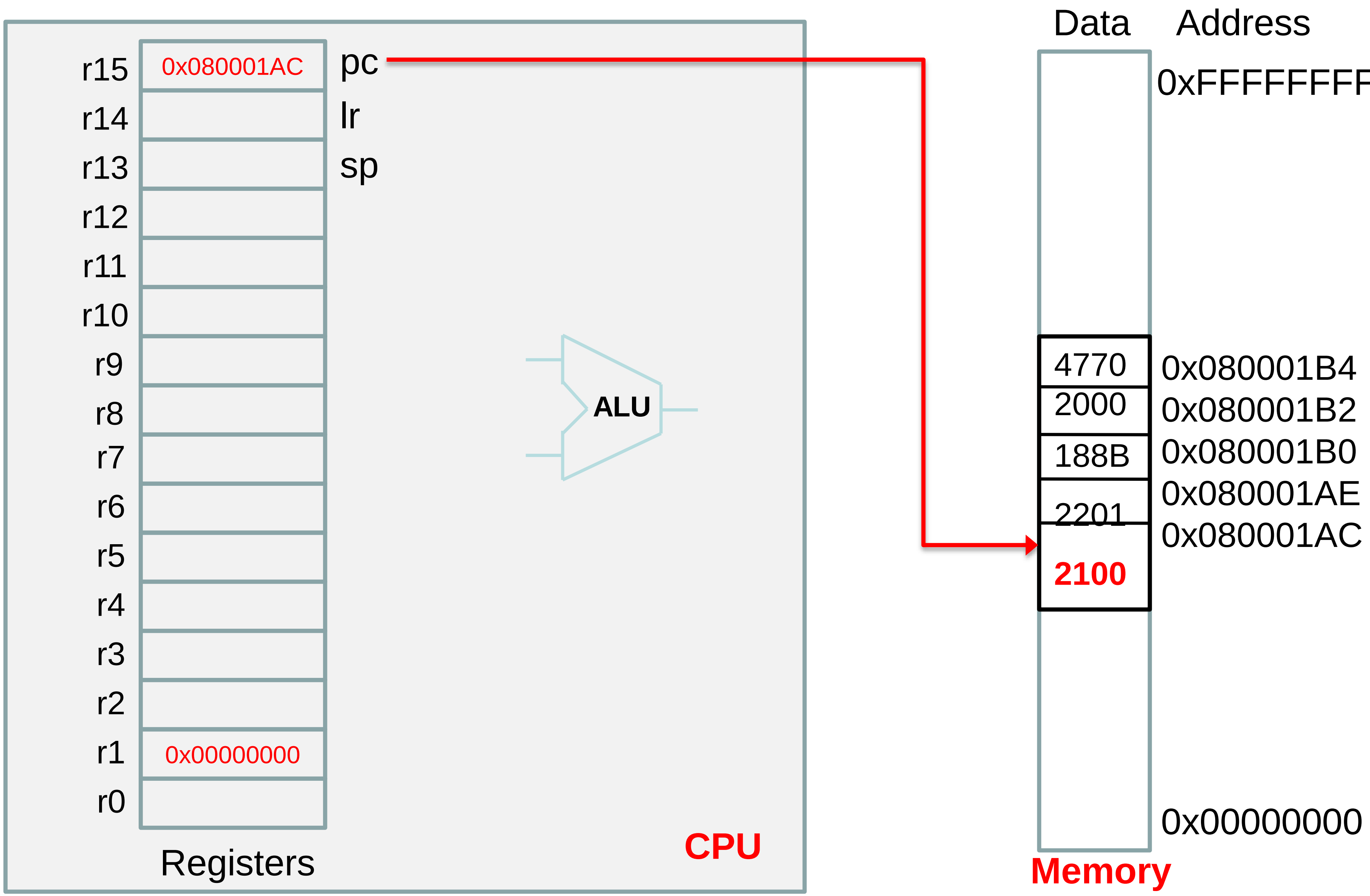
Instruction execution for ARM



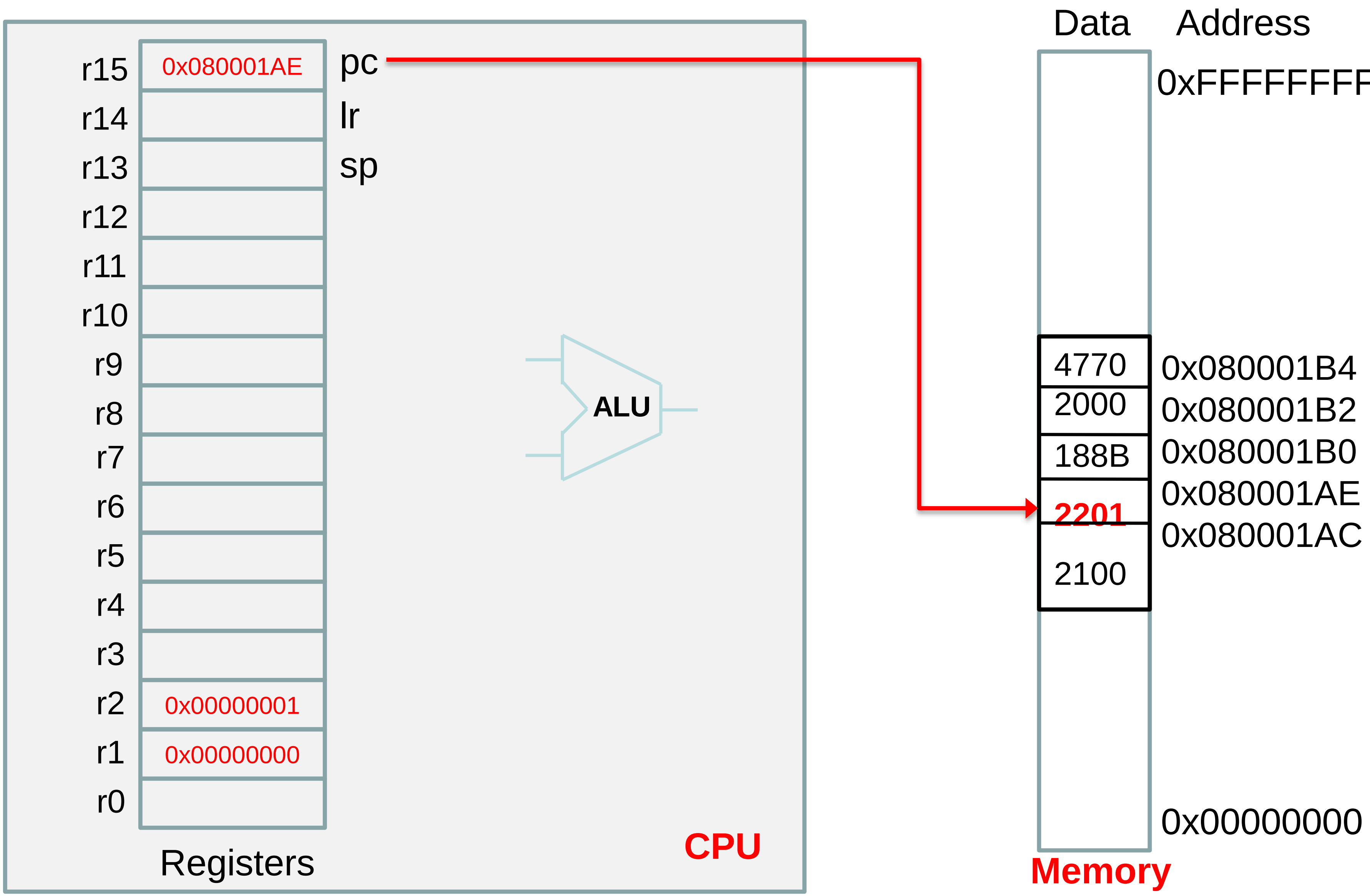
Fetch Instruction: pc = 0x08001AC
Decode Instruction: 2100 = MOVS r1, #0x00



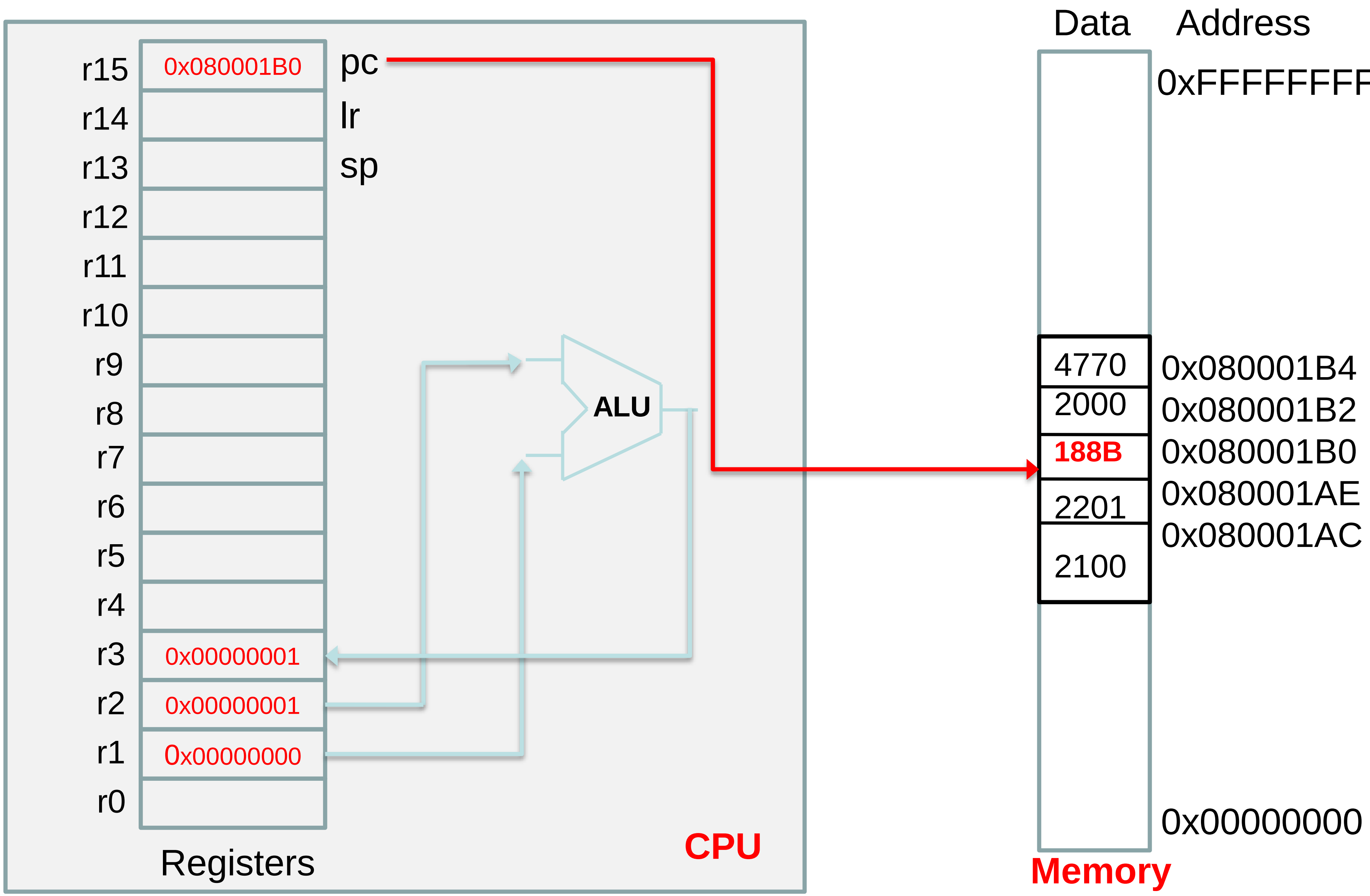
Execute Instruction:
MOVS r1, #0x00



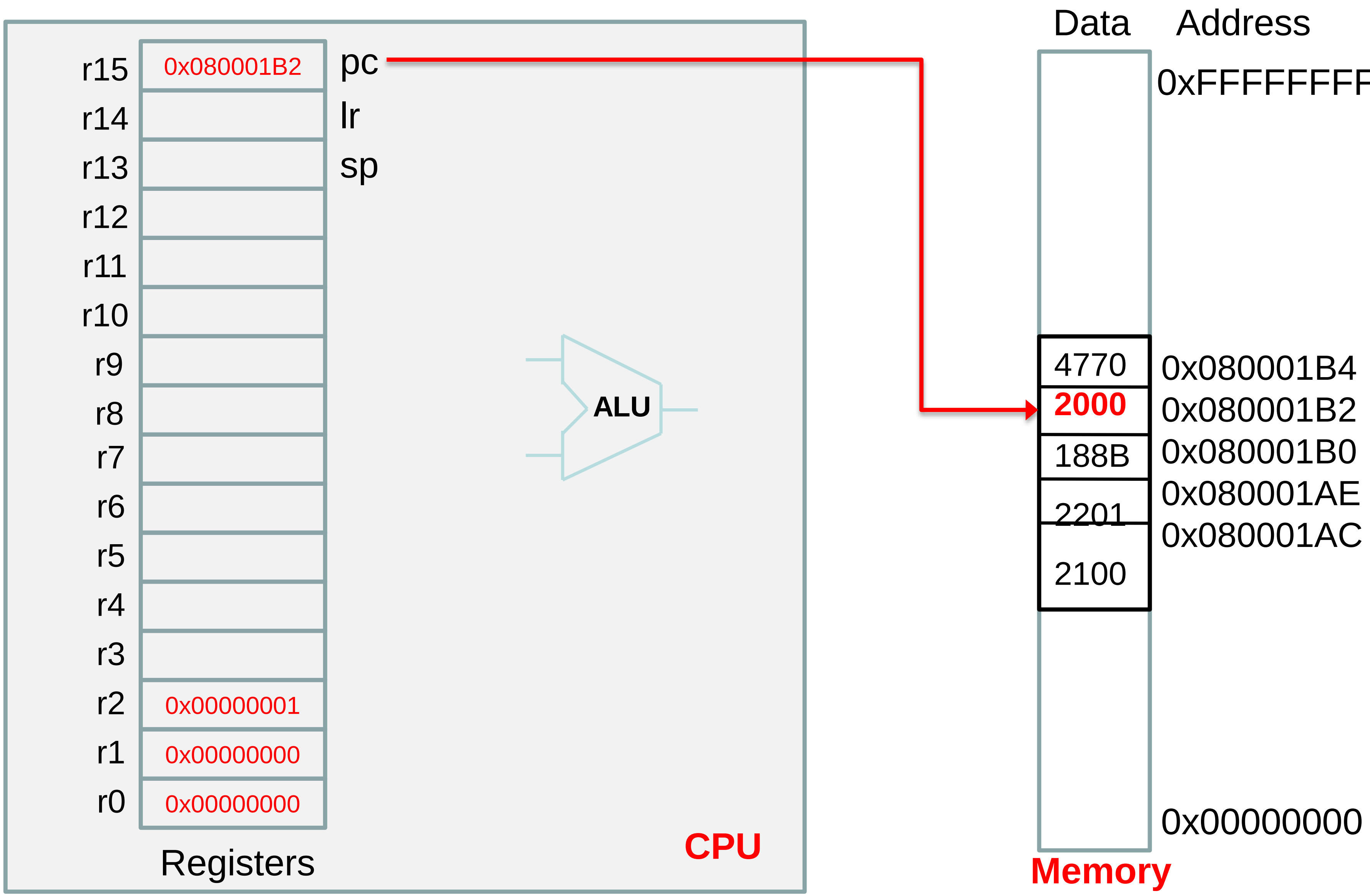
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2201 = MOVS r2, #0x01



