ΕΠΛ221:
Οργάνωση Υπολογιστών και Συμβολικός Προγραμματισμός

Εργαστήριο Αρ. 2

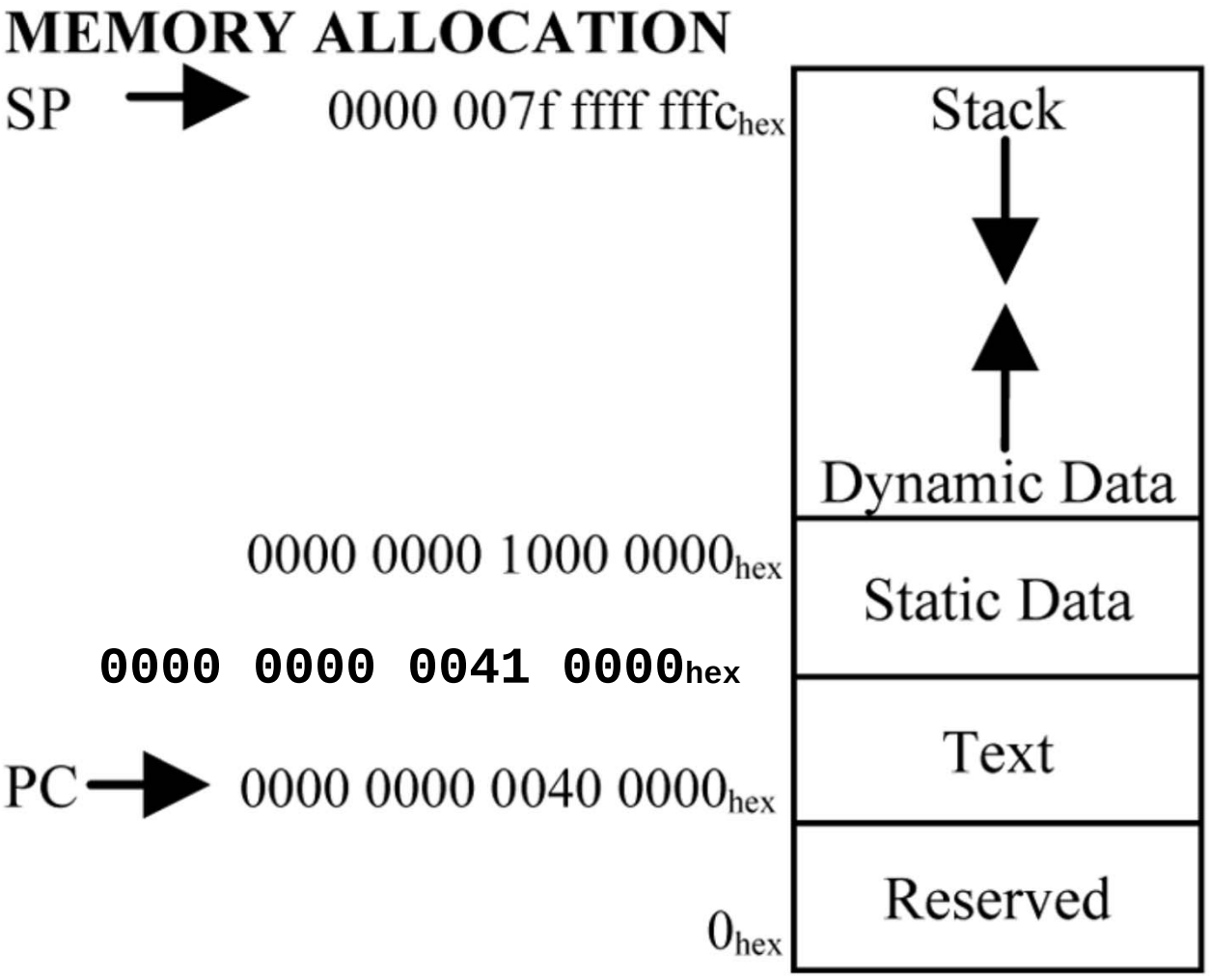
Εισαγωγή στην Αρχιτεκτονική
ARMv8-A

Arithmetic and Logic Instr.

.data, Branch and Loops

Πέτρος Παναγή, PhD

Memory Allocation LEGv8



Βασικές Εντολές

Arithmetic and logical operations

Type	Instructions
Arithmetic	ADD, SUB, ADC, SBC, NEG
Logical	AND, BIC, ORR, ORN, EOR, EON
Comparison	CMP, CMN, TST
Move	MOV, MVN

Some instructions also have an S suffix, indicating that the instruction sets flags. Of the instructions in Table 6-1, this includes ADDS, SUBS, ADCS, SBCS, ANDS, and BICS. There are other flag setting instructions, notably CMP, CMN and TST, but these do not take an S suffix.

The operations ADC and SBC perform additions and subtractions that also use the carry condition flag as an input.

```
ADC{S}: Rd = Rn + Rm + C
SBC{S}: Rd = Rn - Rm - 1 + C
```

Example 6-1 Arithmetic instructions

```
ADD W0, W1, W2, LSL #3          // W0 = W1 + (W2 << 3)
SUBS X0, X4, X3, ASR #2         // X0 = X4 - (X3 >> 2), set flags
MOV X0, X1                      // Copy X1 to X0
CMP W3, W4                      // Set flags based on W3 - W4
ADD W0, W5, #27                 // W0 = W5 + 27
```

Special dedicated registers

2 dedicated registers:

- **sp**, the stack pointer register: holds pointer to bottom of the stack
  - preferred register to access the stack
  - must be 16-bytes aligned

```
STR     W0, [SP, #4]    ; Stores W0 into the stack at address SP + 4.
ADD     SP, SP, #8      ; WARNING: SP is now "unusable": it is not aligned anymore!
STR     X0, [SP]        ; ERROR: cannot use unaligned SP!
```

zr: the zero register

- when used as source register it always returns the integer value zero.

```
MOV     W0, #0          ; W0 = 0
MOV     W0, WZR         ; W0 = 0, same effect as previous instruction
```

- when used as a destination register it discards the value

```
SUBS    WZR, W10, W11   ; Does W10 - W11, set the flags and discard the result
CMP     W10, W11        ; Compare two numbers: CMP is an alias for the SUBS above
```

Two ways of writing the same instruction

Multiplication and division

- Regular 32-bit and 64-bit multiplication:
  - `MUL Rd, Rn, Rm` → `Rd = Rn*Rm` (alias of `MADD`: `Ra = ZR`)
  - `MADD Rd, Rn, Rm, Ra` → `Rd = Ra + Rn*Rm`
  - `MSUB Rd, Rn, Rm, Ra` → `Rd = Ra - Rn*Rm`
  - `MNEG Rd, Rn, Rm` → `Rd = -Rn*Rm` (alias of `MSUB`: `Ra = ZR`)
- Long result multiplication: 32-bit source registers, 64-bit destination register.
  - Signed variants: `SMULL`, `SMADDL` , `SMSUBL`, `SMNEGL`
  - Unsigned variants: `UMULL`, `UMADDL` , `UMSUBL`, `UMNEGL`
  - Upper 64 bits in 128-bit multiplication result: `UMULH`, `SMULH`
- Signed and unsigned 32-bit and 64-bit division
  - `SDIV` / `UDIV Rd, Rm, Rn` → `Rd = Rn/Rm`
  - Division by 0 returns 0 (with no exception)
  - `MAXNEG` integer divided by -1 overflows (returns `MAXNEG`)

