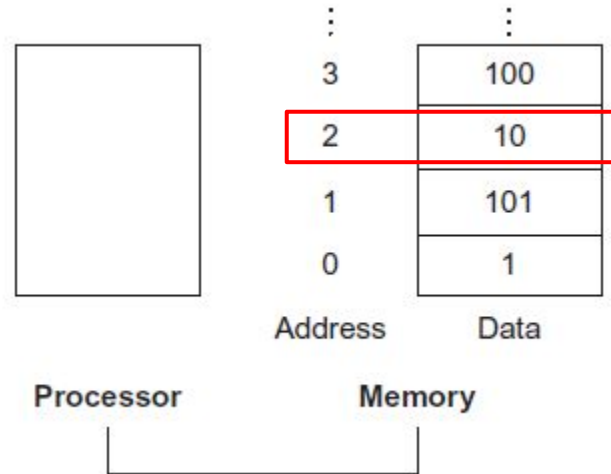


# LEGv8 básico

OdC - 2022

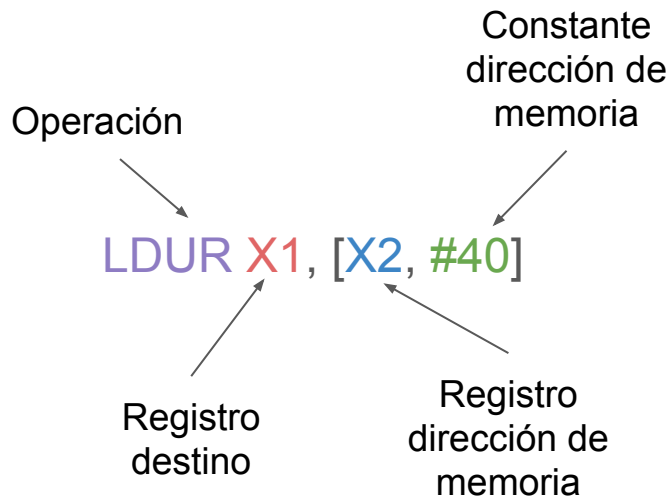
# Accediendo a la memoria

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.



# Instrucción Load (“cargar”)

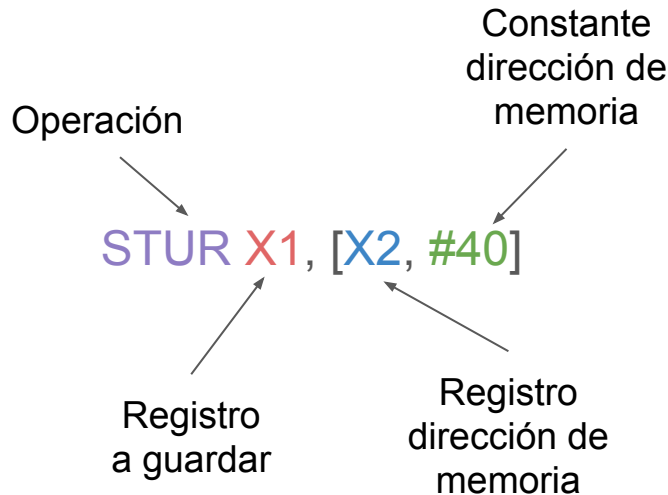
Copia al registro **X1** el contenido de la memoria direccionada por el contenido del registro **X2** sumado a la constante **#40**.



$$\mathbf{X1} = \text{Memory}[\mathbf{X2} + \mathbf{\#40}]$$

# Instrucción Store (“guardar”)

Copia el contenido del registro **X1** en la posición de memoria direccionada por el contenido del registro **X2** sumado a la constante **#40**.



