ARQUITECTURA DE COMPUTADORAS I

Ing. Gamboa Leonardo

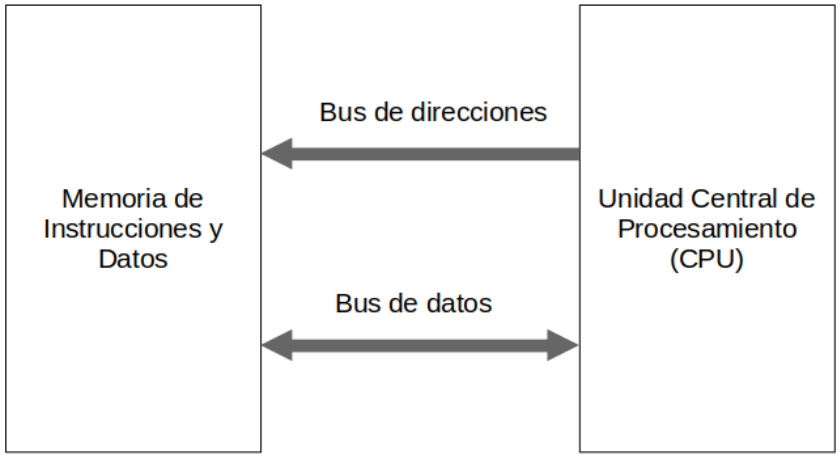
Carrera: Ingeniería Informática

2025

Unidad 1: Introducción al Microprocesador

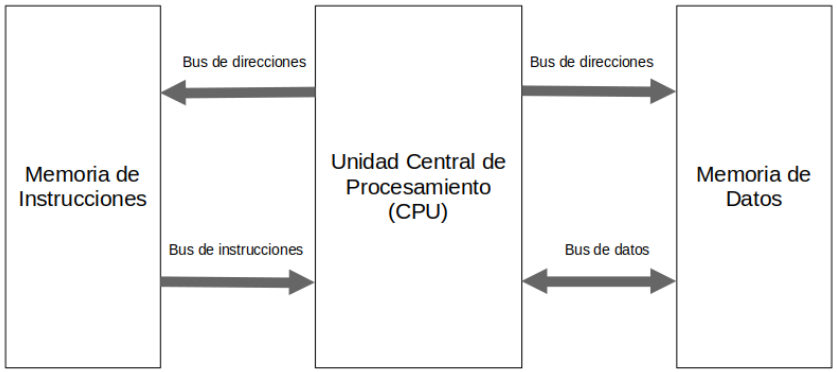
- Repaso de Arquitecturas
- Microprocesadores ARM
- Arquitectura básica del microprocesador ARMv8.
- ISA (Instruction Set Architecture) LEGv8
- Operaciones aritméticas y lógicas en LEGv8
- La memoria y el microprocesador.
- Formato de Instrucciones LEGv8
- Emulador Qemu.
- Bibliografía.

Arquitectura Von Neumann



- Memoria única para instrucciones y datos.
- Unidad de Control, ALU, Registro, Bus común.
- Ejecución secuencial de instrucciones.
- Ventajas: simplicidad y flexibilidad.
- Desventajas: cuello de botella del bus (von Neumann bottleneck).

Arquitectura Harvard



- Memorias separadas para instrucciones y datos.
- Acceso concurrente a instrucciones y datos.
- Unidad de Control, ALU, registros separados.
- Mejora el rendimiento al eliminar cuello de botella.
- Más compleja en hardware.

Comparación: Von Neumann vs Harvard

Característica	Von Neumann	Harvard
Memoria	Única (instrucciones y datos)	Separada
Acceso a memoria	Secuencial	Concurrente
Complejidad hardware	Menor	Mayor
Rendimiento	Limitado por bus	Mejor rendimiento
Uso típico	Computadoras generales	Sistemas embebidos

Arquitectura RISC (Reduced Instruction Set Computer)

- **Instrucciones simples**, formato fijo. Esto facilita la decodificación rápida y sencilla.
- **Ejecución en un ciclo**. Cada instrucción se diseña para ser ejecutada en un único ciclo de reloj o en un número reducido, lo que mejora la velocidad y permite mayor uso de pipeline.
- **Pipelining eficiente**. La uniformidad y simplicidad de las instrucciones permiten que el procesador se organice en etapas (fetch, decode, execute, etc.), procesando varias instrucciones simultáneamente.
- **Uso intensivo de registros**. Para minimizar accesos a memoria, RISC dispone de un gran banco de registros, y se usa un modelo load/store donde solo instrucciones explícitas acceden a memoria.
- Ventajas: *Hardware simple, alta eficiencia energética, excelente para dispositivos móviles y embebidos*.
- Desventajas: *El código puede ser más largo, ya que se necesitan más instrucciones para realizar tareas complejas*.
- *Ejemplos: ARM, MIPS, SPARC, RISC-V*.

Arquitectura CISC (Complex Instruction Set Computer)

- **Conjunto de instrucciones amplio y complejo**: CISC tiene cientos de instrucciones con diferentes longitudes y formatos, muchas capaces de realizar tareas complejas en una sola instrucción.
- **Instrucciones multifase** y formatos variados: Las instrucciones pueden requerir varios ciclos de reloj para su ejecución debido a su complejidad (carga, cálculo, y almacenamiento en una sola instrucción).
- **Microprogramación**: La unidad de control descentralizada usa microprogramas para traducir instrucciones complejas en operaciones básicas internas.
- Ventajas: Código compacto. Al realizar más trabajo por instrucción, el código tiende a ser más compacto, lo que reduce el uso de memoria.
- Desventajas: Hardware más complejo, mayor consumo energético, decodificación complicada que puede causar pérdidas de rendimiento.
- *Ejemplos: Intel x86, AMD64*.