Examples

Lab2_example1.s
Lab2_example2.s
Lab2_example3.s

Data processing

- **Values in registers can be processed using many different instructions**
  - Arithmetic, logic, data moves, bit field manipulations, shifts, conditional comparisons, and more
  - These instructions always operate between registers, or between a register and an immediate

Example bit manipulation:

```
; Clear bit 4, set bit 7 at X1
LDR   X0, [X1]
AND   X0, X0, #~(1 << 4)
ORR   X0, X0, #(1 << 7)
STR   X0, [X1]
```

Example countdown loop:

```
  ; add W3 to all elements of an
  ; array of loop_count ints in X2
  MOV   X0, #<loop_count>
loop:
  LDR   W1, [X2]
  ADD   W1, W1, W3
  STR   W1, [X2], #4
  SUB   X0, X0, #1
  CBNZ  X0, loop
```

Shifts and rotates

- **Standalone instructions for shifts and rotates**
  - Source register may be an `Xn` or `Wn` register
  - Also used for flexible second operands, such as to shift an `LDR` / `STR` `Xn` register offset
- **Shift amount may be an immediate or a register**
  - Immediate shifts up to (register_size – 1)
  - Register values taken modulo 32-bit or 64-bit

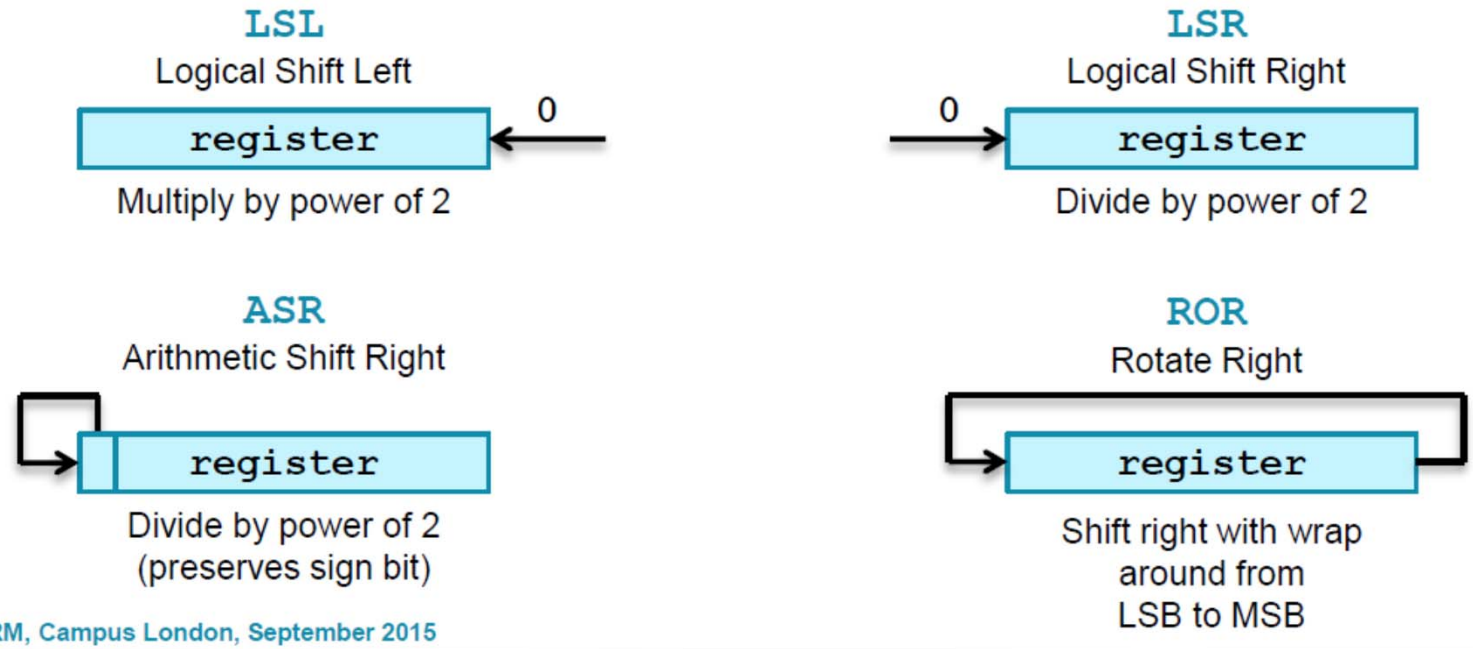


Table 6-3 Shift and move operations

Instruction	Description
Shift	
ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROR	Rotate right
Move	
MOV	Move
MVN	Bitwise NOT

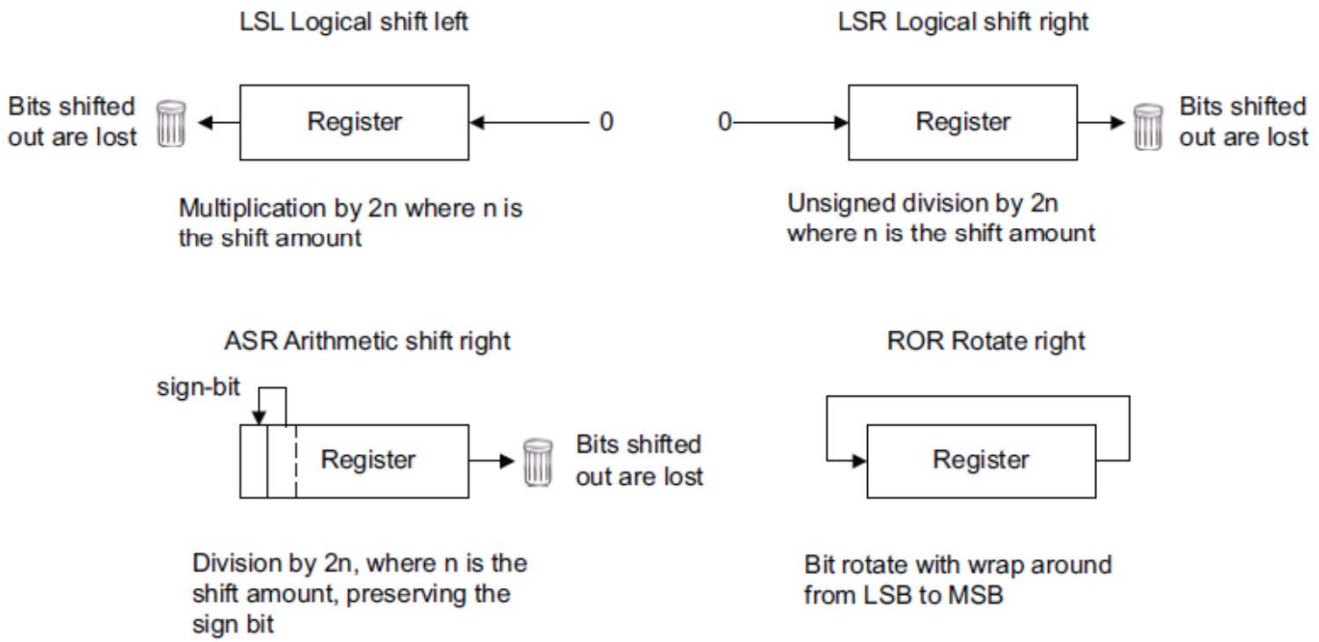


Figure 6-1 Shift operations

Extension

- **SXTB / SXTH / SXTW**
  - Sign-extend byte / half-word / single-word
- **UXTB / UXTH / UXTW**
  - Zero-extend byte / half-word / single-word
- **Destination register may be an Xn or Wn register**
  - Wn destination extends source to 32-bits, Xn destination extends source to 64-bits
  - Source register must always be a Wn register

```
SXTB   X3, W2          ; Sign-extend low byte of W2 to 64-bits
UXTH   W4, W5          ; Zero-extend low half-word of W5 to 32-bits
SXTW   X6, W7          ; Sign-extend word in W7 to 64-bits
```

