Prof. Dr.-Ing. A. Siggelkow

Computer Technology: Lab Notes

(October 8, 2024)

Hochschule Ravensburg-Weingarten
University of Applied Sciences

B

Contents

These lab notes describe the experiments conducted in the lab Computer Technology at the **University of Applied Sciences Ravensburg-Weingarten** . The experiments are drafted for students of electrical engineering in their second term (bachelor's program) and cover the areas taught in the lecture Computer Technology ([1]). Every student has to show all experiments. All experiments must be:

- prepared,
- conducted,
- documented as a scientific report and
- presented.

The result will be a part of the final grade of "Computer Technology".

The experiments covers the topics of lecture "Computer Technology": Basic programming, ARM assembler, load and store, sub-program call.

Chapter 1

General Remarks

1.1 Date and Time

This lab takes place at the University, in presence. We are starting with an all together introduction and Herr Martin Meier organizes a timetable. At the end of the semester there is one special time slot for the presentation.

1.2 Contact Persons

Contact persons are:

- Mr. Martin Meier , H215, Tel. 9611, martin.meier@rwu.de
- Prof. Dr-Ing. Stephan Jobke , H230, Tel. 9625, bruemmer@hs-weingarten.de
- Prof. Dr-Ing. Andreas Siggelkow , H218, Tel. 9633, andreas.siggelkow@rwu.de

1.3 Preparation, Report, Passing the Lab

- Preparation: The experiments of the labs are grouped in 4 tasks. Every task is focused on a specific topic. This means the contents of the lecture "Computer Technology" have to be well understood. It makes no sense to do this Lab without knowledges of the ISA of the ARM. The tasks itself will be described later in this script. The preparation sheet will be checked with the begin of the lab, a poor preparation leads to a rejection of the student from the lab.
- Reports: The projects have to be documented in a scientific report and an presentation. The presentation takes place at the end of the semester

according a special time schedule. The report and the presentation will be graded. The first report will be checked twice: The first version will be commented and graded, then the student can correct it, the second version will be graded finally. The remaining reports will be graded with the first version.

- Grading the Lab: Every student has to conduct each experiment successfully. The results must be demonstrated during the lab. The preparation, the success of the experiment, the quality of the report and the presentation will be graded.

Chapter 2

Introduction to the Lab

2.1 Introduction

The lab computer technology is based on the correspondent lecture. So it is necessary to be familiar with the content of this lecture to do the lab. The lab consists of five tasks. In the labs you will learn to write small assembly routines for the ARM processor. The programs you will develop are a combination of an assembly- and a c-routine. For each exercise you will get a assembly program body which you have to complete. Even though the c-code is given, it is helpful to have a look at it to understand how the assembly program is called and which parameters are passed to it.

2.2 Using the Raspberry Pi at home

If you plan to do some work at home, you either need a Linux device with an ARM processor, or use an ARM emulator. The easiest way is to get yourself a Raspberry Pi minicomputer which costs approx. 35 - 40 EUR (See: http://www.raspberrypi.org). It is useful to have a monitor and keyboard + mouse for the Raspberry Pi although you can do without it.

2.3 Basic Linux Commands

Since your working environment will be Linux (Ubuntu) here is a short introduction of some basic command line commands:

- Getting help for a command (exit with q):
  - man <command>

  - info <command>

- Files:
  - *ls*: lists the files of the current working directory
  - *ls -l*: like ls but more detailed
  - *cp*: copy files
  - *mv*: move or rename files
  - *rm*: delete file(s)
  - *cat*: show the contens of a textfile
  - *more*: like cat but separated by pages – file: show the file type

- Directories:
  - *cd*: change the working directory
  - *pwd*: show the working directory
  - *mkdir*: create a directory
  - *rmdir*: delete a directory

- Run a program as a different user (e.g. with higher permission):
  - *su*: change the user account
  - *sudo*: execute as root

- Miscellaneous:
  - *touch*: update the time stamp of a file
  - *whoami*: show your user account name

2.4 How to program with the Linux operating system

All the programs of the lab are developed under the operating system Linux. The following development tools are used:

- Text editor
  - *geany*: This is the default editor.