Comparación: Arquitectura RISC vs CISC

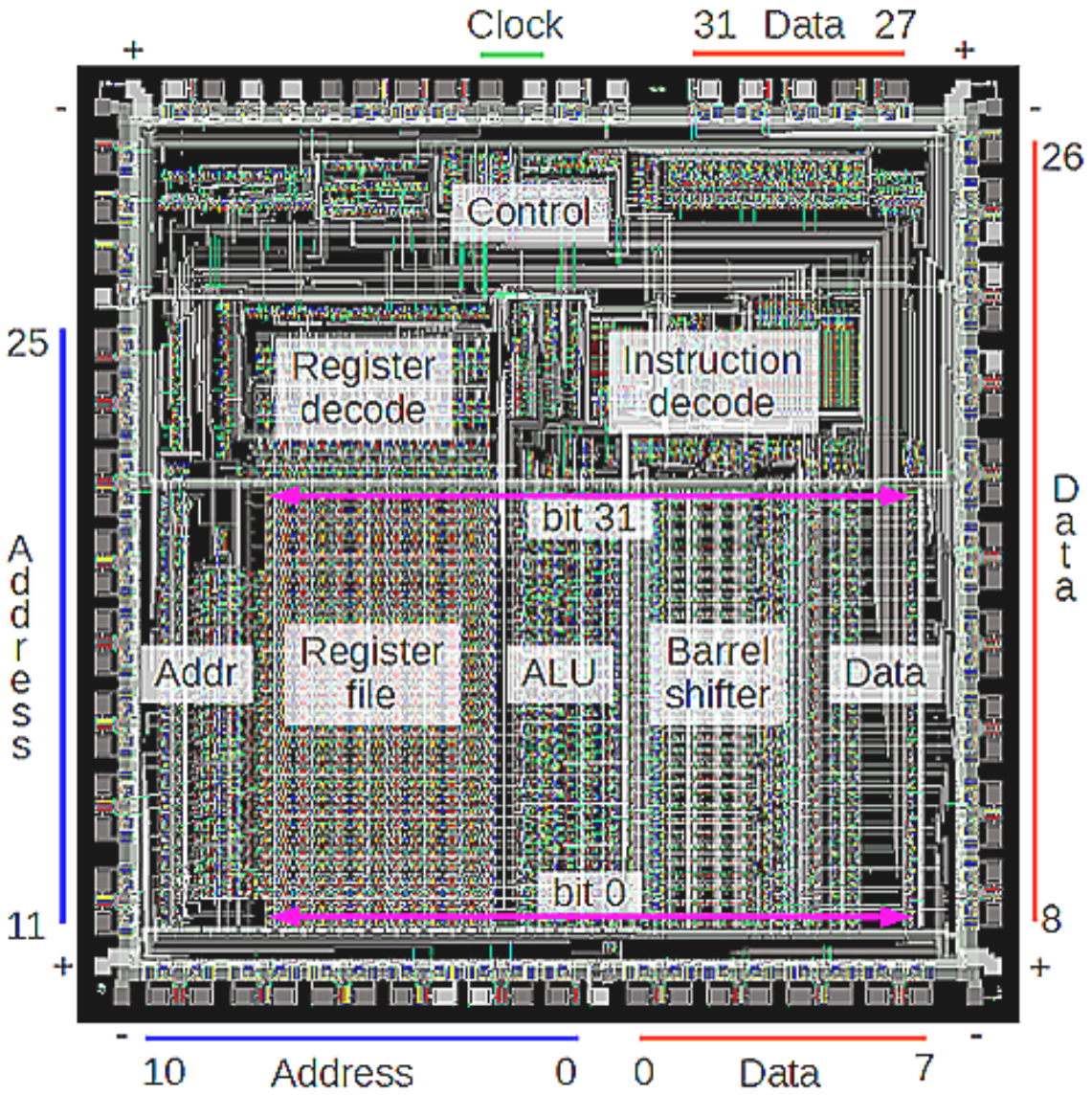
Características	**Arquitectura RISC**	**Arquitectura CISC**
Conjunto de instrucciones	Reducido, simple, número limitado	Amplio, complejo, muchas instrucciones
Tamaño de instrucción	Fijo, uniforme	Variable, múltiple formatos
Complejidad de instrucciones	Baja, una operación por instrucción	Alta, instrucciones pueden realizar múltiples operaciones
Ciclos por instrucción	Generalmente 1 ciclo	Múltiples ciclos
Decodificación	Simple y rápida	Compleja y lenta
Uso de registros	Gran banco de registros para minimizar acceso a memoria	Menor cantidad de registros; más acceso a memoria
Pipelining	Fácil implementación y alta eficiencia	Difícil implementación por instrucciones complejas
Tamaño del código	Mayor debido a instrucciones simples	Más compacto al hacer más por instrucción
Consumo energético	Bajo, por simplicidad y eficiencia	Más alto, debido a complejidad de hardware
Aplicaciones típicas	Sistemas embebidos, móviles, dispositivos de bajo consumo	PCs, servidores, aplicaciones generales

