SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

ENSC254 – Load / Store Operations

Last updated June 13, 2021

© 2017 -- Fabio Campi and Craig Scratchley
School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

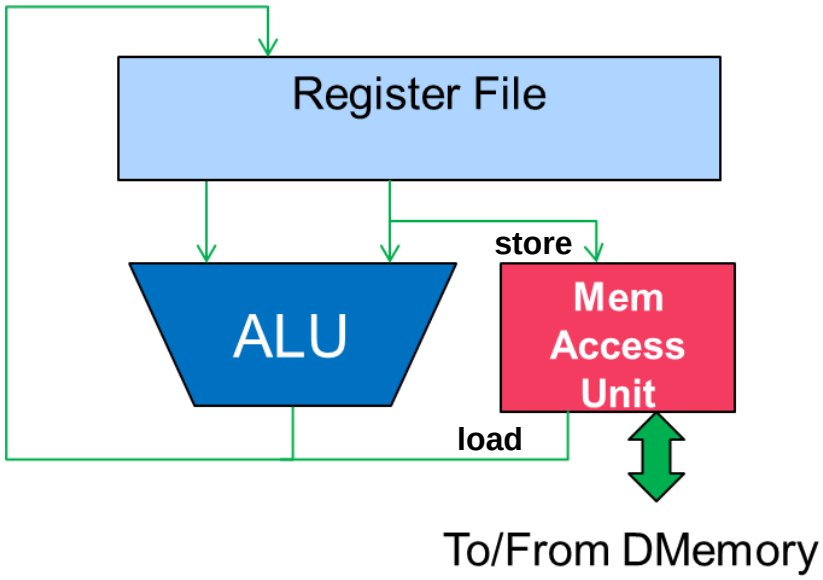
LOAD / STORE Architecture

- The most notable feature of RISC architectures is LOAD/STORE, which means that ***Arithmetical/Logical operations CANNOT access the external memory***

- *That is one of the very few RISC features that ARM left untouched*

- The only instructions that are allowed to access external memory are those for LOAD (transfer data from memory to register) and STORE (transfer data from register to memory)

- ALU operations can only use operands derived from registers, including the Instruction Register. So constants (a.k.a. immediates) are derived from the Instruction Register



LOAD and STORE Operations in ARM

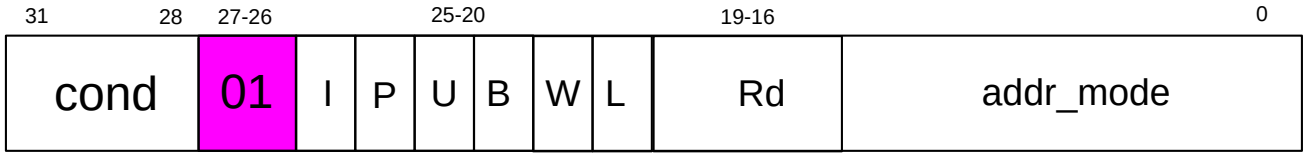
LDR | STR *{cond} {B} <Rd>, <addressing_mode>*

Will load the content of an address to register Rd | store the content of register Rd to an address

- {Rd} can be any register, but loading to R15 (Program Counter) will be special as it usually creates a jump to a code location different than the next instruction
- {B} specifies the dimension of the operation: present for Byte (8-bits) – otherwise word (32-bits)
- There are several ways (modes) to specify the **EFFECTIVE ADDRESS**, but it is based on the sum of a **BASE ADDRESS,** contained in a register, and an **OFFSET,** specified as a constant/immediate or another register. A register offset can be manipulated to some degree
- The base address can be, for example, the start point of the relative section, or the base of a struct, or the 0^{th} element of a given array, such as the beginning of a string. The offset is used to identify the specific address of a given single field/element in the section / struct / vector / string. Often the base address is the Program Counter (i.e. related to the location of the current instruction).

ARM Load / Store Operations

LDR | STR *{cond} {B} <Rd>, <address>*



Note: The IPUBWL flags will be described in the course of the present unit

I = Immediate / register offset
P = Pre / post indexed addressing
U = Upward* (add offset) / downward (subtract offset)
B = unsigned Byte / word – halfword has different instruction format
W= address Write-back
L = Load / store

*address goes Up, which takes you down the screen, not up the screen.

The format of the “addr_mode” field depends on the chosen *addressing mode*

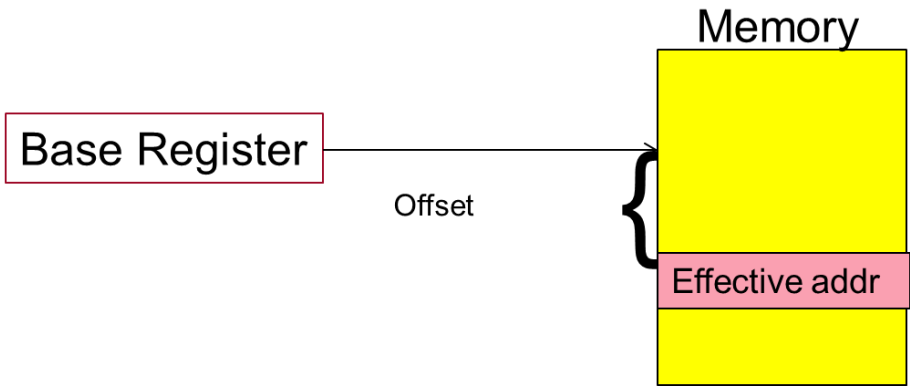
ADDRESSING MODES

Addressing Modes

Wikipedia: ***Addressing modes*** *are an aspect of the instruction set architecture in processor design.*

- ADDRESSING MODES define how machine language instructions identify the operand(s) of each instruction.
- Addressing modes specify how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.
- In computer programming, addressing modes are primarily of interest to compiler writers and to those who write code directly in assembly

ARM Addressing Modes



- *Note: ADDRESSING MODES used to be a very tricky part of CISC ISAs*
- There is basically only one addressing mode in RISC processors:

source/Dest register <=> [RBASE + offset (register or immediate)]

- *As usual, the case of ARM is a bit more complicated than classic RISC*

Wake Up Call 1:

How many bits are necessary to encode a single register address in ARM?

And How many bits are necessary to encode registers involved in an RRR operation?

a) 5 and 15
b) 4 and 12
c) 3 and 9
d) 3 and 12
e) 5 and 12

Wake Up Call 1:

How many bits are necessary to encode a single register address in ARM?

