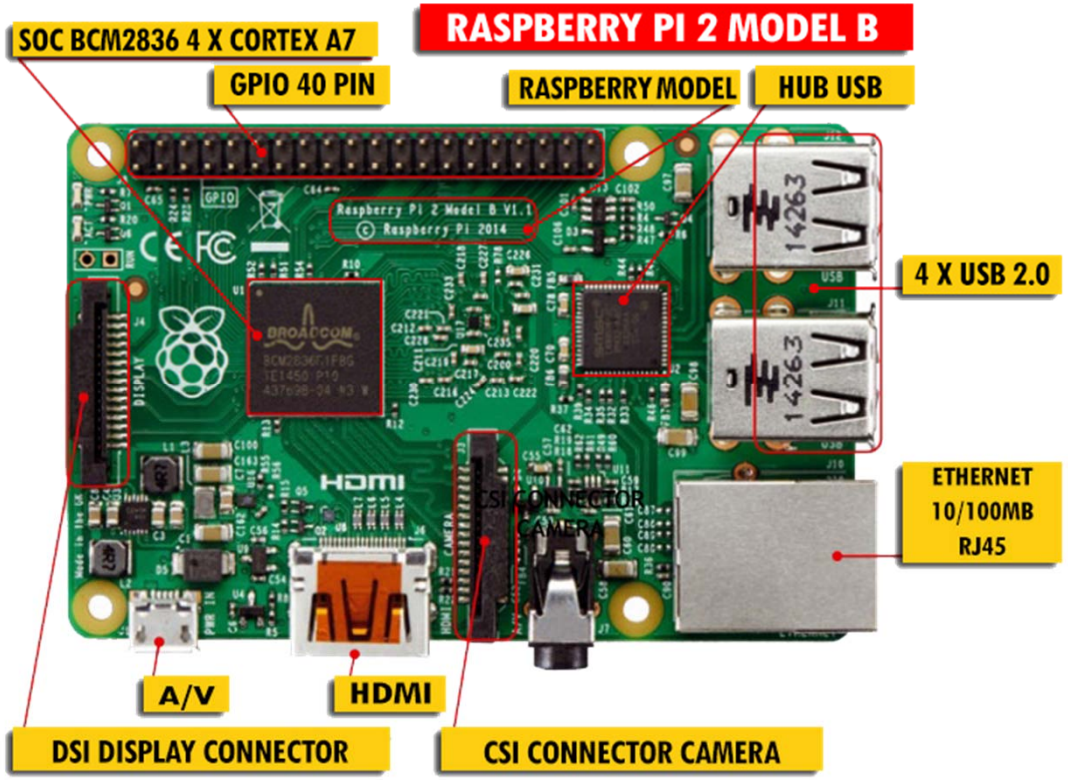
Taller Rasperry Pi

Tecnología de Computadores
Curso 2015-2016

Raspberry Pi 2

- Lleva un procesador ARM Cortex-A7 (ARMv7) Quad core Broadcom BCM2836
  - específico para Raspeberry Pi 2
  - Cortex-A7 pensado para ahorro energético (muy usado junto con Cortex-A15 en móviles)