Microprocesadores ARM

- Desarrollada por empresa ARM Holdings.
- Microprocesadores Arquitectura RISC de 64 bits (v8) /32 bits (vesriones anteriores).
- El negocio principal de ARM Holdings es la venta de núcleos IP (propiedad intelectual), estas IP core son licenciadas por otras empresas (Qualcomm,Samsung, entre otras) para diseñar sus microcontroladores y CPUs basados en este núcleo, los cuales posteriormente se envían a frabricar.
- Su caracteristica principal es la Eficiencia energética.
- Existe una gran cantidad de dispositivos que utilizan la arquitectura ARM (sistemas embebidos, smartphones, tablets, etc).

https://www.arm.com/

Microprocesador ARM1 Layout



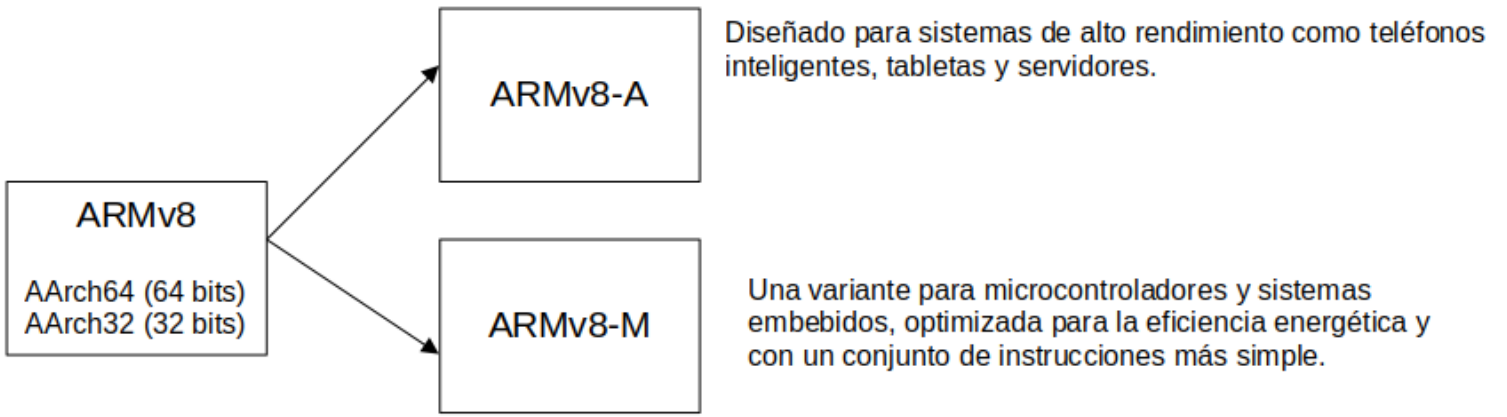
Evolución de la familia de microprocesadores ARM

Arquitectura	Familia
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	StrongARM, ARM7TDMI
ARMv5	ARM7EJ, ARM9E,XScale
ARMv6	ARM11, ARM Cortex-M
ARMv7	ARM Cortex-A, ARM Cortex-R
ARMv8	ARM Cortex-A50

- ARM1: primer diseño (1985).
- ARM2 y ARM3: mejoras en potencia y rendimiento.
- ARM7: popular en móviles (1994).
- ARM9, ARM11: mayor rendimiento y multimedia.
- ARM Cortex-A: procesadores de alto rendimiento.
- ARM Cortex-M: para sistemas embebidos.
- ARMv8: arquitectura de 64 bits.

Detalles de la Familia ARMv8

- Introducción arquitectura de 64 bits (AArch64).
- Soporta ejecución en modo 32 bits (AArch32).
- Mejoras en rendimiento y eficiencia energética.
- Nuevas instrucciones criptográficas.
- Compatible con versiones anteriores.
- Ejecución multinivel (EL0-EL3): Define cuatro niveles de excepción (EL0 a EL3) para la gestión de la seguridad y la virtualización en un sistema
- Usado en smartphones, servidores y dispositivos IoT.