And How many bits are necessary to encode registers involved in an RRR operation?

a) 5 and 15
b) 4 and 12
c) 3 and 9
d) 3 and 12
e) 5 and 12

ARM has 16 addressable GP registers, so we can encode them in 4 bits.
The RRR format needs to specify 3 registers, so 4*3=12bits

ARM Addressing Modes:

Rd is the register where we want to load or from which we want to store data

Rn is the base register for the address calculation

- offset_12 is the 12-bit constant offset for address calculation
- Rm is the offset register for the address calculation

Offset modes:

1. IMMEDIATE OFFSET: LDR/STR <Rd>, [<Rn>, #{+/-}<offset_12>]

2. (REGISTER OFFSET: LDR/STR <Rd>, [<Rn>, {+/-}<Rm>])

3. SCALED REGISTER OFFSET:
   LDR/STR <Rd>, [<Rn>, {+/-}<Rm>, <shift> #<shift_imm>]

ARM Addressing Modes: Immediate Offset

Rd is the register where we want to load or from which we want to store data

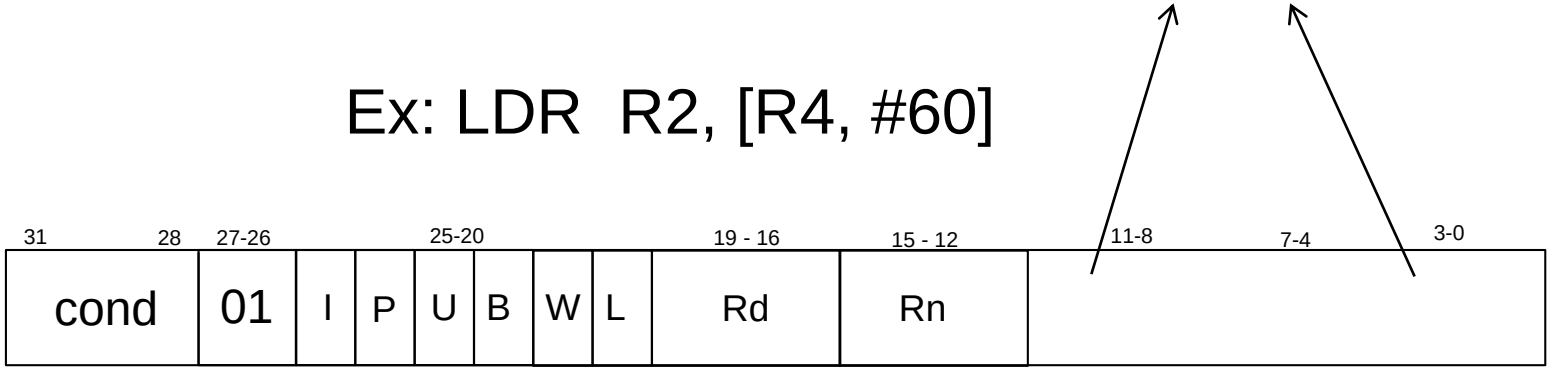
Rn is the base register for the address calculation

Rm is the offset register for the address calculation

Offset is the 12-bit constant offset for address calculation

1. IMMEDIATE OFFSET: <Rd>, [<Rn>, #{+/-}<offset_12>]

Ex: LDR R2, [R4, #60]



ARM Addressing Modes: Register Offset

Rd is the register where we want to load or from which we want to store data

Rn is the base register for the address calculation

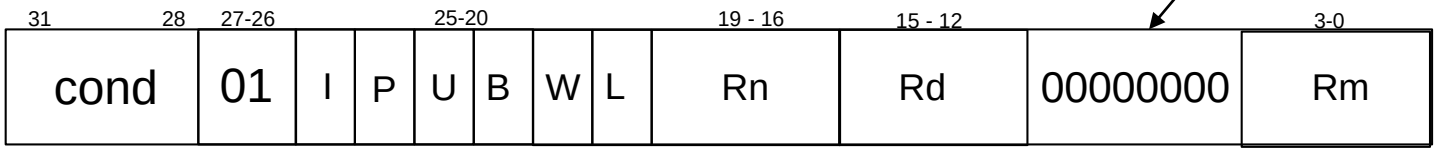
Rm is the offset register for the address calculation

Offset is the 12-bit constant offset for address calculation

2. REGISTER OFFSET: LDR/STR <Rd>, [<Rn>, {+/-}<Rm>]

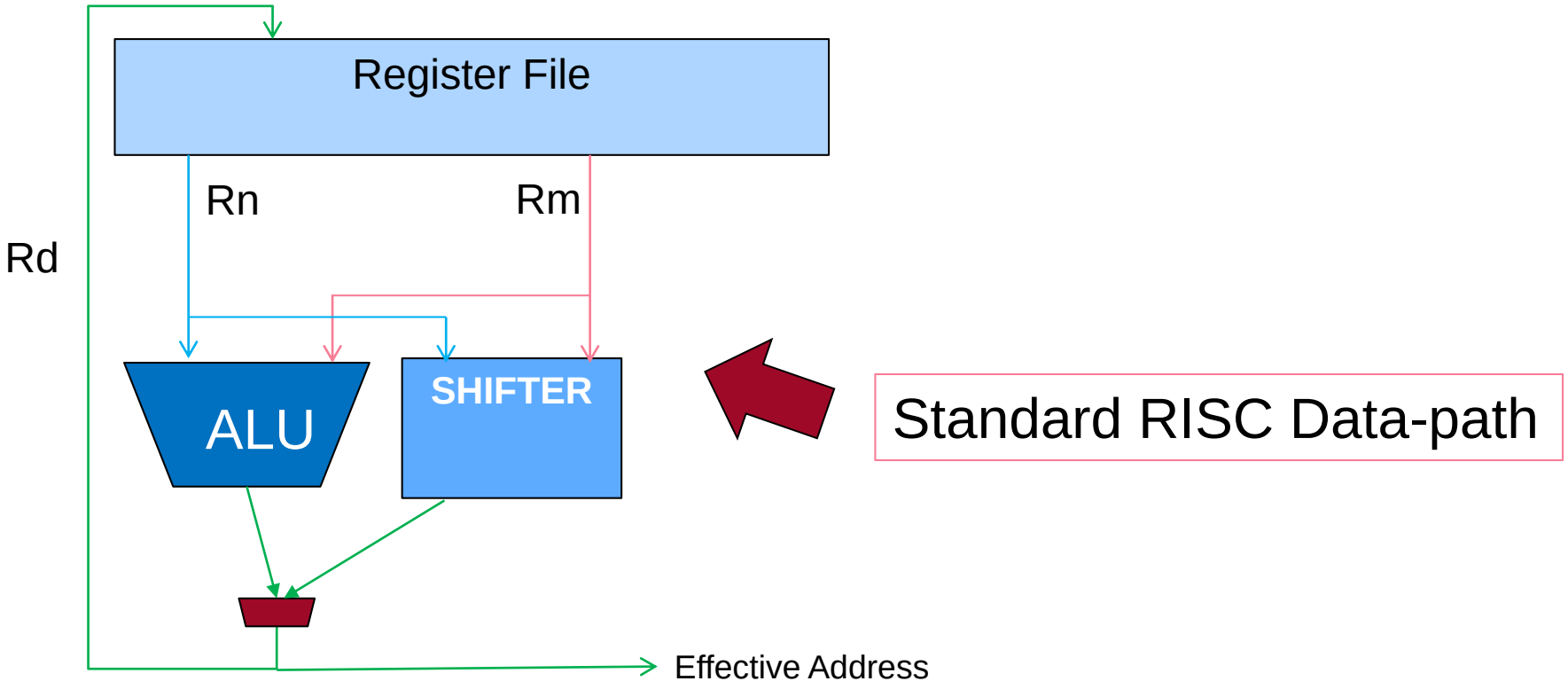
Ex: LDR R2, [R4, R5]