Fetch Next Instruction: pc = pc + 2
Decode & Execute: 188B = ADDS r3, r1, r2



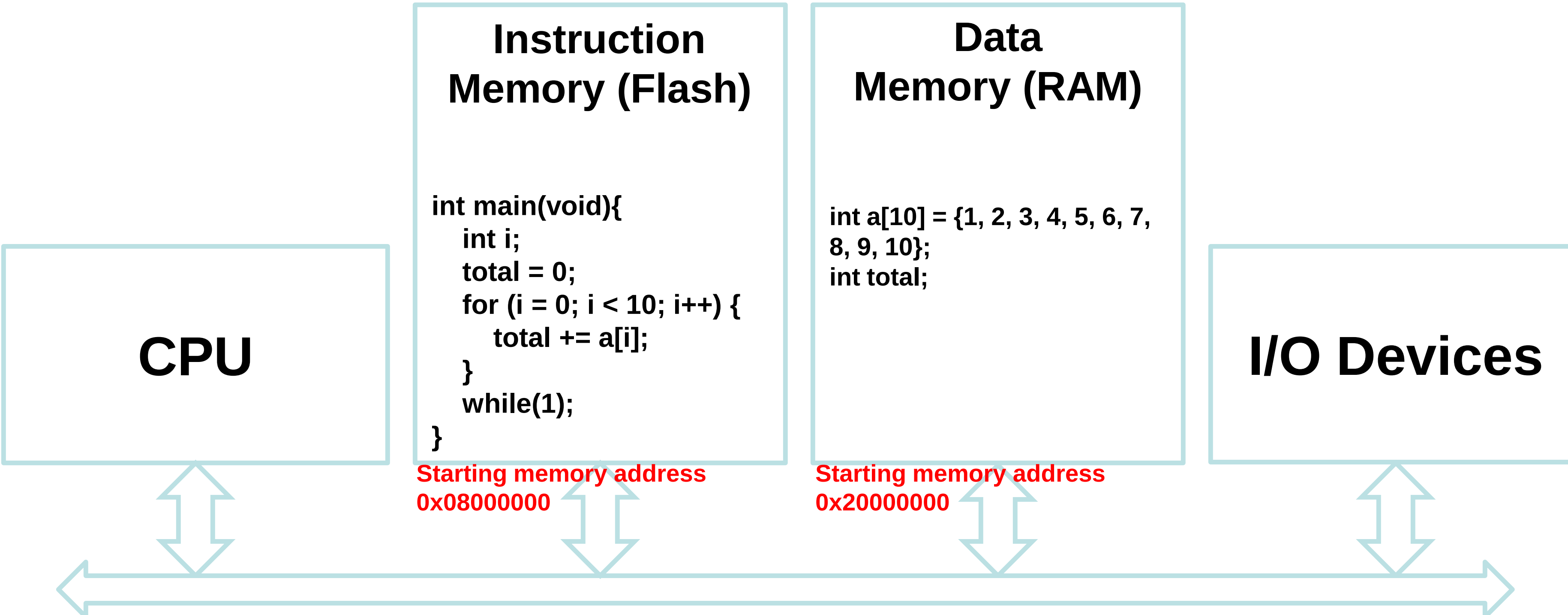
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2000 = MOVS r0, #0x00



Example:
Calculate the Sum of an Array

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;

int main(void){
        int i;
        total = 0;
        for (i = 0; i < 10; i++) {
                total += a[i];
        }
        while(1);
}
```

Example:
Calculate the Sum of an Array



Example:
Calculate the Sum of an Array

Instruction Memory (Flash)

```
int main(void){
    int i;
    total = 0;
    for (i = 0; i < 10; i++) {
        total += a[i];
    }
    while(1);
}
```

Starting memory address 0x08000000

```
0010 0001 0000 0000
0100 1010 0000 1000
0110 0000 0001 0001
0010 0000 0000 0000
1110 0000 0000 1000
0100 1001 0000 0111
1111 1000 0101 0001
0001 0000 0010 0000
0100 1010 0000 0100
0110 1000 0001 0010
0100 0100 0001 0001
0100 1010 0000 0011
0110 0000 0001 0001
0001 1100 0100 0000
0010 1000 0000 1010
1101 1011 1111 0100
1011 1111 0000 0000
1110 0111 1111 1110
```

⟺

```
        MOVS  r1, #0x00
        LDR   r2, = total_addr
        STR   r1, [r2, #0x00]
        MOVS  r0, #0x00
        B     Check
Loop:   LDR   r1, = a_addr
        LDR   r1, [r1, r0, LSL #2]
        LDR   r2, = total_addr
        LDR   r2, [r2, #0x00]
        ADD   r1, r1, r2
        LDR   r2, = total_addr
        STR   r1, [r2,#0x00]
        ADDS  r0, r0, #1
Check:  CMP   r0, #0x0A
        BLT   Loop
        NOP
Self:   B     Self
```

Example: Calculate the Sum of an Array

Data Memory (RAM)

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;
```

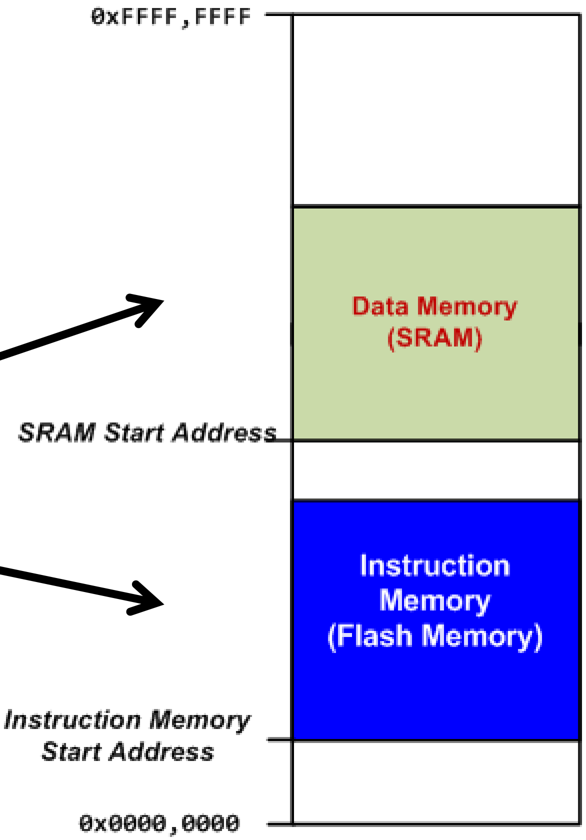
Assume the starting memory address of the data memory is 0x20000000

Memory address in bytes	Memory content	
0x20000000	0x0001	a[0] = 0x00000001
0x20000002	0x0000	
0x20000004	0x0002	a[1] = 0x00000002
0x20000006	0x0000	
0x20000008	0x0003	a[2] = 0x00000003
0x2000000A	0x0000	
0x2000000C	0x0004	a[3] = 0x00000004
0x2000000E	0x0000	
0x20000010	0x0005	a[4] = 0x00000005
0x20000012	0x0000	
0x20000014	0x0006	a[5] = 0x00000006
0x20000016	0x0000	
0x20000018	0x0007	a[6] = 0x00000007
0x2000001A	0x0000	
0x2000001C	0x0008	a[7] = 0x00000008
0x2000001E	0x0000	
0x20000020	0x0009	a[8] = 0x00000009
0x20000022	0x0000	
0x20000024	0x000A	a[9] = 0x0000000A
0x20000026	0x0000	
0x20000028	0x0000	total= 0x00000000
0x2000002A	0x0000	

Loading Code and Data into Memory

```
int counter;

int a[5] = {1, 2, 3, 4, 5};

int main(void){
   int i;

   int b[5];

   counter = 0;
   for (i = 0; i < 5; i++){
      b[i] = a[i];
      counter++
   }
   while(1);
}
```