$$\text{Memory}[\text{X2} + \text{\#40}] = \text{X1}$$

# Tamaño de palabra

## DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

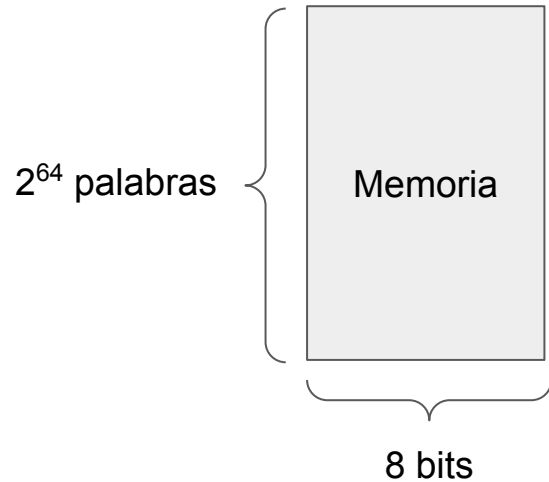
- Tamaño de un registro = 64 bits = Double Word
- Address para load y store = 64 bits = Double Word

# Conjunto de instrucciones - Transferencia de datos

Data transfer	load register	LDUR X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$	Doubleword from memory to register
	store register	STUR X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$	Doubleword from register to memory
	load signed word	LDURSW X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$	Word from memory to register
	store word	STURW X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$	Word from register to memory
	load half	LDURH X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$	Halfword memory to register
	store half	STURH X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$	Halfword register to memory
	load byte	LDURB X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$	Byte from memory to register
	store byte	STURB X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$	Byte from register to memory
	load exclusive register	LDXR X1, [X2,0]	$X1 = \text{Memory}[X2]$	Load; 1st half of atomic swap
	store exclusive register	STXR X1, X3 [X2]	$\text{Memory}[X2] = X1; X3 = 0 \text{ or } 1$	Store; 2nd half of atomic swap
	move wide with zero	MOVZ X1, 20, LSL 0	$X1 = 20 \text{ or } 20 * 2^{16} \text{ or } 20 * 2^{32} \text{ or } 20 * 2^{48}$	Loads 16-bit constant, rest zeros
	move wide with keep	MOVK X1, 20, LSL 0	$X1 = 20 \text{ or } 20 * 2^{16} \text{ or } 20 * 2^{32} \text{ or } 20 * 2^{48}$	Loads 16-bit constant, rest unchanged

# Dimensiones de la memoria

En LEGv8 la memoria se direcciona de a byte:



Ejemplo:

Dirección	Contenido
0	0x00
1	0x01
2	0x02
3	0x03
4	0x04
5	0x05
6	0x06
7	0x07
8	0x08
9	0x09
10	0x0A
11	0x0B
12	0x0C
13	0x0D
14	0x0E
15	0x0F
16	0x10
...	...

# Acceso a memoria de byte (LDURB/STURB)

Dirección	Contenido
0	0x00
1	0x01
2	0x02
3	0x03
4	0x04
5	0x05
6	0x06
7	0x07
8	0x08
9	0x09
10	0x0A
11	0x0B
12	0x0C
13	0x0D
14	0x0E
15	0x0F
16	0x10
...	...

LDURB X1, [XZR, #0]      // X1 = 0x0000000000000000

LDURB X1, [XZR, #3]      // X1 = 0x0000000000000003

LDURB X1, [XZR, #10]     // X1 = 0x000000000000000A

ceros



# Acceso a memoria de doubleword (LDUR/STUR)

En LEGv8 la memoria se direcciona de a byte => si se accede de forma secuencial a datos doubleword las direcciones varían de a 8.

