ΕΠΛ221:
Οργάνωση Υπολογιστών και Συμβολικός Προγραμματισμός

Εργαστήριο Αρ. 1

Εισαγωγή στην Αρχιτεκτονική
ARMv8

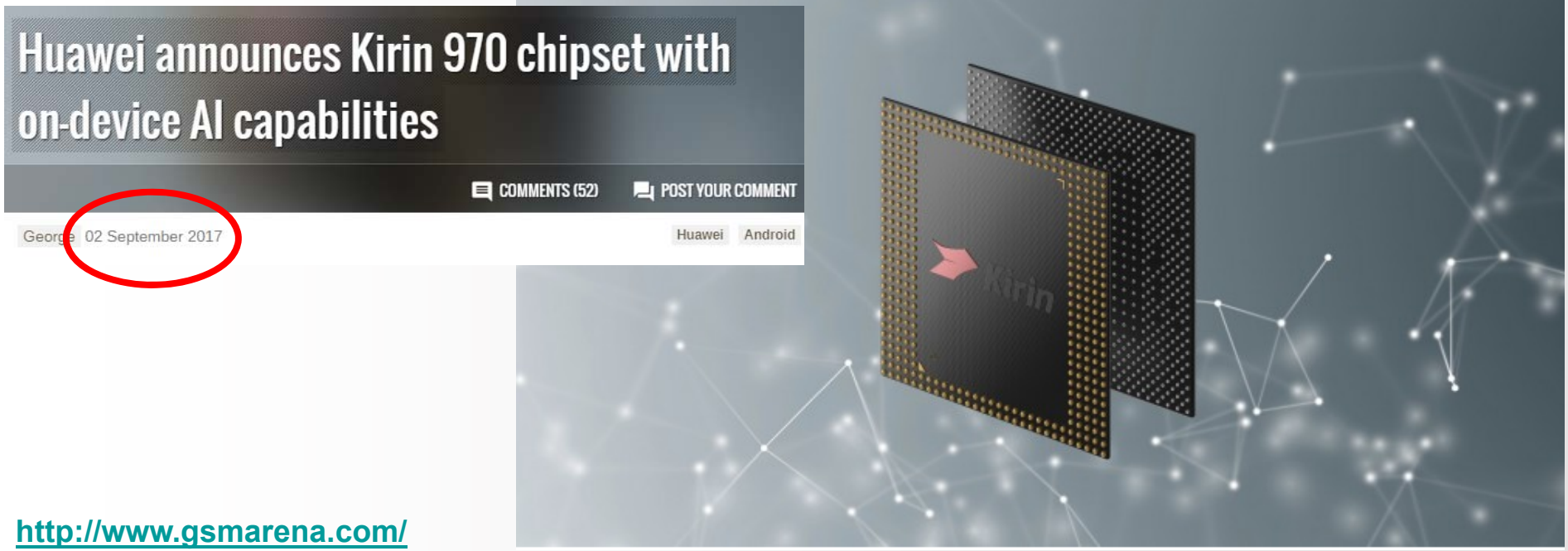
Πέτρος Παναγή, PhD

Kirin 970

Huawei has announced its next-generation chipset for smartphones, the Kirin 970. It bets big on AI and comes with what Huawei calls a neural processing unit (or NPU) bringing on-device AI capabilities.

The new chipset features an octa-core CPU with 4x2.4GHz Cortex-A73 and 4x1.8GHz Cortex-A53 cores, and a 12-core Mali-G72MP12 GPU. Built on a 10nm process, the SoC contains some 5.5 billion transistors on a single square centimeter. Huawei says its 'new heterogeneous computing architecture delivers up to 25x the performance with 50x greater efficiency' than your typical quad-core Cortex-A73 CPU cluster helping it outperform other chips on the market in AI tasks.

And since all this AI prowess needs actual applications, Huawei states it's opening up the Kirin 970 to developers to find practical uses.



http://www.gsmarena.com/

https://www.youtube.com/watch?v=5k566SwWDIc

Ιστοσελίδα Μαθήματος

http://www.cs.ucy.ac.cy/courses/EPL221/

Υλικό Για το Εργαστήριο:

http://www.cs.ucy.ac.cy/courses/EPL221/epl221_lab_schedule.htm

Η παρουσία στα εργαστήρια είναι υποχρεωτική.

Η παράδοση των εργασιών σας θα γίνεται ΜΟΝΟ με την χρήση του ειδικού εντύπου.

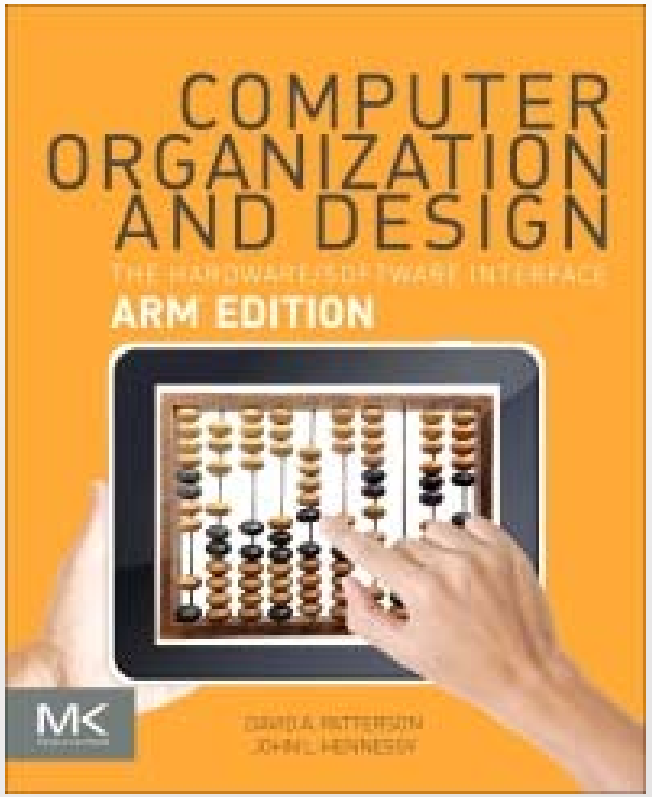
Απαγορεύεται η χρήση ιστοσελίδων εκτός ucy.ac.cy κατα την διάρκεια των εργαστηρίων. (π.χ. Facebook, gmail, κτλ.)

Προσπάθεια αντιγραφής ή αποστολής κώδικα απαγορεύτε.