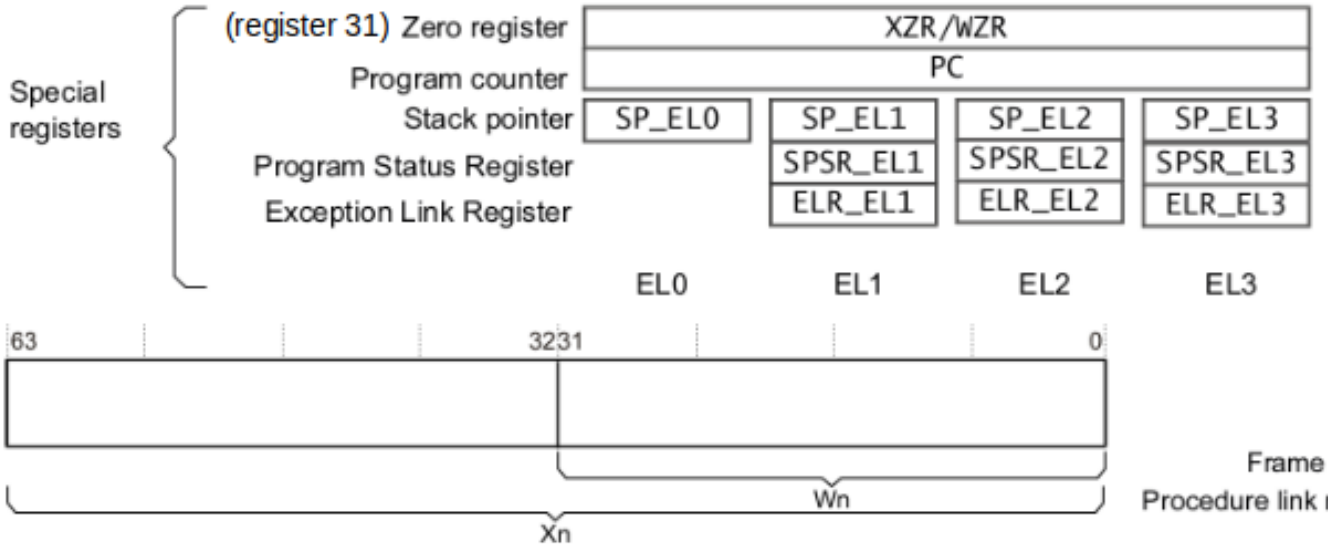


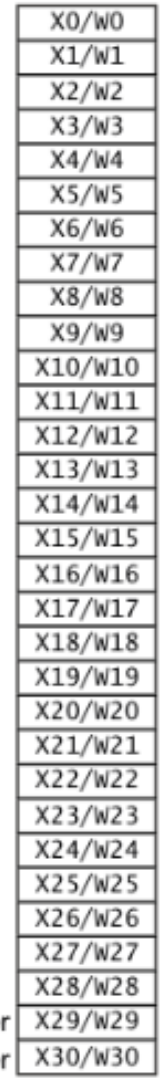
Arquitectura Básica del Microprocesador ARMv8

- Arquitectura RISC de 64 bits diseñada para eficiencia y rendimiento.
- LEGv8 es un subconjunto didáctico de ARMv8-A utilizado para enseñanza.
- Cuenta con 31 registros de 64 bits de propósito general (X0 a X30).
- Registro de solo lectura XZR que siempre vale cero.
- Contador de programa (PC) y registro de estado (PSR).
- Unidad lógica y aritmética (ALU) para operaciones básicas.
- Pipeline simple con etapas de búsqueda, decodificación, ejecución, etc.

Arquitectura ARMv8

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





Arquitectura ARMv8-LEGv8

- El uso de este set de instrucciones permite realizar simulaciones en arquitecturas reales, utilizando QEMU.
- Por razones pedagógicas se propone la utilización de LEGv8. Es un ISA que consiste en un sub-set del ARMv8.
- Tanto ARMv8 como LEGv8 son de 64 bits. Tienen instrucciones de 32 bits y 32 registros de propósito general (X0 - X30, XZR), todos de 64 bits.

Registros LEGv8 (ver greencard para mayor detalle.)

- X0 – X7: procedure arguments/results.
- X8: indirect result location register.
- X9 – X15: temporaries.
- X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register.
- X18: platform register for platform independent code; otherwise a temporary register.
- X19 – X27: saved.
- X28 (SP): stack pointer.
- X29 (FP): frame pointer.
- X30 (LR): link register (return address).
- XZR (register 31): the constant value 0.

ISA (Instruction Set Architecture) LEGv8

El **ISA** de un procesador es el conjunto de instrucciones y reglas que define cómo el software se comunica con el hardware. Especifica aspectos clave como:

- **Conjunto de instrucciones**: Operaciones que el procesador puede ejecutar (aritméticas, lógicas, control de flujo, carga/almacenamiento, etc.).
- **Registros**: Cantidad y tipo de registros disponibles (propósito general, especiales, etc).
- **Modos de direccionamiento**: Cómo se accede a la memoria y se interpretan las direcciones.
- **Formato de instrucciones**: Estructura binaria de cada instrucción (longitud, operandos, codificación).
- **Modelo de ejecución**: Cómo se manejan las instrucciones dentro del pipeline del procesador.

Sintaxis de assembler LEGv8

Supongamos una operación arimética simple:

```
1    ADD X1,X2,X3
```

- ADD: Mnemónico de la operación a realizar
- X2: Registro primer operando
- X3: Registro segundo operando
- X1: Registro destino

Cada instrucción tiene exactamente tres operandos. No más, no menos.
(ver greencard)

Operaciones aritméticas en LEGv8

structionExample		Meaning	Comments
add	`ADD X1, X2, X3`	`X1 = X2 + X3`	Three register operands
subtract	`SUB X1, X2, X3`	`X1 = X2 − X3`	Three register operands
add immediate	`ADDI X1, X2, 20`	`X1 = X2 + 20`	Used to add constants
subtract immediate	`SUBI X1, X2, 20`	`X1 = X2 − 20`	Used to subtract constants
add and set flags	`ADDS X1, X2, X3`	`X1 = X2 + X3`	Add, set condition codes
subtract and set flags	`SUBS X1, X2, X3`	`X1 = X2 − X3`	Subtract, set condition codes
add immediate and set flags	`ADDIS X1, X2, 20`	`X1 = X2 + 20`	Add constant, set condition codes
subtract immediate and set flags	`SUBIS X1, X2, 20`	`X1 = X2 − 20`	Subtract constant, set condition codes

Operaciones lógicas en LEGv8

and	`AND X1, X2, X3`	`X1 = X2 & X3`	Three reg. operands; bit-by-bit AND
inclusive or	`ORR X1, X2, X3`	`X1 = X2 | X3`	Three reg. operands; bit-by-bit OR
exclusive or	`EOR X1, X2, X3`	`X1 = X2 ^ X3`	Three reg. operands; bit-by-bit XOR
and immediate	`ANDI X1, X2, 20`	`X1 = X2 & 20`	Bit-by-bit AND reg. with constant
inclusive or immediate	`ORRI X1, X2, 20`	`X1 = X2 | 20`	Bit-by-bit OR reg. with constant
exclusive or immediate	`EORI X1, X2, 20`	`X1 = X2 ^ 20`	Bit-by-bit XOR reg. with constant
logical shift left	`LSL X1, X2, 10`	`X1 = X2 << 10`	Shift left by constant
logical shift right	`LSR X1, X2, 10`	`X1 = X2 >> 10`	Shift right by constant

Ejemplo 1

1) Dadas las siguientes sentencias en C, escribir la secuencia mínima de código assembler LEGv8, asumiendo que f,g,h e i se almacenan en X0, X1, X2, X3 y X4 respectivamente.