Dirección	Contenido
0	0x00
1	0x01
2	0x02
3	0x03
4	0x04
5	0x05
6	0x06
7	0x07
8	0x08
9	0x09
10	0x0A
11	0x0B
12	0x0C
13	0x0D
14	0x0E
15	0x0F
16	0x10
...	...

palabra 0

palabra 1

Dirección	Contenido							
0	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
8	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
16	0x10	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

64 bits

En este ejemplo el microprocesador LEGv8 está configurado en modo *big-endian*

LDUR X1, [XZR, #0] // X1 = 0x0001020304050607

LDUR X1, [XZR, #8] // X1 = 0x08090A0B0C0D0E0F

# Acceso a memoria de doubleword (LDUR/STUR)

Si se configura el microprocesador en modo little-endian los bits **menos** significativos de una palabra se guardan en las posiciones más bajas de memoria.

Dirección	Contenido
0	0x00
1	0x01
2	0x02
3	0x03
4	0x04
5	0x05
6	0x06
7	0x07
8	0x08
9	0x09
10	0x0A
11	0x0B
12	0x0C
13	0x0D
14	0x0E
15	0x0F
16	0x10
...	...

palabra 0

palabra 1

Dirección	Contenido							
0	0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00
8	0x0F	0x0E	0x0D	0x0C	0x0B	0x0A	0x09	0x08
16	...	...	...	...	...	...	...	0x10
...	...	...	...	...	...	...	...	...

64 bits

En este ejemplo el microprocesador LEGv8 está configurado en modo *little-endian*

LDUR X1, [XZR, #0]

// X1 = 0x0706050403020100

LDUR X1, [XZR, #8]

// X1 = 0x0F0E0D0C0B0A0908

# Acceso a memoria de doubleword (LDUR/STUR)

Dirección    Contenido

...	...	
16	0x10	
15	0x0F	
14	0x0E	
13	0x0D	
12	0x0C	palabra 1
11	0x0B	
10	0x0A	
9	0x09	
8	0x08	
7	0x07	
6	0x06	
5	0x05	
4	0x04	palabra 0
3	0x03	
2	0x02	
1	0x01	
0	0x00	

Microprocesador en modo **big-endian**:

LDUR X1, [XZR, #0]      // X1 = 0x0001020304050607

LDUR X1, [XZR, #8]      // X1 = 0x08090A0B0C0D0E0F

Microprocesador en modo **little-endian**:

LDUR X1, [XZR, #0]      // X1 = 0x0706050403020100

LDUR X1, [XZR, #8]      // X1 = 0x0F0E0D0C0B0A0908

Por defecto asumimos  
que el microprocesador  
está en modo big-endian

# Acceso a memoria de word (LDURSW/STURW)

En LEGv8 la memoria se direcciona de a byte, por lo que si se accede de forma secuencial a datos word las direcciones varían de a 4.

Dirección	Contenido	
0	0x00	palabra 0
1	0x01	
2	0x02	
3	0x03	
4	0x04	palabra 1
5	0x05	
6	0x06	
7	0x07	
8	0x08	palabra 2
9	0x09	
10	0x0A	
11	0x0B	
12	0xEF	palabra 3
13	0x0D	
14	0x0E	
15	0x0F	
16	0x10	
...	...	

Dirección	Contenido			
0	0x00	0x01	0x02	0x03
4	0x04	0x05	0x06	0x07
8	0x08	0x09	0x0A	0x0B
12	0x0C	0x0D	0x0E	0x0F
16	0x10	...	...	...
...	...			

32 bits

En este ejemplo el microprocesador LEGv8 está configurado en modo *big-endian*

LDURSW X1, [XZR, #0]

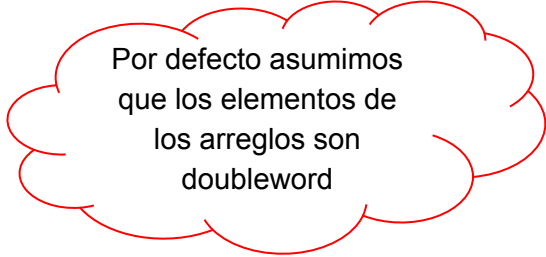
// X1 = 0x00000000000010203

LDURSW X1, [XZR, #12]

// X1 = 0xFFFFFFFFFEF0D0E0F

## Ejercicio 5(a)

Dada la siguiente sentencia en “C”:

$$f = -g - A[4];$$


Por defecto asumimos  
que los elementos de  
los arreglos son  
doubleword

5.1) 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.