Βιβλιογραφία:

Computer Organization and Design ARM Edition

1st Edition



https://www.elsevier.com/books/computer-organization-and-design-arm-edition/patterson/978-0-12-801733-3?utm_source=publicity&utm_medium=pressrls&utm_campaign=ARM_COD_rel

Βιβλιογραφία (από διαδίκτυο):

ARM® Architecture Reference Manual

ARMv8, for ARMv8-A architecture profile (6354 σελίδες !!!!)

https://static.docs.arm.com/ddi0487/b/DDI0487B_a_armv8_arm.pdf

ARM® Cortex®-A Series Version: 1.0

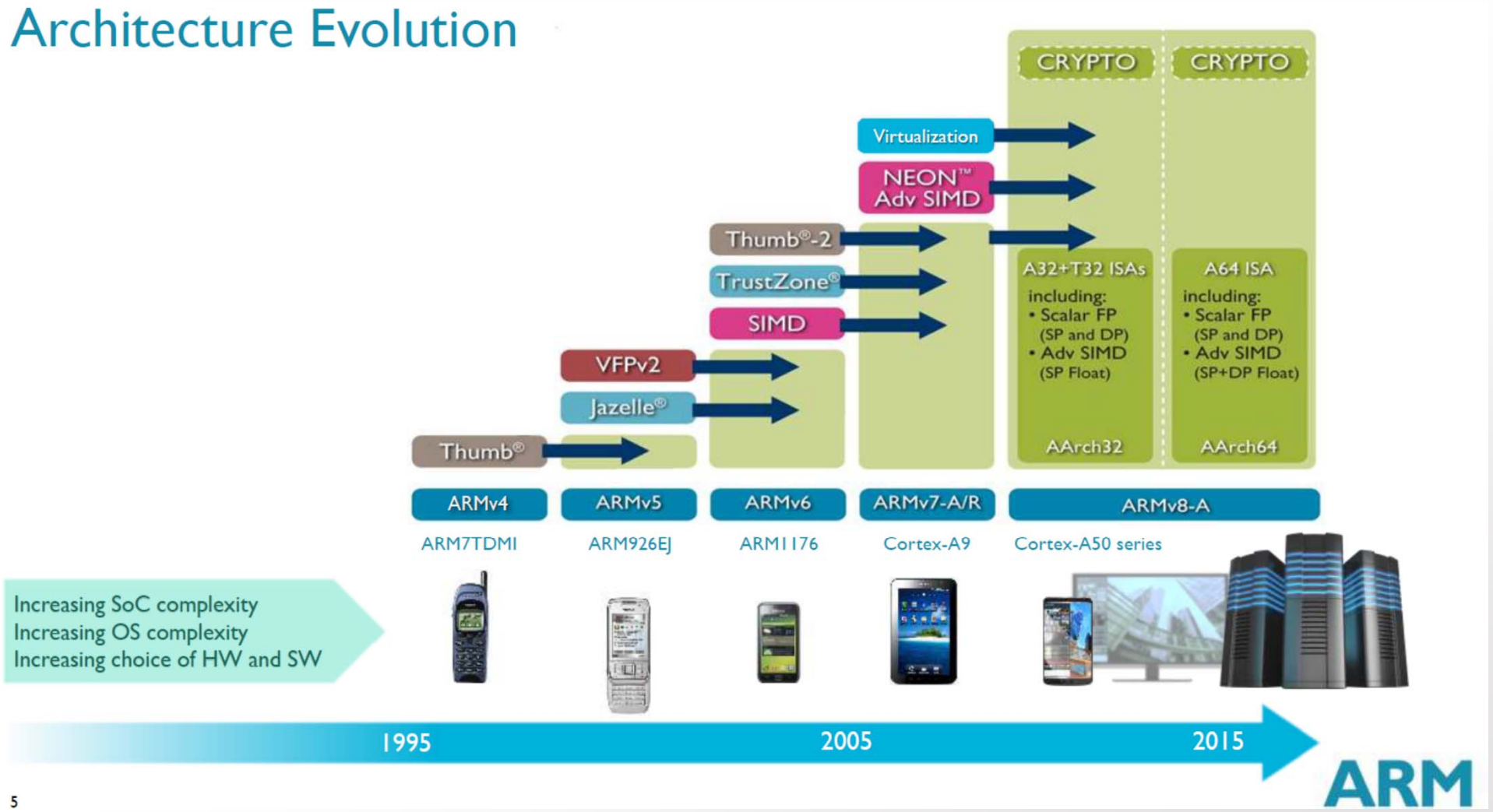
Programmer's Guide for ARMv8-A

http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf

ARMv8 Instruction Set Overview

https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf

Αρχιτεκτονική ARM



http://media.corporate-ir.net/media_files/IROL/19/197211/ARMv8-A%20IR%20webcast%2024_03_2014.pdf

RASPBERRY PI 3 MODEL B

https://www.raspberrypi.org/

https://www.raspberrypi.org/magpi/

[Username@]clusterpi01.in.cs.ucy.ac.cy

SPECIFICATIONS

The Raspberry Pi 3 is the third-generation Raspberry Pi. It replaced the Raspberry Pi 2 Model B in February 2016.

Quad Core 1.2GHz Broadcom BCM2837 **64bit** CPU (**ARM Cortex-A53**)

1GB RAM

BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board

40-pin extended GPIO

4 USB 2 ports

4 Pole stereo output and composite video port

Full size HDMI

CSI camera port for connecting a Raspberry Pi camera

DSI display port for connecting a Raspberry Pi touchscreen display

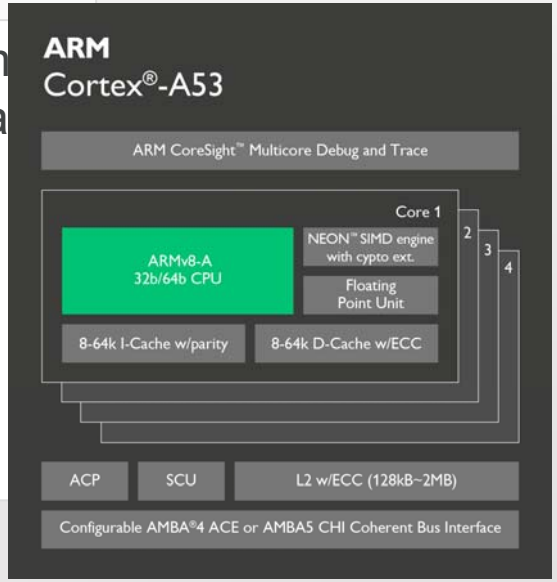
Micro SD port for loading your operating system and storing data

Upgraded switched Micro USB power source up to 2.5A

Cortex-A53

The Cortex-A53 processor is a high efficiency processor that implements the Armv8-A architecture.

