ECSE324 : Computer Organization

Instruction Set Architecture

Christophe Dubach
Fall 2020

Original slides from Prof. Warren Gross – 2017.
Updated by Christophe Dubach – 2020.
Some material from Hamacher, Vranesic, Zaky, and Manjikian, *Computer Organization and Embedded Systems, 6 th ed*, 2012, McGraw Hill and Patterson and Hennessy, *Computer Organization and Design, ARM Edition*, Morgan Kaufmann, 2017, and notes by A. Moshovos

Timestamp: 2020/09/20 15:28:08

Disclaimer

Lectures are recorded live and will be posted unedited on *mycourses* on the same day.

It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the book, the course webpage or ask on Piazza for clarifications.

Introduction

Instruction Set Architecture

- Each processor has a predefined set of instructions that it understands called the *instruction set*
- The instruction set, along with the information about how the memory is organized, how to access memory, etc,... is called the programming model or instruction set architecture (ISA).
- The ISA forms a contract between the machine and the programmer).
- There are a relatively small number of ISAs (e.g. x86-64, ARMv7-A, Power ISA 3.0, RISC-V), but many processor implementations that conform to each ISA.

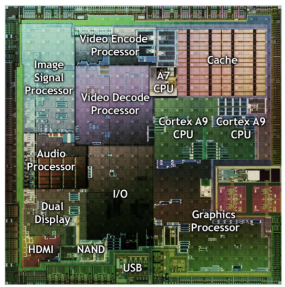
Different implementations of an ISA

- The ISA tells you what the processor does. The implementation is how it does it.
- The ISA is the interface between the hardware and software.

Machine language software (assembly) is portable between two processors if they implement the same ISA .

The ARM Architecture

- A family of RISC processors used in many devices, especially smartphones and tablets
- There have been 150 billion ARM processors shipped as of 2019 (∼15 billion per year in 2015/2016)
- ARM provides the processor design to chip manufacturers, who fabricate it in their own products:
  - *e.g.* Apple A5 chip has a dual-core ARM Cortex-A9 processor
  - *e.g.* Nvidia Tegra 2 SoC also has the same ARM processor



Nvidia Tegra 2 SoC

source: www.anadtech.com

ARM ISA

ARM has developed several ISAs, and many flavors of implementations based on each ISA

- ARMv7-A is the ISA for the ARM Cortex-A9 processors in Apple A5 (iPhone 5) and the Altera Cyclone V SoC (*the one from the labs!*)

DE1-SoC Altera Cyclone V

There are other implementations of the ARMv7-A ISA that have different characteristics: speed, power, cost, etc, ...

In the lab you will program an ARM Cortex-A9 processor implementing the ARMv7-A ISA.

- The "Introduction to the ARM Processor Using Altera Toolchain" document contains most of what you need for this course.
- Appendix D of the textbook describes ARMv4, which is very similar, and should be adequate for this course. Some of the terminology is slightly different and I will use the correct terms in the lecture slides.
- The *complete* ISA is described in the ARMv7-AR Architecture Reference Manual.
  - The interesting part for us are : A1–A4.

From now on, I will just refer to "ARM ISA" or "ARM assembly language".

ARM ISA

Textbook§D.1,D.2

ARM ISA basics

The word word length is 32-bits and processor registers are 32-bits.

The ISA is (mostly) RISC:

- All instructions are 32-bits long.
- Only load and store instructions access memory.
- All arithmetic and logic instructions operate on registers.
- There are some features which normally are seen in CISC ISAs.

ARM ISA Memory

- The memory is byte-addressable using 32bit addresses.
- Memory is *litte-endian*.
- Memory accesses are word-aligned.
- Word, half-word, and byte data transfers to and from processor registers are supported.

ARM programmer-visible registers

Sixteen 32-bit processor registers labelled R0 through R15

- R15 is the program counter (PC)
- R14 is the link register (LR)
- R13 is the stack pointer (SP)

In practice, use only R0...R12 as GPRs (General Purpose Registers) and only use and refer to R13, R14, and R15 as `SP`, `LR`, and `PC`.

In addition, there is a special status register called the CPSR (Current Program Status Register) that indicates various useful information (we will come back to this later).

ARM ISA

Textbook§D.1,D.2

Syntax

Assembly language syntax

Assembly language consists of shorthand instruction names called mnemonics, and a syntax for using them.

A program called an assembler translates the mnemonics into machine language instructions (we will see this in a future lecture).

Here is a (short) ARM assembly program:

```
ADD  R1, R2, R3    // R1 <- R2 + R3
```

- `ADD` is a mnemonic
- `R1` is a destination register; the first operand
- `R2` and `R3` are source registers; the second and third operand
- `// R1 <- R2 + R3` is a comment (not a very useful one)

There are different ways to use each instruction.

```
ADD  R1, R2, R3   // R1 <- R2 + R3
```

Here, the syntax of the instruction is `ADD Rd, Rn, Rm` where

- `Rd` specifies the destination register
- `Rn` and `Rm` specify the source registers

```
ADD  R4, R5, #24   // R4 <- R5 + 24
```

Here, the syntax of the instruction is `ADD Rd, Rn, Imm` where

- `Rd` specifies the destination register
- `Rn` specifies the source register
- `Imm` specifies an immediate value (constant)

Operands type

We will use the following convention when specifying the instructions:

- `Rd` always refers to a destination register which is written to.
- `Rn` or `Rm` refers to source registers; their value do not change (unless the register is the same as `Rd`).
- `Imm` refers to an immediate value (the maximum number of bits might be specified, *e.g.* `Imm16` for a 16-bits value).
- `Op2` refers to a flexible source operand, which is either:
  - an 8-bit immediate value (with optional rotation)
  - a register (with optional shift)

ARM ISA

Textbook§D.1,D.2

General Data Processing Instructions

Move instructions

These instructions copy data into registers.

```
MOV  Rd, Op2       // MOVes value of Op2 into Rd
MOV  Rd, #Imm16    // MOVes immediate 16-bit value into Rd

MVN  Rd, Op2       // MoVes complement (Not) of Op2 value
                   // into Rd

MOVT Rd, #Imm16    // MOVes Top: moves a 16-bit constant into
                   // the high-order 16 bits of Rd and leaves
                   // the lower bits unchanged
```

Why is the last instruction useful?