5.2) ¿Cuántos registros se utilizan para llevar a cabo las operaciones anteriores?

## Ejercicio 5(a)

$f = -g - A[4];$

$X0 \leftarrow f$

$X1 \leftarrow g$

$X6 \leftarrow \text{dirección base del arreglo } A = \& A[0]$

$\& A[4] = \& A[0] + 4 \cdot 8$

LDUR  $X0, [X6, \#32]$  //  $f = A[4]$

ADD  $X0, X0, X1$  //  $f = A[4] + g$

SUB  $X0, XZR, X0$  //  $f = -g - A[4]$

5.2) Se utilizaron 4 registros.

## Ejercicio 5(b)

Dada la siguiente sentencia en “C”:

$$B[8] = A[i - j];$$

5.1) 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.

5.2) ¿Cuántos registros se utilizan para llevar a cabo las operaciones anteriores?

## Ejercicio 5(b)

$B[8] = A[i - j];$

$X2 \leftarrow i$

$X3 \leftarrow j$

$X6 \leftarrow \text{dirección base del arreglo } A = \&A[0]$

$X7 \leftarrow \text{dirección base del arreglo } B = \&B[0]$

$\&A[i-j] = \&A[0] + (i-j)*8$

SUB X9, X2, X3 // X9=i-j

LSL X9, X9, #3 ★ // X9=(i-j)\*8

ADD X10, X6, X9 // X10=&A+[(i-j)\*8]

LDUR X11, [X10, #0] // X11=A[i-j]

STUR X11, [X7, #64] // B[8]=A[i-j]

$\&B[8] = \&B[0] + 8*8$

5.2) Se utilizaron 7 registros.



## ★ Usando LSL para multiplicar...

ADDI X0, XZR, #15 //  $X0_{10} = 15$ , o su equivalente en binario:

[illegible]

Luego:

```
LSL X1, X0, 1    // X1= 0b0...00011110 →  $X1_{10} = 30$  → equivale a  $X0 \cdot 2^1$ 
```

```
LSL X2, X0, 2    // X2= 0b0...00111100 → X210= 60 → equivale a X0*22
```

```
LSL X3, X0, 3    // X3= 0b0...01111000 → X310 = 120 → equivale a X0*23
```

## Ejercicio 6(b)

Dadas las siguientes sentencias en assembler LEGv8:

```
LSL X9, X3, #3
ADD X9, X6, X9
LSL X10, X4, #3
ADD X10, X7, X10
LDUR X12, [X9, #0]
ADDI X11, X9, #8
LDUR X9, [X11, #0]
ADD X9, X9, X12
STUR X9, [X10, #0]
```

6.1) Escribir la secuencia mínima de código “C” asumiendo que los registros X0, X1, X2, X3 y X4 contienen las variables f, g, h, i y j respectivamente, y los registros X6, X7 contienen las direcciones base de los arreglos A y B.

6.2) Para las instrucciones LEGv8 anteriores, re-escriba el código para minimizar (de ser posible) la cantidad de instrucciones manteniendo la funcionalidad.