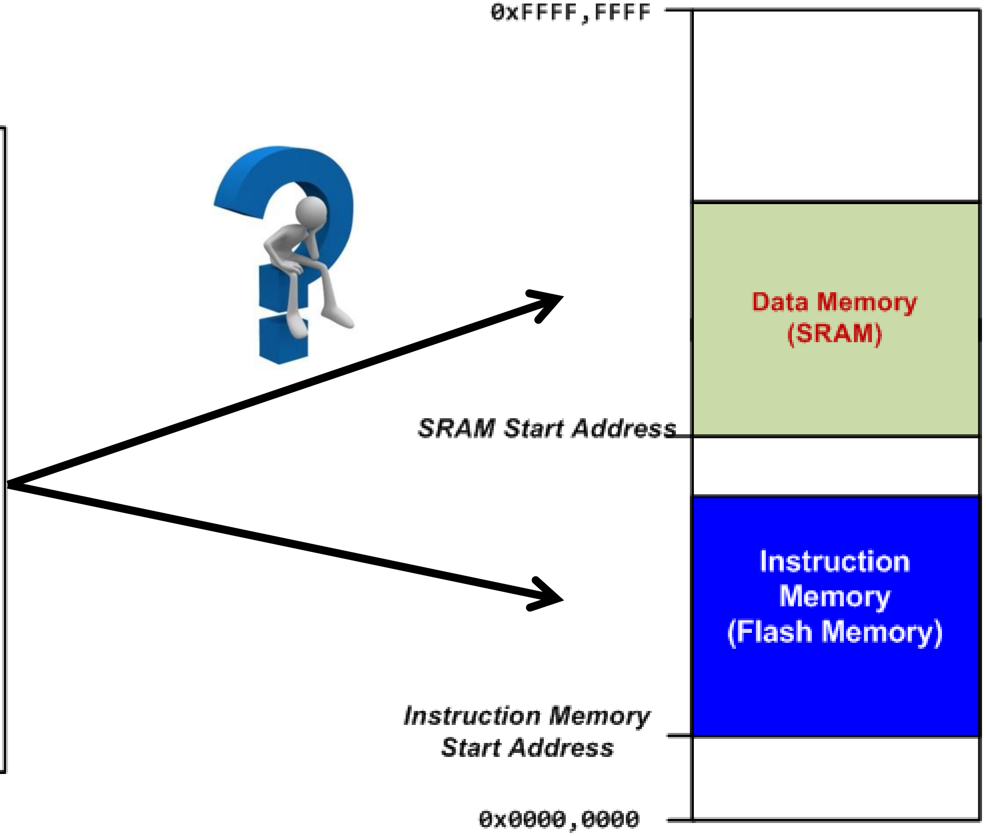
C Program:
Copying an Array



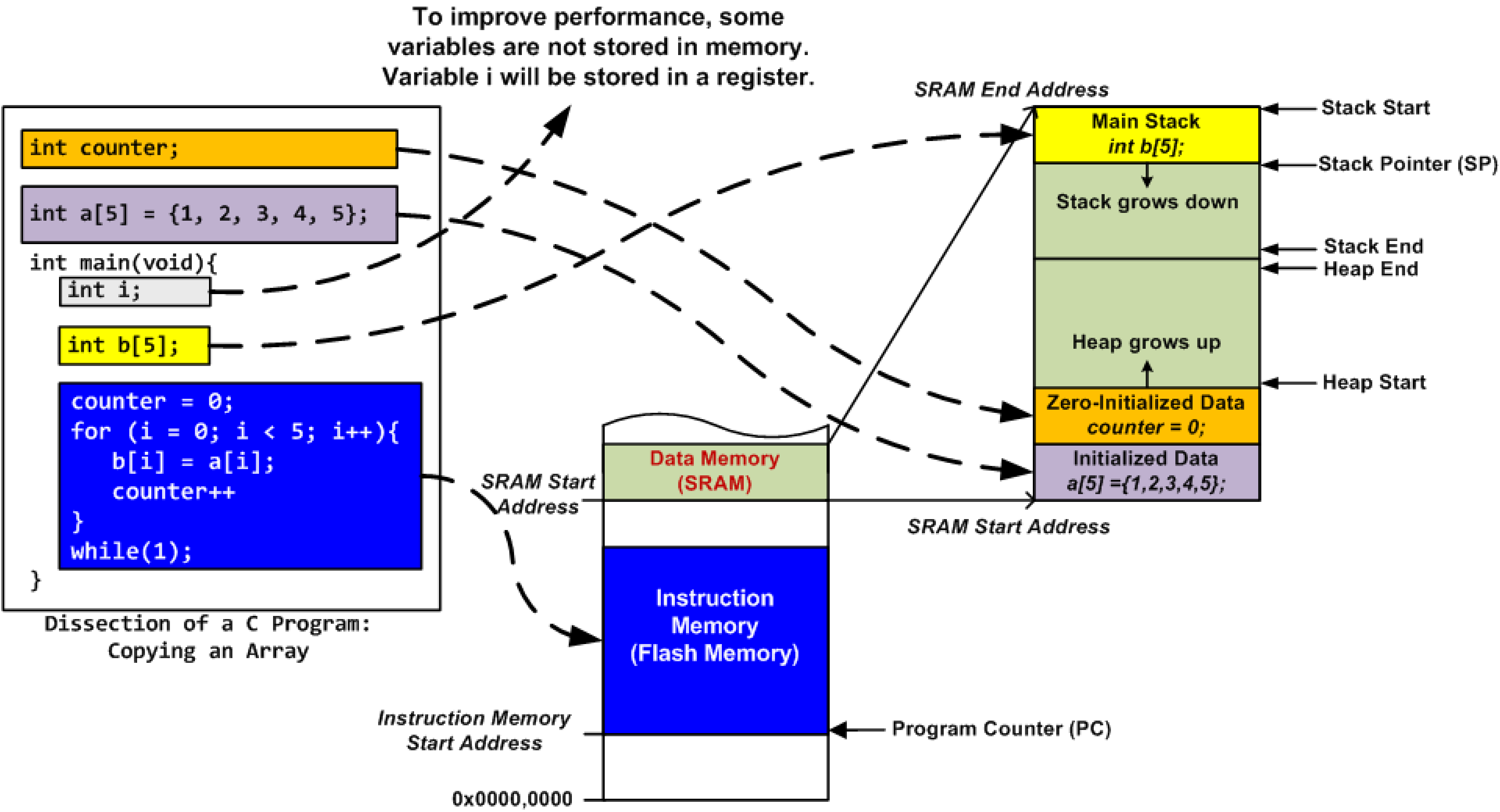
Loading Code and Data into Memory

```c
int counter;

int a[5] = {1, 2, 3, 4, 5};

int main(void){
   int i;

   int b[5];

   counter = 0;
   for (i = 0; i < 5; i++){
      b[i] = a[i];
      counter++
   }
   while(1);
}
```

Dissection of a C Program:
Copying an Array



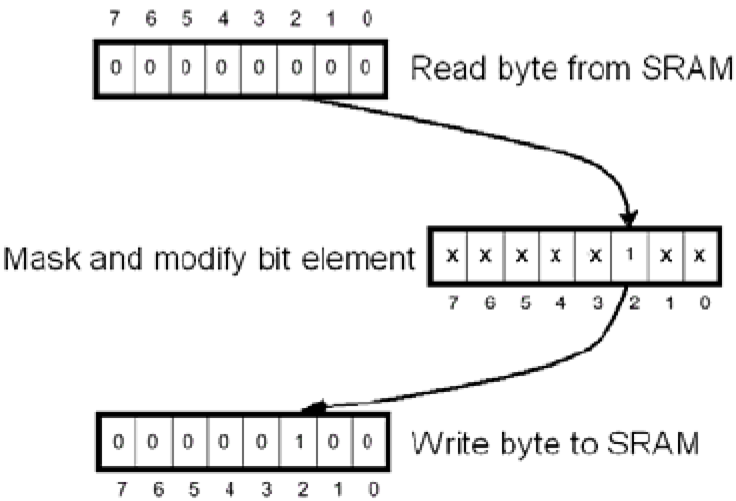
Loading Code and Data into Memory



Bit banding

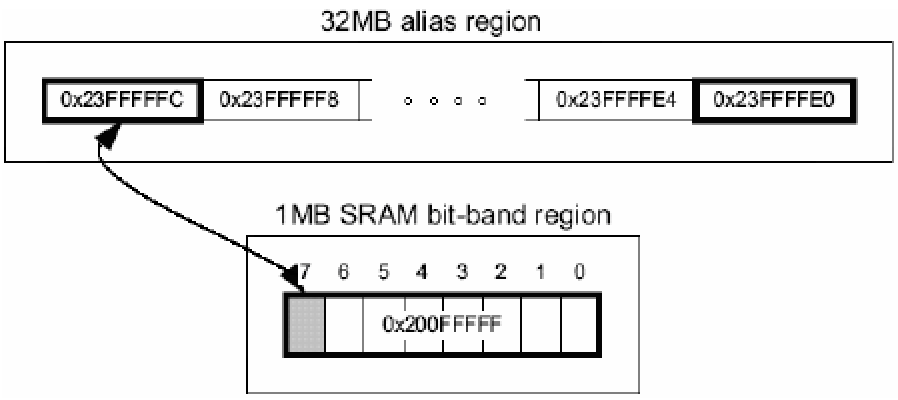
- The device takes a region of memory (the Bit-band region) and maps each bit in that region to an entire word in a second memory region (the Bit-band Alias Region).
- The benefit of Bit-banding is that a write to a word in the alias region performs a write to the corresponding bit in the Bit-band region.
  - Writing 1/0 to bit[0], writes a 1/0 to the bit-band bit,
  - Bits [31-1] of the alias word have no effect on thebit-band bit.
- Also, reading a word in the alias region will return the value of the corresponding bit in the Bit-band region.
  - If word in the alias region is read 0x00000000/0x00000001, the targetted bit in the bit-band region is 0/1.

Bit banding




```
LDR R0,=0x200FFFFF ; Setup address
MOV R2, #0x4       ; Setup data
LDR R1, [R0]       ; Read
ORR R1, R2         ; Modify bit
STR R1, [R0]       ; Write back result
```

Traditional bit manipulation method




```
LDR R0,=0x23FFFFFC ; Setup address
MOV R1, #0x1       ; Setup data
STR R1, [R0]       ; Write
```

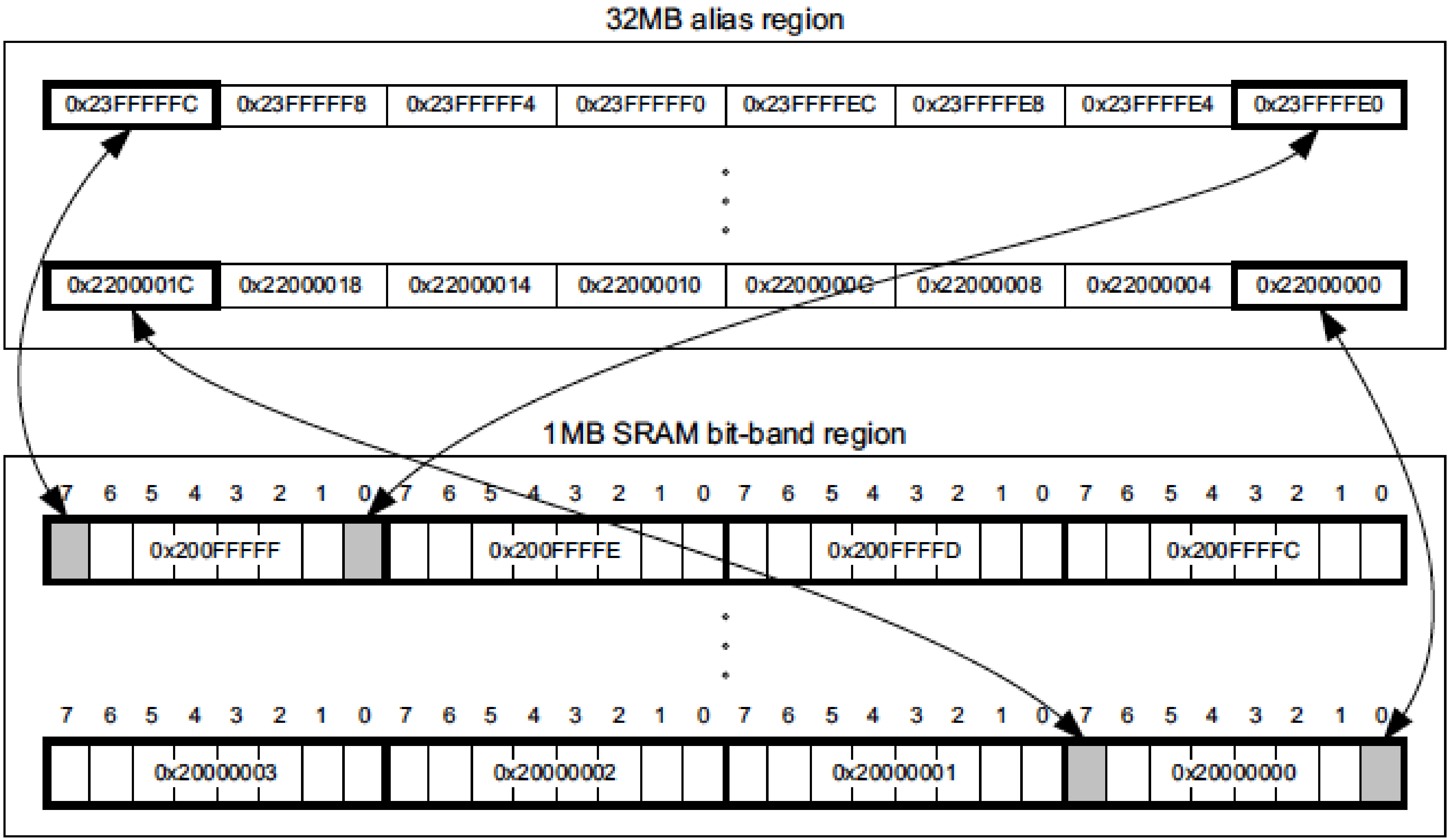
Direct, single cycle access with bit banding

- Instruction and data address:
  - 0x20000000 – 0x200FFFFF SRAM bit band
  - 0x22000000 – 0x23FFFFFF SRAM bit band alias
  - 0x40000000 – 0x400FFFFF Peripheral bit band alias
  - 0x42000000 – 0x43FFFFFF Peripheral bit band
- Bit word offset =

Byte offset x 32 + Bit number x 4

- Bit word address =

Bit band base + Bit word offset



the alias word at `0x23FFFFE0` maps to bit[0] of the bit-band byte at `0x200FFFFF`: `0x23FFFFE0` = `0x22000000` + (`0xFFFFF`*32) + (0*4)

EEE447 Introduction to Microprocessors

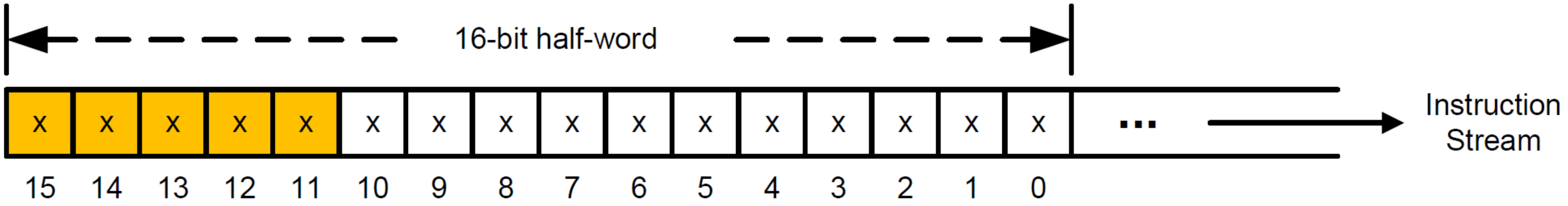
Chapter 3

Assembly Programming Details

Thumb2 instructions

- Either 16-bit or 32-bit.
- Bits[15:11] of the halfword that the PC points to determine whether it is a 16-bit instruction, or whether the following halfword is the second part of a 32-bit instruction.

hw1[15:11]	Function
0b11100	Thumb 16-bit unconditional branch instruction, defined in all Thumb architectures.
0b111xx	Thumb 32-bit instructions, defined in Thumb-2.
0bxxxxx	Thumb 16-bit instructions.



If bit[`15-11`] = `11101`, `11110`, or `11111`, then this half-word is the first half-word of a 32-bit instruction.
Otherwise, this half-word is a 16-bit instruction.

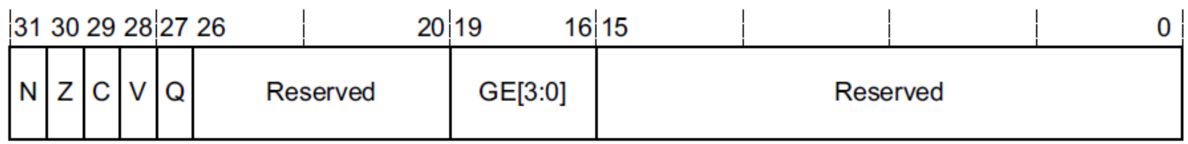
Thumb Instruction types and Instruction Set Overview

- Data movement operations
  - memory-to-register and register-to-memory
    - includes different memory “addressing” options
    - “memory” includes peripheral function registers
  - register-to-register
  - constant-to-register (or to memory in some CPUs)
- 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
- Miscellaneous
  - Wait for events
  - Interrupts
  - Others

- Most instructions are 16 bits long, some are 32 bits
- Most 16-bit instructions can only access low registers (R0-R7), but some can access high registers (R8-R15)
- Half-word aligned instructions
- Some instructions can be followed by suffixes.
  - For the Cortex-M3/M4 processors, a data processing instruction can optionally update the APSR (flags) by using the suffix ‘S’.