Logic instructions

```
AND  Rd, Rn, Op2  // bitwise AND operation
ORR  Rd, Rn, Op2  // bitwise OR operation
EOR  Rd, Rn, Op2  // bitwise Exclusive OR (xor) operation
BIC  Rd, Rn, Op2  // BIt Clear: Rd <-- Rn & NOT(Op)
```

Shift Instructions

```
LSL  R1, R2, #5   // Logical shift left
LSR  R1, R2, R3   // Logical shift right
ASR  R1, R2, #4   // Arithmetic shift right
```

Note: Last operand can be a register or an immediate value.

- Logical ⇒ pad with 0, Arithmetic ⇒ extend sign bit

Shift Instructions

```
LSL   R1, R2, #5   // Logical shift left
LSR   R1, R2, R3   // Logical shift right
ASR   R1, R2, #4   // Arithmetic shift right
```

Note: Last operand can be a register or an immediate value.

- Logical ⇒ pad with 0, Arithmetic ⇒ extend sign bit

Logical shift left by 2 of 0000 0011 =

Logical shift right by 1 of 0000 0011 =

Logical shift right by 3 of 1111 0000 =

Arithmetic shift right by 3 of 1111 0000 =

Observation

Shifting left by k = multiplication by 2^k
Arithmetic shifting right by k = division by 2^k

Rotate Instruction

Rotate instruction: ROR

```
ROR  R1, R2, #2  // Circular rotate right
```

Note: Last operand can be a register or an immediate value.

Arithmetic Instructions

Addition/subtraction instructions

```
ADD   R0, R1, R2          // R0 <-- R1 + R2
ADD   R0, R1, #-24        // R0 <-- R1 + (-24)
SUB   R0, R1, #24         // R0 <-- R1 - (24)
ADD   R0, R1, R2, LSL#2   // R0 <-- R1 + R2*4
```

Arithmetic Instructions

Addition/subtraction instructions

```
ADD   R0, R1, R2          // R0 <-- R1 + R2
ADD   R0, R1, #-24        // R0 <-- R1 + (-24)
SUB   R0, R1, #24         // R0 <-- R1 - (24)
ADD   R0, R1, R2, LSL#2   // R0 <-- R1 + R2*4
```

Multiply instruction

```
MUL   R2, R3, R4          // R2 <-- R3 * R4
```

Multiply-accumulate instruction

```
MLA   R2, R3, R4, R5      // R2 <-- (R3 * R4) + R5
```

Both multiply instruction only returns the 32 least significant bits!

Addressing modes (Textbook§2.4, D3)

The different ways an instruction can specify its operands are called addressing modes. For instance:

```
ADD  R0, R1, R2
```

uses register mode for all of its operands.

```
ADD  R0, R1, #24
```

uses register mode for the destination and first source operand, and immediate mode (#24) for the other source operand.

```
ADD  R0, R1, R2, LSL#2
```

uses scaled register mode for its last operand R2.

ARM ISA

Textbook§D.1,D.2

Memory Instructions

Arrays in C (recap)

```
short arr[5] = {1, 2, 3, 4 ,5}
```

Implemented as elements one after the other in memory (watchout for Endianess!)

For a 1D array, `arr[i]` is at address: `&arr[0]+sizeof(TYPE)*i` where

- & means *address of*
- `&arr[0]` is the address of the first array element, which is also the start address of the array, written simply as `arr`.

Address	Content
	…
0x1000	0x01
0x1001	0x00
0x1002	0x02
0x1003	0x00
0x1004	0x03
0x1005	0x00
0x1006	0x04
0x1007	0x00
0x1008	0x05
0x1009	0x00
	…

Byte view

Address	Content
	…
0x1000	1
0x1002	2
0x1004	3
0x1006	4
0x1008	5
	…

Half-word view

A first example: Load Instruction

```
LDR  Rd, [Rn]      // Rd <-- Mem[Rn], Rn = address in byte
```

A first example: Load Instruction

```
LDR  Rd, [Rn]     // Rd <-- Mem[Rn], Rn = address in byte
```

C code:

```
int array[8]; // sizeof(int) = 4 byte
...
array[i];
```

A first example: Load Instruction

```
LDR  Rd, [Rn]     // Rd <-- Mem[Rn], Rn = address in byte
```

C code:

```
int array[8]; // sizeof(int) = 4 byte
...
array[i];
```

Assembly program:

```
// R0 = variable i, R1 = base address of array
MUL  R2, R0, #4 // R2 = i*4
ADD  R3, R1, R2 // R3 = array + i*4
LDR  R4, [R3]   // R4 = array[i]
```

A first example: Load Instruction

```
LDR  Rd, [Rn]     // Rd <-- Mem[Rn], Rn = address in byte
```

C code:

```
int array[8]; // sizeof(int) = 4 byte
...
array[i];
```

Assembly program:

```
// R0 = variable i, R1 = base address of array
MUL  R2, R0, #4 // R2 = i*4
ADD  R3, R1, R2 // R3 = array + i*4
LDR  R4, [R3]   // R4 = array[i]
```

When accesing an array, we need to multiply the index by the element size. This is a very common case: the actual address we are interested to access is composed of a base address and an offset (`i*4`).

```
int array[8] = {17, 58, 79, 15, ...}
...
array[i];
```

```
// R0 = variable i, R1 = base address of array
MUL  R2, R0, #4 // R2 = i*4
ADD  R3, R1, R2 // R3 = base address of array + i*4
LDR  R4, [R3]   // R4 = array[i]
```

Address	Content
	...
0x0000	`MUL R2,R0,#4`
0x0004	`ADD R3,R1,R2`
0x0008	`LDR R4,[R3]`
	...
0x1000	17
0x1004	58
0x1008	79
0x100C	15
	...

Example for array base address =`0x1000` and `i=3` after execution of the load:

Registers

R0	0x00000003
R1	0x00001000
R2	0x0000000C
R3	0x0000100C
R4	0x0000000F

Load/Store instructions

The most common ARM load/store instructions for 32-bit words have the following form:

```
LDR  Rd, EA  // Rd <-- Mem[EA]
STR  Rm, EA  // Mem[EA] <-- Rn
```

Loads/stores do not specify a memory address explicitly, rather they generally compute an effective address (EA) from a base address and an offset.

Effective Address Calculation
$EA = base + offset$

Calculating an EA is very convenient for implementing common program structures such as loops and data structures such as arrays as just seen.

Offset (addressing) mode

- The base address is always stored in a register (`Rn`).
- There are three kinds of offset:
  - Immediate: a 12-bit number that can badded or subtracted from the base register value
  - Index: the offset is stored in a register (`Rm`).
  - Scaled index: the value in the index register is shifted by a specified immediate value, then added to or subtracted from the base register.