Examples

Lab2_example4.s
Lab2_example5.s
Lab2_example6.s

Βασικές Εντολές

Arithmetic and logical operations

The logical operations are essentially the same as the corresponding boolean operators operating on individual bits of the register.

The BIC (Bitwise bit Clear) instruction performs an AND of the register that is the first after the destination register, with the inverted value of the second operand. For example, to clear bit [11] of register X0, use:

```
MOV X1, #0x800
BIC X0, X0, X1
```

ORN and EON perform an OR or EOR respectively with a bitwise-NOT of the second operand.

The comparison instructions only modify the flags and have no other effect. The range of immediate values for these instructions is 12 bits, and this value can be optionally shifted 12 bits to the left.

Examples

Lab2_example7.s

Useful assembler directives and macros for the GNU assembler (https://community.arm.com/processors/b/blog/posts/useful-assembler-directives-and-macros-for-the-gnu-assembler https://sourceware.org/binutils/docs/as/index.html#Top)

The **.text** directive switches the current section to the .text section.

The .text section is normally used for storing code in.

This is usually going in your flash-memory of your microcontroller (but you can customize your linker-script, so that it puts it somewhere else)

```
.text
        (put your code here)
```

The **.data** directive switches the current section to the .data section.

You can use the .data section for storing all kind of various data, which will be copied to the microcontroller's RAM, when your program starts up:

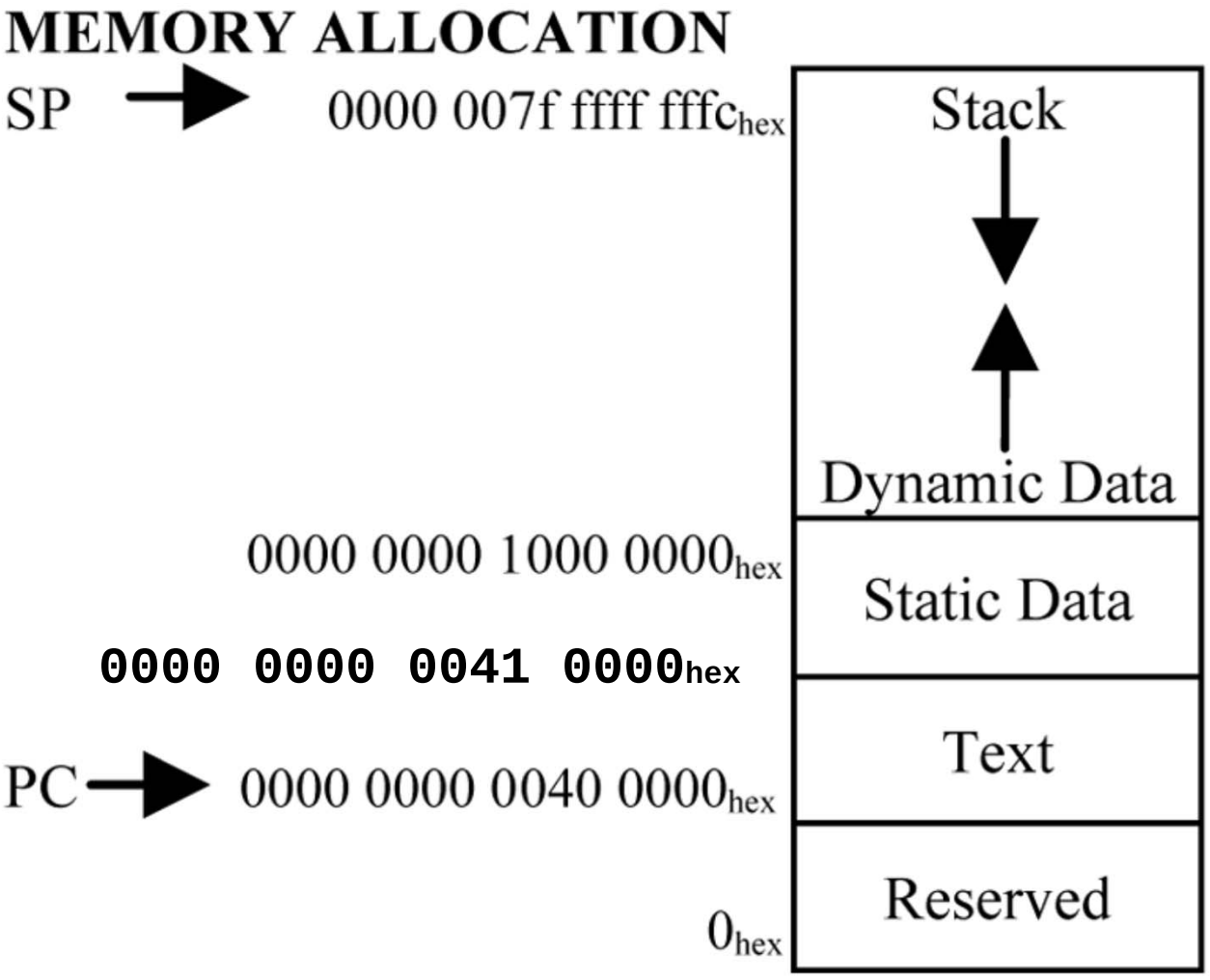
Binary values, strings, pointers, etc.

```
        .data
hello_string:       .asciz              "Hello World!\n“
```

The **`.space`** directive reserves a number of bytes in the current section. By default, it will be filled with zeroes.