SUB R1, R1, R2

SUBS R1, R1, R2



  - Conditional execution of instructions by using suffixes: conditional branches, as well as conditional execution of instructions by putting the conditional instructions in an IF-THEN (IT) instruction block.

```
ITTEE   EQ              ; Next 4 instructions are conditional
MOVEQ   R0, R1          ; Conditional move
ADDEQ   R2, R2, #10     ; Conditional add
ANDNE   R3, R3, #1      ; Conditional AND
BNE.W   dloop           ; Branch instruction can only be used in the last
                        ; instruction of an IT block
```

Encoding 16-bit Thumb Instructions

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shift by immediate, move register	0	0	0	opcode [1]		imm5					Rm			Rd		
Add/subtract register	0	0	0	1	1	0	opc	Rm			Rn			Rd		
Add/subtract immediate	0	0	0	1	1	1	opc	imm3			Rn			Rd		
Add/subtract/compare/move immediate	0	0	1	opcode		Rdn			imm8							
Data-processing register	0	1	0	0	0	0	opcode				Rm			Rdn		
Special data processing	0	1	0	0	0	1	opcode [1]		DN	Rm				Rdn		
Branch/exchange instruction set	0	1	0	0	0	1	1	1	L	Rm				(0)	(0)	(0)
Load from literal pool	0	1	0	0	1	Rd			PC-relative imm8							
Load/store register offset	0	1	0	1	opcode			Rm			Rn			Rd		
Load/store word/byte immediate offset	0	1	1	B	L	imm5					Rn			Rd		
Load/store halfword immediate offset	1	0	0	0	L	imm5					Rn			Rd		
Load from or store to stack	1	0	0	1	L	Rd			SP-relative imm8							
Add to SP or PC	1	0	1	0	SP	Rd			imm8							
Miscellaneous:	1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x
Load/store multiple	1	1	0	0	L	Rn			register list							
Conditional branch	1	1	0	1	cond [2]				imm8							
Undefined instruction	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x
Service (system) call	1	1	0	1	1	1	1	1	imm8							
Unconditional branch	1	1	1	0	0	imm11										
32-bit instruction	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x
32-bit instruction	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x

32-bit Instructions

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	1	0	0	op		0	W	L	Rn				x		0	Register list													Load/store multiple
1	1	1	0	1	0	0	op1		1	op2		Rn				x								op3								Load/store dual or exclusive, table branch
1	1	1	0	1	0	1	op				S	Rn				x	imm3			Rd				imm 2				Rm				Data processing (shifted register)
1	1	1	0	1	1	op1										x				coproc							op					Coprocessor instructions
1	1	1	0	1	x	0	op					Rn				0	imm3			Rd				imm8								Data processing (modified immediate)
1	1	1	1	0	x	1	op					Rn				0																Data processing (plain binary immediate)
1	1	1	1	0	op											1	op1															Branches and miscellaneous control
1	1	1	1	1	0	0	0	op1			0					x				op2												Store single data item
1	1	1	1	1	0	0	op1		0	0	1	Rn				Rt				op2												Load byte, memory hints
1	1	1	1	1	0	0	op1		0	1	1	Rn				Rt				op2												Load halfword, memory hints
1	1	1	1	1	0	0	op1		1	0	1	Rn				x				op2												Load world
1	1	1	1	1	0	0	x	x	1	1	1					x																Undefined
1	1	1	1	1	0	1	0	op1				Rn				x	1	1	1	1				op2				Rm				Data processing (register)
1	1	1	1	1	0	1	1	0	op1							x		Ra						0	0	op2		Rm				Multiply, multiply accumulate, and absolute difference
1	1	1	1	1	0	1	1	1	op1							x								op2				Rm				Long multiply, long multiply accumulate, divide
1	1	1	1	1	1	op1										x				coproc							op					Coprocessor instructions

Examples

Inst1: **Pc** **Instruction** **Machine code**

0x00000290 **mov r3,#1** **0xF04F0301**

31					16					7 0
11110	0	0	0010	0	1111		0	000	0011	00000001

Move immediate (12 bit) Rd=3 constant=1

1 1 1 1 0	i	0	OP	S	Rn	0	imm3	Rd	imm8

Data processing, modified 12-bit immediate

Inst2: 0x00000294 ldr r1,[r3] 0x6819

15					0
011	0	1	00000	011	001
Ldr			offset	Rn=3	Rdest=1

0 1 1	B	L	imm5	Rn	Rd

Load/store word/byte immediate offset

Inst3: 0x00000296 ldr r1,[r3,#4] 0x6859

15					0
011	0	1	00001	011	001

Ldr offset=4 Rsource=3 Rdest=1

0 1 1	B	L	imm5	Rn	Rd

Load/store word/byte immediate offset

Inst4: 0x00000298 ldr r1,[r2,#0xFF] 0xF8D210FF

11111	00	0	1	10	1	0010	0001	000011111111

Ldr by Rn+imm12

Rsource=2 Rdest=1 constant=0xFF

instruction is load

sign extended or 0 extended

any including R15

1	1	1	1	1	0	0	S	1	size	L	Rn	Rt	imm12

Rn + imm12 (2)

Inst5: 0x0000029c ldr r1,[r3],#4 0xF8531B04

11111	00	0	0	10	1	0011	0001	1011	00000100

Ldr by Rn+imm8 Rsource=3 Rd=1 constant=4

Post indexed

Same as above example

1 1 1 1 1	0 0	S	0	size	L	Rn	Rt	1	0		1	imm8

Rn post-indexed by +/– imm8 (5)

Data Movement Instructions

MOV	Rd ← operand2
MVN	Rd ← NOT operand2

```
MOV r4, r5              ; Copy r5 to r4
MVN r4, r5              ; r4 = bitwise logical NOT of r5
MOV r1, r2, LSL #3      ; r1 = r2 << 3
MOV r0, #num            ; Copy num to r0
MOV r1, SP              ; Copy SP (r14) to r1
```

```
MOV        r1, #64          ; copy 64 (0x00000040) to r1
MOV        r0, r1           ; r0=0x00000040, r1 doesn't change
MOVT       r0, #0x1234      ; # -> r0[31:16.Now r0=0x12340040
MVN        r0, #0x13        ; copy NOT(0x00000013).r0=0xFFFFFFEC
MOV32      r0, #0x20008000         ; r0=0x20008000
```

Pseudo instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM or Thumb instructions at assembly time.

In loading data

- Load a 32-bit constant (data, address, etc .) into a register. Cannot embed 32-bit value in a 32-bit instruction
- Use a “pseudo-op” to load the constant: Either LDR with relative addressing or MOV

Example:

Addres	Source Program	Debug Disassembly
	MOV**32** r0,#0x12345678	MOVW r0,#0x5678
		MOVT r0,#0x1234
0x0000050C	LDR.W r3,=0x12345678	LDR r3,[PC,#offset]

.....

```
0x0000051C   dcw 0x5678           ;in literal pool following the code
0x0000051E   dcw 0x1234
```

****

Load and store instructions

- ARM is a load/store architecture, so must process data in registers, not memory
- General load/store instruction format

({type} : none is word; B is byte; H is half-word)

LDR{type}	Rd,[Rn]	;load memory at [Rn] to Rd
STR{type}	Rt,[Rn]	;store Rt to memory at [Rn]
LDR{type}	Rd,[Rn, #n]	;load memory at [Rn+n] to Rd, Rn unchanged
STR{type}	Rt,[Rn, #n]	;store Rt to memory [Rn+n], Rn unchanged
LDR{type}	Rd,[Rn,Rm,LSL #n1]	;load [Rn+Rm<<n1] to Rd, Rn unchanged
STR{type}	Rt,[Rn,Rm,LSL #n1]	;store Rt to [Rn+Rm<<n1], Rn unchanged
LDR{type}	Rd,[Rn, #n]!	;load memory at [Rn+n] to Rd, update Rn
STR{type}	Rt,[Rn, #n]!	;store Rt to memory [Rn+n], update Rn
LDR{type}	Rd,[Rn], #n	;load memory at [Rn] to Rd, update Rn
STR{type}	Rt,[Rn], #n	;store Rt to memory [Rn], update Rn

Offset range n: -255 to 255, n1: 0 to 3

LDRD, STRD is used for loading/storing 2 words

Single register data transfer

LDR	Load Word
LDRB	Load Byte
LDRH	Load Halfword
LDRSB	Load Signed Byte
LDRSH	Load Signed Halfword

STR	Store Word
STRB	Store Lower Byte
STRH	Store Lower Halfword

Load a Byte, Half-word, Word

`r1 before load:`

0x12	0x34	0x56	0x78

31 0

0x02000003	0x87
0x02000002	0x65
0x02000001	0xE3
0x02000000	0xE1

Little Endian

Assume
r0 = 0x02000000

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

Sign Extension

Load a Signed Byte

```
LDRSB r1, [r0]
```

31			0
0xFF	0xFF	0xFF	0xE1

Load a Signed Halfword

```
LDRSH r1, [r0]
```

31			0
0xFF	0xFF	0xE3	0xE1

0x02000003	0x87
0x02000002	0x65
0x02000001	0xE3
0x02000000	0xE1

Little Endian

Assume
r0 = 0x02000000

Facilitate subsequent 32-bit signed arithmetic!

- Address accessed by LDR/STR can be specified by a base register plus an offset
- Offset can be
  - A number

Instruction Type	Immediate Offset	Pre-Indexed	Post-Indexed
Word, halfword, signed halfword, byte, or signed byte	−255 to 4095	−255 to 255	−255 to 255

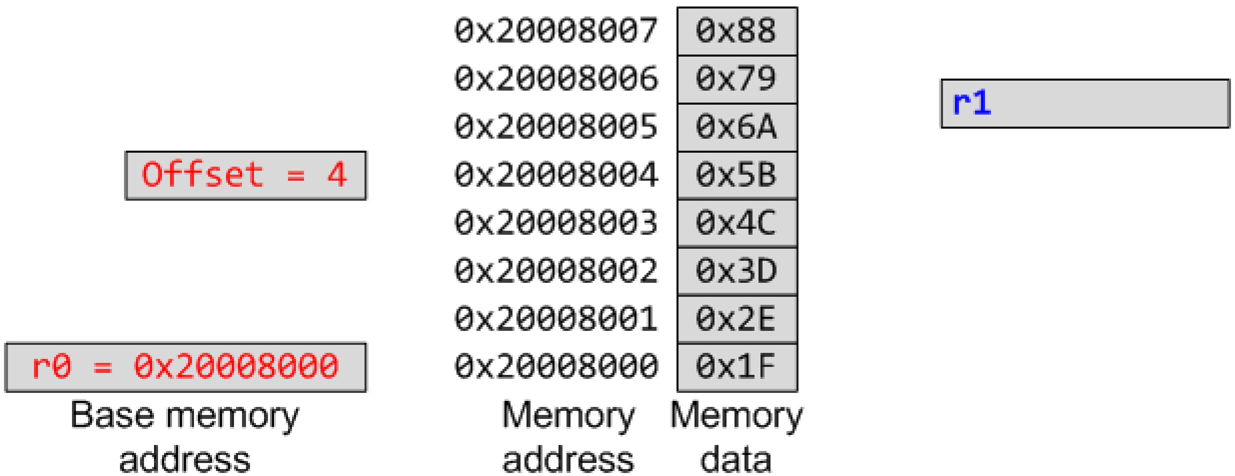
```
LDR r0,[r1,#8]
```

  - A register, optionally shifted by an immediate value

```
LDR r0,[r1,r2]
LDR r0,[r1,r2,LSL#2]
```