Effective address:

Name	Assembler syntax	Address generation
register indirect	`[Rn]`	EA = `Rn`
immediate offset	`[Rn, #offset]`	EA = `Rn` + offset
offset in `Rm`	`[Rn, ± Rm, shift]`	EA = `Rn` ± shifted(`Rm`)

Coming back to our example

C code:

```
int array[8];
...
array[i];
```

Immediate (with #0): $EA = R3$

```
// R0 = variable i, R1 = base address of array
MUL  R2, R0, #4        // R2 = i*4
ADD  R3, R1, R2        // R3 = array + i*4
LDR  R4, [R3,0]        // R4 = array[i]
```

Coming back to our example

C code:

```
int array[8];
...
array[i];
```

Immediate (with #0): $EA = R3$

```
// R0 = variable i, R1 = base address of array
MUL  R2, R0, #4         // R2 = i*4
ADD  R3, R1, R2         // R3 = array + i*4
LDR  R4, [R3,0]         // R4 = array[i]
```

Index: $EA = R1 + R2$

```
MUL  R2, R0, #4         // R2 = i*4
LDR  R4, [R1,R2]        // R4 = array[i]
```

Coming back to our example

C code:

```
int array[8];
...
array[i];
```

Immediate (with #0): $EA = R3$

```
// R0 = variable i, R1 = base address of array
MUL  R2, R0, #4          // R2 = i*4
ADD  R3, R1, R2          // R3 = array + i*4
LDR  R4, [R3,0]          // R4 = array[i]
```

Index: $EA = R1 + R2$

```
MUL  R2, R0, #4          // R2 = i*4
LDR  R4, [R1,R2]         // R4 = array[i]
```

Scaled Index: $EA = R1 + (R0 << 2) = R1 + (R0 \times 4)$

```
LDR  R4, [R1,R0,LSL#2] // R4 = array[i]
```

Same for the store instruction

C code:

```
int array[8];
...
array[i] = 44;
```

Scaled Index: $EA = R1 + (R0 << 2) = R1 + (R0 \times 4)$

```
// R0 = variable i, R1 = base address of array
MOV  R2, #44           // R2 = 44
STR  R2, [R1,R0,LSL#2] // array[i] = R2
```

Checkpoint

For each instruction below, calculate the EA (Effective Address) given the following register content:

R2 = 0x1A4DDA38

R6 = 0x10004008

R8 = 0x10004000

R10 = 0x00000002

```
LDR R2, [R6, #-12]

LDR R2, [R6, #0x200]

STR R2, [R6, -R8]

STR R2, [R8]

LDR R2, [R8, R10, LSL#3]
```

Pointers in C (recap)

- A pointer (`int* ptr;`) is an address
- You can perform pointer arithmetic (`ptr+2`)
  - Including pre-increment (`++ptr`) and post-increment (`ptr++`)
- You can dereference a pointer (`*ptr`), *i.e.* access the data contained in memory location pointed by the pointer.

In C, you declare that a variable is a pointer with `*`

```
int *p;     // p is a pointer to an integer
            // i.e. the memory address of a 32-bit variable
            // since p contains an address, it is also 32-bits
int x;
int a[5] = {20,35,0,42,12};

p = &a[3]; // the address of the 4th element of a is stored in p

x = *p;     // here, * means indirection (the value addressed by p)
            // tricky, C uses * to mean different things !
```

What is the value stored in `x`?

C code

```
x = *p;
```

Assembly equivalent:

```
LDR R0, p
LDR R1, [R0]
STR R1, x
```

Why is it important to know the pointer type?

```
int *p;
```

Because we can do arithmetic on the pointer:

```
p = 0x1000;
```

What is p+1?

```
int arr[8] = {56,26,88,45,-45,77,98,13};
print(arr);
print(&arr[1]);

int* ptr = &arr[1];
print(ptr);
print(*ptr);

print(ptr+2);
print(*(ptr+2));

print(ptr++);
print(ptr);

print(++prt);
print(ptr);

print(*(ptr++));
print(*(++ptr));
```

Address	**Content**
	...
0x1000	56
0x1004	26
0x1008	88
0x100C	45
0x1010	-45
0x1014	77
0x1018	98
0x101C	13
	...

Assuming `arr` starts at address `0x1000`, what is printed by this C code?

Pointers into assembly

C code (without pointer):

```
int arr[8] = ...;
for (int i=0; i<8; i++) {
  v = arr[i];
  ...
}
```

loop body in assembly:

```
// R0 = i
// R1 = base address of arr
// R2 = v
LDR R2,[R1,R0,LSL#2] //v=arr[i]
ADD R0,R0,#1         //i++
```

Pointers into assembly

C code (without pointer):

```
int arr[8] = ...;
for (int i=0; i<8; i++) {
  v = arr[i];
  ...
}
```

loop body in assembly:

```
// R0 = i
// R1 = base address of arr
// R2 = v
LDR R2,[R1,R0,LSL#2] //v=arr[i]
ADD R0,R0,#1         //i++
```

Pointer equivalent C code:

```
int arr[8] = ...;
int* ptr = arr;
while (ptr<(arr+8)) {
  v = *(ptr++);
  ...
}
```

Pointers into assembly

C code (without pointer):

```
int arr[8] = ...;
for (int i=0; i<8; i++) {
  v = arr[i];
  ...
}
```

loop body in assembly:

```
// R0 = i
// R1 = base address of arr
// R2 = v
LDR R2,[R1,R0,LSL#2] //v=arr[i]
ADD R0,R0,#1         //i++
```

Pointer equivalent C code:

```
int arr[8] = ...;
int* ptr = arr;
while (ptr<(arr+8)) {
  v = *(ptr++);
  ...
}
```

loop body in assembly:

```
// R0 = ptr
// R1 = v
LDR R1, [R0]    // v = *ptr
ADD R0, R0, #4  // ptr=ptr+4
```

Using a pointer instead `arr[i]` uses one less register in assembly!
A good compiler will do this transformation to pointer-based code automatically for you.

Post/Pre-indexed Addressing Mode

Offset addressing mode (register indirect)

```
// R0 = ptr
// R1 = v
LDR R1, [R0]     // v = *ptr
ADD R0, R0, #4   // ptr=ptr+4
```

Post-indexed addressing mode (immediate offset)

```
LDR R1,[R0],#4 // v = *(ptr++)
```

Pre-indexed addressing mode (immediate offset)

```
LDR R1,[R0,#4]! // v = *(++ptr)
```

We can use a single instruction to perform both the read and the increment of the pointer! Very useful in the presence of loops!