https://sourceware.org/binutils/docs/as/Pseudo-Ops.html#Pseudo-Ops

Memory Allocation LEGv8



objdump

```
gcc -g Lab2_example1.s
objdump -s -j .data ./a.out
./a.out:     file format elf64-littleaarch64
Contents of section .data:
 410a00 00000000 57656c63 6f6d6520 746f2045  ....Welcome to E
 410a10 504c3232 3120616e 64204861 76652061  PL221 and Have a
 410a20 206e6963 65204461 79202121 21002563   nice Day !!!.%c
 410a30 0a00                                 ..
objdump -d ./a.out -j .text | less
0000000004005b0 <main>:
  4005b0:       a9be7bfd        stp     x29, x30, [sp,#-32]!
  4005b4:       910003fd        mov     x29, sp
  4005b8:       90000080        adrp    x0, 410000 <__FRAME_END__+0xf838>
  4005bc:       91281000        add     x0, x0, #0xa04
  4005c0:       aa0003f3        mov     x19, x0
From gdb (when break on main)
sp              0x7fffffff1b0
```

```
(gdb) info file
Symbols from “./a.out".
Local exec file: `./a.out', file type elf64-littleaarch64.
        Entry point: 0x4004c0
        0x0000000000400200 - 0x000000000040021b is .interp
        0x000000000040021c - 0x000000000040023c is .note.ABI-tag
        0x000000000040023c - 0x0000000000400260 is .note.gnu.build-id
        0x0000000000400260 - 0x0000000000400294 is .gnu.hash
        0x0000000000400298 - 0x0000000000400328 is .dynsym
        0x0000000000400328 - 0x0000000000400372 is .dynstr
        0x0000000000400372 - 0x000000000040037e is .gnu.version
        0x0000000000400380 - 0x00000000004003a0 is .gnu.version_r
        0x00000000004003a0 - 0x00000000004003b8 is .rela.dyn
        0x00000000004003b8 - 0x0000000000400430 is .rela.plt
        0x0000000000400430 - 0x0000000000400444 is .init
        0x0000000000400450 - 0x00000000004004c0 is .plt
        0x00000000004004c0 - 0x00000000004006bc is .text
        0x00000000004006bc - 0x00000000004006cc is .fini
        0x00000000004006d0 - 0x00000000004006f8 is .rodata
        0x00000000004006f8 - 0x0000000000400734 is .eh_frame_hdr
        0x0000000000400738 - 0x0000000000400824 is .eh_frame
        0x0000000000410828 - 0x0000000000410830 is .init_array
        0x0000000000410830 - 0x0000000000410838 is .fini_array
        0x0000000000410838 - 0x0000000000410840 is .jcr
        0x0000000000410840 - 0x0000000000410a10 is .dynamic
        0x0000000000410a10 - 0x0000000000410a20 is .got
        0x0000000000410a20 - 0x0000000000410a60 is .got.plt
        0x0000000000410a60 - 0x0000000000410a96 is .data
        0x0000000000410a96 - 0x0000000000410a98 is .bss
```

Memory access instructions

The general form of a Load instruction is as follows:

```
LDR Rt, <addr>
```

For loads into integer registers, you can choose a size to load. For example, to load a size smaller than the specified register value, append one of the following suffixes to the LDR instruction:

- LDRB (8-bit, zero extended).
- LDRSB (8-bit, sign extended).
- LDRH (16-bit, zero extended).
- LDRSH (16-bit, sign extended).
- LDRSW (32-bit, sign extended).

Similarly, the general form of a Store instruction is as follows:

```
STR Rn, <addr>
```

Register load/store

- `LDR`
  - Load data from an address into a register
- `STR`
  - Store data from a register to an address

```
LDR    X0, <addr>          ; Load from <addr> into X0
STR    X0, <addr>          ; Store contents of X0 to <addr>
```

- **By default, the size of the load/store is determined by the source/destination register name**
  - `Xn` will load/store 64 bits, `Wn` will load/store 32 bits
  - Instruction can be suffixed to force a smaller load/store size
    - `'B'` for byte, `'H'` for half-word, `'W'` for word
    - Result will be zero-extended by default, combine with the `'S'` suffix for sign-extension

```
LDRSB  X0, <addr>          ; Load byte from <addr> into X0 and sign-extend
STRH   W1, <addr>          ; Store half-word from W1 to <addr>
```

Memory access instructions

Example: Byte loads

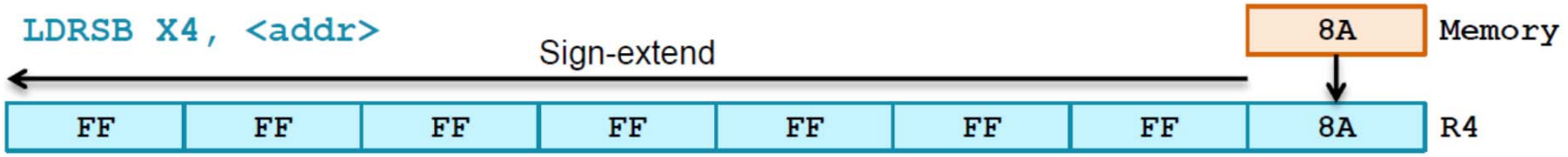
Hex 8A is decimal **-118** or **138** depending on whether it is considered **signed** or **unsigned**

Sign-extended 8-bit load to a `Wn` register:

```
LDRSB W4, <addr>
```

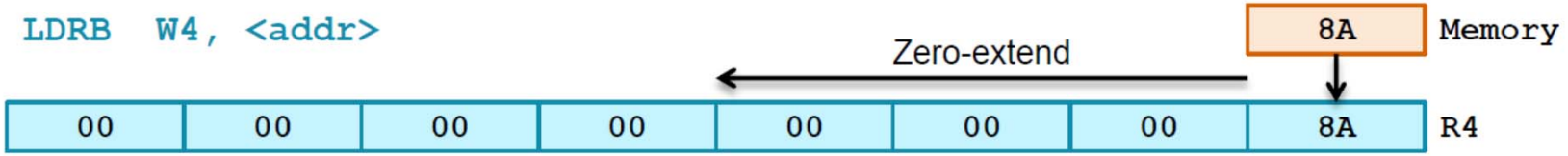
Sign-extended 8-bit load to an `Xn` register:

```
LDRSB X4, <addr>
```



Zero-extended 8-bit load to a `Wn` register:

```
LDRB  W4, <addr>
```



Memory access instructions

Offset modes

Offset addressing modes add an immediate value or an optionally-modified register value to a 64-bit base register to generate an address.

Table 6-8 Offset addressing modes

Example instruction	Description
`LDR X0, [X1]`	Load from the address in X1
`LDR X0, [X1, #8]`	Load from address X1 + 8
`LDR X0, [X1, X2]`	Load from address X1 + X2
`LDR X0, [X1, X2, LSL, #3]`	Load from address X1 + (X2 << 3)
`LDR X0, [X1, W2, SXTW]`	Load from address X1 + sign_extend(W2)
`LDR X0, [X1, W2, SXTW, #3]`	Load from address X1 + (sign_extend(W2) << 3)

Memory access instructions

There are two variants: pre-index modes which apply the offset *before* accessing the memory, and post-index modes which apply the offset *after* accessing the memory.

Table 6-9 Index addressing modes

Example instruction	Description
`LDR X0, [X1, #8]!`	Pre-index: Update X1 first (to X1 + #8), then load from the new address
`LDR X0, [X1], #8`	Post-index: Load from the unmodified address in X1 first, then update X1 (to X1 + #8)
`STP X0, X1, [SP, #-16]!`	Push X0 and X1 to the stack.
`LDP X0, X1, [SP], #16`	Pop X0 and X1 off the stack.