Dispositivos con Cortex A-7 y A-15

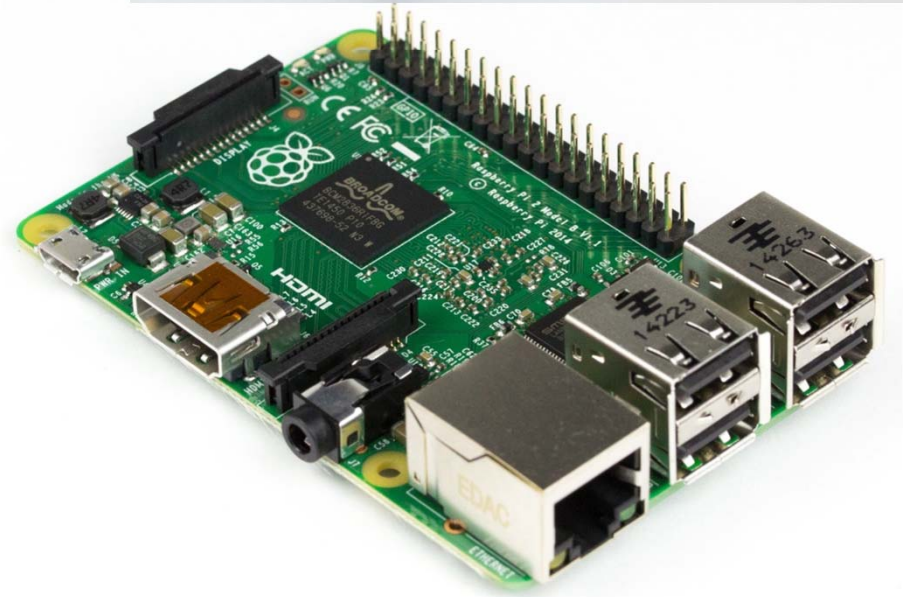
2013

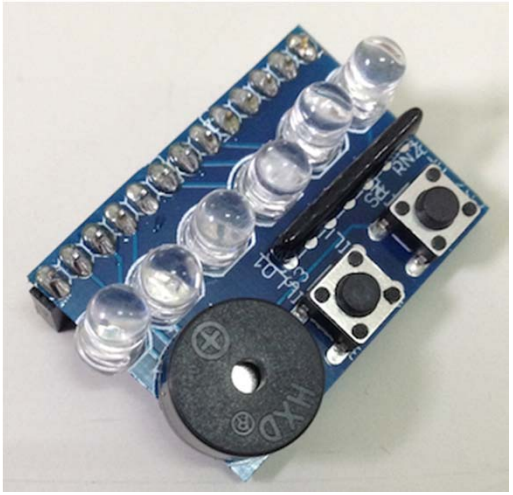
2013

2014

2014

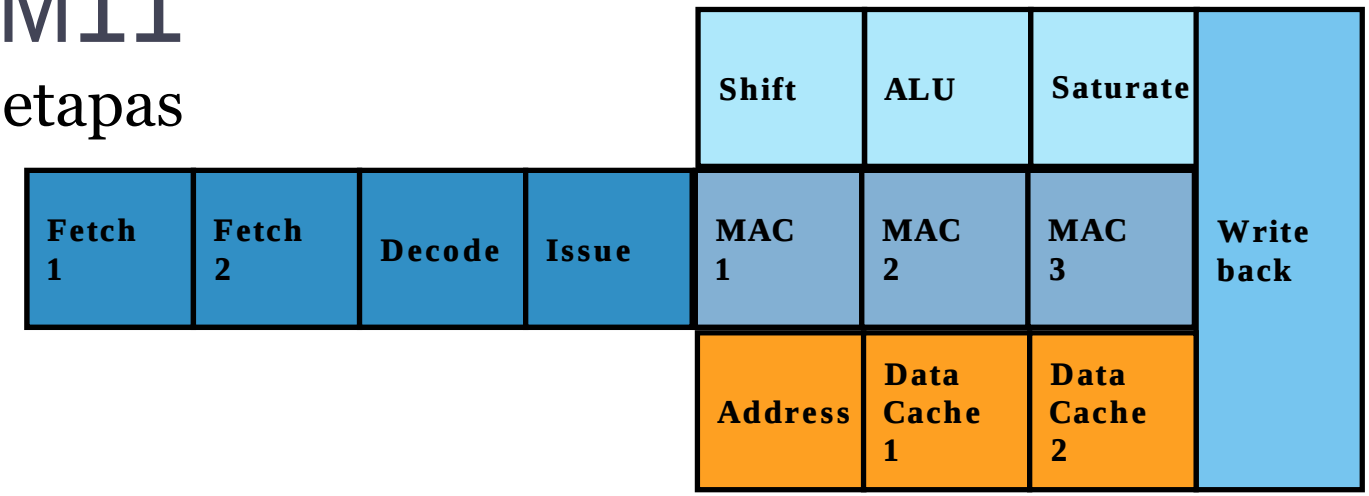
Raspberry Pi: Interrupciones y GPIO





ARM11

- 8 etapas



- 1. Fe1 – Se manda dirección y se recibe la instrucción
- 2. Fe2 – Predicción de salto
- 3. De – Decodificación instrucción
- 4. Iss – Lectura de registros
- 5. Sh – Se realizan las operaciones de desplazamiento (shift operations)
- 6. ALU – Se realizan operaciones con enteros
- 7. Sat – Saturación de resultados
- 8. WB – Se escribe resultado en los registros

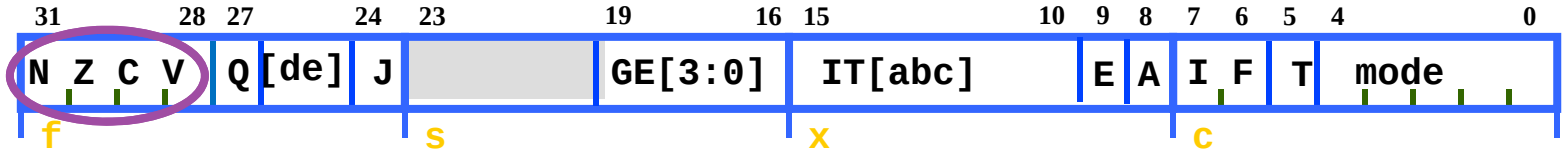
Tamaño de los datos e instrucciones

- ARM es una arquitectura RISC de 32-bit, Load-Store (como el MIPS!!).
- En un ARM (= MIPS):
  - **Byte** son 8 bits
  - **Halfword** son 16 bits (dos bytes)
  - **Word** son 32 bits (cuatro bytes)