Post-indexed addressing mode (immediate offset)

```
LDR R1,[R0],#4 // v = *(ptr++)
```

Pre-indexed addressing mode (immediate offset)

```
LDR R1,[R0,#4]! // v = *(++ptr)
```

Assuming R0=0x1008 before the `LDR` instruction executes, what's the content of R0 and R1 after the instruction executes?

Address	**Content**
	…
0x1004	26
0x1008	88
0x100C	45
	…

Load/Store Addressing Mode Summary (Textbook§2.4, D3)

Name	Assembler Syntax	Address generation
Register indirect:	`[Rn]`	Address = `Rn`
Offset:		
immediate offset	`[Rn,#offset]`	Address = `Rn` + offset
offset in `Rm`	`[Rn,±Rm,shift]`	Address = `Rn` ± shifted(`Rm`)
Pre-indexed:		
immediate offset	`[Rn,#offset]!`	Address = `Rn` + offset `Rn` ← Address
offset in `Rm`	`[Rn,±Rm,shift]!`	Address = `Rn` ± shifted(`Rm`) `Rn` ← Address
Post-indexed:		
immediate offset	`[Rn],#offset`	Address = `Rn` `Rn` ← `Rn` + offset
offset in `Rm`	`[Rn],±Rm,shift`	Address = `Rn` `Rn` ← `Rn` ± shifted(`Rm`)

- offset = a signed number (∼13-bit)
- shift = `direction # integer`
  where `direction` is `LSL` for left shift or `LSR` for right shift,
  and `integer` is a 5-bit unsigned number specifying the shift amount

Loading/Storing half-word/byte

Dedicated instructions to load/store values smaller than a word:

`LDRB` (Load Register Byte) – zero padded to 32 bits
`LDRH` (Load Register Halfword) – zero padded to 32 bits

`LDRSB` (Load Register Signed Byte) – sign extended to 32 bits
`LDRSH` (Load Register Signed Halfword) – sign extended to 32 bits

`STRB` (Store Register Byte) – stores low byte of Rd
`STRH` (Store Register Halfword) – Store the low halfword of Rd

Loading/Storing multiple words

The `LDM` and `STM` instructions load and store blocks of words in consecutive memory addresses into multiple registers.

Registers are always stored by `STM` in order from largest-to-smallest index (`R15..R0`) and by `LDM` in order from smallest to largest index (`R0..R15`)

To determine the direction in which memory addresses are computed, you must use one of the following suffixes for the mnemonic to determine how to update the address:

- IA – Increment After the transfer
- IB – Increment Before the transfer
- DA – Decrement After the transfer
- DB – Decrement Before the transfer

Example:

```
LDMIA R3!, {R4, R6–R8, R10}
```

```
R4 ← Mem[R3]
R6 ← Mem[R3 + 4]
R7 ← Mem[R3 + 8]
R8 ← Mem[R3 + 12]
R10 ← Mem[R3 + 16]
R3 ← R3 + 20 // increment after
```

PC-relative addressing

- The `PC` can be used as the base register to access memory locations in terms of their distance relative to `PC+8`.
  - The processor updates PC ← PC+4, and then fetches the next instruction at that address, which starts executing before the current instruction is finished, so it also increments it's PC by 4.
  - This is called pipelining (covered later).
- PC-relative addressing when accessing variable declared statically.

Address	**Content**
	...
0x0FF0	96
0x0FF4	-8
0x0FF8	78
0x0FFC	26
0x1000	`LDR R0, [PC, #-16]`
	...

What's the content of `R0` after executing this instruction?

```
LDR R0, [PC, #-16]
```

ARM ISA

Textbook§D.1,D.2

Data/Text section

Assembler directives

We are almost ready to write out first assembly language program.

The assembler also accepts commands about how it should assemble your program – these are not machine instructions and are never translated to executable machine language.

Some common ones (see the Altera documentation for more):

```
.global symbol    // makes symbol visible outside object file
.word expressions // reserves space for words in memory
.text             // marks the beginning of the code
.end              // marks the end of the code
```

- Text section = where code goes
- Data section = where data goes (everything except code)

Loading 32-bit constants into register

The assembler uses the pseudo-instruction:

```
LDR Rd, =value  // pseudo-instruction
```

to load a 32-bit value into register `Rd`.

- If the value fits within the range allowed in a `MOV` instruction, the assembler will produce a `MOV` instruction.
- Otherwise, the assembler places the constant `value` into a literal pool in memory, in the data section, where it can be read at runtime:

  ```
  LDR Rd, [PC, #offset]
  ```

  where `Mem[PC + offset]` = value

Example of 32-bit constants (and our first programs!)

Loading a small constant:

```
.global _start
.text
_start: LDR R0, =0x00000020
.end
```

address	content	code
0x00000000	0xE3A00020	MOV R0, #32

Loading a large constant:

```
.global _start
.text
_start: LDR R0, =0xF0F0F0F0
.end
```

address	content	code
0x00000000	0xE51F0004	LDR R0, [PC, #-4]
0x00000004	0xF0F0F0F0	.word 0xF0F0F0F0

Declaring variable (with initialization) = label (= address):

```
.global _start
n: .word 7
_start:
  LDR R0, n
  LDR R1, =n
```

address	**content**	**code**
0x00000000	0x00000007	.word 7
0x00000004	0xE54F000c	LDR R0, [PC, #-12]
0x00000008	0xE54F1004	LDR R1, [PC, #-4]
0x0000000C	0x00000000	.word 0x00000000

- `LDR R0,n` is real instruction where `n = PC-12`
- `LDR R1,=n` is pseudo-instruction

After execution:

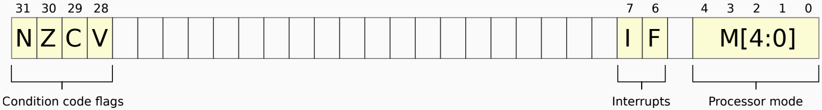
- R0 = 0x00000007
- R1 = 0x00000000

ARM ISA

Textbook§D.1,D.2

CPSR & Branching

Current Program Status Register (CPSR)



- Condition code flags (bit sets to 1 when condition is true)
  - N = Negative, Z = Zero, C = Carry, V = Overflow
- Interrupt flags
  - I = IRQ mask bit, F = FRQ (Fast interrupt) mask bit
- Processor mode
  - 10000 = User (most of user code)
  - 10001 = Serving fast interrupt (when dealing with I/O)
  - 10010 = Serving normal interrupt (when dealing with I/O)
  - 10011 = Supervisor (used by the Operating System)

No direct control over the CPSR

Some instructions will modify the CPSR as a side-effect, while others will behave differently depending the CPSR content.