Specifying the load/store address

- Address to load/store from is a 64-bit base register plus an optional offset

```
LDR    X0, [X1]                ; Load from address held in X1
STR    X0, [X1]                ; Store to address held in X1
```

- Offset can be an immediate or a register

```
LDR    X0, [X1, #8]            ; Load from address [X1 + 8 bytes]
LDR    X0, [X1, #-8]           ; Load with negative offset
LDR    X0, [X1, X2]            ; Load from address [X1 + X2]
```

- A `Wn` register offset needs to be extended to 64 bits

```
LDR    X0, [X1, W2, SXTW]      ; Sign-extend offset in W2
LDR    X0, [X1, W2, UXTW]      ; Zero-extend offset in W2
```

- Both `Xn` and `Wn` register offsets can include an optional left-shift

```
LDR    X0, [X1, W2, UXTW #2]   ; Zero-extend offset in W2 & left-shift by 2
LDR    X0, [X1, X2, LSL #2]    ; Left-shift offset in X2 by 2
```

Addressing modes

```
/* Analogous C code */
int *intptr = …;     // X1
int out;             // W0
```

Simple: X1 is not changed

```
LDR W0, [X1]
```

```
out = *intptr;
```

Offset: X1 is not changed

```
LDR W0, [X1, #4]
```

```
out = intptr[1];
```

Pre-indexed: X1 changed before load

```
LDR W0, [X1, #4]!   ⇒   ADD X1, X1, #4
                        LDR W0, [X1]
```

```
out = *(++intptr);
```

Post-indexed: X1 changed after load

```
LDR W0, [X1], #4    ⇒   LDR W0, [X1]
                        ADD X1, X1, #4
```

```
out = *(intptr++);
```

Accessing multiple memory locations

In A64 code, there are the ***Load Pair*** **(LDP)** and ***Store Pair*** **(STP)** instructions

Table 6-11 Register Load/Store pair

Load and Store pair	Description
`LDP W3, W7, [X0]`	Loads word at address X0 into W3 and word at address X0 + 4 into W7. See Figure 6-6.
`LDP X8, X2, [X0, #0x10]!`	Loads doubleword at address X0 + 0x10 into X8 and the doubleword at address X0 + 0x10 + 8 into X2 and add 0x10 to X0. See Figure 6-7.
`LDPSW X3, X4, [X0]`	Loads word at address X0 into X3 and word at address X0 + 4 into X4, and sign extends both to doubleword size.
`LDP D8, D2, [X11], #0x10`	Loads doubleword at address X11 into D8 and the doubleword at address X11 + 8 into D2 and adds `0x10` to X11.
`STP X9, X8, [X4]`	Stores the doubleword in X9 to address X4 and stores the doubleword in X8 to address X4 + 8.

Register pair load/store

- **New Load Pair and Store Pair instructions**
  - Support both integer and scalar FP / SIMD registers
  - Both source/destination registers must be the same width

```
LDP    W3, W7, [X0]          ; [X0] => W3, [X0 + 4 bytes] => W7
STP    Q0, Q1, [X4]          ; Q0 => [X4], Q1 => [X4 + 16 bytes]
```

- **No Load Multiple, Store Multiple, or `PUSH` / `POP` instructions in AArch64**
  - Construct these operations using `STP` and `LDP` instructions

```
STP    X0, X1, [SP, #-16]!   ; Push X0 and X1 onto the stack
LDP    X0, X1, [SP], #16     ; Pop X0 and X1 from the stack
```

  - **There are variants of `LDR` to load `PC` relative data**
    - Use a label operand rather than a 64-bit base address register
    - Linker generates a `PC` relative load from the address of the label in the executable image

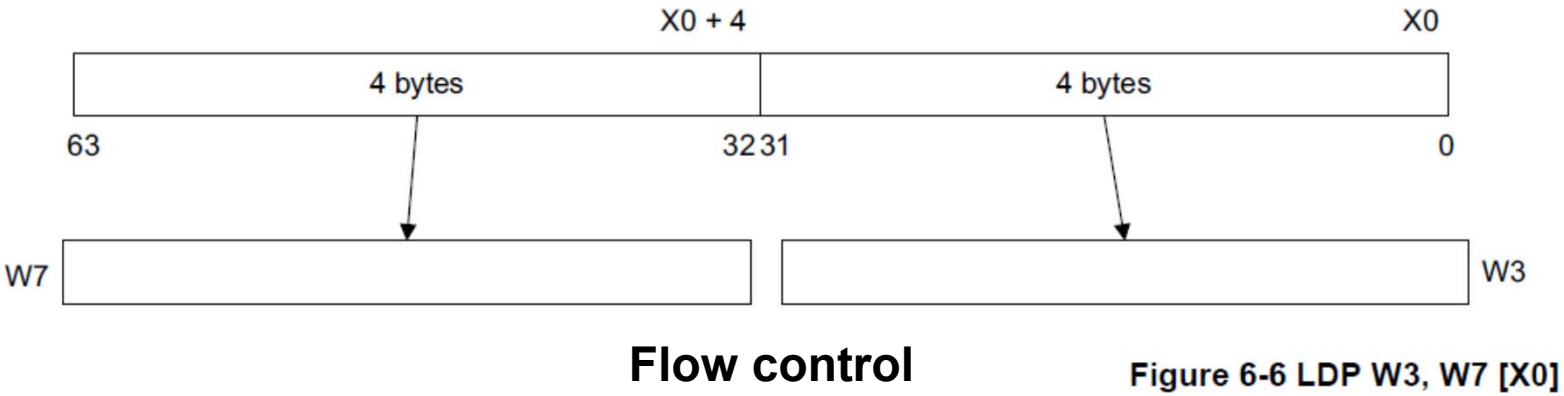
```
LDR    X0, label             ; Load value at <label>
```

  - **Assemblers may support a “Load (immediate)” pseudo-instruction**
    - Creates a `PC` relative load, and a literal pool containing the value to be loaded

```
LDR    X0, =imm              ; Load from literal containing imm
```

Accessing multiple memory locations

In A64 code, there are the ***Load Pair*** **(LDP)** and ***Store Pair*** **(STP)** instructions



Flow control

Figure 6-6 LDP W3, W7 [X0]

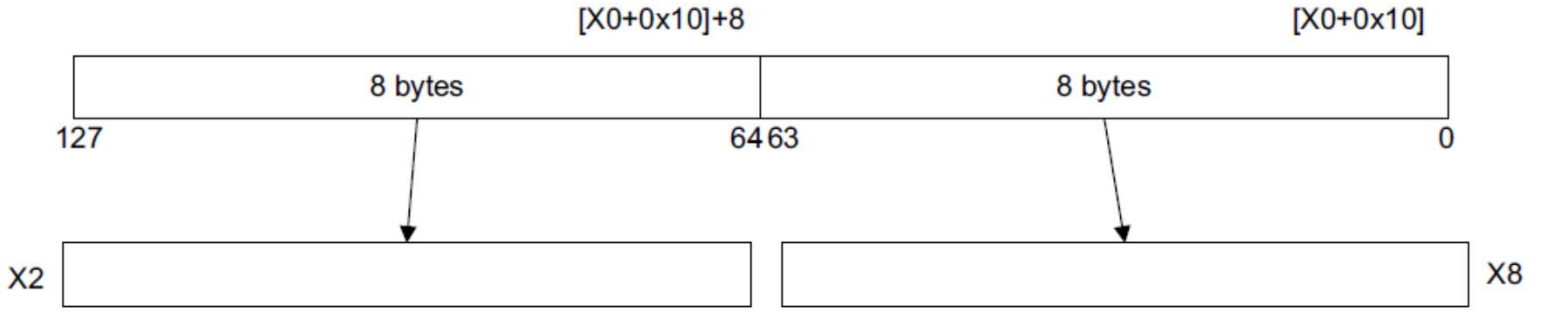


Figure 6-7 LDP X8, X2, [X0 + #0x10]!