A shift operand is included in this area in mode #3



Note: R15 can never be an offset: Being the PC it must always be a base register for addressing

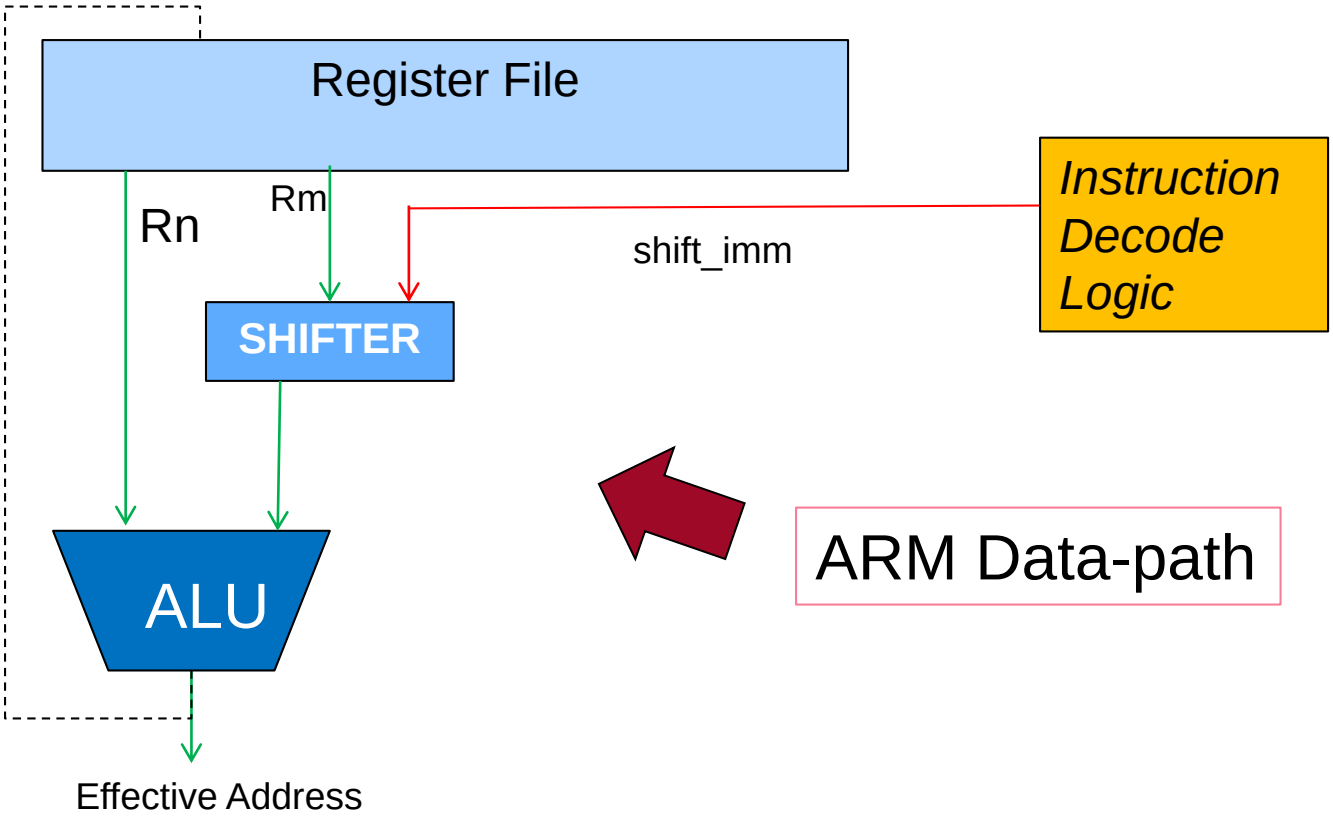
IMPORTANT: ARM vs Standard RISC Datapath



- LOAD/STORE Addresses are computed by the ALU, similarly to an ADD operation (Base+Offset)

The ARM data-path has a very peculiar unique feature: The shifter is not part of the ALU, or another functional unit parallel to the ALU: it is actually moved to be used BEFORE the ALU