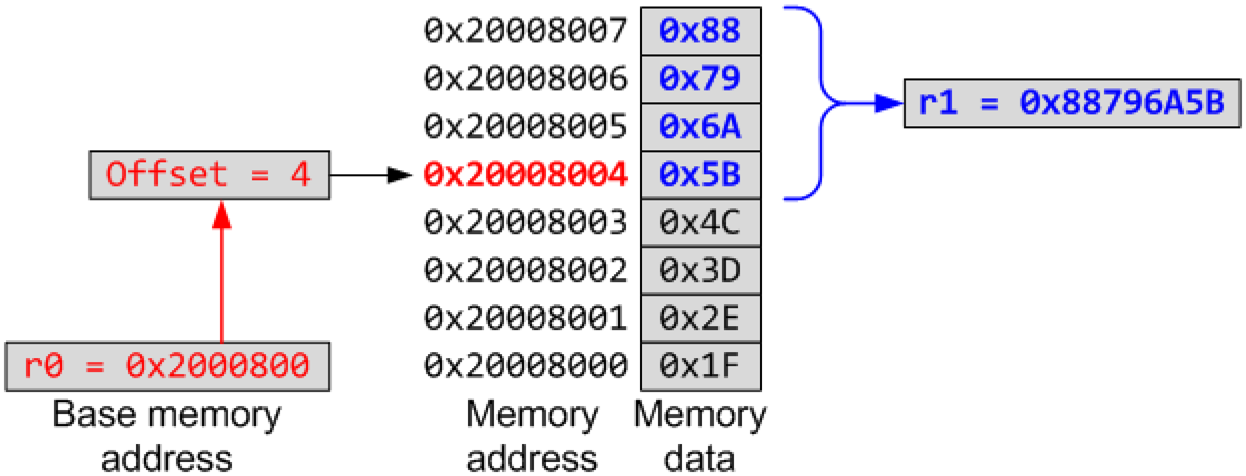
Pre-index

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



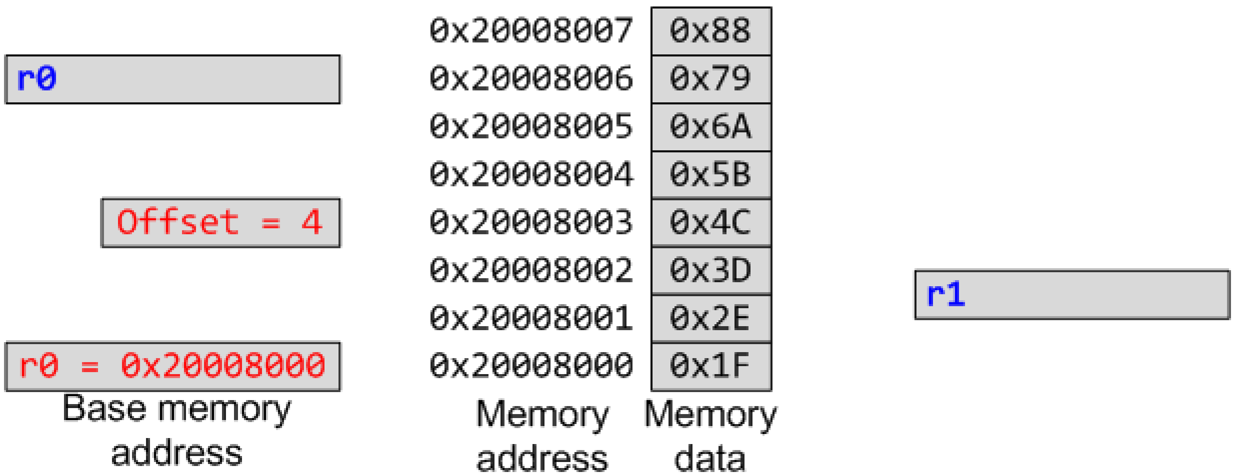
Pre-index

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



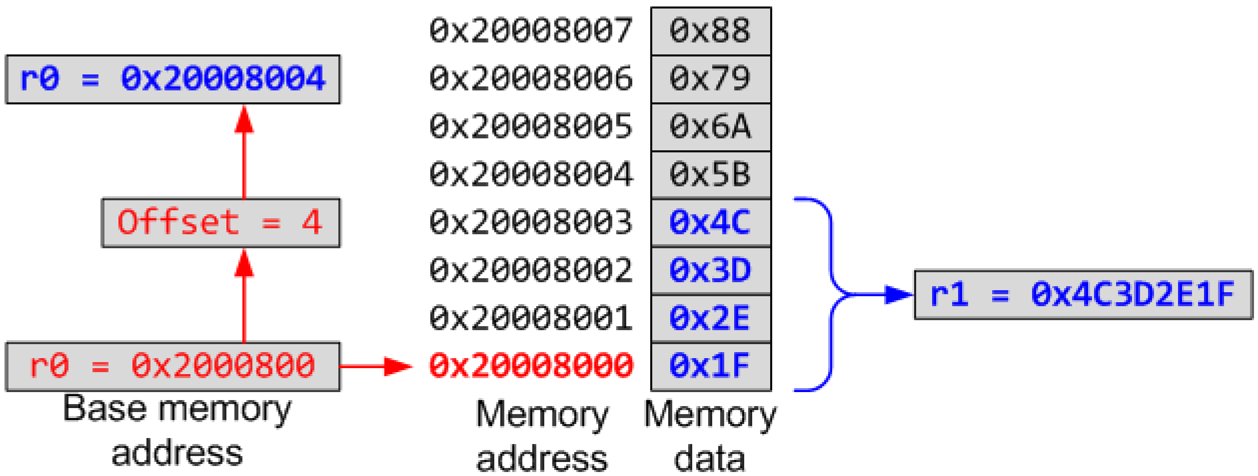
Post-index

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



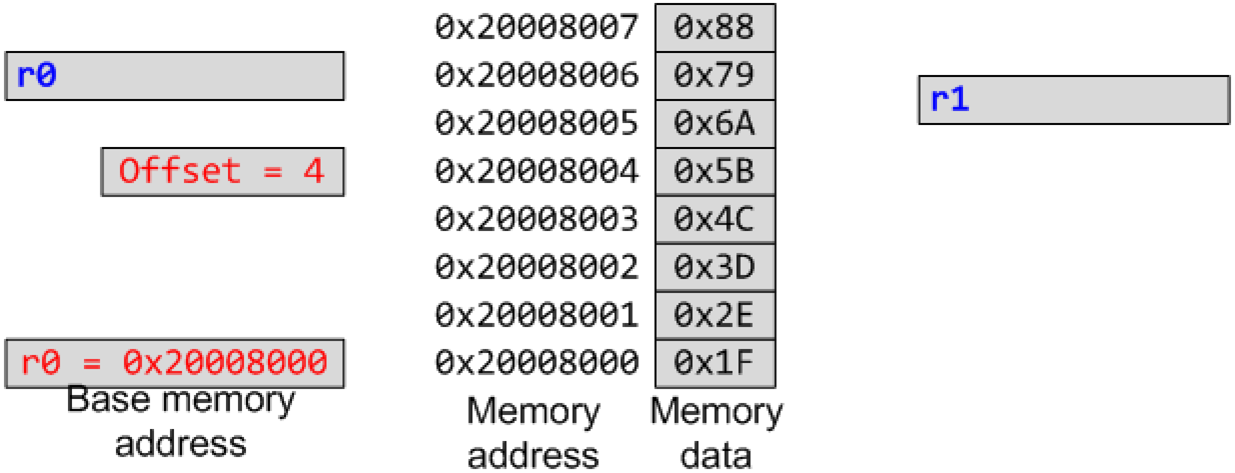
Post-index

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



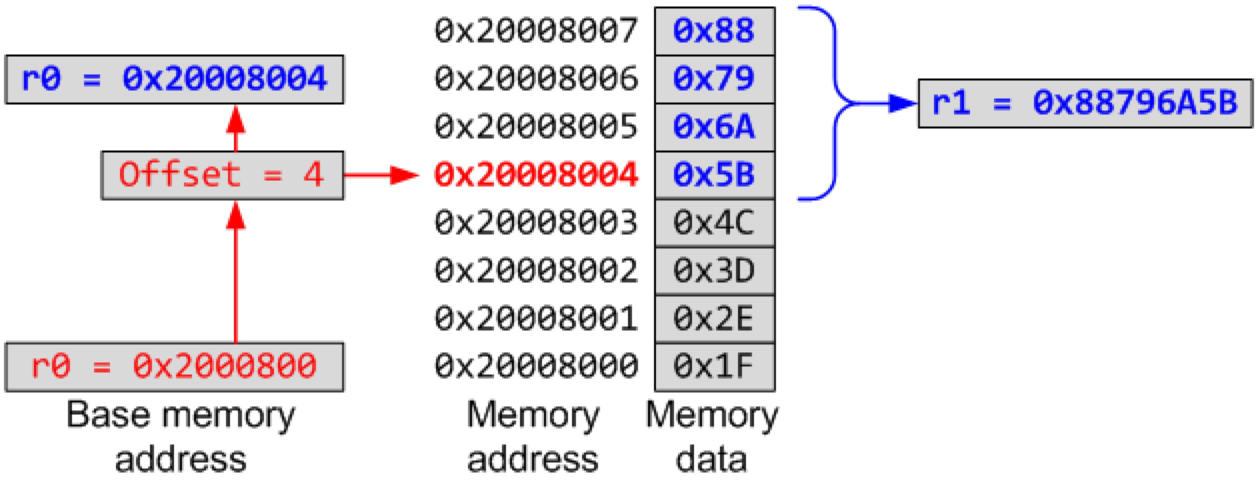
Pre-index with Updates

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



Pre-index with Updates

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



Summary of Pre-index and Post-index

Index Format	Example	Equivalent
Pre-index	LDR r1, [r0, #4]	r1 ← memory[r0 + 4], r0 is unchanged
Pre-index with update	LDR r1, [r0, #4]!	r1 ← memory[r0 + 4] r0 ← r0 + 4
Post-index	LDR r1, [r0], #4	r1 ← memory[r0] r0 ← r0 + 4

Store Instructions

- STR rt, [rs]:
  - save data in register rt into memory
  - The memory address is specified in a base register rs.
  - Pre and post-indexed addressing modes can be used like ldr
  - For Example:

Examples

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

r0 before store

0x20008000

r0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

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

r0 before store

$\boxed{\texttt{0x20008000}}$

r0 after store

$\boxed{\texttt{0x20008000}}$

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	**0x76**
0x20008002	**0x54**
0x20008001	**0x32**
0x20008000	**0x10**

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

r0 before store

0x20008000

r0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

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

r0 before store

$\boxed{\texttt{0x20008000}}$

r0 after store

$\boxed{\texttt{0x20008004}}$

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	**0x76**
0x20008002	**0x54**
0x20008001	**0x32**
0x20008000	**0x10**

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

r0 before the store

0x20008000

r0 after the store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

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

r0 before store

$\boxed{\texttt{0x20008000}}$

r0 after store

$\boxed{\texttt{0x20008000}}$

Memory Address	Memory Data
0x20008007	**0x76**
0x20008006	**0x54**
0x20008005	**0x32**
0x20008004	**0x10**
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

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

r0 before store

0x20008000

r0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

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

r0 before store

0x20008000

r0 after store

0x20008004

Memory Address	Memory Data
0x20008007	0x76
0x20008006	0x54
0x20008005	0x32
0x20008004	0x10
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Load/store double registers

STRD R1, R2, [R0] ; Store R1 to address in R0, and store R2 to
; a word 4 bytes above the address in R0

```
STRD r1,r2, [r0]
; r0 = 0x20008000,  r1=0x76543210, r2=0xABCDEF10
```

Memory Address	Memory Data
0x20008007	0xAB
0x20008006	0xCD
0x20008005	0xEF
0x20008004	0x10
0x20008003	**0x76**
0x20008002	**0x54**
0x20008001	**0x32**
0x20008000	**0x10**

Load/Store Multiple

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

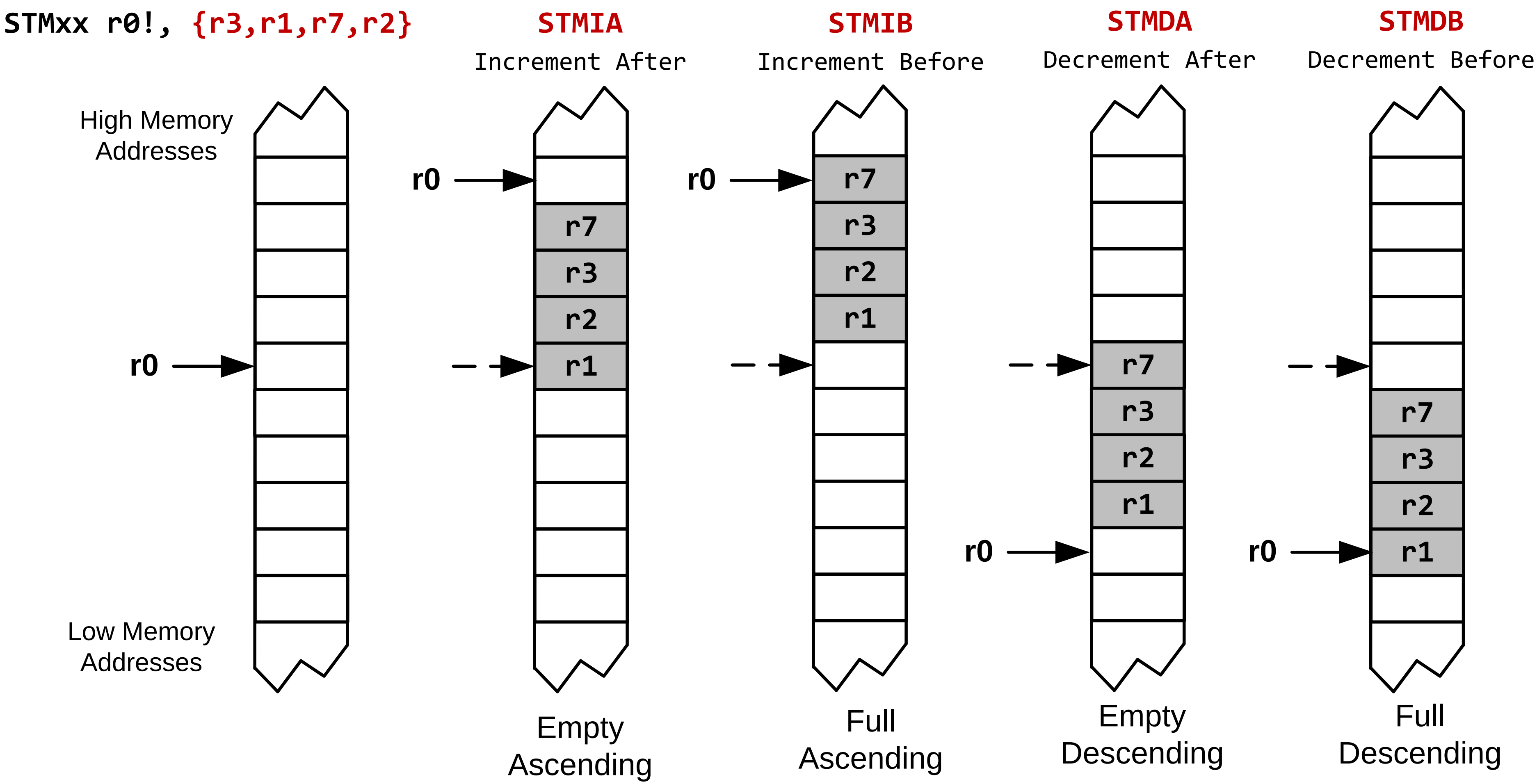
- xx = IA, IB, DA, or DB

Addressing Modes	Description	Instructions
IA	Increment After	STMIA, LDMIA
IB	Increment Before	STMIB, LDMIB
DA	Decrement After	STMDA, LDMDA
DB	Decrement Before	STMDB, LDMDB

- 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.