Architecture	Armv8-A
Multicore	1-4x Symmetrical Multiprocessing (SMP) within a single processor cluster, and multiple coherent SMP processor clusters through AMBA 4 technology
ISA Support	•AArch32 for full backward compatibility with Arm •AArch64 for 64-bit support and new architectura features •TrustZone security technology •NEON advanced SIMD •DSP & SIMD extensions •VFPv4 floating point •Hardware virtualization support



https://developer.arm.com/products/processors/cortex-a/cortex-a53

Applications

Built as a low-power processor with 64-bit capabilities, the Cortex-A53 processor is applicable in a range of devices requiring high performance in power-constrained environments. Some notable deployments include:

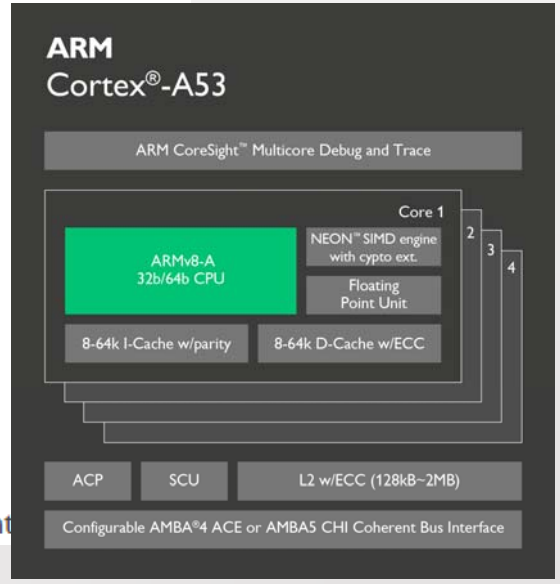
Premier to mid-range smartphones

Aerospace

Networking

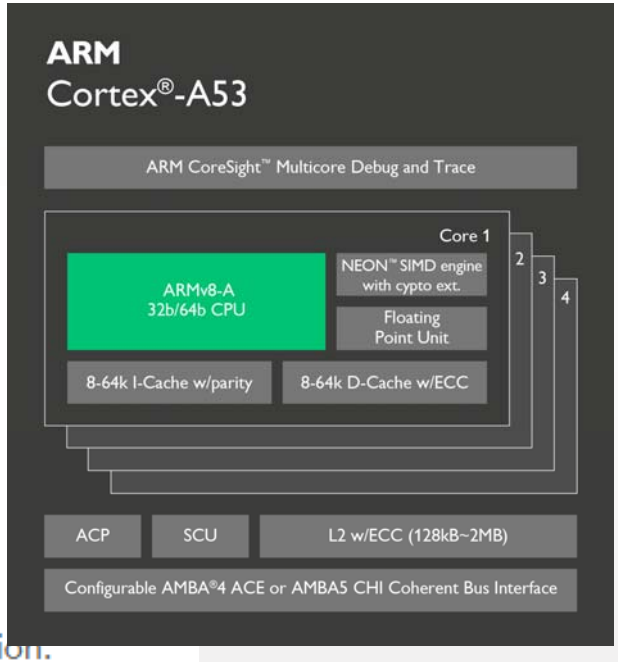
Storage networking (HDD, SDD)

Automotive infotainment



https://developer.arm.com/products/processors/cortex-a/cortex-a53

Cortex-A53



Key features

In-order pipeline

Lower power consumption.

Extensive dual-issue capability

Increased peak instruction throughput via dual instruction decode and execution.

Advanced branch predictor

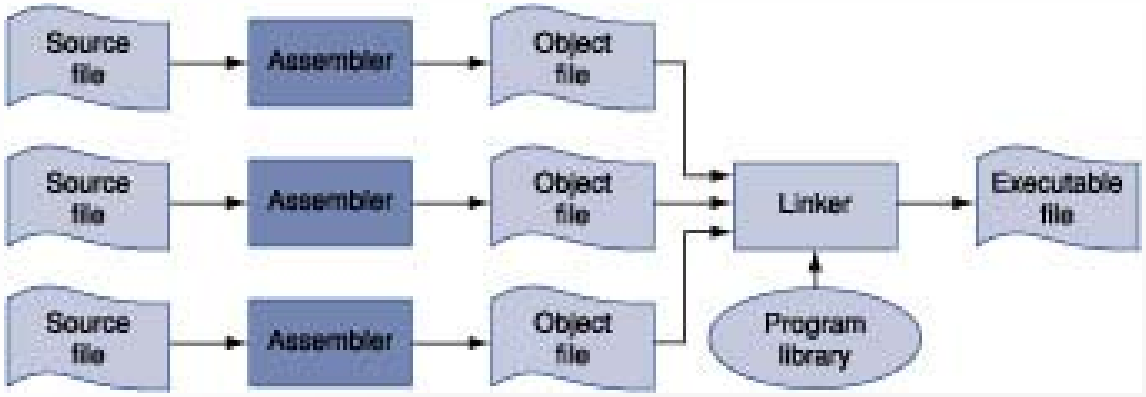
Increased branch hit rate with 6Kb Conditional Predictor and 256 entry indirect predictor.

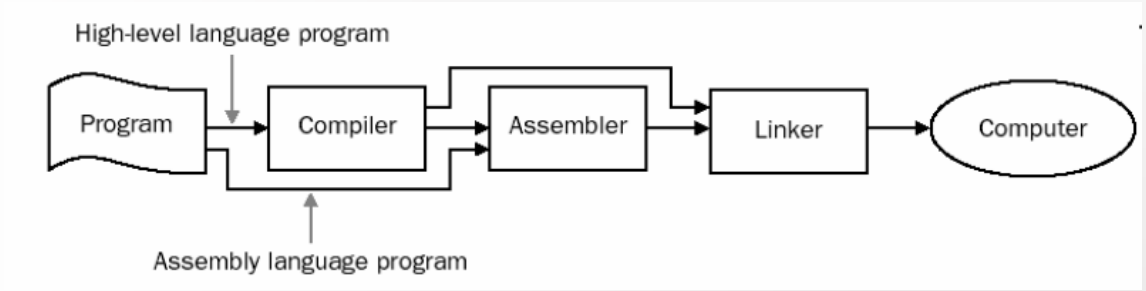
Extensive power-saving features

Hierarchical clock gating, power domains, advanced retention modes.

https://developer.arm.com/products/processors/cortex-a/cortex-a53

Πώς Δημιουργείτε ένα Executable Αρχείο





Source File: Ένα TEXT αρχείο με τον πυγαίο κώδικα.