## Ejercicio 6(b)

$X3 \leftarrow i$

$X4 \leftarrow j$

$X6 \leftarrow$  dirección base del  
arreglo  $A = \& A[0]$

$X7 \leftarrow$  dirección base del  
arreglo  $B = \& B[0]$


LSL X9, X3, #3      //X9 =  $i * 8$

ADD X9, X6, X9      //X9 =  $\&A + (i * 8) = \&A[i]$

LSL X10, X4, #3      //X10 =  $j * 8$

ADD X10, X7, X10      //X10 =  $\&B + (j * 8) = \&B[j]$

LDUR X12, [X9, #0]      //X12 =  $A[i]$

 ADDI X11, X9, #8      //X11 =  $\&A[i] + 8 = \&A[i+1]$

LDUR X9, [X11, #0]      //X9 =  $A[i+1]$

ADD X9, X9, X12      //X9 =  $A[i+1] + A[i]$

STUR X9, [X10, #0]      //B[j] =  $A[i+1] + A[i]$

6.1)  $B[j] = A[i+1] + A[i]$

# Ejercicio 7

Dirección	Valor
0x0000000040080030	0x64
0x0000000040080038	0xC8
0x0000000040080040	0x12C

Dadas las siguientes sentencias en assembler LEGv8:

```
ADDI X9, X6, #8
ADD  X10, X6, XZR
STUR X10, [X9, #0]
LDUR X9, [X9, #0]
ADD  X0, X9, X10
```

7.1) Asumiendo que los registros X0, X6 contienen las variables f y A (dirección base del arreglo), escribir la secuencia mínima de código “C” que representa.

7.2) Asumiendo que los registros X0, X6 contienen los valores 0xA, 0x40080030, y que la memoria contiene los valores de la tabla, encuentre el valor del registro X0 al finalizar el código assembler.

# Ejercicio 7

$X0 \leftarrow f$

$X6 \leftarrow \text{dirección base del arreglo } A = \&A[0]$

ADDI X9, X6, #8      //X9 = &A[0]+8 = &A[1]

ADD X10, X6, XZR      //X10 = &A[0]+0 = &A[0]

STUR X10, [X9, #0]      //A[1] = &A[0]

 LDUR X9, [X9, #0]      //X9 = A[1] = &A[0]

ADD X0, X9, X10      //f = &A[0] + &A[0]


Dirección	Valor
0x0000000040080030	0x64
0x0000000040080038	0xC8
0x0000000040080040	0x12C

7.1)  $f = \&A[0] + \&A[0]$


7.2) resuelto con QEMU

# Memoria en QEMU


0x40080000	+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
<b>Memory</b>																	
0x0000000040080000																	
0x0000000040080000	40	02	00	58	26	01	00	58	c6	00	00	8b	c9	20	00	91	@..X&..X.....
0x0000000040080010	ca	00	1f	8b	2a	01	00	f8	29	01	40	f8	20	01	0a	8b	....*... ) .@.....
0x0000000040080020	00	00	00	14	00	00	00	00	30	00	00	00	00	00	00	00	.....0.....
0x0000000040080030	64	00	00	00	00	00	00	00	c8	00	00	00	00	00	00	00	d.....
0x0000000040080040	2c	01	00	00	00	00	00	00	00	00	08	40	00	00	00	00	, .....@....
0x0000000040080050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000040080060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x0000000040080070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....



Dirección Base



Palabra 0 (64 bits)



Palabra 1 (64 bits)

## Ejercicio 8.1(a)

Dado el contenido de los siguientes registros:

$X9 = 0x55555555$  y  $X10 = 0x12345678$

¿Cuál es el valor del registro X11 luego de la ejecución del siguiente código assembler en LEGv8?

`LSL X11, X9, #4`

`ORR X11, X11, X10`

# Ejercicio 8.1(a)

- X9= 0x0000000055555555 =

0b0000 0000 0000 0000 0000 0000 0000 0000 0101 0101 0101 0101 0101 0101 0101 0101

- X10= 0x0000000012345678 =

0b0000 0000 0000 0000 0000 0000 0000 0000 0001 0010 0011 0100 0101 0110 0111 1000

**LSL** X11, X9, #4 //X11 = 0x0000000555555550

X11=0b0000 0000 0000 0000 0000 0000 0000 0101 0101 0101 0101 0101 0101 0101 0101 0000