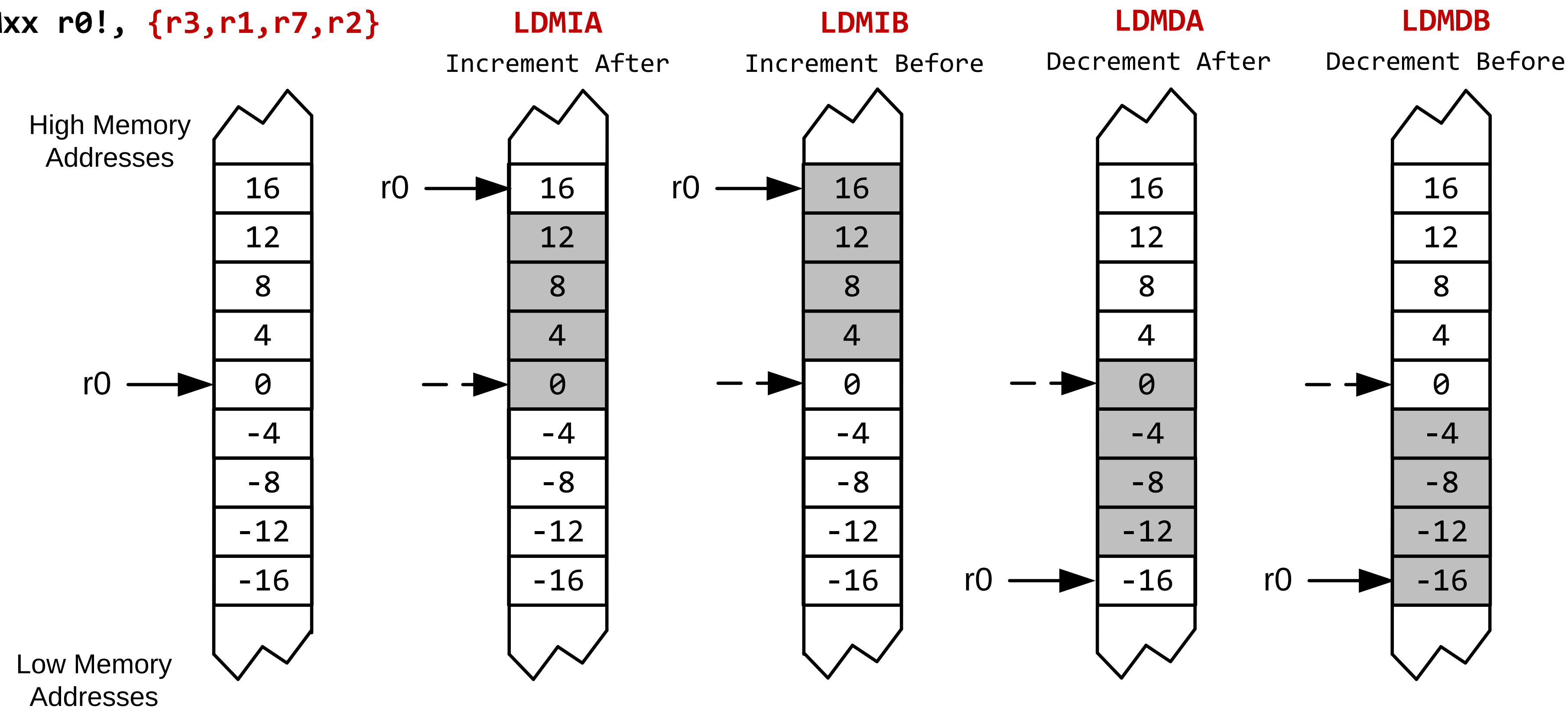
LDMIA and STMIA are pseudo-instructions, translated by assembler

Store Multiple Registers



Load Multiple Registers

`LDMxx r0!, {r3,r1,r7,r2}`



LDMIA	LDMIB	LDMDA	LDMDB
r1 = 0	r1 = 4	r1 = -12	r1 = -16
r2 = 4	r2 = 8	r2 = -8	r2 = -12
r3 = 8	r3 = 12	r3 = -4	r3 = -8
r7 = 12	r7 = 16	r7 = -0	r7 = -4

Arithmetic and Logic Instructions

Shift

LSL (logic shift left), LSR (logic shift right), ASR (arithmetic shift right), ROR (rotate right), RRX (rotate right with extend)

Logic

AND (bitwise and), ORR (bitwise or), EOR (bitwise exclusive or), ORN (bitwise or not), MVN (move not)

Bit set/clear

BFC (bit field clear), BFI (bit field insert), BIC (bit clear), CLZ (count leading zeroes)

Bit/byte reordering

RBIT (reverse bit order in a word), REV (reverse byte order in a word), REV16 (reverse byte order in each half-word independently), REVSH (reverse byte order in each half-word independently)

Addition

ADD, ADC (add with carry)

Subtraction

SUB, RSB (reverse subtract), SBC (subtract with carry)

Multiplication

MUL (multiply), MLA (multiply-accumulate), MLS (multiply-subtract), SMULL (signed long multiply-accumulate), SMLAL (signed long multiply-accumulate), UMULL (unsigned long multiply-subtract), UMLAL (unsigned long multiply-subtract)

Division

SDIV (signed), UDIV (unsigned)

Sign extension

SXTB (signed), SXTH, UXTB, UXTH

Bit field extract

SBFX (signed), UBFX (unsigned)

Syntax

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

Commonly Used Arithmetic Operations

ADD {Rd,} Rn, Op2	Add. Rd $\leftarrow$ Rn + Op2
ADC {Rd,} Rn, Op2	Add with carry. Rd $\leftarrow$ Rn + Op2 + Carry
SUB {Rd,} Rn, Op2	Subtract. Rd $\leftarrow$ Rn - Op2
SBC {Rd,} Rn, Op2	Subtract with carry. Rd $\leftarrow$ Rn - Op2 + Carry - 1
RSB {Rd,} Rn, Op2	Reverse subtract. Rd $\leftarrow$ Op2 - Rn
MUL {Rd,} Rn, Rm	Multiply. Rd $\leftarrow$ (Rn × Rm)[31:0]
MLA Rd, Rn, Rm, Ra	Multiply with accumulate. Rd $\leftarrow$ (Ra + (Rn × Rm))[31:0]
MLS Rd, Rn, Rm, Ra	Multiply and subtract, Rd $\leftarrow$ (Ra – (Rn × Rm))[31:0]
SDIV {Rd,} Rn, Rm	Signed divide. Rd $\leftarrow$ Rn / Rm
UDIV {Rd,} Rn, Rm	Unsigned divide. Rd $\leftarrow$ Rn / Rm

Example: Add

```
–   ADD r1, r2, r3    ; r1 = r2 + r3
    ADD r1, r2, #4    ; r1 = r2 + 4
```

Ex: Assume R3= 1000, R2=4000, R4=3000 then

ADDS R3, #250 ; R3=1250, N=Z=C=V=0

ADDS R1, R2, R4 ;results in R1=7000, N=Z=C=V=0

Ex: Assume R3= 1000, R2=250, R4=3000 then

SUBS R3, R2 ; R3=750, N=Z= V=0 , C=1

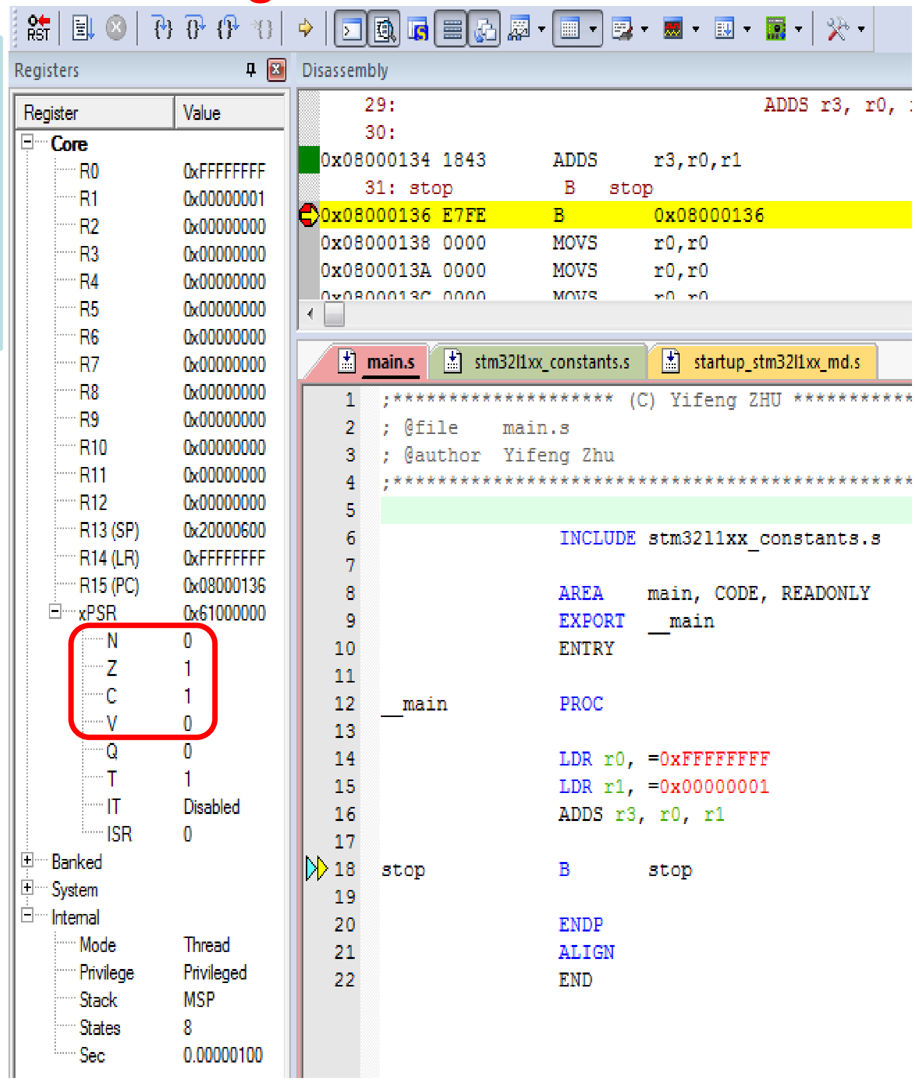
SUBS R1, R4, R2 ; R1=2750, N=Z= V=0 , C=1

Example:

S: Set Condition Flags

```
start
        LDR     r0, =0xFFFFFFFF
        LDR     r1, =0x00000001
        ADDS    r0, r0, r1
stop    B       stop
```

- In this example, the Z and C bits are set.



Program Status Register

- Application PSR (**APSR**), Interrupt PSR (**IPSR**), Execution PSR (**EPSR**)

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

- Combine them together into one register (**PSR**)
- Use PSR in code

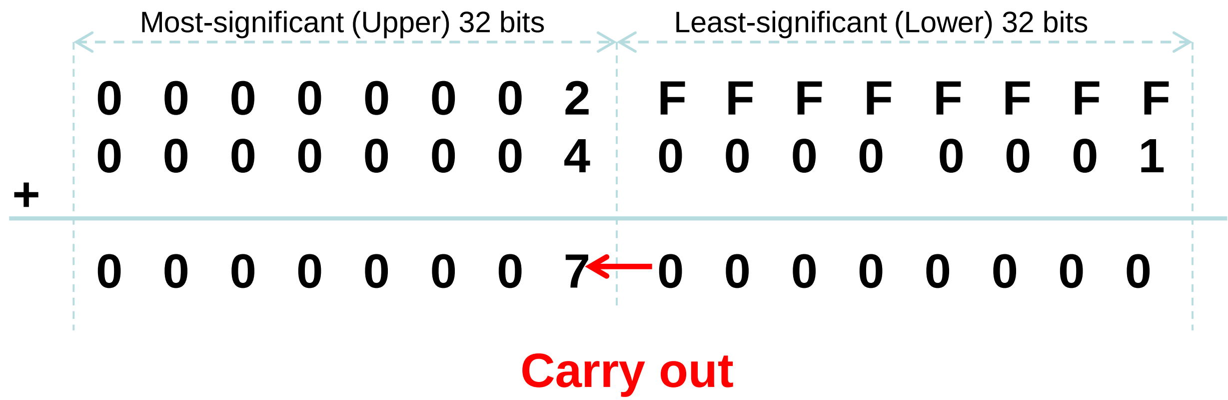
	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
PSR	N	Z	C	V	Q	ICI/IT	T		GE	ICI/IT		Exception Number				

Note: GE flags are only available on Cortex-M4 and M7

Example: Short Multiplication and Division

```
; MUL: Signed multiply
MUL  r6, r4, r2         ; r6 = LSB32( r4 × r2 )

; UMUL: Unsigned multiply
UMUL r6, r4, r2         ; r6 = LSB32( r4 × r2 )

; MLA: Multiply with accumulation
MLA  r6, r4, r1, r0     ; r6 = LSB32( r4 × r1 ) + r0

; MLS:  Multiply with subtract
MLS  r6, r4, r1, r0     ; r6 = LSB32( r4 × r1 ) - r0
```

Example: 64-bit Addition



- A register can only store 32 bits
- A 64-bit integer needs two registers
- Split 64-bit addition into two 32-bit additions

Example: 64-bit Addition

```
start
  ; C = A + B
  ; Two 64-bit integers A (r1,r0) and B (r3, r2).
  ; Result C (r5, r4)
  ; A = 00000002FFFFFFFF
  ; B = 0000000400000001
  LDR  r0, =0xFFFFFFFF  ;  A's lower 32 bits
  LDR  r1, =0x00000002  ;  A's upper 32 bits
  LDR  r2, =0x00000001  ;  B's lower 32 bits
  LDR  r3, =0x00000004  ;  B's upper 32 bits

  ;  Add A and B
  ADDS r4, r2, r0  ; C[31..0] = A[31..0] +  B[31..0],  update Carry
  ADC  r5, r3 r1   ; C[64..32] = A[64..32] + B[64..32]  +  Carry

stop  B  stop
```