Using the PC

5.3.4 Address Generation

`ADRP Xd, label`

Address of Page: sign extends a 21-bit offset, shifts it left by 12 and adds it to the value of the PC with its bottom 12 bits cleared, writing the result to register `Xd`. This computes the base address of the 4KiB aligned memory region containing label, and is designed to be used in conjunction with a load, store or `ADD` instruction which supplies the bottom 12 bits of the label's address. This permits position-independent addressing of any location within ±4GiB of the PC using two instructions, providing that dynamic relocation is done with a minimum granularity of 4KiB (i.e. the bottom 12 bits of the label's address are unaffected by the relocation). The term "page" is short-hand for the 4KiB relocation granule, and is not necessarily related to the virtual memory page size.

`ADR Xd, label`

Address: adds a 21-bit signed byte offset to the program counter, writing the result to register `Xd`. Used to compute the effective address of any location within ±1MiB of the PC.

Using the PC

- **Obtaining the address of a label**
  - PC relative loads and ADR are limited in range to ±1MB, whereas ADRP has range ±4GB

```
LDR    X0, =label          ; Load address of label from literal pool
ADR    X0, label           ; Calculate address of label (PC relative)
ADR    X0, .               ; Get current PC (address of ADR instruction)
ADRP   X0, label           ; Calculate address of 4KB page containing label
```

Our Assembly Code

```
adrp    x0, message_str
add     x0, x0, :lo12:message_str
```

Disassembly Code

```
4005b8:       90000080        adrp    x0, 410000 <__FRAME_END__+0xf838>
4005bc:       91281000        add     x0, x0, #0xa04 //x0= 0x410a04
```

```
Contents of section .data:
 410a00 00000000 57656c63 6f6d6520 746f2045  ....Welcome to E
 410a10 504c3232 3120616e 64204861 76652061  PL221 and Have a
 410a20 206e6963 65204461 79202121 21002563   nice Day !!!.%c
 410a30 0a00                                 ..
```

Examples

Lab2_example8.s
Lab2_example9.s

5.1 Control Flow

5.1.1 Conditional Branch

Unless stated, conditional branches have a branch offset range of ±1MiB from the program counter.

`B.cond label`

Branch: conditionally jumps to program-relative label if cond is true.

`CBNZ Wn, label`

Compare and Branch Not Zero: conditionally jumps to program-relative `label` if `Wn` is not equal to zero.

`CBNZ Xn, label`

Compare and Branch Not Zero (extended): conditionally jumps to `label` if `Xn` is not equal to zero.

`CBZ Wn, label`

Compare and Branch Zero: conditionally jumps to `label` if `Wn` is equal to zero.

`CBZ Xn, label`

Compare and Branch Zero (extended): conditionally jumps to `label` if `Xn` is equal to zero.

`TBNZ Xn|Wn, #uimm6, label`

Test and Branch Not Zero: conditionally jumps to `label` if bit number `uimm6` in register `Xn` is not zero. The bit number implies the width of the register, which may be written and should be disassembled as `Wn` if `uimm` is less than 32. Limited to a branch offset range of ±32KiB.

`TBZ Xn|Wn, #uimm6, label`

Test and Branch Zero: conditionally jumps to `label` if bit number `uimm6` in register `Xn` is zero. The bit number implies the width of the register, which may be written and should be disassembled as `Wn` if `uimm6` is less than 32. Limited to a branch offset range of ±32KiB.

5.1.2 Unconditional Branch (immediate)

Unconditional branches support an immediate branch offset range of ±128MiB.

`B label`

Branch: unconditionally jumps to pc-relative `label`.

`BL label`

Branch and Link: unconditionally jumps to pc-relative `label`, writing the address of the next sequential instruction to register `X30`.

5.1.3 Unconditional Branch (register)

`BLR Xm`

Branch and Link Register: unconditionally jumps to address in `Xm`, writing the address of the next sequential instruction to register `X30`.

`BR Xm`

Branch Register: jumps to address in `Xm`, with a hint to the CPU that this is not a subroutine return.

`RET {Xm}`

Return: jumps to register `Xm`, with a hint to the CPU that this is a subroutine return. An assembler shall default to register `X30` if `Xm` is omitted.

Branches

- `B <offset>`
  - PC relative branch ±128 MB
  - Conditional version `B.cond` (covered later) has ±1 MB range
- `BL <offset>`
  - Similar to `B` (branch range ±128 MB) but also stores return address in `LR` (`X30`), hinting that this is a function call
  - No conditional version
- `BR Xm`
  - Absolute branch to address in `Xm`
- `BLR Xm`
  - Similar to `BR`, but also stores return address in `LR` (`X30`), hinting that this is a function call
- `RET Xm` or simply `RET`
  - Similar to `BR`, but also hints that this is a function return
  - Use `LR` (`X30`) if register is omitted, but can use other register

Conditional execution

- **A64 does not allow instructions to be conditionally executed**
  - Except for branch instructions
  - Unlike A32, which allows for most instructions to include a condition code, for example `ADDEQ R0, R1, R2`
  - Unlike T32, which supports the `IT` (If Then) instruction
- **A64 has conditional operations**
  - These instructions are always executed, but their result depends on the ALU flags
- **Some data processing instructions will set the ALU flags after execution**
  - Mnemonics appended with 's', for example `SUBS`
  - Some encodings have preferred syntax for disassembly to aid in clarity

```
SUBS   X0, X1, X2          ; X0 = (X1 - X2), and set ALU flags
TST    X0, #(1 << 20)      ; Alias of ANDS XZR, X0, #(1 << 20)
CMP    X0, #5              ; Alias of SUBS XZR, X0, #5
```

Flow control using Branch Inst.

Branch instructions	
`B (offset)`	Program relative branch forward or back 128MB. A conditional version, for example B.EQ, has a 1MB range.
`BL (offset)`	As B but store the return address in X30, and hint to branch prediction logic that this is a function call.
`BR Xn`	Absolute branch to address in Xn.
`BLR Xn`	As BR but store the return address in X30, and hint to branch prediction logic that this is a function call.
`RET{Xn}`	As BR, but hint to branch prediction logic that this is a function return. Returns to the address in X30 by default, but a different register can be specified.
Conditional branch instructions	
`CBZ Rt, label`	Compare and branch if zero. If Rt is zero, branch forward or back up to 1MB.
`CBNZ Rt, label`	Compare and branch if non-zero. If Rt is not zero, branch forward or back up to 1MB.
`TBNZ Rt, bit, label`	Test and branch if zero. Branch forward or back up to 32kB.
`TBNZ Rt, bit, label`	Test and branch if non-zero. Branch forward or back up to 32kB.