Assemble: Μεταφράζει τις assembly εντολές σε γλώσσα μηχανής

Linker: Ο Linker παίρνει διάφορα objects files και program library files για να δημιουργήσει το εκτελέσιμο αρχείο (executable file)

Executable File: Είναι το Binary File που αντιγράφεται στην μνήμη του υπολογιστή και εκτελείται.

Assembly with Labels → Assembly → Machine Language

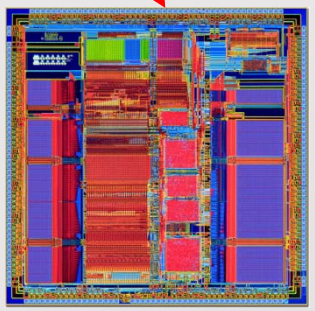
```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

```
addiu   $29, $29, -32
sw      $31, 20($29)
sw      $4,  32($29)
sw      $5,  36($29)
sw      $0,  24($29)
sw      $0,  28($29)
lw      $14, 28($29)
lw      $24, 24($29)
multu   $14, $14
addiu   $8,  $14, 1
slti    $1,  $8, 101
sw      $8,  28($29)
mflo    $15
addu    $25, $24, $15
bne     $1,  $0, -9
sw      $25, 24($29)
lui     $4,  4096
lw      $5,  24($29)
jal     1048812
addiu   $4,  $4, 1072
lw      $31, 20($29)
addiu   $29, $29, 32
jr      $31
move    $2,  $0
```

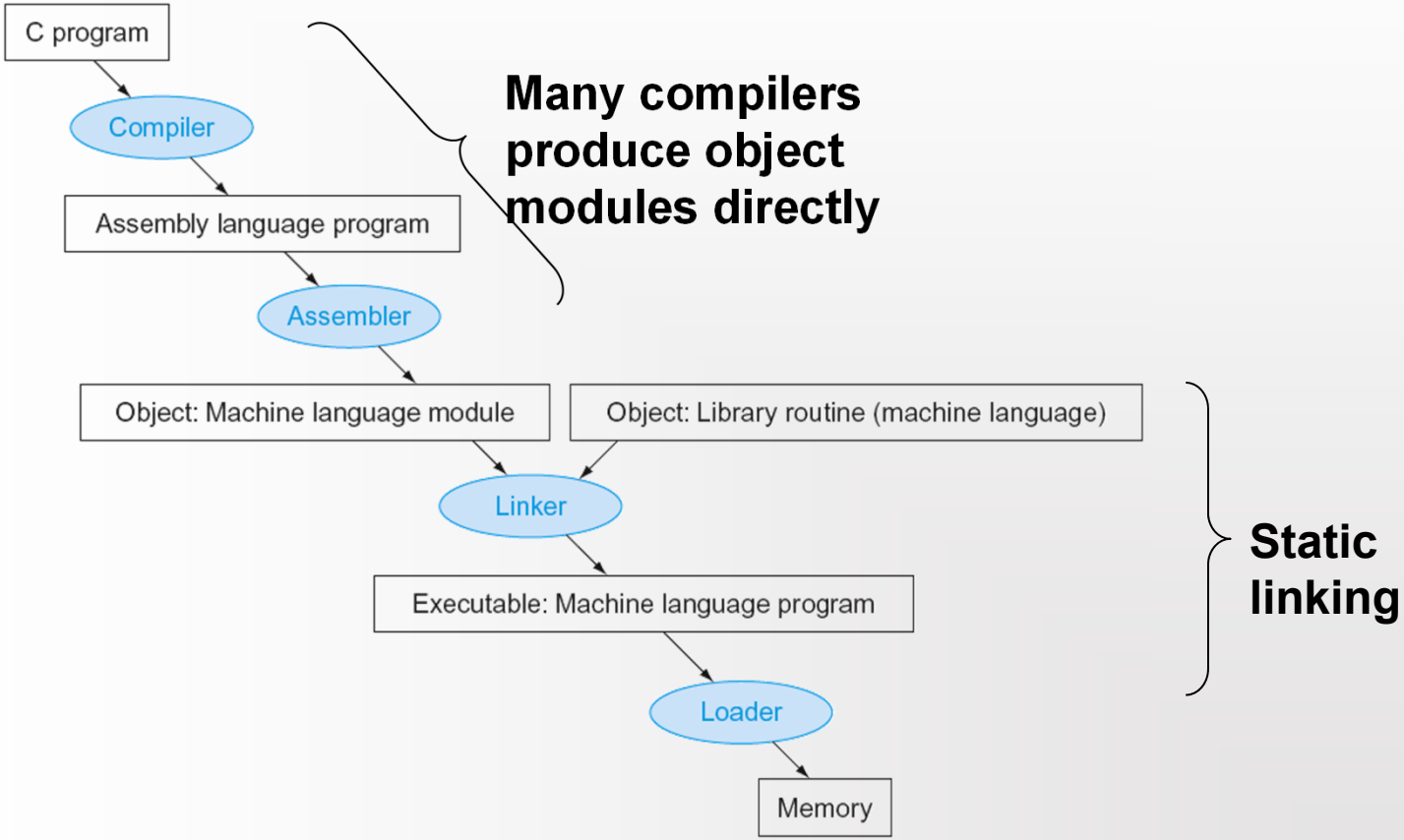
```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

```
   #include <stdio.h>

int
main (int argc, char *argv[])
{
        int i;
        int sum = 0;

        for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
        printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

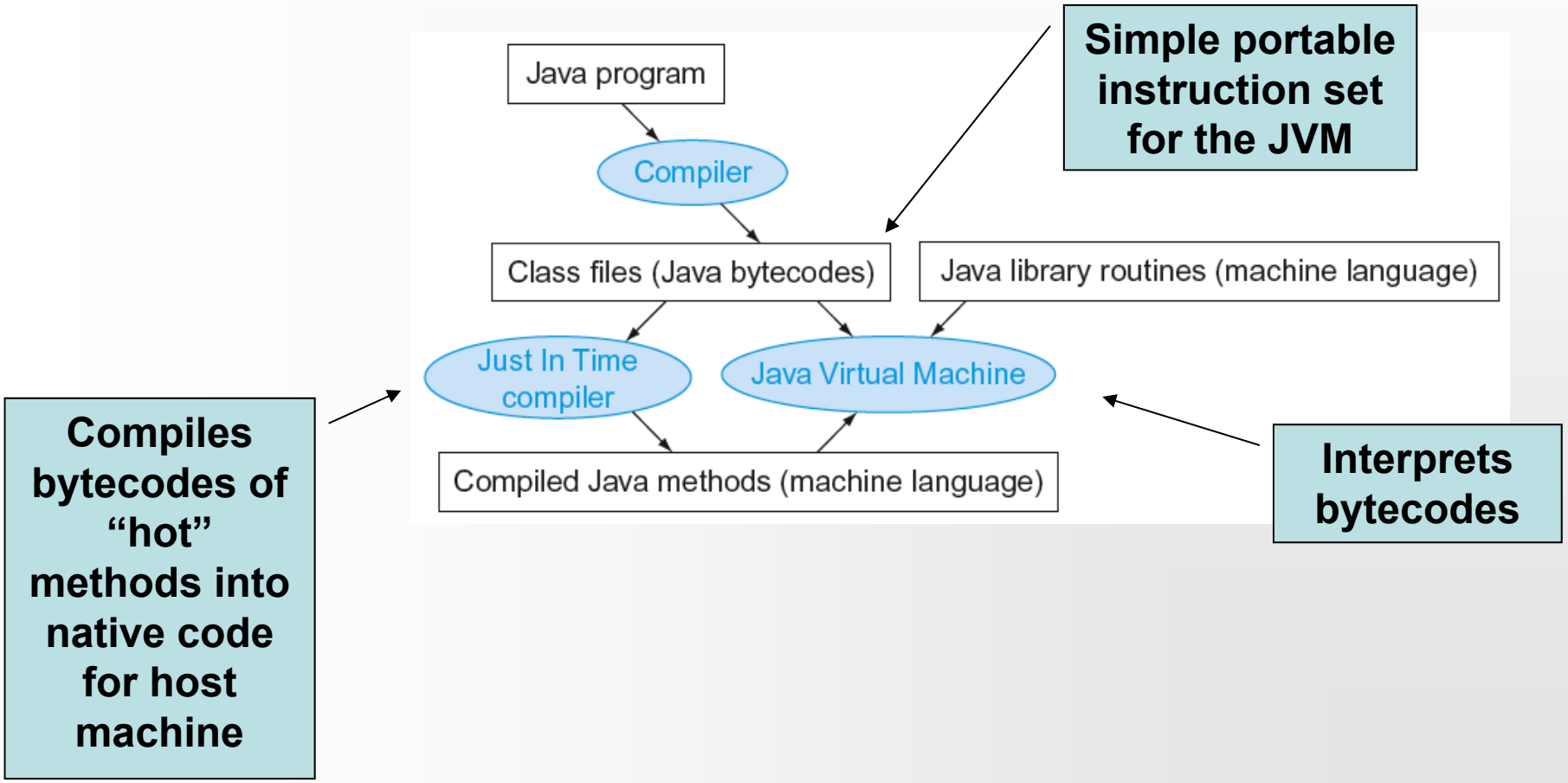
C ή C++



Translation and Startup



Starting Java Applications



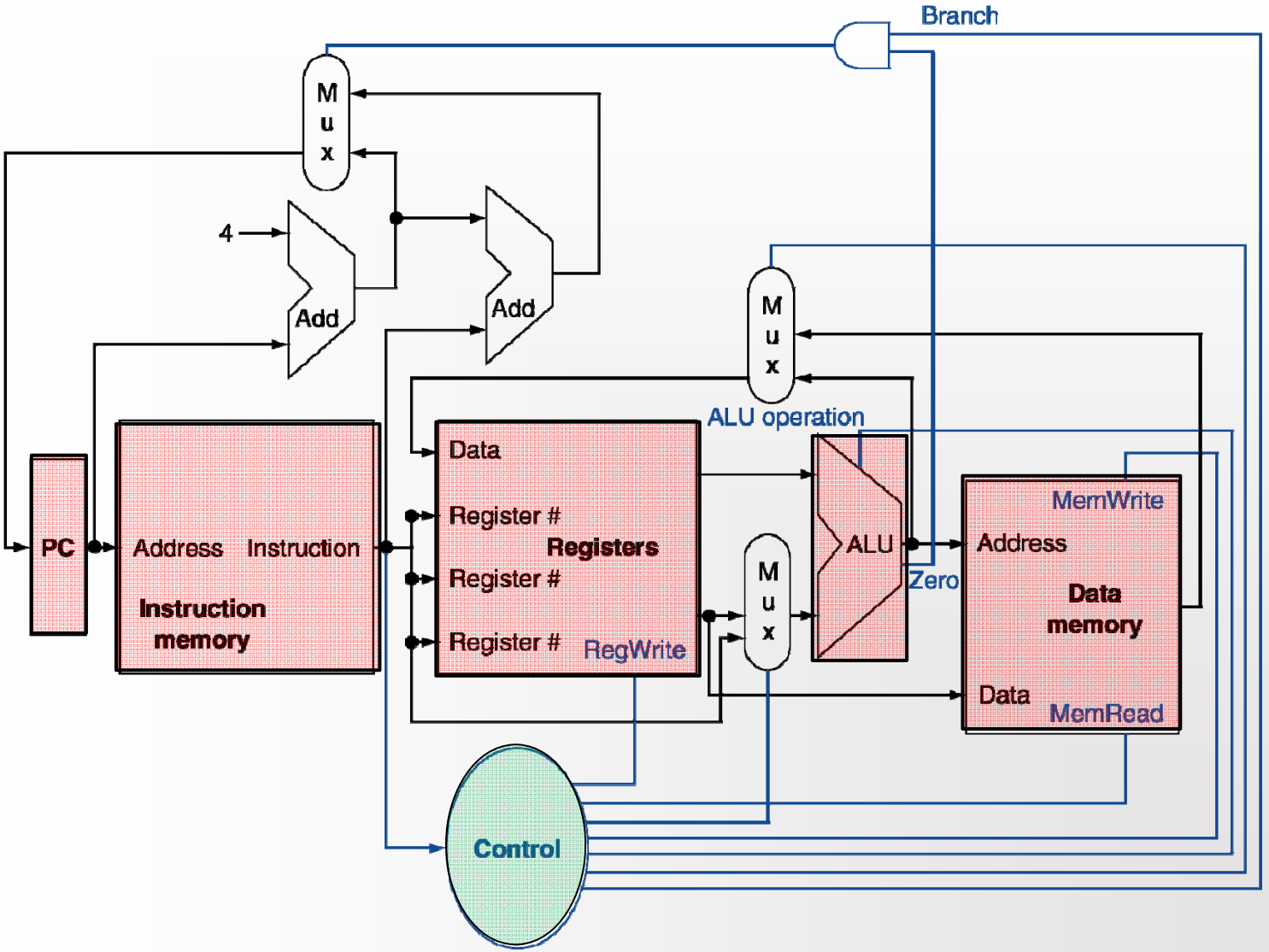
Assembly Language/Η Γλώσσα Μηχανής

Η Συμβολική γλώσσα (Assembly) είναι η συμβολική αντιπροσώπευση της δυαδικής γλώσσας μηχανών (Machine Language). Η Assembly είναι πιο αναγνώσιμη από τη γλώσσα μηχανών επειδή χρησιμοποιεί σύμβολα. Επιπλέον, η Assembly επιτρέπει στους προγραμματιστές να χρησιμοποιούν τις ετικέτες (Labels) για να προσδιορίζουν και να ονομάσουν τις μεταβλητές και τοποθεσίες μνήμης που φυλάσσουν τις οδηγίες ή τα στοιχεία.

Πλεονεκτήματα της Assembly: Το μικρό μέγεθος και η ταχύτητα των Assembly προγραμμάτων είναι τα κύρια πλεονεκτήματα σε σχέση με τις High Level Languages όπως C++ και Java. Αυτό είναι ιδιαίτερα σημαντικά για ενσωματωμένους υπολογιστής (embedded computer) όπου η υπολογιστική ισχύς και η μνήμη είναι περιορισμένη.

Μειονεκτήματα της Assembly: Ίσως το σημαντικό μειονέκτημά του είναι ότι τα προγράμματα που γράφονται στη Assembly είναι συγκεκριμένα για μια μηχανή και πρέπει να ξαναγραφούν συνολικά για να τρέξουν σε μια άλλη αρχιτεκτονική υπολογιστών. Ένα άλλο μειονέκτημα είναι ότι τα προγράμματα Assembly είναι πιο χρονοβόρα στο να γραφτούν από τα ισοδύναμα προγράμματα που γράφονται σε μια υψηλού επιπέδου γλώσσα.

Οργάνωση μιας Κεντρικής Μονάδας Επεξεργασίας

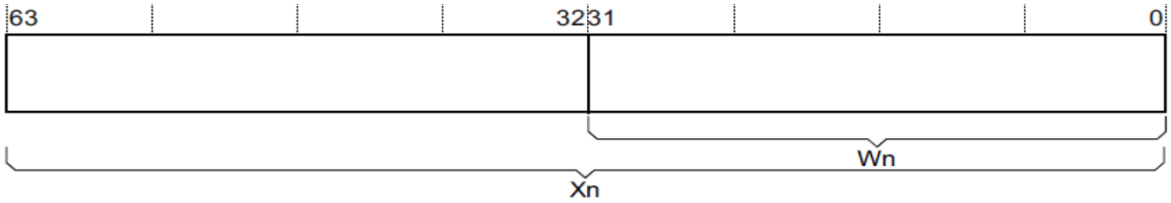


Η αρχιτεκτονική ARMv8 έχει 32 καταχωρητές