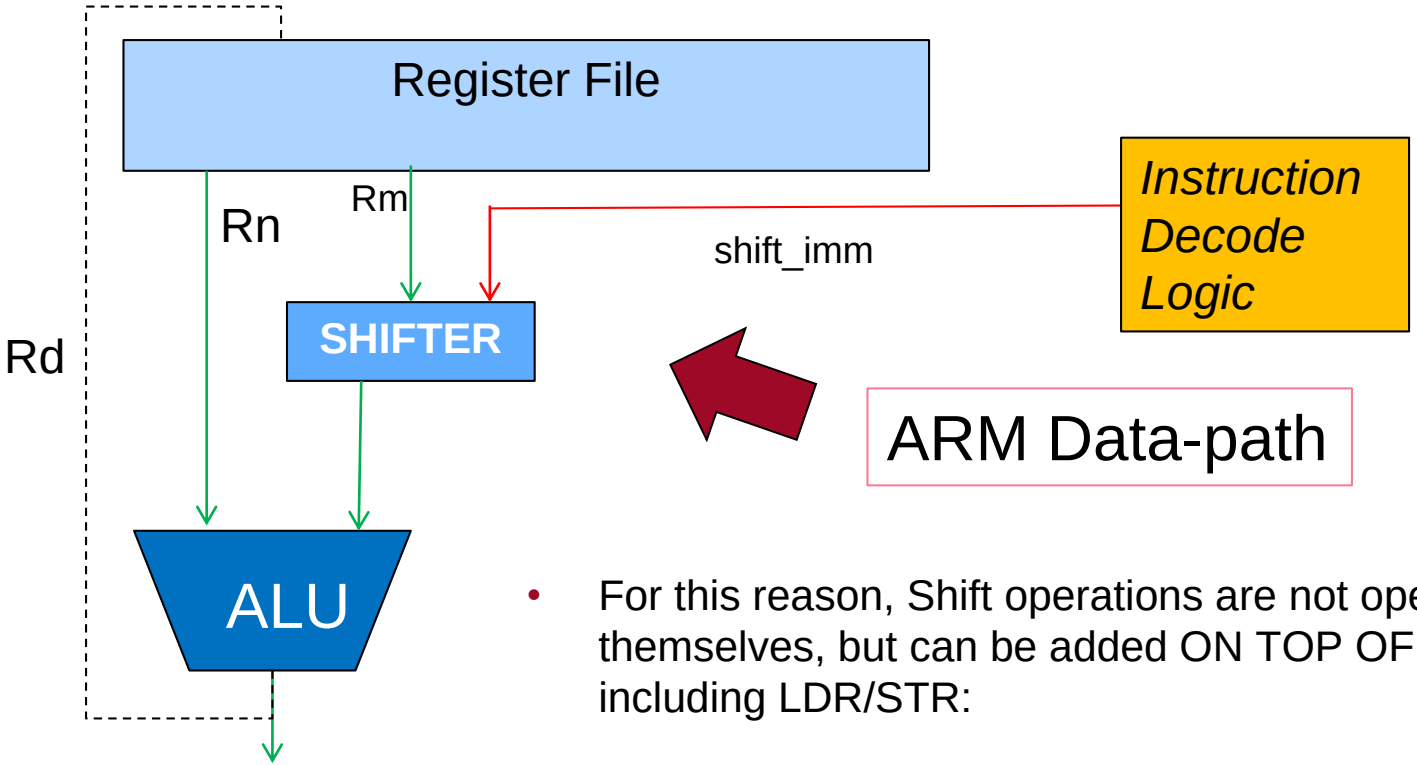
IMPORTANT: ARM Datapath



The ARM data-path has a very peculiar unique feature: The shifter is not part of the ALU, or another functional unit parallel to the ALU: it is actually moved to be used BEFORE the ALU.

This would make the processor slower, AND the instruction format more complex, but will allow for much more compact program size

SHIFT OPERATIONS



- For this reason, Shift operations are not operations in themselves, but can be added ON TOP OF existing operations, including LDR/STR:

LDR Rd, [Rn, Rm, LSL #2] ;@ Rd=[Rn+(Rm << 2)] -- shift 2 bits

where Rn is the base register and Rm provides the offset before shifting

Quiz (1)

- Suppose that, *ultimately as part of a loop,* you want to access the 7th element of an array of bytes (i.e. the element with an index of 6), with the array at a base of 0x4000: write the assembly code that is capable of performing that including code to initialize some registers. Later, while looping, smaller-numbered elements will be accessed.

Addressing modes:

1. IMMEDIATE OFFSET: LDR/STR Rd, [Rn, #{+/-}<offset_12>]
2. REGISTER OFFSET: LDR/STR Rd, [Rn, {+/-}Rm]
3. SCALED REGISTER OFFSET: LDR/STR Rd, [Rn, {+/-}Rm, <shift> #<shift_imm >]

Note: If this access will be part of a loop, we want to use a register as offset!

Quiz (1b)

- Suppose that, *ultimately as part of a loop,* you want to access the 7th element of an array of bytes (i.e. the element with an index of 6), with the array at a base of 0x4000: write the assembly code that is capable of performing that including code to initialize some registers. Later, while looping, smaller-numbered elements will be accessed.

```
        ;@ initialize registers
        MOV r0, #0x4000
        MOV r1, #6
loopTop:
        ;@ top of loop
        LDRB r2, [r0, r1]
        ;@ …
```

Quiz (1c)

- ... Later, while looping, smaller-numbered elements will be accessed. We're getting a bit ahead of ourselves here, but below is what such a loop would look like.

```
            ;@ initialize registers
            MOV r0, #0x4000
            MOV r1, #6
loopTop:
            ;@ top of loop
            LDRB r2, [r0, r1]
            ;@ process r2
            SUB r1, r1, #1 ;@ or SUBS
            CMP r1, #0     ;@ not needed with SUBS
            BNE loopTop
            ;@ …
```

Quiz (2)

- Suppose that, *ultimately as part of a loop*, you want to access the 7th element of an array of bytes (i.e. the element with an index of 6) with the array at a base of 0x4000: write the assembly code that is capable of performing that
  - Now suppose you want to do the same with an array of **words (32 bits)**

a)
```
MOV r0,#0x4000
MOV r1,#6

LDR r2,[r0,r1]
```

b)
```
MOV r0,#0x4000
MOV r1,#6

LDR r2,[r0,r1,LSL #1]
```

c)
```
MOV r0,#0x4000
MOV r1,#6

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

d)
```
MOV r0,#0x4000
MOV r1,#6

LDR r2,[r0,r1,LSL #4]
```

e)
```
MOV r0,#0x4000
MOV r1,#6

LDR r2,[r0,r1,LSR #1]
```

Quiz (2)

- Suppose that, *ultimately as part of a loop*, you want to access the 7th element of an array of bytes (i.e. the element with an index of 6) with the array at a base of 0x4000: write the assembly code that is capable of performing that.
  - Now suppose you want to do the same with an array of **words (32 bits)**

a)
```
MOV r0,#0x4000
MOV r1,#6

LDR r2,[r0,r1]
```

b)
```
MOV r0,#0x4000
MOV r1,#6

LDR r2,[r0,r1,LSL #1]
```

c)
```
MOV r0,#0x4000
MOV r1,#6

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

d)
```
MOV r0,#0x4000
MOV r1,#6

LDR r2,[r0,r1,LSL #4]
```

e)
```
MOV r0,#0x4000
MOV r1,#6

LDR r2,[r0,r1,LSR #1]
```

Quiz (3)