**ORR** X11, X11, X10 //X11 = 0x0000000557755778

X11→ 0000 0000 0000 0000 0000 0000 0000 0101 0101 0101 0101 0101 0101 0101 0101 0000

X10→ 0000 0000 0000 0000 0000 0000 0000 0000 0001 0010 0011 0100 0101 0110 0111 1000

---

X11→ 0000 0000 0000 0000 0000 0000 0000 0101 0101 0111 0111 0101 0101 0111 0111 1000



# Ejercicio 11

Utilizar MOVZ, MOVK para cargar los registros:

10.1) {X0 = 0x1234000000000000}

10.2) {X1 = 0xBBB00000000000AAA}

10.3) {X2 = 0xA0A0B1B10000C2C2}

10.4) {X3 = 0x0123456789ABCDEF}

# Conjunto de instrucciones - Transferencia de datos

Data transfer	load register	LDUR X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$	Doubleword from memory to register
	store register	STUR X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$	Doubleword from register to memory
	load signed word	LDURSW X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$	Word from memory to register
	store word	STURW X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$	Word from register to memory
	load half	LDURH X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$	Halfword memory to register
	store half	STURH X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$	Halfword register to memory
	load byte	LDURB X1, [X2,40]	$X1 = \text{Memory}[X2 + 40]$	Byte from memory to register
	store byte	STURB X1, [X2,40]	$\text{Memory}[X2 + 40] = X1$	Byte from register to memory
	load exclusive register	LDXR X1, [X2,0]	$X1 = \text{Memory}[X2]$	Load; 1st half of atomic swap
	store exclusive register	STXR X1, X3 [X2]	$\text{Memory}[X2] = X1; X3 = 0 \text{ or } 1$	Store; 2nd half of atomic swap
	move wide with zero	MOVZ X1, 20, LSL 0	$X1 = 20 \text{ or } 20 * 2^{16} \text{ or } 20 * 2^{32} \text{ or } 20 * 2^{48}$	Loads 16-bit constant, rest zeros
	move wide with keep	MOVK X1, 20, LSL 0	$X1 = 20 \text{ or } 20 * 2^{16} \text{ or } 20 * 2^{32} \text{ or } 20 * 2^{48}$	Loads 16-bit constant, rest unchanged

# Ejercicio 11

11.1) {X0 = 0x1234000000000000}

```
MOVZ X0, 0x1234, LSL 48 // X0 = 0x1234000000000000
```

11.3) {X2 = 0xA0A0B1B10000C2C2}

```
MOVZ X2, 0xA0A0, LSL 48 // X2 = 0xA0A0000000000000
```

```
MOVK X2, 0xB1B1, LSL 32 // X2 = 0xA0A0B1B100000000
```


```
MOVK X2, 0xC2C2, LSL 0 // X2 = 0xA0A0B1B10000C2C2
```

# Ejercicio 11 - para tener en cuenta!

- Se puede alterar el orden en que se carga el registro:

11.3)    `MOVZ X2, 0xB1B1, LSL 32    // X2 = 0x0000B1B100000000`  
         `MOVK X2, 0xC2C2, LSL 0     // X2 = 0x0000B1B10000C2C2`  
         `MOVK X2, 0xA0A0, LSL 48    // X2 = 0xA0A0B1B10000C2C2`

- No es lo mismo si usamos sólo `MOVK`:

11.3)    `MOVK X2, 0xB1B1, LSL 32    // X2 = 0x????B1B1????????`  
         `MOVK X2, 0xC2C2, LSL 0     // X2 = 0x????B1B1????C2C2`  
         `MOVK X2, 0xA0A0, LSL 48    // X2 = 0xA0A0B1B1????C2C2`  
 `MOVK X2, 0x0000, LSL 16    // X2 = 0xA0A0B1B10000C2C2`

# Bibliografía

Patterson and Hennessy, “Computer Organization and Design: The Hardware/Software Interface ARM Edition”, Morgan kaufmann, 2016.