- Assembler

- C Compiler, linker
  - *gcc*: GNU project c-compiler.
- Debugging with graphical user interface. With the help of a debugger, you can execute your program step by step, while tracking the values of variables and registers. This help to find error (bugs) in your programs.
  - *nemiver*
  - *ddd*

2.5 Basic File Structure

For each exercise there is a seperate folder. This is helpful to keep track of the needed files for each program. In the folders there are at least the following files:

- Makefile: The makefile is used for the compilation of the programcode. The make- file contains the instructions and parameters for thecompiler, assembler and linker to create the executable file. In the next chapter you will find a more detailed description of the makefile.
- .c-File: In this file you will find the c-code. The c-file contains the main program from which the assembly routine is called. All the input and output (reading input from the keyboard, printing results to the screen) is done by the c-file as well. You don't have to change anything in this file.
- .asm-File: In this file you will find the assembly program body. It is your task to add here the missing assembly code.

2.6 The Makefile

Make is a utility for automatically building executable programs from source code. The program *make* operates with a *Makefile*, which contains all the necessary translation steps and parameters.

So you only have to type *make* and the whole compilation process is done automatically.

In line 14 after the 'all' command, the file name of the executable to be generated is defined, here 'add'. This is the Makefile for the first task in which two values are being added.

Line 3 and 4 define which assembler (here as) will be used, and which flags will be passed to the assembler. Lines 5 and 6 define the C-compiler (here gcc)

and the compiler flags. Then from line 8 to 12 compiler and assembler are called which generate the two object files (.o) for the assembly part and the c part. After that these two object files are linked together at line 17 and the executable 'add' is generated.

If you want to execute your compiled code you have to type '*./add*' on the console. After compilation you can remove no longer used .o files by typing '*make clean*' (line 28 and 29).

If you want to start the debugger you have to type '*make debug*'. In the example below, the debugger '*ddd*' would be started (lines 31 and 32).

Listing 2.1: Makefile

```
.SUFFIXES: .o .asm .c

AS=as
ASFLAGS= -g -o

CFLAGS= -g
CC= gcc

.asm.o:
$(AS) $(ASFLAGS) $*.o $*.asm -al=$*.lst

.c.o:
$(CC) -c $(CFLAGS) $*.c

all: Add

Add : add.o add_asm.o
$(CC) $(CFLAGS) -o Add add.o add_asm.o

add_asm . o : add_asm.asm

add.o : add.c

clean :
rm *.o
rm *.lst
rm Add

debug :
gdb ./Add -x gdb_command
```

Chapter 3

Lab: STR/LDR

The scientific question is: Examine all possibilities, how in an ARM data can be stored and read to/from the main memory. **Which possibilities exists in ARMV8 to store and read variables from the main memory?** Figure 3 shows a hint from the ARM documentation. In the following (appendix), the load (ldr) and store (str) is copied from the ARMv8 ISA.

4.5 Load/Store Addressing Modes

Load/store addressing modes in the A64 instruction set broadly follows T32 consisting of a 64-bit `base` register (`Xn` or `SP`) plus an immediate or register offset.

Type	Immediate Offset	Register Offset	Extended Register Offset
Simple register (exclusive)	`[base{,#0}]`	n/a	n/a
Offset	`[base{,#imm}]`	`[base,Xm{,LSL #imm}]`	`[base,Wm,(S|U)XTW {#imm}]`
Pre-indexed	`[base,#imm]!`	n/a	n/a
Post-indexed	`[base],#imm`	n/a	n/a
PC-relative (literal) load	`label`	n/a	n/a

- An immediate offset is encoded in various ways, depending on the type of load/store instruction:

Bits	Sign	Scaling	Write-back?	Load/Store Type
0	-	-	-	exclusive / acquire / release
7	signed	scaled	option	register pair
9	signed	unscaled	option	single register
12	unsigned	scaled	no	single register

Figure 3.1: Load and store in ARM taken from: ARM ARMv8 Instruction Set Overview, PRD03-GENC-010197 15.0, 11. Nov. 2011

3.1 Simple register load and store

Write a program which demonstrates a simple register store and load. The load and store must be done in assembler. The first argument is a variable which must

be stored, the second variable is a pointer to a memory location which holds the re-loaded variable (see table 3.1). Display both numbers. Prove by means of the debugger step by step, what the content of the relevant registers is, that the values are at the intended data memory locations (make screen shots for your report).

Table 3.1: Parameters for store/load

Parameter	Description	Register
int int_val1	Variable to be storedd	X0
int *int_val2	Variable for the load	X1

Einige Debuggerbefehle / Some debugger instructions:

- l: list
- i r: list all registers
- i r X0: list register X0
- r: run
- s: step
- b strldr_asm: set breakpoint
- x 0x555557777: show content of address
- p variable: show variable
- p &variable: show address of the variable

Ein beispielhaftes Vorgehen / An example procedure:

1. make debug (starts the debugger; after make all)
2. b strldr_asm (sets a breakpoint to the begin of the assembler function)
3. r (runs the program)
4. do all necessary inputs
5. i r (list all registers; look at $X0$, it is the first argument of the function, what is it?)
6. s (step to the next instruction)
7. i r (look at $X6$, explain)

8. s
9. i r (look at $X7$, explain)
10. s
11. i r (look at $X7$, explain)
12. s
13. i r (look at $X9$, explain)
14. s
15. i r
16. x 0x7ffffff330 (see the address in the data memory (could differ, find your value), explain)
17. s
18. i r (look at $X8$, explain)
19. s
20. i r
21. x 0x7ffffff328 (watch the address of the second function argument (could differ, find your value), explain)
22. s
23. i r (look at $X0$, the function return, explain)
24. q (quit)

3.2 Pre-indexed load and store

Do the same for the pre-indexed store and load.

3.3 Offset load and store

Do the same for the offset store and load.

3.4 Offset and shift load and store

Do the same for the offset and shift store and load.

3.5 Post-indexed load and store

Do the same for the post-indexed store and load.

3.6 Results and Presentation

Please show the results and write a scientific documentation about the experiment.

The scientific report must show the usage of the debugger. Go step by step through the program and show the content of the involved registers and the flags. This is a must for all further tasks and experiments.

In the presentation explain the Code.

3.7 Preparation

- Write all codes.
- Explain the codes.

Chapter 4

Lab: Flags

The scientific question is: **Can a computer react on different number situations, e.g. positive numbers, negative numbers, overflow conditions, and how?**

4.1 Positive Number

Write a program which demonstrates a simple usage of the flags oft the ARM Cortex A53. Show the usage on positive numbers (for the first function argument, see table 4.1), negative numbers(for the first function argument) and on overflow (addition of both function arguments). Display both numbers. Prove by means of the debugger step by step, what the content of the relevant registers is, that the values are at the intended data memory locations (make screen shots for your report).

Table 4.1: Parameters for flags

Parameter	Description	Register
long long int_va0	Variable 1	X0
long long int_val1	Variable 2	X1
long long *int_val2	pointer to variable for the result	X2

Hints and procedure / Hinweise und Vorgehen:

- Get informations about the NZCV register (also look for the CPSR register). / Suchen Sie Informationen über das NZCV Register (auch über das CPSR Register).
- Add two positive numbers without overflow (the first number will be checked about the sign). The return status (by argument) is 42. / Addieren Sie zwei

positive Zahlen ohne Bereichsüberlauf (das Vorzeichen der ersten Zahl wird ermittelt). Der Rückgabewert ist 42 (als Funktionsargument).