Register	Special	Role in the procedure call standard
SP		The Stack Pointer.
r30	LR	The Link Register.
r29	FP	The Frame Pointer
r19…r28		Callee-saved registers
r18		The Platform Register, if needed; otherwise a temporary register. See notes.
r17	IP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r9…r15		Temporary registers
r8		Indirect result location register
r0…r7		Parameter/result registers

Size (bits)	32b	64b
Name	WSP	SP

Size (bits)	32b	64b
Name	WZR	XZR



Size (bits)	32b	64b
Name	Wn	Xn

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

ARMv8 καταχωρητές

- The names `Xn` and `Wn` refer to the same architectural register.
- There is no register named `W31` or `X31`.
- For instruction operands where register 31 in interpreted as the 64-bit *stack pointer*, it is represented by the name `SP`. For operands which do not interpret register 31 as the 64-bit stack pointer this name shall cause an assembler error.
- The name `WSP` represents register 31 as the *stack pointer* in a 32-bit context. It is provided only to allow a valid disassembly, and should not be seen in correctly behaving 64-bit code.
- For instruction operands which interpret register 31 as the *zero register,* it is represented by the name `XZR` in 64-bit contexts, and `WZR` in 32-bit contexts. In operand positions which do not interpret register 31 as the zero register these names shall cause an assembler error.
- Where a mnemonic is overloaded (i.e. can generate different instruction encodings depending on the data size), then an assembler shall determine the precise form of the instruction from the size of the *first* register operand. Usually the other operand registers should match the size of the first operand, but in some cases a register may have a different size (e.g. an address base register is always 64 bits), and a source register may be smaller than the destination if it contains a word, halfword or byte that is being widened by the instruction to 64 bits.
- The architecture does not define a special name for register 30 that reflects its special role as the link register on procedure calls. Such software names may be defined as part of the Procedure Calling Standard.

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

Η αρχιτεκτονική ARMv8 έχει 32 καταχωρητές

The first eight registers, r0-r7, are used to pass argument values into a subroutine and to return result values from a function. They may also be used to hold intermediate values within a routine (but, in general, only *between* subroutine calls).