- Suppose that, *ultimately as part of a loop*, you want to access the 7th element of an array of bytes (i.e. the element with an index of 6) with the array at a base of 0x4000: write the assembly code that is capable of performing that.
  - Now suppose you want to do the same with an array of **half words (16-bit)**

a)
```
MOV r0,#0x4000
MOV r1,#6

LDRH r2,[r0,r1]
```

b)
```
MOV r0,#0x4000
MOV r1,#6

LDRH r2,[r0,r1,LSL #1]
```

c)
```
MOV r0,#0x4000
MOV r1,#6

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

d)
```
MOV r0,#0x4000
MOV r1,#6

LDRH r2,[r0,r1,LSL #4]
```

e)
```
MOV r0,#0x4000
MOV r1,#6

LDRH r2,[r0,r1,LSR #1]
```

Quiz (3)

- Suppose that *as part of a loop* you want to access the 7th element of an array of bytes (i.e. the element with an index of 6) with the array at a base of 0x4000: write the assembly code that is capable of performing that.
  - Now suppose you want to do the same with an array of **half words (16-bit)**

a)
```
MOV r0,#0x4000
MOV r1,#6

LDRH r2,[r0,r1]
```

b)
```
MOV r0,#0x4000
MOV r1,#6

LDRH r2,[r0,r1,LSL #1]
```

c)
```
MOV r0,#0x4000
MOV r1,#6

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

d)
```
MOV r0,#0x4000
MOV r1,#6

LDRH r2,[r0,r1,LSL #4]
```

e)
```
MOV r0,#0x4000
MOV r1,#6

LDRH r2,[r0,r1,LSR #1]
```

ARM Addressing Modes: Scaled Register Offset

Rd is the register where we want to load or from which we want to store data

Rn is the base register for the address calculation

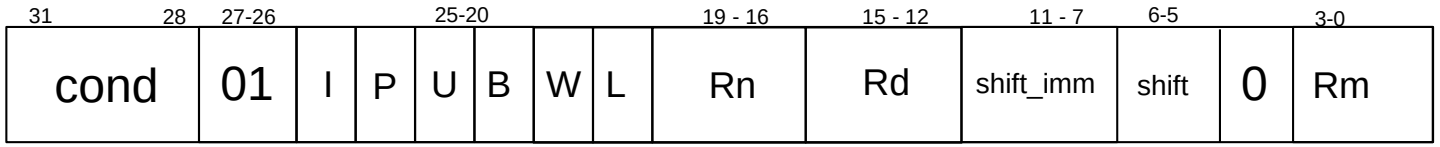
Rm is the offset register for the address calculation

shift_imm and **shift** op express the actual offset for address calculation as Rm, **shift** #**shift_imm**

3. SCALED REGISTER OFFSET:

   LDR/STR <Rd>, [<Rn>, #{+/-}<Rm>, <shift> #<shift_imm>]

Ex: LDR R2, [R4, R5, LSL #2]



ARM Addressing Modes (List)

(See TextBook for a complete Description)

1. IMMEDIATE OFFSET
2. REGISTER OFFSET
3. SCALED REGISTER OFFSET

4. IMMEDIATE PRE-INDEXED
5. REGISTER PRE-INDEXED
6. SCALED PRE-INDEXED

7. IMMEDIATE POST-INDEXED
8. REGISTER POST-INDEXED
9. SCALED POST-INDEXED

Assembly level Load/Store Specification

- All Load/Store operations in ARM are based on the BASE + Offset format, and can have two classes of encodings:
  - LDR/STR rdata, [rbase, #immed_offset]
    or rather… LDR/STR rdata, [rbase{,#immed_offset}]
  - LDR/STR rdata, [rbase, roffset]

- From the Assembly point of view, we NEED TO CHOSE, ourselves, from options including:
  1. If the location we want to access is in the vicinity of the program counter [R15] we can use PC as base register and a constant immediate as offset

     ldr r4, constant_location

     - *This will force the Assembler to use PC as a base register, and an immediate-offset – this is pseudo-direct addressing.*
  2. If the location we want to access is too far to be accessed by PC+immed, we **can** load address and/or offset on a register – this is pseudo-immediate addressing.

     ldr r3, =variable_location

     ldr r4, [r3]

Assembly level Load/Store Specification (2)

- All Load/Store operations in ARM are based on the BASE + Offset format, and can have two classes of encodings:
  - LDR/STR rdata, [rbase, #immed_offset]
  - LDR/STR rdata, [rbase, roffset]

- From the Assembly point of view, we NEED TO CHOSE OURSELVES from options including the following pseudo instructions:
  1. If the location we want to access is in the vicinity of the program counter [R15] we can use PC as base register and a constant immediate as offset

     ldr r4, constant_location

  2. If the location we want to access is too far to be accessed by PC+immed, we **can** load address and/or offset on a register – this is pseudo-immediate addressing.

     ldr r3, =variable_location

     ldr r4, [r3]

     - *Will always work, regardless of the position of the variable location WRT the PC*
     - *Unfortunately, will cost more in terms of instruction/data count and execution time*
       - *[LDR operation takes 3 times as much time as arithmetic operations on NXP LPC2104 microcontroller with ARM7TDMI, for example]*

SIGNED / UNSIGNED DATA

Signed / Unsigned Data

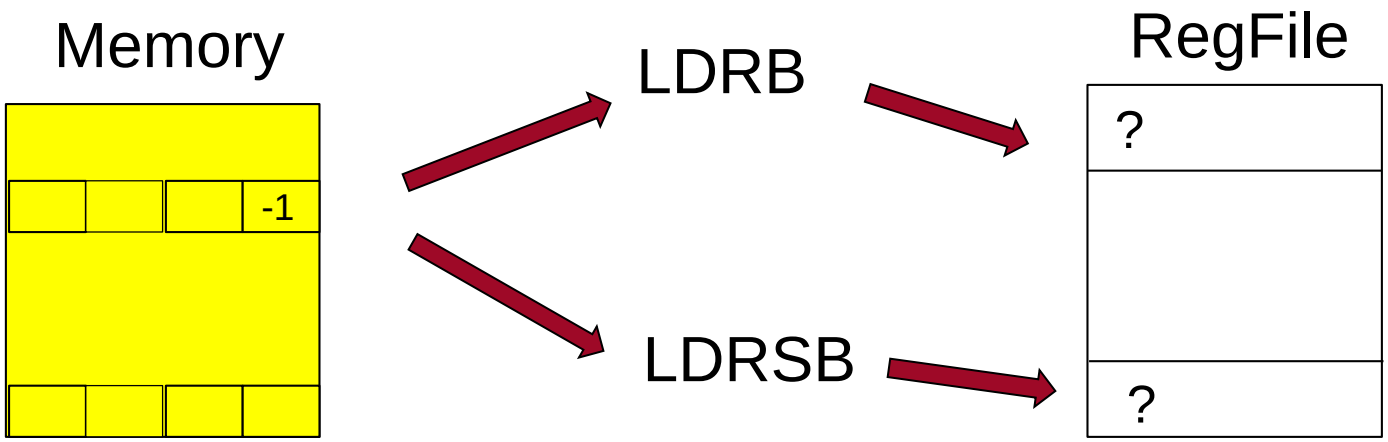
The content of a word of memory (or a fraction thereof) in a 32-bit ARM system can be defined, in a high level programming language like *C*, as