Test & Compare instructions

```
TST  Rs, Op2
```

Zero flag (Z) in the condition code flags set by result of `AND(Rs, Op2)`

```
TEQ  Rs, Op2
```

Zero flag (Z) set by result of `XOR(Rs, Op2)`

```
CMP  Rs, Op2
```

Sets condition code flags by result of `Rs - Op2` (`Rs` unchanged)

```
CMN  Rs, Op2
```

Sets condition code flags by result of `Rs + Op2` (`Rs` unchanged)

All these instructions are useful in conjunction with branch instructions.

Branch instructions

```
B{cond} LABEL
```

- The condition `cond` specifies a test of the condition code bit.
- If the condition is true, the next instruction executed will be at address `LABEL`, the target
- If the conditon is false, the processor simply executes the next instruction (fallthrough).

Condition code

Suffix	Meaning	CSPR Flags
EQ	EQual(zero)	Z=1
NE	Not Equal (nonzero)	Z=0
CS/HS	Carry Set/ unsigned Higher or Same	C=1
CC/LO	Carry Clear / 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 AND 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 AND (N=V)
LE	signed Less or Equal	Z=1 OR (N!=V)
AL/	ALways (usually ommitted)	any
	not used	

Example

Corresponding ARM assembly code:

C code:

```
if (a>3)
  b = 7;
else
  b = 13;
```

```
       LDR R0, a
       CMP R0, #3   // R0-#3, only update CPSR
       BLE ELSE     // if R0-#3<=0 then branch
       MOV R1, #7
       B   END      // branch to END
ELSE:  MOV R1, #13
END:   STR R1, b
```

Show the content of each register after each instruction (including the CPSR), assuming:

1) a = 6,

2) a = 3, and

3) a = 2

Setting conditions codes

Test and compare instructions always set the condition codes in the `CPSR`, but so do other instructions

Data processing instructions (arithmetic, logic, move) affect the condition codes if the suffix S is appended to the mnemonic.

Example:

```
ADDS  R0, R1, R2   // sets condition codes
ADD   R0, R1, R2   // does not
```

Note that the following two instructions are equivalent:

```
SUBS  R0, R1, R2
CMP   R1, R2
```

Unless the results of the subtraction is required, `CMP` is preferred since one less register is used.

Conditional execution

Most ARM instructions can be executed conditionally

If the condition is true, then the instruction executes, otherwise the instruction has no effect

This can save some branches, resulting in compact and fast code.

Instruction format: `OP{S}{cond} Rd, Rn, Op2`

```
if (a>3)
  b = 7;
else
  b = 13;
```

```
LDR   R0, a
CMP   R0, #3  //
MOVGT R1, #7  // if R0 >  3
MOVLE R1, #13 // if R0 <= 3
STR   R1, b
```

This is a pretty advanced and somewhat ARM-specific technique. Recommend thinking in terms of branches to keep things simple.

ARM ISA

Textbook§D.1,D.2

Putting it all together: calculating dot product in assembly

Dot product

The dot product of two vectors A and B is defined as:

$$\sum_{i=0}^{n-1} A(i) \cdot B(i)$$

Corresponding C program for two vector of six integers:

```c
void main() {
  int n = 6;
  int vectorA[6] = {5, 3, -6, 19, 8, 12};
  int vectorB[6] = {2, 14, -3, 2, -5, 36};
  int dotP;
  int i;

  dotP = 0;
  for (i = 0; i<n; i++)
    dotP += vectorA[i] * vectorB[i];

  printf("Dot product = %d\n", dotP);
}
```

C variable declarations:

```
int n = 6;
int vectorA[6] = {5, 3, -6, 19, 8, 12};
int vectorB[6] = {2, 14, -3, 2, -5, 36};
int dotP;
int i;
```

Assembly memory allocation:

```
n:        .word 6
vectorA:  .word 5,3,-6,19,8,12
vectorB:  .word 2,14,-3,2,-5,36
dotP:     .space 4
// i will be stored in a register, no memory allocation needed
```

- `.word a b c ...`
  allocate storage for 1 or more words (4 byte each) and initialize with the values a,b, c, ...
- `.space 4`
  allocate 4 bytes without initialization
- `n`, `vectorA`, ... are addresses corresponding to the start of the allocated space

Loop:

```
dotP = 0;
for (i = 0; i<n; i++)
  dotP += vectorA[i] * vectorB[i];
```

```
 MOV R3, #0        // register R3 will accumulate the product

 LDR R0, =vectorA  // R0 = vectorA start address (pseudo-instruction)
 LDR R1, =vectorB  // R1 = vectorB start address (pseudo-instruction)
 LDR R2, n         // R2 is content of memory at address n (R2=6)

 MOV R6, #0        // iteration variable i

LOOP:
 CMP R6, R2         // i-n
 BGE END            // i>=n ?
 LDR R4, [R0], #4   // post-index mode
 LDR R5, [R1], #4   // post-index mode
 MLA R3,R4,R5,R3    // R3 = (R4*R5)+R3
 ADD R6,R6,#1       // i++
 B   LOOP

END:
 STR R3, dotP
```

Alternative approach using `SUBS`:

```
dotP = 0;
i = n;
do {
  dotP += vectorA[i] * vectorB[i];
  i--;
} while (i>0) // assumes there is at least one element in each array
```

```
MOV R3, #0        // register R3 will accumulate the product

 LDR R0, =vectorA // R0 = vectorA start address (pseudo-instruction)
 LDR R1, =vectorB // R1 = vectorB start address (pseudo-instruction)
 LDR R2, n        // R2=6 (R2 is out loop iteration variable i)

LOOP:
 LDR  R4, [R0], #4  // post-index mode
 LDR  R5, [R1], #4  // post-index mode
 MLA  R3,R4,R5,R3   // R3 = (R4*R5)+R3
 SUBS R2,R2,#1      // decrement counter and set condition flags
 BGT  LOOP          // i>0 ?

 STR R3, dotP
```

- One less register used
- 5 vs 7 instructions in the loop body

Last bit, printing the result:

```
printf("Dot product = %d\n", dotP);
```

Use a call to a sub-routine to print the results. This usually requires an operating system to print information on a terminal, or direct access an I/O device in assembly (*e.g.* a screen). We will see that in another lecture.