Registers r16 (IP0) and r17 (IP1) may be used by a linker as a scratch register between a routine and any subroutine it calls (for details, see *§5.3.1.1, Use of IP0 and IP1 by the linker*). They can also be used within a routine to hold intermediate values *between* subroutine calls.

The role of register r18 is platform specific. If a platform ABI has need of a dedicated general purpose register to carry inter-procedural state (for example, the thread context) then it should use this register for that purpose. If the platform ABI has no such requirements, then it should use r18 as an additional temporary register. The platform ABI specification must document the usage for this register.

Note Software developers creating platform-independent code are advised to avoid using r18 if at all possible. Most compilers provide a mechanism to prevent specific registers from being used for general allocation; portable hand-coded assembler should avoid it entirely. It should not be assumed that treating the register as callee-saved will be sufficient to satisfy the requirements of the platform. Virtualization code must, of course, treat the register as they would any other resource provided to the virtual machine.

A subroutine invocation must preserve the contents of the registers r19-r29 and SP.

In all variants of the procedure call standard, registers r16, r17, r29 and r30 have special roles. In these roles they are labeled IP0, IP1, FP and LR when being used for holding addresses (that is, the special name implies accessing the register as a 64-bit entity).

Note The special register names (IP0, IP1, FP and LR) should be used only in the context in which they are special. It is recommended that disassemblers always use the architectural names for the registers.

The NZCV register is a global condition flag register with the following properties:

- The N, Z, C and V flags are undefined on entry to and return from a public interface.

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

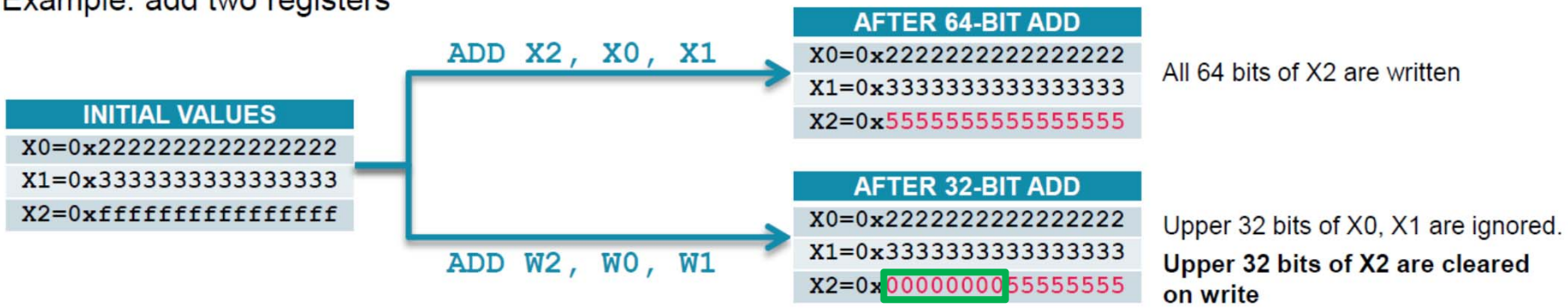
General purpose and dedicated registers

- 31 general purpose + 2 dedicated registers, each 64-bit wide

same register

	GENERAL PURPOSE	DEDICATED	NOTES
Architectural names	R0, R1, R2, …, R30	SP, ZR	• names rarely used in practice
64-bit views	X0, X1, X2, …, X30	SP, XZR	• "x" stands for e**X**tended word • all 64 bits are "used"
32-bit views	W0, W1, W2, …, W30	WSP, WZR	• "w" stands for **W**ord • only **bottom** 32 bits are "used"

- Example: add two registers



Special registers in AArch64

Integer Class	
W	32-bit integer
X	64-bit integer

Table 4-1 Special registers in AArch64

Name	Size	Description
WZR	32 bits	Zero register
XZR	64 bits	Zero register
WSP	32 bits	Current stack pointer
SP	64 bits	Current stack pointer
PC	64 bits	Program counter