- Integer (int) / Unsigned Integer (unsigned int) – 32-bits;
- short int / unsigned short int – 16-bits;
- Byte (char) / Unsigned Byte (unsigned char) – 8-bits;

What difference does it make in the assembly translation of the code ?

a) Add/Sub operations have a different handling of the overflow
b) Right Shift operations need to know what to insert (at the most-significant end)
c) When translating data from a smaller to a larger format, the processor needs to know whether to Sign Extend

Sign Extension



- Data loaded from memory can be 8, 16 or 32 bits. Registers are always 32 bits.
- When loading a byte / half-word into a register, we need to fill 16 or 24 extra bits
  - We can not fill them with zeroes if the number is signed, otherwise we would mess up the number significantly
  - We can not fill them with bit 7 for bytes and 15 for half-words, as in case of unsigned numbers that would change the number value.

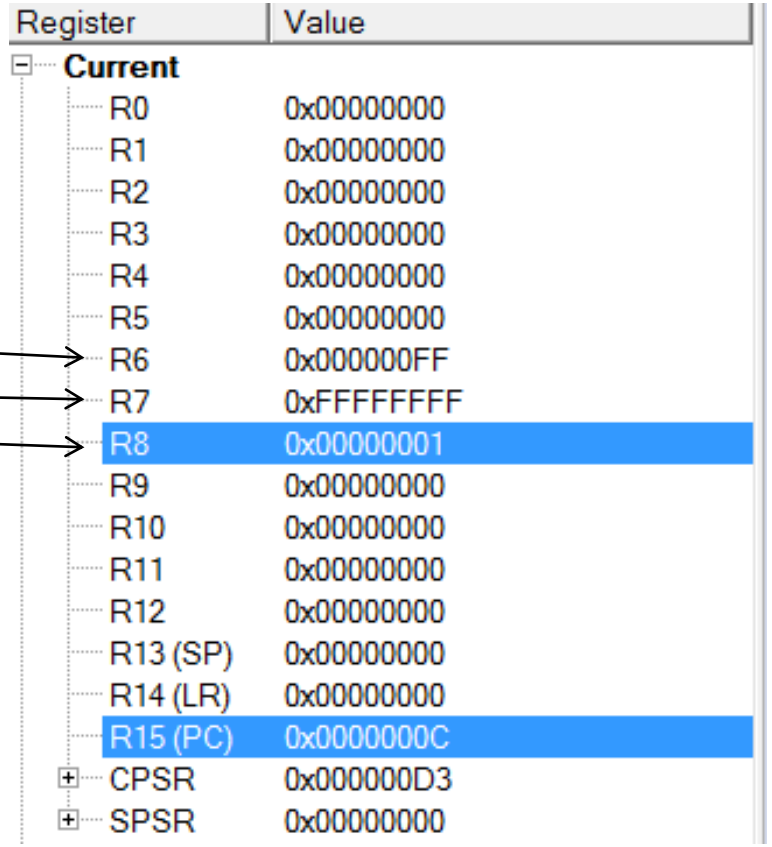
we must differentiate in the case of loads between Signed or Unsigned operands

This is accomplished by using special “S” instruction variants for signed numbers

Sign Extension Example

```
    LDRB  r6,A1
    LDRSB r7,A1
    LDRSB r8,A2

Stop:
    b stop
A1  .byte -1
A2   .byte 1
```



Quiz

Which of the following instructions needs to specify the type of sign-extension?

a) Load
b) Store
c) Mov
d) Load and Store
e) Load and Store and Mov

Quiz

Which of the following instruction needs to specify the type of sign-extension?

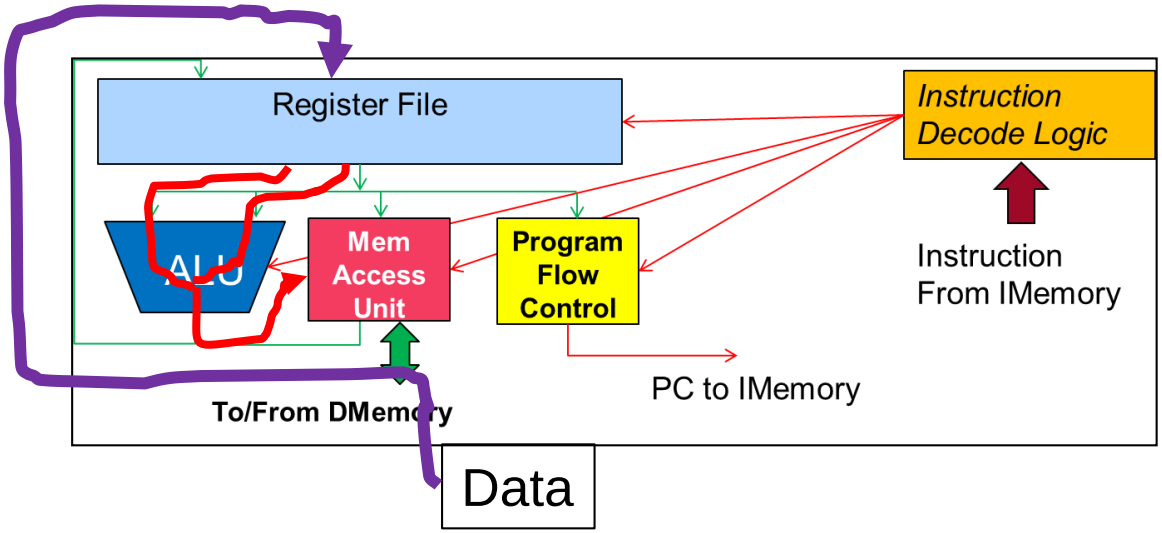
a) Load
b) Store
c) Mov
d) Load and Store
e) Load and Store and Mov

Sign Extension Specification

Load	Store	Size and Type
LDR	STR	Word (32-bit)
LDRB	STRB	Byte (8-bits)
LDRH	STRH	Half-word (16-bit)
LDRSB		Signed Byte
LDRSH		Signed Half-word
LDM	STM	Multiple Words