Example: 64-bit Subtraction

```
start
  ; C = A - B
  ; Two 64-bit integers A (r1,r0) and B (r3, r2).
  ; Result C (r5, r4)
  ; A = 00000002FFFFFFFF
  ; B = 0000000400000001
  LDR  r0, =0xFFFFFFFF  ;  A's lower 32 bits
  LDR  r1, =0x00000002  ;  A's upper 32 bits
  LDR  r2, =0x00000001  ;  B's lower 32 bits
  LDR  r3, =0x00000004  ;  B's upper 32 bits

  ;  Subtract B from A
  SUBS r4, r0, r2  ; C[31..0]= A[31..0] - B[31..0],  update Carry
  SBC  r5, r1, r3  ; C[64..32]= A[64..32] - B[64..32] -  Carry

stop  B  stop
```

Bitwise Logic

AND {Rd,} Rn, Op2	Bitwise logic AND. Rd ← Rn & operand2
ORR {Rd,} Rn, Op2	Bitwise logic OR. Rd ← Rn | operand2
EOR {Rd,} Rn, Op2	Bitwise logic exclusive OR. Rd ← Rn ^ operand2
ORN {Rd,} Rn, Op2	Bitwise logic NOT OR. Rd ← Rn | (NOT operand2)
BIC {Rd,} Rn, Op2	Bit clear. Rd ← Rn & NOT operand2
BFC Rd, #lsb, #width	Bit field clear. Rd[(width+lsb–1):lsb] ← 0
BFI Rd, Rn, #lsb, #width	Bit field insert. Rd[(width+lsb–1):lsb] ← Rn[(width-1):0]
MVN Rd, Op2	Move NOT, logically negate all bits. Rd ← 0xFFFFFFFF EOR Op2

Example: AND r2, r0, r1

32 bits

r0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

r1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1

r2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Bit-wise Logic AND

Example: ORR r2, r0, r1

32 bits

r0	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
r1	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**
r2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Bit-wise Logic OR

Example: BIC r2, r0, r1

Bit Clear

r0= r0 & NOT r1

Step 1:

r1	**0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1**
NOT r1	**1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0**

Step 2:

r0	**1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**
NOT r1	**1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0**
r2	**1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0**

Example: BFC and BFI

- Bit Field Clear (BFC) and Bit Field Insert (BFI).
- Syntax
  - `BFC Rd, #lsb, #width`
  - `BFI Rd, Rn, #lsb, #width`

- Examples:

```
BFC R4, #8, #12
```

; Clear bit 8 to bit 19 (12 bits) of R4 to 0

```
BFI R9, R2, #8, #12
```

; Replace bit 8 to bit 19 (12 bits) of R9 with bit 0 to bit 11 from R2.

Assume [r0]=0x0F = 0b00001111 = 15
[r4]=0xF0=0b11110000
[r1]=0xAD=0b10101101

```
AND r0,r0,#5        ; perform AND;  r0=0b00000101
ORR r4,r0,r4        ; perform OR; r4=0xF5
BFI   r4,r0,#8,#4; r4=0b0000010111111111
BFC r4,#1,#5        ; r4=0b0000000011000001
ORN r4,r0,r1        ; r4=0xFFFFFF57
```

Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] & 0xFFFF

RBIT Rd, Rn

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

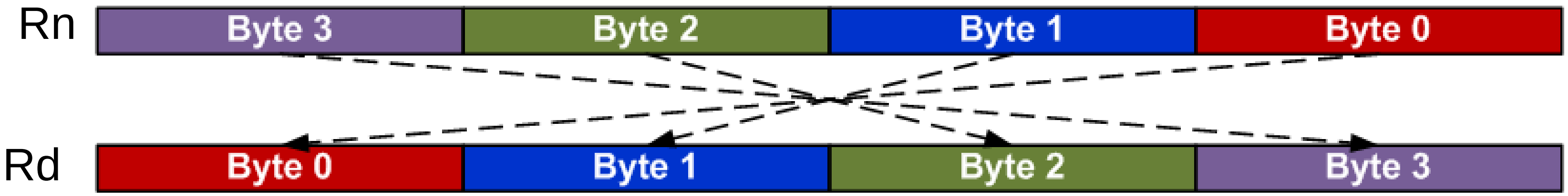
Example:

```
LDR  r0, =0x12345678  ; r0 = 0x12345678
RBIT r1, r0           ; Reverse bits, r1 = 0x1E6A2C48
```

Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] & 0xFFFF

REV Rd, Rn



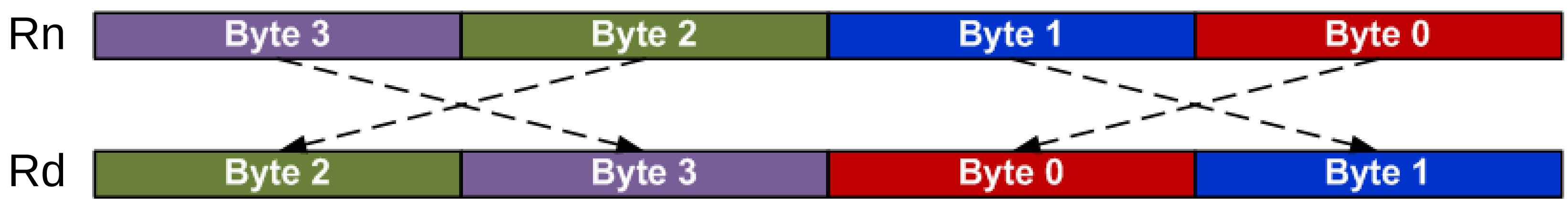
Example:

```
LDR R0, =0x12345678
REV R1, R0      ; R1 = 0x78563412
```

Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] & 0xFFFF

REV16 Rd, Rn



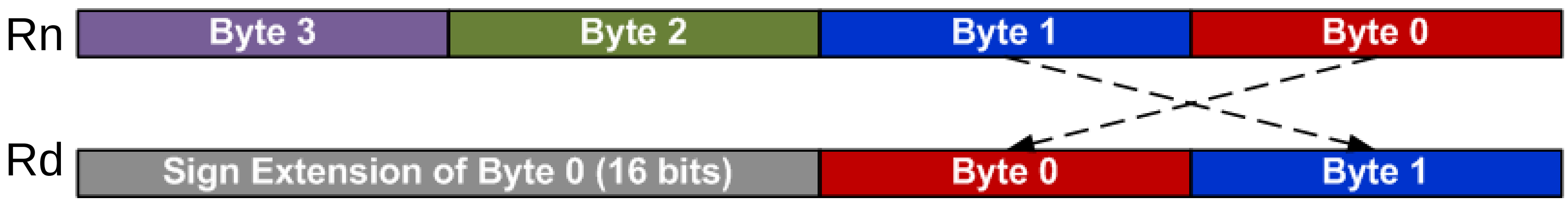
Example:

```
LDR R0, =0x12345678
REV16 R2, R0   ; R2 = 0x34127856
```

Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] & 0xFFFF

REVSH Rd, Rn



Example:

```
LDR R0, =0x33448899
REVSH R1, R0   ; R0 = 0xFFFF9988
```

Sign and Zero Extension

```
signed int_8  a = -1;   // a signed 8-bit integer,  a = 0xFF
signed int_16 b = -2;   // a signed 16-bit integer, b = 0xFFFF
signed int_32 c;        // a signed 32-bit integer

c = a;                  // sign extension required, c = 0xFFFFFFFF
c = b;                  // sign extension required, c = 0xFFFFFFFF
```

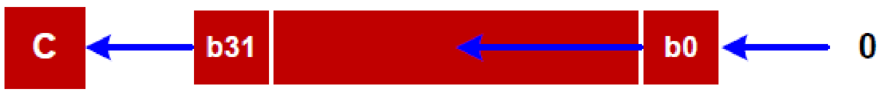
Sign and Zero Extension

SXTB {Rd,} Rm {,ROR #n}	Sign extend a byte. Rd[31:0] ← Sign Extend((Rm ROR (8 × n))[7:0])
SXTH {Rd,} Rm {,ROR #n}	Sign extend a half-word. Rd[31:0] ← Sign Extend((Rm ROR (8 × n))[15:0])
UXTB {Rd,} Rm {,ROR #n}	Zero extend a byte. Rd[31:0] ← Zero Extend((Rm ROR (8 × n))[7:0])
UXTH {Rd,} Rm {,ROR #n}	Zero extend a half-word. Rd[31:0] ← Zero Extend((Rm ROR (8 × n))[15:0])

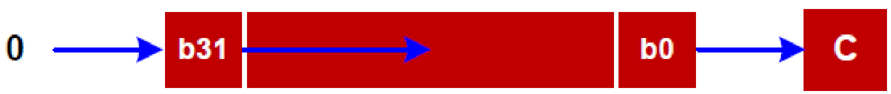
```
LDR R0, =0x55AA8765
SXTB R1, R0    ; R1 = 0x00000065
SXTH R1, R0    ; R1 = 0xFFFF8765
UXTB R1, R0    ; R1 = 0x00000065
UXTH R1, R0    ; R1 = 0x00008765
```

Shift and rotate instructions

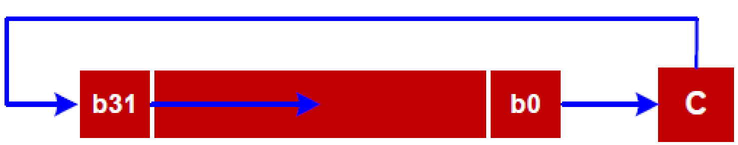
Logical Shift Left (**LSL**)



Logical Shift Right (**LSR**)

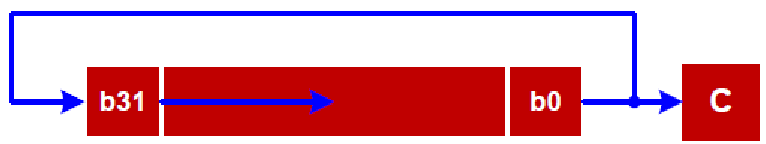


Rotate Right Extended (**RRX**)



Arithmetic Shift Right (**ASR**)

Rotate Right (**ROR**)



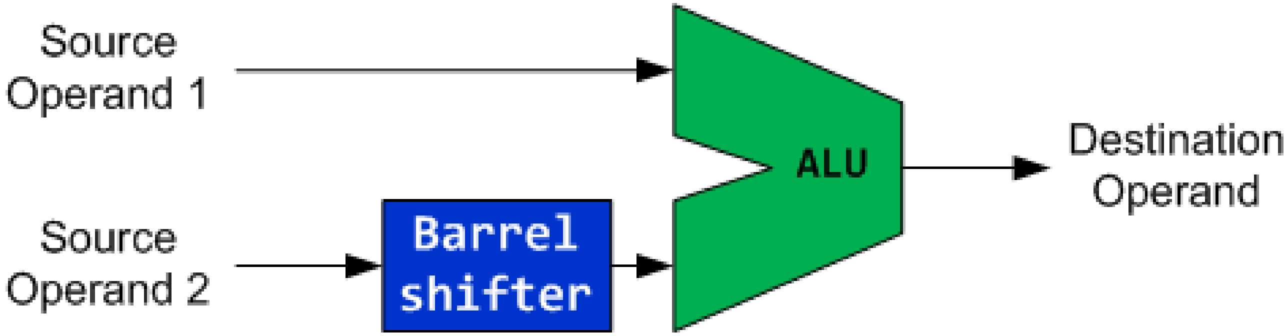
Why is there rotate right but no rotate left?

Rotate left can be replaced by a rotate right with a different rotate offset.

Examples

```
LSL r1, r0, #3      ; Before r0 = 0x0F = 0b00001111 = 15
                    ; After   r1 = 0b01111000 = 0x78 = 120 = 15*2^3

LSR r3, r4, #2      ;Before  r4=0xF0=0b11110000=240
                    ; After r3 = 0b00111100 = 0x3C = 60 = 240 / 2^2

ASR r3, r4, #2      ;Before [r4]=0xF0=0b11110000=240
                    ; r3 = 0b00111100 = 0x3C = 60 = 240 / 2^2
```

Using Barrel Shifter in Arithmetic



- The second operand of ALU has a special hardware called Barrel shifter
- Example:

```
ADD r1, r0, r0, LSL #3  ; r1 = r0 + r0 << 3 = 9 × r0
```

Shift left

Examples

```
ADD r1, r0, r0, LSL #3
 ; r1 = r0 + r0 << 3 = r0 + 9 × r0

ADD r1, r0, r0, LSR #3
 ; r1 = r0 + r0 >> 3 = r0 + r0/8 (unsigned)

ADD r1, r0, r0, ASR #3
 ; r1 = r0 + r0 >> 3 = r0 + r0/8 (signed)
```

- Use Barrel shifter to speed up the application

```
ADD r1, r0, r0, LSL #3 <=>MOV r2, #9      ; r2 = 9
                          MUL r1, r0, r2 ; r1 = r0 * 9
```

Comparison Instructions

Instruction	Operands	Brief description	Flags
CMP	Rn, Op2	Compare	N,Z,C,V
CMN	Rn, Op2	Compare Negative	N,Z,C,V
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C

- The only effect of the comparisons is to **update the condition flags**.
  - No need to set S bit.
  - No need to specify Rd.
- Operations are:
  - CMP operand1 - operand2, but result not written
  - CMN operand1 + operand2, but result not written
  - TST operand1 & operand2, but result not written
  - TEQ operand1 ^ operand2, but result not written
- Examples:
  - `CMP r0, r1`
  - `TST r2, #5`

CMP and CMN

CMP{cond} Rn, Operand2

CMN{cond} Rn, Operand2

- The CMP instruction subtracts the value of Operand2 from the value in Rn.
  - This is the same as a SUBS instruction, except that the result is discarded.
- The CMN instruction adds the value of Operand2 to the value in Rn.
  - This is the same as an ADDS instruction, except that the result is discarded.
- These instructions update the N, Z, C and V flags according to the result.