Full dot product code in ARM assembly

```
.global _start // tells the assembler/linker where to start execution

n:        .word 6
vectorA:  .word 5,3,-6,19,8,12
vectorB:  .word 2,14,-3,2,-5,36
dotP:     .space 4

_start:
 MOV R3, #0        // register R3 will accumulate the product
 LDR R0, =vectorA  // R0 = vectorA start address (pseudo-instruction)
 LDR R1, =vectorB  // R1 = vectorB start address (pseudo-instruction)
 LDR R2, n         // R2=6 (R2 is out loop iteration variable i)

LOOP:
 LDR   R4, [R0], #4  // post-index mode
 LDR   R5, [R1], #4  // post-index mode
 MLA   R3,R4,R5,R3   // R3 = (R4*R5)+R3
 SUBS  R2,R2,#1      // decrement counter and set condition flags
 BGT   LOOP          // i>0 ?

 STR R3, dotP

STOP:
 B     STOP // infinite loop whe done
```

ARM ISA

Textbook§D.1,D.2

Subroutine calls

Textbook§2.6,2.7,D.4.8

Subroutines

It is usual programming practice to reuse blocks of code in a subroutine (*i.e.* procedure, function, method) that can be called from many places in a program.

```
int add3(int a, int b, int c) {
  return a + b + c;
}

void main() {
  int sum = 0;

  sum += add3(1, 2, 3);
  sum += 10;
  sum += add3(10, 20, 30);

  printf("Sum = %d\n", sum);
}
```

- We should be able to call a subroutine from anywhere in our program, i.e. change the PC so that the routine is executed.
- A subroutine must be able to return from subroutine, i.e. change the PC so that execution continues immediately after the point where it was called.
- We should be able to pass parameters (or arguments) that may take different values across different calls.
- A subroutine must be able to return a value.

```
int add3(int a,int b,int c)
{
  return a + b + c;
}

void main() {
  int sum = 0;

  sum += add3(1, 2, 3);
  sum += 10;
  sum += add3(10, 20, 30);

  printf(”Sum = %d\n”, sum);
}
```

Calling and returning

A subroutine call is implemented with the Branch and Link instruction `BL` that stores the address of the next instruction (return address) in the link register `LR` (`R14`).

```
BL addr  // LR <- PC +4; PC <- addr
```

To return from subroutine, branch to the address stored in the link register with BX instruction (branches to the address stored in a register).

```
BX Rn  // Pc <- Rn
```

C code:

```
boo() {
  coo();
   ...
}
coo() {
   ...
  return;
}
```

ARM assembly:

```
boo:  BL coo  // LR <- PC +4; PC <- coo
      ...
coo:  ...
      BX LR  // PC <- LR
```

Multiple nested calls

```
boo() {
     coo();
B1:  doo();
B2:  return;
}
coo() {
    doo();
C:  return;
}
doo() {
    return;
}
```

- The calls are nested, *i.e.* boo calls coo, which calls doo. If we save the return address when boo calls coo in LR, then when we are in coo, the return address we save when calling doo will overwrite LR, and we loose the return address back to boo!
- doo() is called from two different places, and is expected to return to different places for each call.
- How do we remember the return addresses for each call, in the correct order ? *i.e.* the reverse call order.

boo calls coo	save B1
coo calls doo	save C
doo returns to coo	PC ← C
coo returns to boo	PC ← B1
boo calls doo	save B2
doo returns to boo	PC ← B2

Which data structure shall we use to save these addresses?

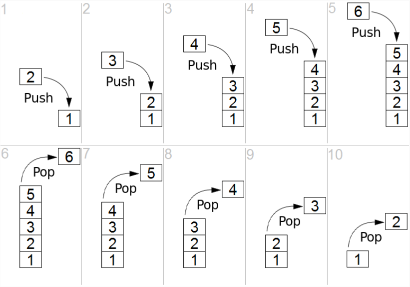
We need a way to recall return addresses in the reverse order they were saved (and later, also their parameters and return values).

We will use a Last-in-First-out (LIFO) data structure called a stack.

source: Mk2010 / CC 4.0 BY-SA

Stack operations

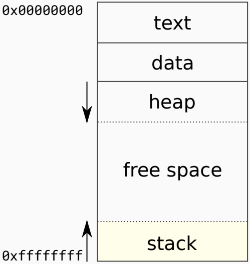
- `push(value)` : adds a new element on the top of the stack (TOS)
- `value = pop` : removes the top element
- `value = peek(distance)` : returns the value of an element at a distance relative to TOS. `peek(0)` returns the element at the TOS.



source: Maxtremus / CC0

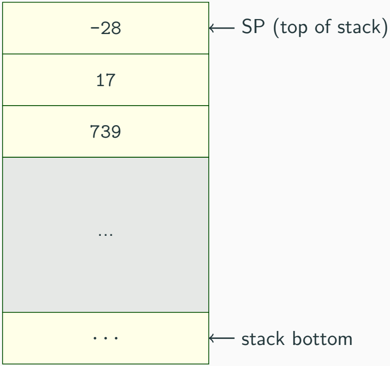
ARM Memory layout

- The heap starts at lower addresses and grows “downward”.
- The bottom of stack is at a fixed address and the top of stack grows “upward”, towards lower memory addresses.



Stack in ARM

- The stack is used to support subroutines.
- The data elements on the stack are always words.
- Register `R13` is used as a stack pointer to point to TOS, also called `SP`.