There is no register called X31 or W31. Many instructions are encoded such that the number 31 represents the zero register, ZR (WZR/XZR). There is also a restricted group of instructions where one or more of the arguments are encoded such that number 31 represents the *Stack Pointer* (SP).

The Program Counter (PC)

The current **Program Counter (PC)** cannot be referred to by number as if part of the general register file and therefore cannot be used as the source or destination of arithmetic instructions, or as the base, index or transfer register of load/store instructions. The only instructions which read the PC are those whose function is to compute a PC-relative address (ADR, ADRP, literal load, and direct branches), and the branch-and-link instructions which store it in the link register (BL and BLR). The only way to modify the Program Counter is using explicit control flow instructions: conditional branch, unconditional branch, exception generation and exception return instructions.

Where the PC is read by an instruction to compute a PC-relative address, then its value is the address of the instruction, i.e. unlike A32 and T32 there is no implied offset of 4 or 8 bytes.

Saved Process Status Register

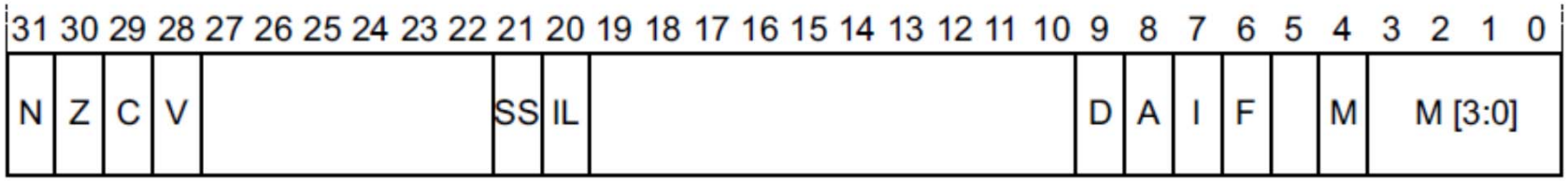


Figure 4-4 SPSR

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).

Endianness

There are two basic ways of viewing bytes in memory, either as ***Little-Endian*** **(LE)** or ***Big-Endian*** **(BE).** On big-endian machines, the most significant byte of an object in memory is stored at the lowest address, that is the address closest to zero. On little-endian machines, the least significant byte is stored at the lowest address. The term byte-ordering can also be used rather than endianness.

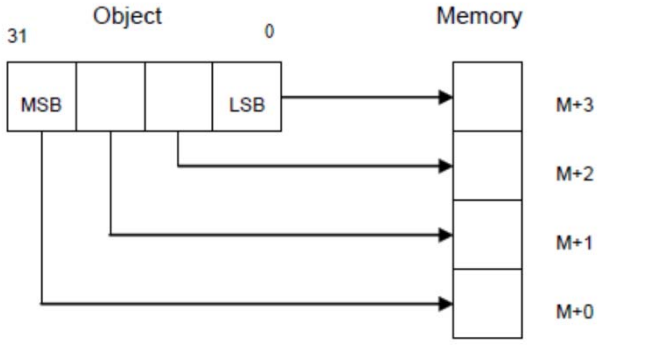


Figure 1, Memory layout of big-endian data object

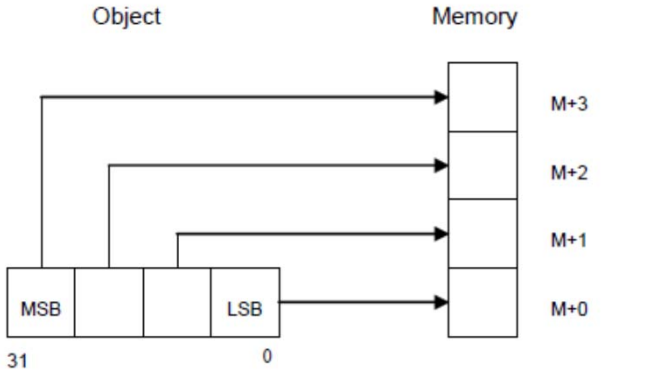
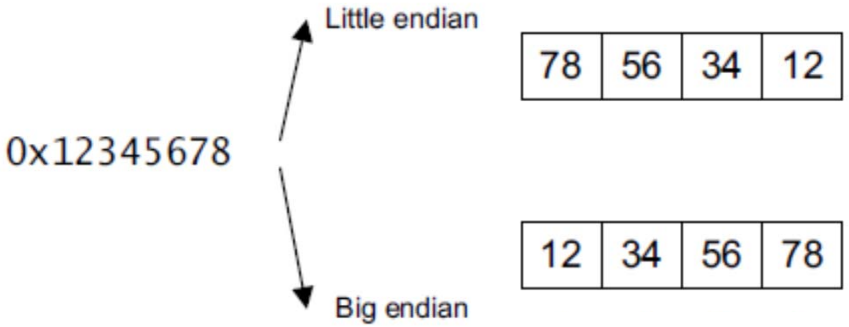


Figure 2, Memory layout of little-endian data object



https://en.wikipedia.org/wiki/Endianness

Program counter and stack pointer alignment

PC alignment checking

PC alignment checking generates a PC alignment fault exception associated with the instruction fetch if, in AArch64 state, there is an attempt to architecturally execute an instruction that was fetched with a misaligned PC. **A misaligned PC is when bits[1:0] of the PC are not 0b00.**

SP alignment checking

A misaligned stack pointer is where bits[3:0] of the stack pointer are not 0b0000, when the stack pointer is used as the base address of the calculation, regardless of any offset applied by the instruction. In other words **sp%16 = 0**

Δομή ενός προγράμματος σε Assembly