1. f = g + h + i + j;
2. f = g + (h + 5);
3. f = (g + h) - (g + h);

$$f \rightarrow X0$$
$$g \rightarrow X1$$
$$h \rightarrow X2$$
$$i \rightarrow X3$$
$$j \rightarrow X4$$

Ejemplo 1

1. f = g + h + i + j;

```
1 ADD X9, X1, X2
2 ADD X10, X3, X4
3 ADD X0, X9, X10
```

¿Es posible escribir de otra manera en assembler la misma instrucción en C?

2. f = g + (h + 5);

```
1 ADDI X0, X2, #5
2 ADD X0, X0, X1
```

3. f = (g + h) - (g + h);

Código sin optimizar

```
1     ADD X10, X1, X2
2     ADD X11, X1, X2
3     SUB X0, x10, x11
```

Código optimizado

```
1     ADD X0, XZR, XZR
```

Ejercitación 1

A) Dadas las siguientes sentencias en C, escribir la secuencia mínima de código assembler LEGv8 asumiendo que f y g se asignan en los registros X0 y X1 respectivamente.

1. f = -g -f;
2. f = g + (-f - 5);

B) Dadas las siguientes sentencias en assembler LEGv8, escribir la secuencia mínima de código C asumiendo que los registros X0 , X1 y X2 respectivamente.

1. sentencia en assembler legv8

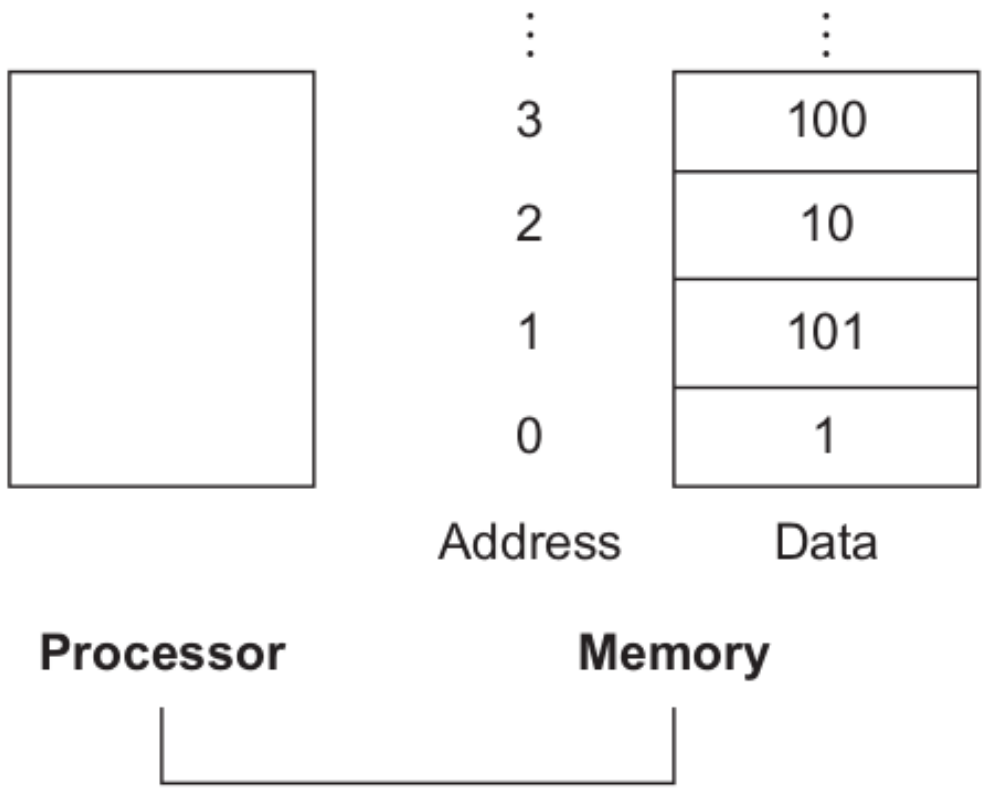
```
1     SUB   X1, XZR, X1
2     ADD   X0, X1, X2
```

2. sentencia en assembler legv8

```
1     ADDI  X2, X0, \#1
2     SUB   X0, X1, X2
```

Accediendo a memoria

La memoria es un gran array unidimensional, donde la dirección actúa como índice de ese array, comenzando en 0. Por ejemplo, en la figura la dirección del tercer elemento es 2 y el contenido o valor de la memoria es 10.