Stack operations

Push from `Rj`

```
STR Rj, [SP, #-4]!
```

`SP ← SP - 4`

`Mem[SP] ← Rj`

Pop into `Rj`

```
LDR Rj, [SP], #4
```

`Rj ← Mem[SP]`

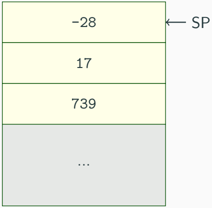
`SP ← SP + 4`

Peek(i) into `Rj`

```
LDR Rj, [SP, #const]
```

where `const` $= i * 4$

`Rj ← Mem[SP+const]`



Assuming Rj=19 and i=2, what's the content of the stack, registers Rj and SP after each instruction executes?
(consider them separately)

Pushing/Popping multiple elements

Often, several elements needs to be pushed/popped onto/from the stack. There are two pseudoinstructions that are useful:

- `PUSH {R1, R3-R5}` is a pseudoinstruction equivalent to
  `STMDB SP!, R1, R3-R5`
  (`R1` ends up at the top of the stack)
- `POP {R1, R3-R5}` is a pseudoinstruction equivalent to
  `LDMIA SP!, R1, R3-R5`
  (top of the stack ends up in `R1`)

Multiple nested calls

```
main() {
     boo();
A:   ...;
}
boo() {
     push(LR);
     coo();
B1:  doo();
B2:  LR = pop();
     return;
}
coo() {
    push(LR);
    doo();
C:  LR = pop();
    return;
}
doo() {
    return;
}
```

If you are a subroutine that will call another subroutine, follow this convention:

- Before you call anybody: Push the return address stored in LR on the stack
- When you are done calling: Pop the return address off the stack into LR

Action	Stack (TOS on left)	LR
main calls boo		A
boo saves LR	A	A
boo calls coo	A	B1
coo saves LR	B1 A	B1
coo calls doo	B1 A	C
doo returns	B1 A	C
coo restores LR	A	B1
coo returns	A	B1
boo calls doo	A	B2
doo returns	A	B2
boo restores LR		A
boo returns		A

Passing parameters and return values

For a small number of parameters you can use the ARM calling convention: use `R0` – `R3` for passing parameters, and use `R0` for the return value.

```
int add3(int a, int b, int c) {
  return a + b + c;
}
```

```
        MOV   R0, #1
        MOV   R1, #2
        MOV   R2, #3
        STR   LR, [SP, #-4]! // save return address
        BL    add3
        STR   R0, SUM        // return value is in R0
        LDR   LR, [SP], #4   // restore return address
        ...

add3:   ADD R0, R0, R1
        ADD R0, R0, R2
        BX  LR
```

Callee-save convention

```
add3:    ADD R0, R0, R1
         ADD R0, R0, R2
         BX  LR
```

- In the previous example, the callee overwrote `R0`, which was OK, since the caller knew that the return value would be in `R0`.
- In general, the caller may need the register values after the callee returns, so the rule is a callee is responsible for leaving the registers as it found them.

Callee-save convention:

A subroutine should save any registers it wants to use on the stack and then restore the original values to the registers after it is finished using them.

Passing parameters on the stack

When you have more than 4 parameters, you can pass 4 in registers, and the additional ones on the stack. Or you could pass all parameters and the return value on the stack. Passing parameters by registers will always be faster.

When you want to pass large data structure which does not fit into four words, you may also have to use the stack. Example:

```
struct largeDataStruct {
  int a;
  int b;
  int c;
  int d;
  int e;
}
```

Let's illustrate how to pass everything on the stack. Write a program to sum a list of numbers. The number of entries in the list is stored in the variable N and the list is stored starting at address ARRAY.

```
ARRAY:    .word 6,5,4,3,2,1,14,13,12,11,10,9,8,7
N:        .word 14
SUM:      .space 4

          .global _start
_start:   LDR   R0, =ARRAY      // R0 points to ARRAY
          LDR   R1, N           // R1 contains number of elements to add
          PUSH  {R0, R1, LR}    // push parameters and LR
          BL    listadd         // call subroutine
          LDR   R0, [SP, #4]    // get return value from stack
          STR   R0, SUM         // store in memory
          LDR   LR, [SP, #8]    // restore LR
          ADD   SP, SP, #12     // remove params and LR from stack
stop:     B     stop

listadd:  PUSH  {R0-R3}          // callee-save registers listadd uses
          LDR   R1, [SP, #20]    // load param N from stack
          LDR   R2, [SP, #16]    // load param ARRAY from stack
          MOV   R0, #0           // clear R0 (sum)
loop:     LDR   R3, [R2], #4     // get next value from ARRAY
          ADD   R0, R0, R3       // form the partial sum
          SUBS  R1, R1, #1       // decrement loop counter
          BGT   loop
          STR   R0, [SP, #20]    // store sum on stack, replacing N
          POP   {R0-R3}          // restore registers
          BX    LR
```

Passing by value / reference

Recap from C:

- Passing by value: a copy of the value is passed to the caller. If the copy is modified, no effect on the callee side.
- Passing by reference: an address in memory where the value is stored is passed. The caller may modify the value.

```
int add3Val(int a) {
  a = a+3;
  return a;
}
void add3Ref(int* a) {
  *a = (*a)+3
}
void main() {
  int i=77;
  int j;

  j = add3Val(i);
  print(i);
  print(j);

  add3Ref(&i);
  print(i);
  print(j);
}
```

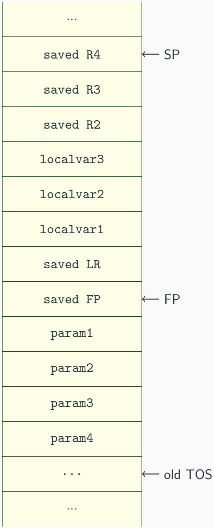
```
ARRAY:    .word 6,5,4,3,2,1,14,13,12,11,10,9,8,7
N:        .word 14
...

LDR    R0, =ARRAY     // R0 points to ARRAY
LDR    R1, N          // R1 contains the number of scores
PUSH   {R0, R1, LR}  // push parameters and LR
BL     listadd        // call subroutine
```

- The parameter N was passed by value, *i.e.* the actual value of N (14) was passed to the subroutine.
- The parameter ARRAY was passed by reference, *i.e.* a pointer to the first element of the array was passed

Stack frame

- The subroutine can also allocate local variables, only accessible by the subroutine, on the stack
- Using a frame pointer (usually `R11`) gives a consistent reference to parameters `[FP, #const]` and local variables `[FP, #-const]`, which move around relative to the `SP`.
- When nesting, the stack frame also includes the return address and frame pointer
- `FP` not scrictly required, mainly used to make assembly program easier to write, and to help with the debugger.
- FP remains constant while in the same subroutine



ARM Cheatsheet

Reading the cheatsheet

https://developer.arm.com/documentation/qrc0001/m/

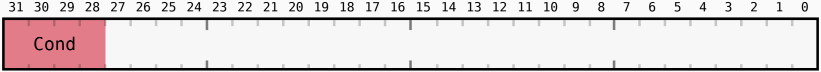
ARM Instruction Encoding

ARM Assembly vs. Binary

The machine language instruction are encoded as binary of 32 bits per instruction (ARM ISA is RISC).