```
1  //*****************************************************************
2  // Program Name : Basic program to print "Hello Wolrd!!!\n" on the Screen
3  // Programmer   : Petros Panayi Stud. ID: 000000
4  // Date  Modif. : 1 Sep 2017
5  //*****************************************************************
6  // Comments: A simple program that prints a string message on the screen
7  //                        and the terminates
8  //*****************************************************************
9          .data                                  // The Start of the .data Segment
10 Hello_Label: .string    "Hello Wolrd!!!\n"
11
12
13 //*****************************************************************

14         .text                              // The Start of the .test Segment
15         .global main
16 main:
17         stp     x29, x30, [sp, -16]!
18         add     x29, sp, 0
19         adrp    x0, Hello_Label
20         add     x0, x0, :lo12:Hello_Label
21         bl      printf
22         ldp     x29, x30, [sp], 16
23         ret
24 //*****************************************************************
```

ftp://ftp.vim.org/pub/vim/doc/book/vimbook-OPL.pdf

Σύνταξη για τις οδηγίες Συμβολικής γλώσσας ARMv8

```
{label:*} {opcode {dest{, source1{, source2{, source3}}}}}
```

{ } Any item bracketed by curly braces { and } is optional. A description of the item and of how its presence or absence affects the instruction is normally supplied by subsequent text. In some cases curly braces are actual symbols in the syntax, for example surrounding a register list, and such cases will be called out in the surrounding text.

- Τα κόμματα είναι υποχρεωτικά.
- Στη Συμβολική γλώσσα (Assembly) ARMv8, οτιδήποτε σε μια γραμμή μετά από το σημάδι (//) είναι σχόλιο.

S.O.S. Όταν εκτυπώνουμε τα reports ο κώδικας και τα σχόλια δεν πρέπει να ξεπερνούν την μία γραμμή.

An A64 assembler will recognise both upper and lower-case variants of instruction mnemonics and register names, but not mixed case. An A64 disassembler may output either upper or lower-case mnemonics and register names. The case of program and data labels **is** significant.

The A64 assembly language does not require the '#' symbol to introduce immediate values, though an assembler must allow it. An A64 disassembler shall always output a '#' before an immediate value for readability.

Χρήσιμες Εντολές

gcc -g example1.s [https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html]

objdump -d ./a.out | less

[https://sourceware.org/binutils/docs/binutils/objdump.html]

```
0000000004005b0 <main>:
  4005b0:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  4005b4:       910003fd        mov     x29, sp
  4005b8:       90000080        adrp    x0, 410000 <__FRAME_END__+0xf850>
  4005bc:       9127b000        add     x0, x0, #0x9ec
  4005c0:       97ffffa8        bl      400460 <printf@plt>
  4005c4:       a8c17bfd        ldp     x29, x30, [sp],#16
  4005c8:       d65f03c0        ret
  4005cc:       00000000        .inst   0x00000000 ; undefined
```

hexdump -C ./a.out | less

[http://www.linuxdevcenter.com/cmd/cmd.csp?path=h/hexdump]

```
000009d0  10 04 40 00 00 00 00 00  10 04 40 00 00 00 00 00  |..@.......@.....|
000009e0  10 04 40 00 00 00 00 00  00 00 00 00 48 65 6c 6c  |..@.........Hell|
000009f0  6f 20 57 6f 6c 72 64 21  21 21 0a 00 47 43 43 3a  |o Wolrd!!!..GCC:|
00000a00  20 28 47 4e 55 29 20 36  2e 32 2e 31 20 32 30 31  | (GNU) 6.2.1 201|
00000a10  36 30 39 31 36 20 28 52  65 64 20 48 61 74 20 36  |60916 (Red Hat 6|
00000a20  2e 32 2e 31 2d 32 29 00  00 00 00 00 00 00 00 00  |.2.1-2).........|
00000a30  2c 00 00 00 02 00 00 00  00 00 08 00 00 00 00 00  |,...............|
00000a40  b0 05 40 00 00 00 00 00  1c 00 00 00 00 00 00 00  |..@.............|
```