Accediendo a memoria

Load: Permite almacenar en los registros información proveniente desde la memoria.

```
1 LDUR X9 , [ X22, # 8 ]
```

- **LDUR**: operación
- X9: registro destino
- X22: registro dirección de memoria
- #8: offset de memoria

Resultado: X9 = *(X22 + 8);

Store: Permite almacenar en la memoria el contenido de los registros.

```
1     STUR X9 , [ X22, # 8 ]
```

- **STUR**: operación
- X9: registro a guardar en la memoria
- X22: registro dirección de memoria base
- #8: offset de memoria

Resultado: *(X22 + 8)=X9;

Accediendo a memoria

En LEGv8 se considera que:

- byte: 8bits
- halfword: 16bits
- word: 32bits
- doubleword: 64bits

- LDURB/STURB: acceso a memoria de a bytes
- LDUR/STUR: acceso a memoria de a doubleword
- LDUW/STUW: acceso a memoria de a word
- LDUH/STUH: acceso a memoria de a halfword

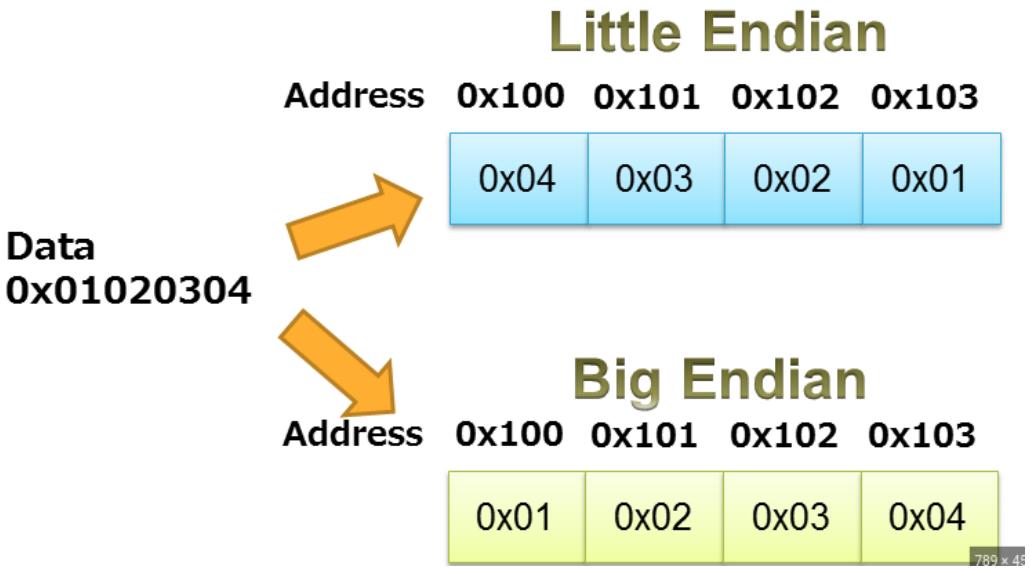
En LEGv8 la memoria se direcciona de byte.

```
1 A[12] = h + A [8] ;
```

```
1 LDUR X9 , [X22, #64]
2 ADD X9 , X21 , X9
3 X9 , [X22, #96]
```

Accediendo a memoria

Endianess
Los procesadores se dividen en las que consideran que la ubicación del byte mas significativo está a la izquierda (big-endian) o a la derecha (little-endian)



Ejercitación

I) Escribir la secuencia mínima de código assembler LEGv8 asumiendo que f,g, i y j se asignan en los registros X0, X1, X2 y X3 respectivamente, y que la dirección base de los arreglos A y B se almacenan en los registros X6 y X7 respectivamente.

1. f = -g - A[4];
2. B[8] = A[i -j];

Instrucciones en LEGv8

Las instrucciones se codifican en binario, lo que se conoce como código de máquina.
Las instrucciones de LEGv8 se codifican como palabras de instrucción de 32 bits.

- Tipo R (Register): Instrucciones aritmético-lógicas entre registros. Ej:

```
1 ADD X0, X1, X2
```

- Tipo I (Immediate): Instrucciones con un operando inmediato (constante). Ej:

```
1 ADDI X0, X1, #10
```

- Tipo D (Data Transfer): Instrucciones de carga y almacenamiento desde/hacia memoria. Ej:

```
1 LDUR X0, [X1, #8]
```

- Tipo B (Branch): Saltos incondicionales relativos. Ej:

```
1 B label
```

Instrucciones en LEGv8

- Tipo CB (Conditional Branch): Saltos condicionales basados en comparaciones con cero. Ej:

```
1 CBNZ X1, label
```