Conditionally executed Instructions

Saved Process Status Register

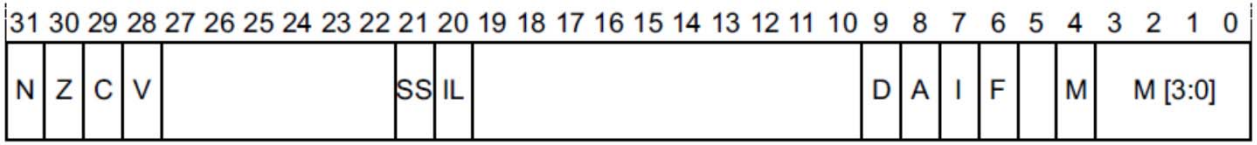


Figure 4-4 SPSR

The individual bits represent the following values for AArch64:

N Negative result (N flag).

Z Zero result (Z) flag.

C Carry out (C flag).

V Overflow (V flag).

Table 6-4 Condition flag

Flag	Name	Description
N	Negative	Set to the same value as bit[31] of the result. For a 32-bit signed integer, bit[31] being set indicates that the value is negative.
Z	Zero	Set to 1 if the result is zero, otherwise it is set to 0.
C	Carry	Set to the carry-out value from result, or to the value of the last bit shifted out from a shift operation.
V	Overflow	Set to 1 if signed overflow or underflow occurred, otherwise it is set to 0.

The C flag is set if the result of an unsigned operation overflows the result register.

The V flag operates in the same way as the C flag, but for signed operations.

Setting the ALU flags

- **The ALU flags are part of PSTATE**
  - NZCV → Negative, Zero, Carry, Overflow

```
MOV    X0, #1                    ; NZCV
SUBS   X1, X0, X0                ; 0100

MOV    W0, #0xFFFFFFFF
MOV    W1, #1                    ; NZCV
ADDS   W2, W0, W1                ; 0110

MOV    W0, #0
MOV    W1, #1                    ; NZCV
SUBS   W2, W0, W1                ; 1000
```

Using the ALU flags

- **Condition codes change the behaviour of some instructions based on the ALU flags**
  - Suffixed to conditional branches, for example `B.EQ label`
  - Passed as an operand to conditional operations, for example `CSINC W0, EQ`
- **Some of the available condition codes are shown below**
  - See appendix for complete list

Condition Code	Description	ALU Flags
`EQ`	Equal	`Z == 1`
`NE`	Not Equal	`Z == 0`
`CS / HS`	Unsigned Higher or Same	`C == 1`
`CC / LO`	Unsigned Lower	`C == 0`
`MI`	Minus	`N == 1`

Conditional branches

- `B.cond`
  - Branch to label if condition code evaluates to true

```
CMP    X0, #5
B.EQ   label                ; Branch to label if (X0 == #5)
```

- `CBZ` / `CBNZ`
  - Branch to label if operand register is equal to zero (`CBZ`) or not equal to zero (`CBNZ`)

```
CBZ    X0, label            ; Branch to label if (X0 == #0)
CBNZ   W0, label            ; Branch to label if (W0 != #0)
```

- `TBZ` / `TBNZ`
  - Branch to a label if a specific bit in the operand register is set (`TBNZ`) or cleared (`TBZ`)

```
TBZ    W0, #20, label       ; Branch to label if (W0[20] == #0b0)
TBNZ   X0, #50, label       ; Branch to label if (X0[50] == #0b1)
```

Example: Condition execution

C Source Code:

```
if (a == 0)
{
        y = y + 1;
}
else
{
        y = y - 1;
}
```

A64 Conditional Branching:

```
        CMP     W0, #0
        B.NE    else
        ADD     W1, W1, #1
        B       end
else:
        SUB     W1, W1, #1
end:
```

Instructions that set the Flags

Arithmetic and logical operations

Type	Instructions
Arithmetic	ADD, SUB, ADC, SBC, NEG
Logical	AND, BIC, ORR, ORN, EOR, EON
Comparison	CMP, CMN, TST
Move	MOV, MVN

Some instructions also have an S suffix, indicating that the instruction sets flags. Of the instructions in Table 6-1, this includes ADDS, SUBS, ADCS, SBCS, ANDS, and BICS. There are other flag setting instructions, notably CMP, CMN and TST, but these do not take an S suffix.

The operations ADC and SBC perform additions and subtractions that also use the carry condition flag as an input.

```
ADC{S}: Rd = Rn + Rm + C
SBC{S}: Rd = Rn - Rm - 1 + C
```

Example 6-1 Arithmetic instructions

```
ADD W0, W1, W2, LSL #3          // W0 = W1 + (W2 << 3)
SUBS X0, X4, X3, ASR #2         // X0 = X4 - (X3 >> 2), set flags
MOV X0, X1                      // Copy X1 to X0
CMP W3, W4                      // Set flags based on W3 - W4
ADD W0, W5, #27                 // W0 = W5 + 27
```

Instructions that set the Flags

The A64 ISA has instructions which set or test condition codes. Those that do will be identified as follows:

1. Instructions which set the condition flags are notionally different instructions, and will continue to be identified by appending an 'S' to the base mnemonic, e.g. `ADDS`.
2. Instructions which are truly conditionally executed (i.e. when the condition is false they have no effect on the architectural state, aside from advancing the program counter) have the condition appended to the instruction with a '.' delimiter. For example `B.EQ`.
3. If there is more than one instruction extension, then the conditional extension is always last.
4. Where a conditional instruction has qualifiers, the qualifiers follow the condition.
5. Instructions which are unconditionally executed, but use the condition flags as a source operand, will specify the condition to test in their final operand position, e.g. `CSEL Wd,Wm,Wn,NE`

Contitional Execution

The full list of condition codes is as follows:

Encoding	Name (& alias)	Meaning (integer)	Meaning (floating point)	Flags
0000	EQ	Equal	Equal	Z==1
0001	NE	Not equal	Not equal, or unordered	Z==0
0010	HS (CS)	Unsigned higher or same (Carry set)	Greater than, equal, or unordered	C==1
0011	LO (CC)	Unsigned lower (Carry clear)	Less than	C==0
0100	MI	Minus (negative)	Less than	N==1
0101	PL	Plus (positive or zero)	Greater than, equal, or unordered	N==0
0110	VS	Overflow set	Unordered	V==1
0111	VC	Overflow clear	Ordered	V==0
1000	HI	Unsigned higher	Greater than, or unordered	C==1 && Z==0
1001	LS	Unsigned lower or same	Less than or equal	!(C==1 && Z==0)
1010	GE	Signed greater than or equal	Greater than or equal	N==V
1011	LT	Signed less than	Less than or unordered	N!=V
1100	GT	Signed greater than	Greater than	Z==0 && N==V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z==0 && N==V)
1110	AL	Always	Always	Any
1111	NV†			