- ***Notice that only Load operations of "narrow data" need to specify a Sign.***
- That is because the Sign is only used to extend the Most Significant Bit of a given byte or half-word when loaded into a larger "container" (i.e., a register).
- When writing a register into smaller locations, we only need to truncate insignificant data, and that is done in the same fashion regardless the signed/unsigned nature of the data

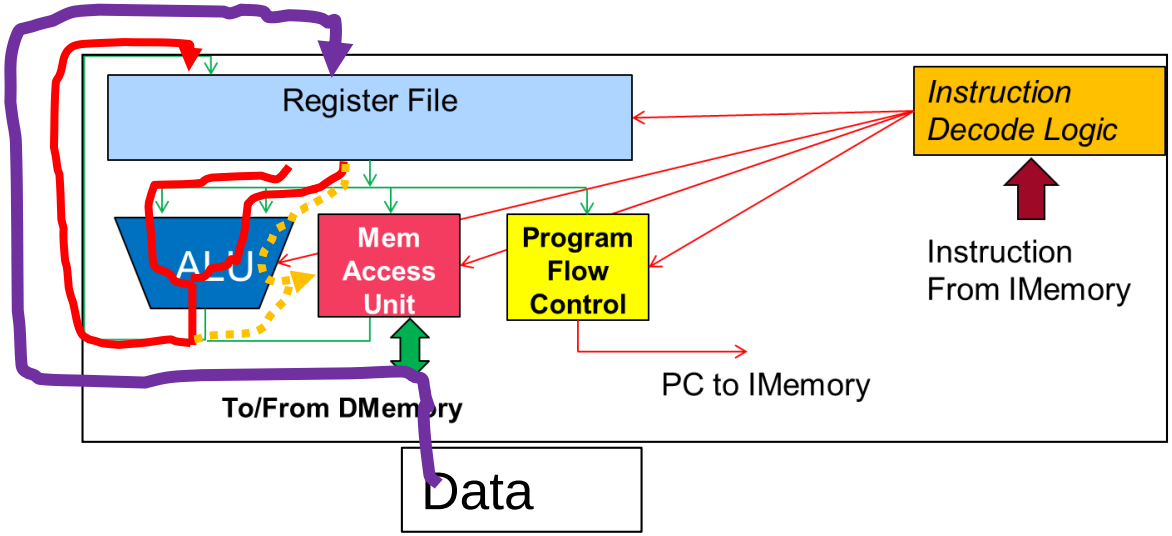
Writeback



The Load operation stores the data Read from memory into the register file

Address Writeback

- Differently from most RISC architectures, *ARM allows for address write-back on the base registers*
- This is done in order to avoid the need for a separate add operation following the memory access *in case of Loops Accessing ARRAYS*



- ***The addition can be done either BEFORE or AFTER sending the address to the memory system, and it is called PRE or POST Indexed Addressing***

Address Write-back

- **PRE-INDEXED Addressing**: **LDR|STR{<cond>}{B} <Rd>,[<Rn>, <offset>] !**

In this case, the effective address calculation is computed BEFORE the memory access and used for both the memory access and for writing back to Rn. If ! is not specified, we have already seen this before and there is no writeback.

Example: LDRB r6,[r1,#1]!

- **POST-INDEXED Addressing**: **LDR|STR{<cond>}{B} <Rd>,[<Rn>], <offset>**

In this case, the addition is done “AFTER” the address in Rn has been provided to the memory system, and the sum is written back on Rn. There is no !, as it would be pointless here to do an addition if there was no write back

Example: LDRB r6,[r1],#1

Address Writeback example

EX: Suppose we want to write a Multiply-accumulate operation of two byte arrays, which is extremely common in Signal Processing:

That can be rendered in C with

char A1[5] = {2, 4, 1, 1, 3}; char A2[5]= {1, 0, 1, 1, 2};

int Acc = 0; for(i=0; i<5; i++) {Acc += A1[i] * A2[i]}

$$Out = 2 * 1 + 4 * 0 + 1 * 1 + 1 * 1 + 3 * 2 = 10$$

```
        MOV   r0,#5   ;@ array size
        LDR   r1,=A1
        LDR   r2,=A2
        MOV   r8,#0
Loop    LDRSB r6,[r1] ;@ offset is zero
        LDRSB r7,[r2]
        MUL   r5,r6,r7
        ADD   r8,r8,r5 ;@ result goes back in r8
        ADD   r1,r1,#1
        ADD   r2,r2,#1
        SUBS  r0,r0,#1
        BNE   Loop
Stop    B     Stop
A1      DCB 2,4,1,1,3
A2      DCB 1,0,1,1,2
```

Address Writeback example

- In the code on the previous page, we are taking care of incrementing the addresses with specific instructions in each iteration

```
ADD   r1, r1, #1
ADD   r2, r2, #1
```

- Of course, we would be most happy if the load operation could do that for us, even though that would be a very un-RISC-ish thing to do

- *Note that this Address Writeback is against much of what RISC stands for, because we have two writeback operations in the same cycle. We can consider this an example of feature creep!*

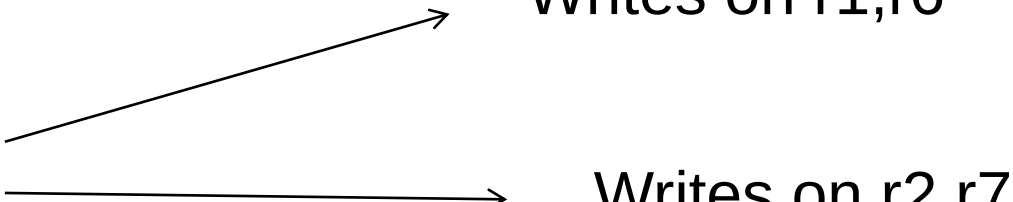
- We may not like it from a hardware design perspective, but we must admit that from the software perspective it makes things easy and fast!!!!!!!!

Address Writeback Example

```
        MOV    r0,#5
        LDR    r1,=A1
        LDR    r2,=A2
        MOV    r8, #0

Loop    LDRSB r6,[r1],#1
        LDRSB r7,[r2],#1
        MUL    r5,r6,r7
        ADD    r8,r8,r5
        ;@ two add instructions deleted here with destinations r1 and r2
        SUBS   r0,r0,#1
        BNE     loop
Stop    b stop

A1      DCB 2,4,1,1,3
A2      DCB 1,0,1,1,2
```



LOADING CONSTANTS

Loading constants

Loading constants value from the program into a register is extremely common and extremely important at assembly level

1. We may need to perform an operation by constant (example a=a*0x2319876)
2. We may need to specify a memory address as base for load/store operations

Remember that ARM is a 32-bit Instruction Set: immediate values need to fit into a single instruction, so they are necessarily smaller that 32 bits. Actually, with condition codes and other tricks, ARM has even smaller space in instructions than standard RISC for constants

Loading a large constant into a register.