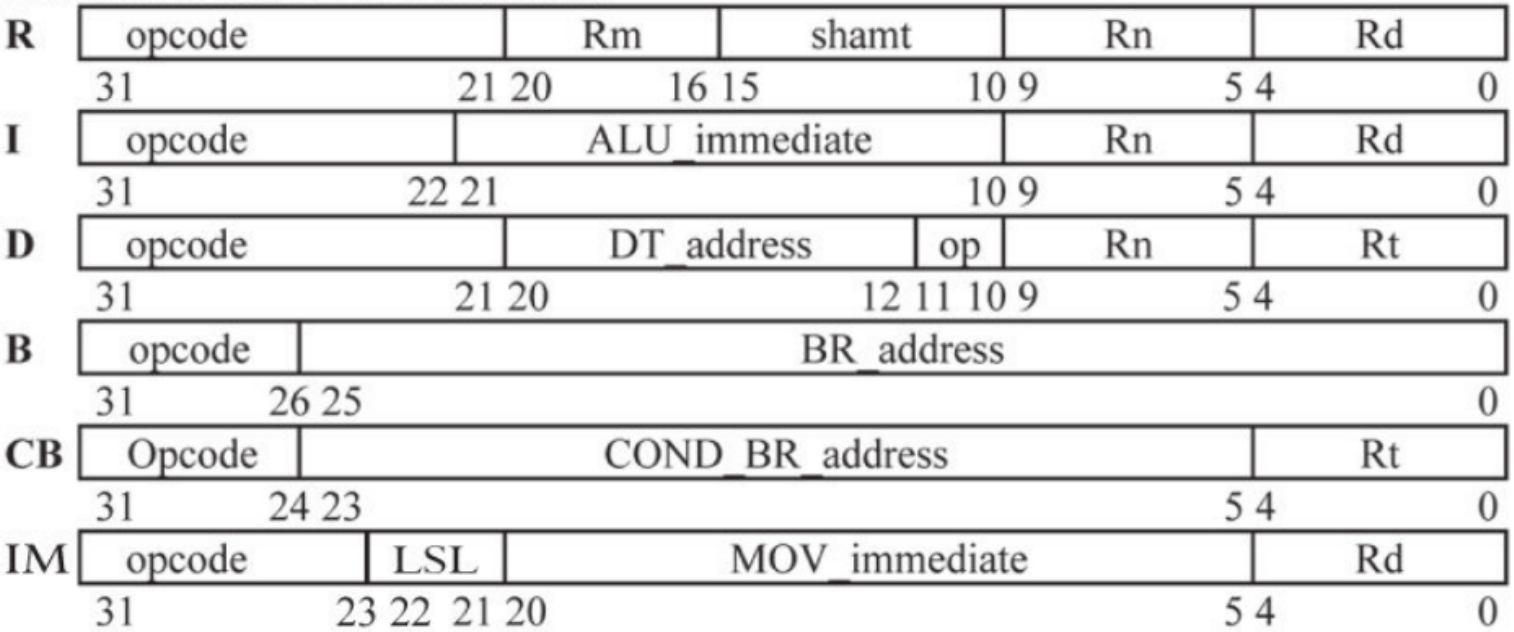
- IM (Move, Literal): Instrucciones para mover valores inmediatos o constantes.

```
1 MOVZ X0, #100
```

Con estos seis formatos básicos de instrucciones que cubren operaciones aritméticas, de memoria y control de flujo.

Instrucciones en LEGv8

CORE INSTRUCTION FORMATS



Formato R (Register) | Formato I (Immediate)

Formato R (Register)

Instrucción	Ejemplo	Descripción
ADD	ADD X0, X1, X2	Suma X1 + X2 → X0
SUB	SUB X3, X4, X5	Resta X4 - X5 → X3
AND	AND X6, X7, X8	AND lógico entre registros
ORR	ORR X9, X10, X11	OR lógico entre registros
EOR	EOR X12, X13, X14	XOR lógico entre registros
LSL	LSL X0, X1, #3	Desplazamiento lógico a la izquierda

Formato I (Immediate)

Instrucción	Ejemplo	Descripción
ADDI	ADDI X0, X1, #10	Suma X1 + 10 → X0
SUBI	SUBI X2, X3, #5	Resta X3 - 5 → X2
ANDI	ANDI X4, X5, #7	AND lógico con inmediato
ORRI	ORRI X6, X7, #1	OR lógico con inmediato

Formato D | Formato B | Formato CB | Formato IM

Formato D (Data Transfer)

Instrucción	Ejemplo	Descripción
LDUR	LDUR X0, [X1, #8]	Carga X0 desde memoria en X1 + 8
STUR	STUR X2, [X3, #0]	Almacena X2 en memoria en X3 + 0

Formato B (Branch)

Instrucción	Ejemplo	Descripción
B	B etiqueta	Salto incondicional a etiqueta
BL	BL subrutina	Llama subrutina y enlaza

Formato CB (Conditional Branch)

Instrucción	Ejemplo	Descripción
CBZ	CBZ X0, etiqueta	Salta si X0 == 0
CBNZ	CBNZ X1, etiqueta	Salta si X1 != 0

Formato IM (Move Immediate)

Instrucción	Ejemplo	Descripción
MOVZ	MOVZ X0, #100	Carga 100 en X0
MOVK	MOVK X0, #255, LSL 16	Carga valor con desplazamiento

Ejemplo de código assembler LEGv8

El siguiente codigo en assembler Legv8 suma dos numeros y almacena el resultado:

```
1         MOVZ X0, #10          ; X0 = 10
2         MOVZ X1, #20          ; X1 = 20
3         ADD X2, X0, X1        ; X2 = X0 + X1
4
5         LDUR X3, [X2, #0]     ;  Carga valor de memoria en X2
6         CBZ X3, etiqueta      ;  Si X3 == 0 salta a etiqueta
7         B fin                 ;  Salto incondicional
8
9         etiqueta:
10        SUB X2, X2, #1        ;  Decrementa X2
11
12        fin:
13        STUR X2, [X1, #0]     ;  Almacena resultado en memoria
```

If-Else en LEGv8

```
1 if (X1 == X2) {
2              X0 = X3 + X4;
3               }
4     else {
5             X0 = X3 - X4;
6         }
```

```
1 SUB X9, X1, X2          ; Resta para comparar X1 y X2
2 CBNZ X9, else_label     ; Si X9 != 0 salta a else_label
3 ADD X0, X3, X4          ; if: X0 = X3 + X4
4 B end_if                ; Salto para evitar else
5 else_label:
6 SUB X0, X3, X4          ; else: X0 = X3 - X4
7 end_if:                 ; }
```