TST and TEQ

TST{cond} Rn, Operand2 **; Bitwise AND**
TEQ{cond} Rn, Operand2 **; Bitwise Exclusive OR**

- The TST instruction performs a bitwise AND operation on the value in Rn and the value of Operand2.
  - This is the same as a ANDS instruction, except that the result is discarded.
- The TEQ instruction performs a bitwise Exclusive OR operation on the value in Rn and the value of Operand2.
  - This is the same as a EORS instruction, except that the result is discarded.
- Update the N and Z flags according to the result
- Can update the C flag during the calculation of Operand2
- Do not affect the V flag.

Branch Instructions

Instruction	Operands	Brief description	Flags
B	label	Branch	-
BL	label	Branch with Link	-
BLX	Rm	Branch indirect with Link	-
BX	Rm	Branch indirect	-

- *B label*: causes a branch to label.
- *BL label*: instruction copies the address of the next instruction into r14 (lr, the link register), and causes a branch to label.
- *BX Rm*: branch to the address held in Rm
- *BLX Rm*: copies the address of the next instruction into r14 (lr, the link register) and branch to the address held in Rm

Branch With Link

- The "Branch with link (BL)" instruction implements a subroutine call by writing PC+4 into the LR of the current bank.
  - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).
- To return from subroutine, simply need to restore the PC from the LR:
  - **`MOV pc, lr`**
  - Again, pipeline has to refill before execution continues.
- The "Branch" instruction does not affect LR.

Condition Codes

Suffix	Description	Flags tested
EQ	EQual	Z=1
NE	Not Equal	Z=0
CS/HS	Unsigned Higher or Same	C=1
CC/LO	Unsigned LOwer	C=0
MI	MInus (Negative)	N=1
PL	PLus (Positive or Zero)	N=0
VS	oVerflow Set	V=1
VC	oVerflow Clear	V=0
HI	Unsigned HIgher	C=1 & Z=0
LS	Unsigned Lower or Same	C=0 or Z=1
GE	Signed Greater or Equal	N=V
LT	Signed Less Than	N!=V
GT	Signed Greater Than	Z=0 & N=V
LE	Signed Less than or Equal	Z=1 or N!=V
AL	ALways	

Note AL is the default and does not need to be specified

Signed Greater or Equal (N == V)

CMP r0, r1

We in fact perform subtraction r0 – r1, without saving the result.

	N = 0	N = 1
V = 0	• No overflow, implying the result is correct. • The result is non-negative, • Thus r0 – r1 ≥ 0, i.e., r0 ≥ r1	• No overflow, implying the result is correct. • The result is negative. • Thus r0 – r1 < 0, i.e., r0 < r1
V = 1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as non-negative and in fact it should be negative. • Thus r0 – r1 < 0 in reality, i.e., r0 < r1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as negative and in fact it should be non-negative. • Thus r0 – r1 ≥ 0 in reality., i.e. r0 ≥ r1

Conclusions:

- If N == V, then it is signed greater or equal (GE).
- Oherwise, it is signed less than (LT)

Number Interpretation

Which is greater?

`0xFFFFFFFF` or **`0x00000001`**

- If they represent signed numbers, the latter is greater
  (**`1 > -1`**).
- If they represent unsigned numbers, the former is greater
  (**`4294967295 > 1`**).

Which is Greater: `0xFFFFFFFF` or `0x00000001`?

It's **software's reasonability** to tell computer how to interpret data:

- If written in C, declare the signed vs unsigned variable
- If written in Assembly, use signed vs unsigned branch instructions

```
signed int x, y ;
x = -1;
y = 1;
if (x > y)
   ...
```

```
MOVS r6, #0xFFFFFFFF
MOVS r5, #0x00000001
CMP  r5, r6
BLE  Then_Clause
...
```

BLE: Branch if less than or equal, signed ≤

```
unsigned int x, y ;
x = 4294967295;
y = 1;
if (x > y)
   ...
```

```
MOVS r6, #0xFFFFFFFF
MOVS r5, #0x00000001
CMP  r5, r6
BLS  Then_Clause
...
```

BLS: Branch if lower or same, unsigned ≤

Signed vs. Unsigned

Conditional codes applied to branch instructions

Compare	Signed	Unsigned
==	EQ	EQ
≠	NE	NE
>	GT	HI
≥	GE	HS
<	LT	LO
≤	LE	LS

Compare	Signed	Unsigned
==	BEQ	BEQ
!=	BNE	BNE
>	BGT	BHI
>=	BGE	BHS
<	BLT	BLO
<=	BLE	BLS

Branch Instructions

	Instruction	Description	Flags tested
Unconditional Branch	B label	Branch to label	
Conditional Branch	BEQ label	Branch if EQual	Z = 1
	BNE label	Branch if Not Equal	Z = 0
	BCS/BHS label	Branch if unsigned Higher or Same	C = 1
	BCC/BLO label	Branch if unsigned LOwer	C = 0
	BMI label	Branch if MInus (Negative)	N = 1
	BPL label	Branch if PLus (Positive or Zero)	N = 0
	BVS label	Branch if oVerflow Set	V = 1
	BVC label	Branch if oVerflow Clear	V = 0
	BHI label	Branch if unsigned HIgher	C = 1 & Z = 0
	BLS label	Branch if unsigned Lower or Same	C = 0 or Z = 1
	BGE label	Branch if signed Greater or Equal	N = V
	BLT label	Branch if signed Less Than	N != V
	BGT label	Branch if signed Greater Than	Z = 0 & N = V
	BLE label	Branch if signed Less than or Equal	Z = 1 or N = !V

Conditional Execution

Add instruction	Condition	Flag tested
ADDEQ r3, r2, r1	Add if EQual	Add if Z = 1
ADDNE r3, r2, r1	Add if Not Equal	Add if Z = 0
ADDHS r3, r2, r1	Add if Unsigned Higher or Same	Add if C = 1
ADDLO r3, r2, r1	Add if Unsigned LOwer	Add if C = 0
ADDMI r3, r2, r1	Add if Minus (Negative)	Add if N = 1
ADDPL r3, r2, r1	Add if PLus (Positive or Zero)	Add if N = 0
ADDVS r3, r2, r1	Add if oVerflow Set	Add if V = 1
ADDVC r3, r2, r1	Add if oVerflow Clear	Add if V = 0
ADDHI r3, r2, r1	Add if Unsigned HIgher	Add if C = 1 & Z = 0
ADDLS r3, r2, r1	Add if Unsigned Lower or Same	Add if C = 0 or Z = 1
ADDGE r3, r2, r1	Add if Signed Greater or Equal	Add if N = V
ADDLT r3, r2, r1	Add if Signed Less Than	Add if N != V
ADDGT r3, r2, r1	Add if Signed Greater Than	Add if Z = 0 & N = V
ADDLE r3, r2, r1	Add if Signed Less than or Equal	Add if Z = 1 or N = !V

Example of Conditional Execution

```
a ⟶ r0
y ⟶ r1
```

```
if (a <= 0)
  y = -1;
else
  y = 1;
```

```
CMP   r0, #0
MOVLE r1, #-1
MOVGT r1, #1
```

LE: Signed Less than or Equal
GT: Signed Greater Than

Conditional execution in Thumb2

- For the pure ARM, a large part (4 leading bits) are dedicated to conditional execution.
- The *Thumb* instruction set does not support conditional execution.
- For the *Thumb2*, there is a variation on conditional execution. Instead of compiling a condition in each instruction, there is an it instruction which checks 8 bits in the condition register.

- **IT** (If-then) makes up to **four** following instructions conditional (known as the IT block). The conditions can all be the same, or some can be the logical inverse of others.
- Syntax: IT{x{y{z}}} {cond}
  where: *cond* is a condition code. *x* , *y* and *z* specify the condition switch for the second, third and fourth instructions in the IT block, for example, ITTET. The condition switch can be either:
  - T (Then), which applies the condition cond to the instruction.
  - E (Else), which applies the inverse condition of cond to the instruction.
- Ex:

```
ITTE NE              ; Next 3 instructions are conditional
ANDNE R0, R0, R1     ; ANDNE does not update condition flags
ADDSNE R2, R2, #1    ; ADDSNE updates condition flags
MOVEQ R2, R3         ; Conditional move
```

Example of Conditional Execution

a ⟶ r0
y ⟶ r1

```
if (a <= 0)
  y = -1;
else
  y = 1;
```

```
CMP   r0, #0
MOVLE r1, #-1
MOVGT r1, #1
```

LE: Signed Less than or Equal
GT: Signed Greater Than

If-then-else block

```
CMP   r0, #0
ITE    LE
MOVS r1, #-1
MOVS r1, #1
```

Anatomy of an assembly program

Labels

```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR   r1, =srcStr         ; Retrieve address of the source string
        LDR   r0, =dstStr         ; Retrieve address of the destination string
loop    LDRB  r2, [r1], #1        ; Load a byte & increase src address pointer
        STRB  r2, [r0], #1        ; Store a byte & increase dst address pointer
        CMP   r2, #0              ; Check for the null terminator
        BNE   loop                ; Copy the next byte if string is not ended
stop    B     stop                ; Dead loop. Embedded program never exits.

        ENDP

        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB   "The source string.",0         ; Strings are null terminated
dstStr  DCB   "The destination string.",0    ; dststr has more space than srcstr

        END
```

Directives

Program Comments

Code Area

Assembly Instructions

Directives

Data Area

Data

Program Comments

Assembly Directives

- Directives are **NOT** instruction. Instead, they are used to provide key information for assembly.

AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value.
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

Directive: AREA

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- The AREA directive indicates to the assembler the start of a new data or code section.
- Areas are the basic independent and indivisible unit processed by the **linker**.
- Each area is identified by a name and areas within the same source file **cannot share the same name.**
- An assembly program must have **at least one code area**.
- By default, a code area can only be read (READONLY) and a data area may be read from and written to (READWRITE).

Directive: ENTRY

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- The ENTRY directive marks **the first instruction to be executed** within an application program.
- There must be **exactly one** ENTRY directive in an application, no matter how many source files the application has.

Directive: END

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- The END directive indicates the end of a source file.
- Each assembly program must end with this directive.

Directive: PROC and ENDP

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- PROC and ENDP are to mark the start and end of a function (also called subroutine or procedure).
- A single source file can contain multiple subroutines, with each of them defined by a pair of PROC and ENDP.
- PROC and ENDP cannot be nested. We cannot define a function within another function.

Directive: EXPORT and IMPORT

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the begin of a subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

- The EXPORT declares a symbol and makes this **symbol visible** to the linker.
- The IMPORT gives the assembler a symbol that is **not defined locally** in the current assembly file. The symbol must be defined in another file.
- The IMPORT is similar to the “extern” keyword in C.

Directive: Data Allocation

Directive	Description	Memory Space
DCB	Define Constant Byte	Reserve 8-bit values
DCW	Define Constant Half-word	Reserve 16-bit values
DCD	Define Constant Word	Reserve 32-bit values
DCQ	Define Constant	Reserve 64-bit values
SPACE	Defined Zeroed Bytes	Reserve a number of zeroed bytes
FILL	Defined Initialized Bytes	Reserve and fill each byte with a value

Directive: Data Allocation

```
 AREA    myData, DATA, READWRITE
hello   DCB    "Hello World!",0   ; Allocate a string that is null-terminated

dollar  DCB    2,10,0,200         ; Allocate integers ranging from -128 to 255

scores  DCD    2,3.5,-0.8,4.0     ; Allocate 4 words containing decimal values

miles   DCW    100,200,50,0       ; Allocate integers between –32768 and 65535

p       SPACE    255               ; Allocate 255 bytes of zeroed memory space

f       FILL   20,0xFF,1          ; Allocate 20 bytes and set each byte to 0xFF

binary  DCB    2_01010101         ; Allocate a byte in binary

octal   DCB    8_73               ; Allocate a byte in octal

char    DCB    ‘A’                ; Allocate a byte initialized to ASCII of ‘A’
```

Directive: EQU and RN

```
; Interrupt Number Definition (IRQn)
BusFault_IRQn    EQU  -11        ; Cortex-M3 Bus Fault Interrupt
SVCall_IRQn      EQU   -5        ; Cortex-M3 SV Call Interrupt
PendSV_IRQn      EQU   -2        ; Cortex-M3 Pend SV Interrupt
SysTick_IRQn     EQU   -1        ; Cortex-M3 System Tick Interrupt

Dividend         RN    6         ; Defines dividend for register 6
Divisor          RN    5         ; Defines divisor for register 5
```

- The EQU directive associates a symbolic name to a numeric constant. Similar to the use of #define in a C program, the EQU can be used to define a constant in an assembly code.
- The RN directive gives a symbolic name to a specific register.

Directive: ALIGN

```
    AREA example, CODE, ALIGN = 3    ; Memory address begins at a multiple of 8
    ADD r0, r1, r2                   ; Instructions start at a multiple of 8

    AREA myData, DATA, ALIGN = 2     ; Address starts at a multiple of four
a   DCB 0xFF                         ; The first byte of a 4-byte word
    ALIGN 4, 3                       ; Align to the last byte (3) of a word (4)
b   DCB 0x33                         ; Set the fourth byte of a 4-byte word
c   DCB 0x44                         ; Add a byte to make next data misaligned
    ALIGN                            ; Force the next data to be aligned
d   DCD 12345                        ; Skip three bytes and store the word
```

Directive: INCLUDE or GET

```
        INCLUDE constants.s       ; Load Constant Definitions
        AREA main, CODE, READONLY
        EXPORT  __main
        ENTRY
__main  PROC
        ...
        ENDP
        END
```

- The INCLUDE or GET directive is to include an assembly source file within another source file.
- It is useful to include constant symbols defined by using EQU and stored in a separate source file.

Assembly process