Three possible **generic** ways to load a large constant into a register:

(a) LDR $r12, =0x12***3456***78

(b) MOVT $r12, #0x1234
ADD $r12,$r12,#0x***00***78

(c) MOV $r12, #0x12**00**
LSL $r12, #16
ADD $r12,$r12,#0x*00*78

** or, actually, for ARM: **

MOV $r12, #0x12**00**0000
ADD $r12,$r12,#0x*00*78

Note that there is no MOVT in ARMv4, so we'll have to make do with the remaining two until we get to ARMv6T2 and ARMv7

Loading Constants

- A standard way of handling constants is to place the constant in the memory, and then load it with a load operation:

C Code

```
const int c= 0xc0dec0de;
```

ARM ASM Code

```
LDR r1,=0xc0decde
```

Heading towards Machine code this becomes

```
0x040          LDR r1,[PC,#0x100]
…
0x148 0xc0dec0de
```

This is called a literal

Loading Constants

- A standard way of handling constants is to place the constant in the memory, and then load it with a load operation:

C Code

```
int d = c + 0xc0dec0de;
```

ARM ASM Code

```
LDR r1,=0xc0dec0de
```

Heading towards Machine code this becomes

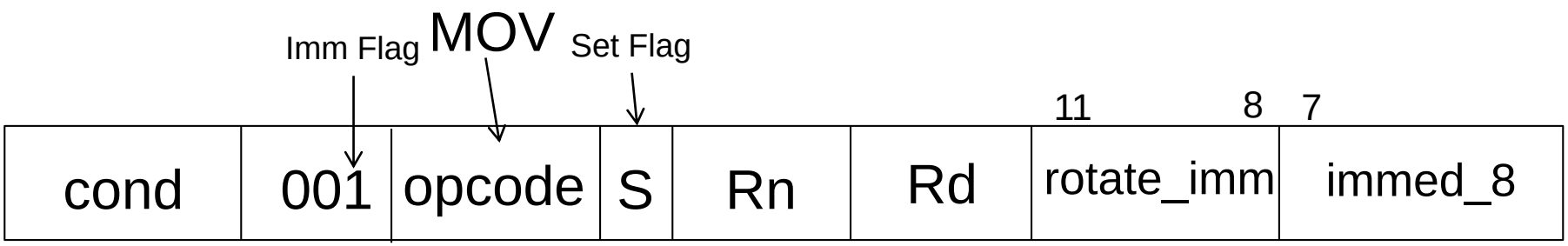
This is called a literal

```
0x040                LDR r1,[PC,#0x100]
…
0x148 0xc0dec0de
```

- But going into memory can be unnecessarily painful. We would like to pack, if possible, our constant in the instruction to save effort: An analysis of the typical software showed ARM that
  - 50% of the constants reside in the range +- 15
  - 90% of the constants reside in the range +-511
  - larger constants are quite often simple power-of-two numbers

Loading Constants (2)

- For this reason, ARM came up with the following option to express constants, which is a bit complex but can be very compact:



- The immediate (Constant) is expressed by an 8-bit value
- The constant can be rotated right by a number of bits. Not every constant between 0 and 2^32 -1 is possible with this scheme, but we can still process it further to load other 32-bit numbers in a register with an additional (ROTATE and ADD) / (ROTATE and SUB) operation.
- The idea is that complicated processing will be necessary only in rare cases

Loading Constants Example

- Although you need to know this level of detail to understand machine code, the Assembler tool can do the calculation for you:

```
MOV r6,#4080
LDR r6, =#4080
```

`0xff shifted left by 4`

- Note that, while in C you can assign any constant, and the compiler and assembler will take care of all the relative processing:

const int con=12345;

```
        LDR     r5,=con
        LDR     r4,[r5]
        …
        AREA C,DATA,READONLY
con     DCD     12345
```

- In assembly language, you will be notified of an error if the constant can not be rendered by 8-bit immediate + rotation:

```
MOV r4,#12345
```

«Immediate can not be represented by 0-255 and rotation»

Loading a constant with a pseudo addressing mode

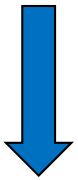
- What if we don't know at the time of choosing an instruction the constant we will eventually choose to load? Are we sometimes risking an assembler error when we change the constant value?
- Actually yes, if we change the value of a constant that we use in a MOV operation, that may lead to an error.
- The best option is to write assembler code at high level, using PSEUDO INSTRUCTIONS or PSEUDO ADRESSING MODES
  - A Pseudo Instruction is an instruction that does not have a direct correspondence to machine code, but may be translated in one or more instructions, or in different instructions depending on the context. In other words, it is a "flexible" form of MACRO, that is translated in different ways depending on the context
  - A Pseudo Addressing Mode is an addressing mode that does not exist in the architecture, but can be transformed into instructions or addressing modes that do exist.
- In this case, using the pseudo addressing mode

  LDR Rd, =CONST

- we never get errors, but only different machine code depending on the type and possibly value of CONST

Loading a constant with a pseudo addressing mode (2)

```
const    EQU, 4800
         LDR r2,=const
```



```
E3A02D4B  MOV       R2,#0x000012C0
```

```
const    EQU, 12345
         LDR r2,=const
```

```
E51F2004  LDR       R2,[PC,#0xNNN]
...
00003039
```

- In this case it can be an Assembler tool to choose, from your flexible definition, the better solution depending on the value of «*const*»
- Doing this, we are losing control on the machine code, giving more responsibility to the Assembler.

Forcing a “Literal Pool” near the code where it is being executed

- We have seen that often we don’t have control on where the Assembler will place the literal constants we are accessing with a “LDR =“ instruction (using a pseudo addressing mode)

The LTORG directive instructs the assembler to place the current literal pool at the current location.

- Without our help, the assembler places an LTORG directive at the end of every code section, so that at the and of any code section we will have all constants related to it
- If the section is too big, distances could be > 2^12, so we can use LTORG to force literal pools closer to our code.

*Know this: place LTORG directives **ONLY** after unconditional branches or similar unconditional instruction (such as unconditional subroutine return instructions) so that the processor does not attempt to execute a literal as an instruction.*