While en LEGv8

```
1 while (X1 != 0) {
2           sum = sum + X2;
3           X1 = X1 - 1;
4           }
```

```
1 loop_start:
2 CBZ X1, loop_end          ; while (X1 == 0) break
3 ADD X3, X3, X2            ; sum += X2
4 SUBI X1, X1, #1           ; X1--
5 B loop_start              ; repetir while
6 loop_end:                  ; }
```

Do-While en LEGv8

```
do {
      sum = sum + X2;
      X1 = X1 - 1;
 } while (X1 != 0);
```

```
loop_start:
ADD X3, X3, X2          ; sum += X2
SUBI X1, X1, #1         ; X1--
CBNZ X1, loop_start     ; repetir mientras X1 != 0
```

For en LEGv8

```
for (i = N; i > 0; i--)
    {
        sum += X2;
    }
```

```
MOVZ X1, #N             ; i = N
MOVZ X3, #0             ; sum = 0

for_loop:
        CBZ X1, for_end         ; if(i == 0) break
        ADD X3, X3, X2          ; sum += X2
        SUBI X1, X1, #1         ; i--
B for_loop

for_end:                         ; }
```

MOVZ - MOVK

MOVZ (Move Wide with Zero): carga un valor inmediato en un registro, llenando con ceros todos los bits no especificados.

```
1
2 MOVZ X0, #0x1234        ; Carga 0x00001234 en X0
3 MOVZ X1, #0xABCD, LSL #16 ; Carga 0xABCD0000 en X1 (desplazamiento 16 bits)
```

Uso típico: inicializar registros con valores de 16 bits, desplazando para armar valores más grandes.

MOVK (Move Wide with Keep): inserta un valor inmediato sin borrar los bits fuera del segmento indicado.

```
1 MOVZ X0, #0x0000          ; Inicializa X0 a 0
2 MOVK X0, #0x5678          ; Inserta 0x5678 en el segmento inferior, X0 =
      0x00005678
3 MOVK X0, #0x9ABC, LSL #16  ; Inserta 0x9ABC en bits [31:16], X0 = 0x9ABC5678
```

Uso típico: construir un valor de 64 bits cargando segmentos de 16 bits uno a uno sin perder los demás bits.

LSL - LSR

LSL - Desplazamiento lógico a la izquierda

```
1 ; Multiplica X1 por 8 desplazando 3 bits a la izquierda:
2 LSL X0, X1, #3     ; X0 = X1 << 3 (equivalente a X1 * 8)
```

Descripción: Desplaza los bits de X1 3 posiciones hacia la izquierda, llenando con ceros a la derecha.

LSR - Desplazamiento lógico a la derecha

```
1 ; Divide X2 por 4 desplazando 2 bits a la derecha:
2 LSR X3, X2, #2     ; X3 = X2 >> 2 (equivalente a X2 / 4)
```

Descripción: Desplaza los bits de X2 2 posiciones hacia la derecha, llenando con ceros a la izquierda.

Introducción a las Flags

- Las **flags** son bits en el registro especial de estado del procesador (SPSR).
- Se modifican automáticamente al ejecutar instrucciones aritméticas y lógicas con postfijo `S` (por ejemplo, `ADDS`).
- Son utilizadas para controlar el flujo condicional dependiendo del resultado de las operaciones.

N -Negativo: se activa si el resultado de una operación es negativo.
SUBS X0, X1, X2 (Si X1 = 3, X2 = 5 entonces X0 = -2 y N=1)

[Z - Zero]: Se activa si el resultado es cero.
SUBS X0, X1, X1 (Si X1 = 4 entonces X0 = 0 y Z=1)

[C - Carry]: Se activa si hay acarreo (en suma) o préstamo (en resta).
ADDS X0, X1, X2 (Si $X1 = 2^{63} - 1$, X2=1 y V=1)

[V - Overflow]: Se activa si hay desbordamiento aritmético con signo (overflow).
ADCS X0, X1, X2 (Si $X1 = 2^{64} - 1$, X2=1 y C=1)

Instrucciones condicionales basadas en flags

```
1 BEQ etiqueta      ; Salta si Z (zero) esta activada, resultado = 0
2 BNE etiqueta      ; Salta si Z esta desactivada, resultado != 0
3 BGT etiqueta      ; Salta si N==V y Z == 0 (mayor que, considerando signos)
4 BLT etiqueta      ; Salta si N!=V (menor que, con signo)
```

Ejemplo: comparar dos registros e ir a etiqueta si son iguales.

```
1 CMP X0, X1
2 BEQ iguales
3
4 B fin
5 iguales:
6
7 fin:
```

Emulador Qemu.

QEMU

- Un emulador y virtualizador de máquinas genérico y de código abierto.
- puede ejecutarse en cualquier tipo de Microprocesador o arquitectura (x86, x86-64, PowerPC, MIPS, SPARC, etc.).
- Programado en C.
- disponible para GNU/Linux, Windows, entre otros.
- sitio web: https://www.qemu.org/

Bibliografía.

Bibliografía Básica

- David A. PATTERSON and John L. HENNESSY, “Computer Organization and Design: The Hardware / Software Interface - ARM Edition” Ed. Morgan Kaufmann (2017).
- Harris & Harris. Digital design and computer architecture: ARM edition. Elsevier, 2015. Cap. 4, 6 y 7.