Examples

Lab2_example10.s
Lab2_example11.s

Επανάληψη

AAPCS64: Role of integer registers

Register	Alternative name	Role
R0		Return value (for integers and pointers)
R0 … R7		Arguments in function calls (for integers and pointers)
R8		Indirect result location register. Used in C++ for returning non-trivial objects (set by the caller).
R9 … R15		Temporary registers (trashed across calls)
R16, R17	IP0, IP1	The intra-procedure-call temporary registers. The linker may use these in PLT code. Can be used as temporary registers between calls
R18		Platform register
R19 … R28		Callee-saved registers: register preserved across calls
R29	FP	Frame pointer. Copy of SP before function stack allocation
R30	LR	Link register. BL and BLR instructions save return address in it
SP		Stack pointer

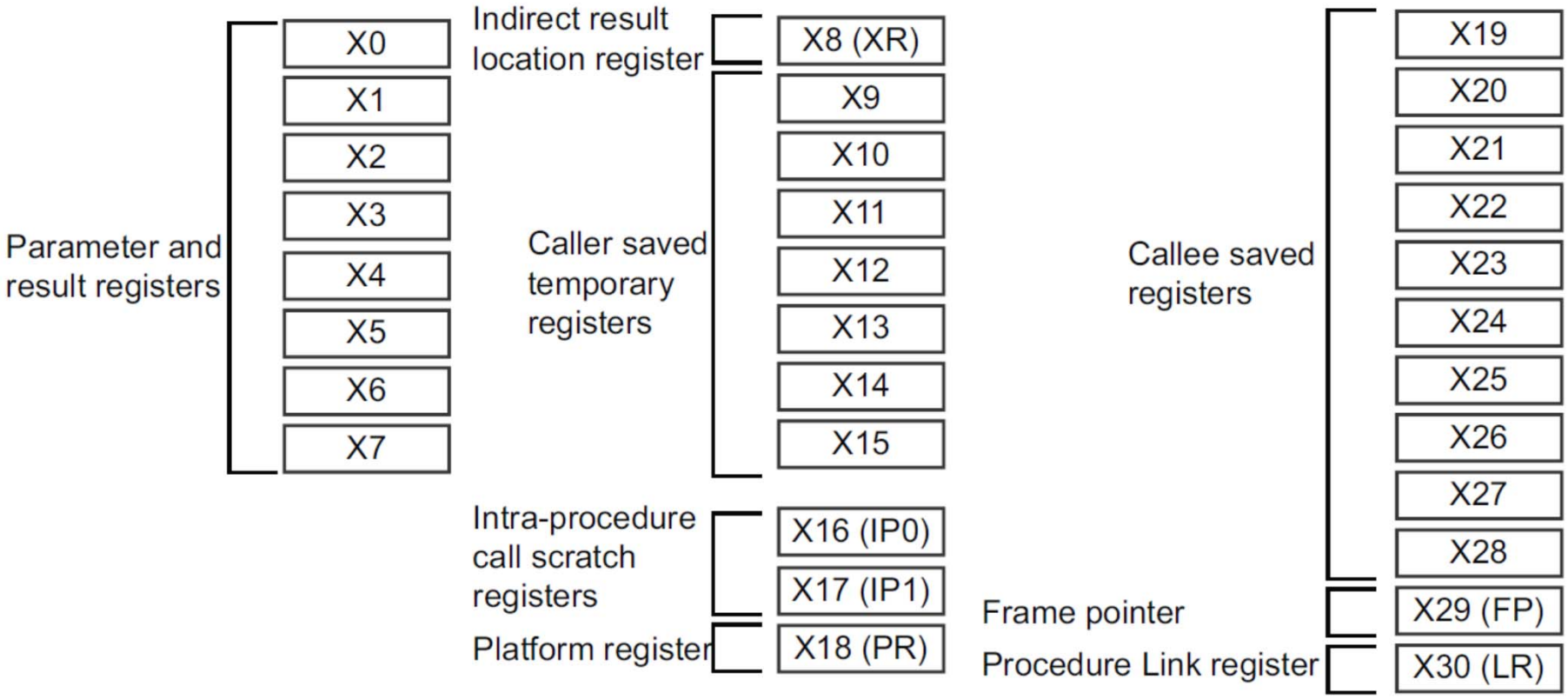


Figure 9-1 General-purpose register use in the ABI

For the purposes of function calls, the general-purpose registers are divided into four groups:

Argument registers (X0-X7)

These are used to pass parameters to a function and to return a result. They can be used as scratch registers or as caller-saved register variables that can hold intermediate values within a function, between calls to other functions. The fact that 8 registers are available for passing parameters reduces the need to spill parameters to the stack when compared with AArch32.

Caller-saved temporary registers (X9-X15)

If the caller requires the values in any of these registers to be preserved across a call to another function, the caller must save the affected registers in its own stack frame. They can be modified by the called subroutine without the need to save and restore them before returning to the caller.

Callee-saved registers (X19-X29)

These registers are saved in the callee frame. They can be modified by the called subroutine as long as they are saved and restored before returning.

Registers with a special purpose (X8, X16-X18, X29, X30)

- X8 is the indirect result register. This is used to pass the address location of an indirect result, for example, where a function returns a large structure.
- X16 and X17 are IP0 and IP1, intra-procedure-call temporary registers. These can be used by call veneers and similar code, or as temporary registers for intermediate values between subroutine calls. They are corruptible by a function. Veneers are small pieces of code which are automatically inserted by the linker, for example when the branch target is out of range of the branch instruction.
- X18 is the platform register and is reserved for the use of platform ABIs. This is an additional temporary register on platforms that don't assign a special meaning to it.
- X29 is the frame pointer register (FP).
- X30 is the link register (LR).

Specifying register load size

Load Size	Extension	Xn	Wn
8-bit	Zero	--	LDRB
	Sign	LDRSB	LDRSB
16-bit	Zero	--	LDRH
	Sign	LDRSH	LDRSH
32-bit	Zero	--	LDR
	Sign	LDRSW	--
64-bit	Zero	LDR	--

- There is no encoding for a zero-extended load of less than 64-bits to an **Xn** register
  - Writing to a **Wn** register automatically clears bits [63:32], which accomplishes the same thing

Store Size	Xn	Wn
8-bit	--	STRB
16-bit	--	STRH
32-bit	--	STR
64-bit	STR	--

Condition codes

Condition Code	Description	Flags Tested
EQ	Equal	Z == 1
NE	Not Equal	Z == 0
CS / HS	Unsigned Higher or Same	C == 1
CC / LO	Unsigned Lower	C == 0
MI	Minus	N == 1
PL	Positive or Zero	N == 0
VS	Overflow	V == 1
VC	No Overflow	V == 0
HI	Unsigned Higher	C == 1 && Z == 0
LS	Unsigned Lower or Same	C == 0 && Z == 1
GE	Greater Than or Equal	N == V
LT	Less Than	N != V
GT	Greater Than	Z == 0 && N == V
LE	Less Than or Equal	Z == 1 || N != V
AL	Always	--

NZCV → Negative, Zero, Carry, Overflow

Data types

Table 8-1 Basic data types

Type	A32	A64	Description
int/long	32-bit	32-bit	integer
short	16-bit	16-bit	integer
char	8-bit	8-bit	byte
long long	64-bit	64-bit	integer
float	32-bit	32-bit	single-precision IEEE floating-point
double	64-bit	64-bit	double-precision IEEE floating-point
bool	8-bit	8-bit	Boolean