- Add two negative numbers without overflow (the first number will be checked about the sign). The return status (by argument) is 43. / Addieren Sie zwei negative Zahlen ohne Bereichsüberlauf (das Vorzeichen der ersten Zahl wird ermittelt). Der Rückgabewert ist 43 (als Funktionsargument).

- Add two positive numbers with overflow (the first number will be checked about the sign). The return status (by argument) is 52. / Addieren Sie zwei positive Zahlen mit Bereichsüberlauf (das Vorzeichen der ersten Zahl wird ermittelt). Der Rückgabewert ist 52 (als Funktionsargument).

- Add two negative numbers with overflow (the first number will be checked about the sign). The return status (by argument) is 53. / Addieren Sie zwei negative Zahlen mit Bereichsüberlauf (das Vorzeichen der ersten Zahl wird ermittelt). Der Rückgabewert ist 53 (als Funktionsargument).

4.2 Results and Presentation

Please show the results and write a scientific documentation about the experiment.

The scientific report must show the usage of the debugger. Go step by step through the program and show the content of the involved registers and the flags. This is a must for all further tasks and experiments.

In the presentation explain the Code.

4.3 Preparation

- Write all codes.
- Explain the codes.

Chapter 5

Lab: Bit Masks

5.1 General

An often used programming style in embedded systems is bit manipulation. Single bits must be set, cleared, checked, etc. This is often the case for interfaces like General-Purpose-Input-Outputs (GPIO). Single LEDs must be switched on (set), switched off (clear). Switches must be read (check). Pulse-width-modulation (PWM) signals must be generated (toggle). The used C-operators are listed in table 5.1. A programming example can be seen in listing 5.1. A question could be: **Can bit manipulation be done using assembler?**

Table 5.1: C-Operators for Bit Manipulation

Operator	Name
&	AND
|	OR
∧	XOR
<<	LSL (logical shift left)
>>	LSR (logical shift right)
∼	NOT

Listing 5.1: masks.c

```
/*
mask.c  14.02.2024

#include <stdio.h>
#include <stdlib.h>
```

```
int int_val1, int_val2;
int result, status;

int main()
{
  ...
  // set bit with mask:
  // mask = 00000100
  // a = 2
  a |= mask; // = 6
  // a = a | mask;

  ...

  // set bit with shift mask:
  // shiftmask = 2 -> shift by 2 bit positions
  // a = 2
  a |= (1 << shiftmask); // = 6

  ...

  // clear bit with mask:
  a &= ~(mask);

  ...

  // clear bit with shift mask:
  a &= ~(1 << shiftmask);

  ...

  // toggle bit with mask:
  a ^= (mask);

    ...

  // toggle bit with shiftmask:
  a ^= (1 << shiftmask);

  ...
```

```
    // update bit:
    a &= ~(1 << shiftmask); // clear
    a |= (new_bit << shiftmask); // new value
       ...

    // check bit:
    a &= (1 << shiftmask);
    // 0101 & 0100 -> true
    // 0101 & 1000 -> false

    return 0 ;

}
```

5.2 Task

Write single programs, C and Assembler for:

- set bit with mask
- set bit with shift mask
- clear bit with mask
- clear bit with shift mask
- toggle bit with mask
- toggle bit with shift mask
- update bit
- check a bit

To make it complete, understand and write the needed Makefiles.

5.3 Results and Presentation

Please show the results and write a scientific documentation about the experiment.

The scientific report must show the usage of the debugger. Go step by step through the program and show the content of the involved registers and the flags. This is a must for all further tasks and experiments.

In the presentation explain the Code. Explain the Makefiles.

5.4 Preparation

- Write all codes.
- Explain the codes.

Chapter 6

Lab: Matrix

6.1 Matrix Addition

The aim of this exercise is to add two matrixes.

The two matrixes are created as a struct ('struct matrix') in a C-program. The structure contains the number of rows, the number of columns and the max number of columns of the matrix in addition to the matrix elements. Because of the definition of a structure, a function call can be realized with only a few parameters.