The binary representation of an instruction is divided into fields. Each field contains some information that encodes information about the instruction.

General format for most instructions:

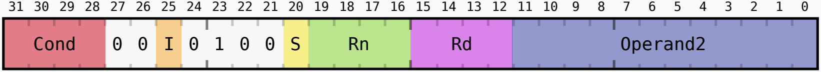


source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Condition field

Cond. field	Suffix	Meaning	CSPR Flags
0000	EQ	EQual(zero)	Z=1
0001	NE	Not Equal (nonzero)	Z=0
0010	CS/HS	Carry Set/ unsigned Higher or Same	C=1
0011	CC/LO	Carry Clear / unsigned Lower	C=0
0100	MI	MInus (negative)	N=1
0101	PL	PLus (positive or zero)	N=0
0110	VS	oVerflow Set	V=1
0111	VC	oVerflow Clear	V=0
1000	HI	unsigned Higher	C=1 AND Z=0
1001	LS	unsigned Lower or Same	C=0 OR Z=1
1010	GE	signed Greater or Equal	N=V
1011	LT	signed Less Than	N!=V
1100	GT	signed Greater Than	Z=0 AND (N=V)
1101	LE	signed Less or Equal	Z=1 OR (N!=V)
1110	AL/	ALways (usually ommitted)	any
1111		not used	

Data processing instructions encoding



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
Cond				0	0	I	0	1	0	0	S	Rn				Rd				Operand2											

source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Examples:

```
ADDGES R1, R2, R3
```

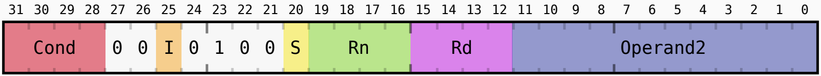
Cond=1010, I=0, S=1, Rn=0010, Rd=0001, Operand2[3-0]=0011

```
ADD R1, R2, #15
```

Cond=1110, I=0, S=1, Rn=0010, Rd=0001, Operand2=000000001111

Why are the register fields 4-bit wise?

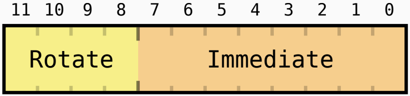
Immediate value encoding



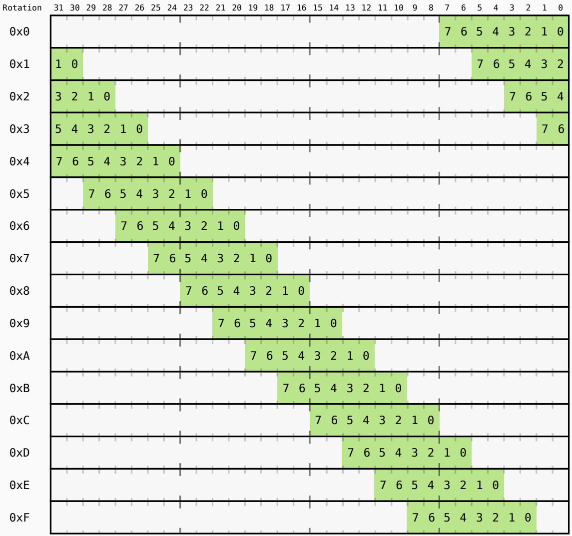
source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

12 bits available to encode immediate value. However, the largest value is not what you think it might be.

The ARM ISA has a very clever way of generating a lot of useful 32-bit constants: 16 possible rotations of an 8-bit value



source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

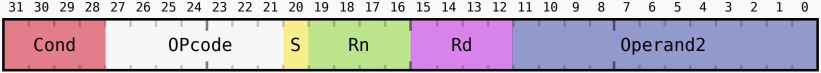


Rotation	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0																									7	6	5	4	3	2	1	0
0x1	1	0																									7	6	5	4	3	2
0x2	3	2	1	0																									7	6	5	4
0x3	5	4	3	2	1	0																									7	6
0x4	7	6	5	4	3	2	1	0																								
0x5			7	6	5	4	3	2	1	0																						
0x6					7	6	5	4	3	2	1	0																				
0x7							7	6	5	4	3	2	1	0																		
0x8									7	6	5	4	3	2	1	0																
0x9											7	6	5	4	3	2	1	0														
0xA													7	6	5	4	3	2	1	0												
0xB															7	6	5	4	3	2	1	0										
0xC																	7	6	5	4	3	2	1	0								
0xD																			7	6	5	4	3	2	1	0						
0xE																					7	6	5	4	3	2	1	0				
0xF																							7	6	5	4	3	2	1	0		

source: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Rotations of an even number of times in a 32-bit word (0, 2, ..., 30)
https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

Load/Store instructions encoding



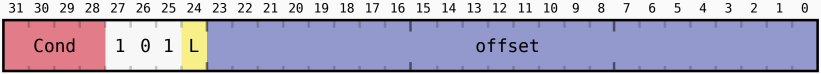
Rn=base

Operand2=Offset or Rm, if Rm, the lower four bits is the register number and uppper bits is the amount of shifting.

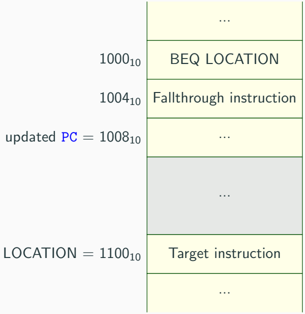
Note that:

- Not every addressing mode is available for every load/store instruction.
- The range of permitted immediate values and the options for scaled registers vary from instruction to instruction.

Branch instructions encoding



Since the offset field is limited to 24-bit, the branch target address is relative to the current value of PC and is left-shifted by 2 since instructions are always 4 byte wide. `L=1` is used for the `BL` instruction.



In this example, we want to jump to address 1100_{10} which is 100 bytes away.

The relative offset is 92 bytes $(100 - 8)$
$= 23$ words
$=$ `0000 0000 0000 0000 0001 0111`.

The condition field is EQ $=$ 0000.

Conclusions

This set of lectures has presented the ARM ISA and introduced:

- the major classes of instructions you will encounter
- the different addressing mode used by instructions
- the way ARM branches work
- the way subroutine calls are implemented in assembly with the stack
- the encoding of instructions in binary

The next lecture will:

- look at the software toolchain used to translate high-level languages to machine code;
- the role of the operating system software.