It is possible to define different matrix forms (rectangular or quadratic) up to a maximum size. In assembly a multidimensional array is stored as an unidimensional array. This means, that for example, a two dimensional array is splitted into each row. Then every row is stored one after the other. A pointer to such an array points always to the first element of the first row.

Task:

- Check the compatibility of the matrixes. The number of rows and the number of columns of each matrix have to be the same. Otherwise it is not possible to add the two matrixes.
- Addition of the matrix elements without handling of an overflow.
- Store the solution in the destination matrix.
- Test the whole program with the debugger.

Table 6.1: Parameters "matrix addition"

Parameter	Description	Register
Matrix_Sum	pointer on destination matrix structure	R2
Matrix_B	pointer on second matrix structure	R1
Matrix_A	pointer on first matrix structure	R0

6.2 Convert Integer to Binary String

The only input parameter here is a 32 bit integer. Your task is to build a string with a length of 32 bytes with the ascii characters '0' and '1' which contains the binary representation of that integer. The ASCII value of a '1' ist '31hex' and of a '0' ist '30hex' (see ascii table). Example: the input parameter '65001' decimal gives '00000000000000001111110111101001' for the output string.

Please note, that you also have to include the call of the assembly function in the c-program yourself.

Table 6.2: Parameters "convbin"

Parameter	Description	Register
value_a	value	R0
result	pointer to string	R1

6.3 Results and Presentation

Please show the results and write a scientific documentation about the experiment. In the presentation explain the Code.

6.4 Preparation

- Write all codes.
- Explain the codes.

Bibliography

[1] A. Siggelkow, “Computer technology,” RWU, Study Script, 2024.

Chapter 7

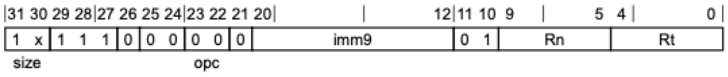
Appendix

7.1 load and store

C6.2.322 STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes* on page C1-234.

Post-index



32-bit variant

Applies when size == 10.

```
STR <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

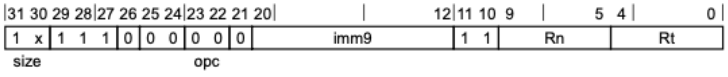
Applies when size == 11.

```
STR <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



32-bit variant

Applies when size == 10.

```
STR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

Applies when size == 11.

```
STR <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

32-bit variant

Applies when size == 10.

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when size == 11.

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

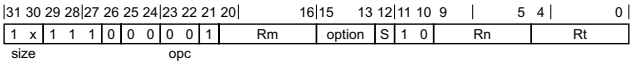
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.275 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes* on page C1-189.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit variant

Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#2` when `S = 1`

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.166 LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes* on page C1-234. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

Post-index

32-bit variant

Applies when size == 10.

```
LDR <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when size == 11.

```
LDR <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

32-bit variant

Applies when size == 10.

```
LDR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

Applies when size == 11.

```
LDR <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

32-bit variant

Applies when `size == 10`.

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDR (immediate)* on page K1-11593.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
```

```
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.167 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/store addressing modes* on page C1-234.

32-bit variant

Applies when `opc == 00`.

```
LDR <Wt>, <label>
```

64-bit variant

Applies when `opc == 01`.

```
LDR <Xt>, <label>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<label>`	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTE2Ext() then
```

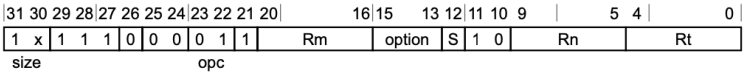
```
case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t, 64] = SignExtend(data, 64);
        else
            X[t, size*8] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.168 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/store addressing modes* on page C1-234.



32-bit variant

Applies when `size == 10`.

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit variant

Applies when `size == 11`.

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#2` when `S = 1`

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

#0	when S = 0
#3	when S = 1

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.