- La mayoría de los ARM implementan dos repertorios de instrucciones:
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set
    (Jazelle cores can also execute Java bytecode)

- Cuando el procesador ejecuta código ARM:
  - Todas las instrucciones tienen tamaño 32 bits
  - Todas las instrucciones están alineadas a nivel de palabra
  - Por lo tanto el valor de PC se almacena en los bits [31:2] estando los bits [1:0] indefinidos (puesto que la instrucción no puede estar alineada a nivel de byte o media-palabra)

Registro de estados



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed

Flags en el registro de estado	Significado
N	Negativo: 1 si el resultado de la operación ha sido negativo
Z	Cero: 1 si el resultado de la operación ha sido 0
C	Carry: 1 si la operación ha generado acarreo
V	Overflow: 1 si el resultado de la operación no se puede representar en complemento a 2 con 32 bits

Ejecución condicional (inst. predicadas)

- Las instrucciones del ARM pueden ejecutarse condicionalmente si se les coloca como sufijo el código de condición adecuado.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
    CMP   r3,#0
    BEQ   skip
    ADD   r0,r1,r2
skip
```

```
CMP   r3,#0
ADDNE r0,r1,r2
```

- Por defecto, las instrucciones de procesado de datos no afectan a los flags de condición. Si queremos que los actualicen debemos utilizar el sufijo "S". Las instrucciones de comparación de registros, como CMP, no necesitan "S".

```
loop
    …
    SUBS r1,r1,#1
    BNE loop
```

decrement r1 and set flags

if Z flag clear then branch

Registro de estados y ejecución condicional

Campo condición {cond}	Significado
EQ	(equal) When **Z** is enabled (**Z is 1**)
NE	(not equal). When **Z** is disabled. (**Z is 0**)
GE	(greater or equal than, in two's complement). When both **V** and **N** are enabled or disabled (**V is N**)
LT	(lower than, in two's complement). This is the opposite of GE, so when **V** and **N** are not both enabled or disabled (**V is not N**)
GT	(greather than, in two's complement). When Z is disabled and **N** and **V** are both enabled or disabled (**Z is 0, N is V**)
LE	(lower or equal than, in two's complement). When **Z** is enabled or if not that, **N** and **V** are both enabled or disabled (**Z is 1. If Z is not 1 then N is V**)
MI	(minus/negative) When **N** is enabled (**N is 1**)
PL	(plus/positive or zero) When **N** is disabled (**N is 0**)
VS	(overflow set) When **V** is enabled (**V is 1**)
VC	(overflow clear) When **V** is disabled (**V is 0**)
HI	(higher) When **C** is enabled and Z is disabled (**C is 1 and Z is 0**)
LS	(lower or same) When **C** is disabled or **Z** is enabled (**C is 0 or Z is 1**) CS/HS (carry set/higher or same) When **C** is enabled (**C is 1**)
CS/HS	(carry set/higher or same) When **C** is enabled (**C is 1**)
CC/LO	(carry clear/lower) When **C** is disabled (**C is 0**)

Escribir registro estado	cmp inst{s}: adds, subs, ands, ...
Salto (b) condicional	b{cond}: b**eq**, b**ne**, b**gt**, b**le** ...
Ejecución condicional	inst{cond}: add**eq**, sub**ne**, ldr**gt**, ...

Ejemplos de ejecución condicional

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1, #1
  ADDNE r2, r2, #1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

ARM assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`ADD r1,r2,r3`	r1 = r2 + r3	3 register operands
	subtract	`SUB r1,r2,r3`	r1 = r2 – r3	3 register operands
Data transfer	load register	`LDR r1, [r2,#20]`	r1 = Memory[r2 + 20]	Word from memory to register
	store register	`STR r1, [r2,#20]`	Memory[r2 + 20] = r1	Word from memory to register
	load register halfword	`LDRH r1, [r2,#20]`	r1 = Memory[r2 + 20]	Halfword memory to register
	load register halfword signed	`LDRHS r1, [r2,#20]`	r1 = Memory[r2 + 20]	Halfword memory to register
	store register halfword	`STRH r1, [r2,#20]`	Memory[r2 + 20] = r1	Halfword register to memory
	load register byte	`LDRB r1, [r2,#20]`	r1 = Memory[r2 + 20]	Byte from memory to register
	load register byte signed	`LDRBS r1, [r2,#20]`	r1 = Memory[r2 + 20]	Byte from memory to register
	store register byte	`STRB r1, [r2,#20]`	Memory[r2 + 20] = r1	Byte from register to memory
	swap	`SWP r1, [r2,#20]`	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	Atomic swap register and memory
	mov	`MOV r1, r2`	r1 = r2	Copy value into register
Logical	and	`AND r1, r2, r3`	r1 = r2 & r3	Three reg. operands; bit-by-bit AND
	or	`ORR r1, r2, r3`	r1 = r2 | r3	Three reg. operands; bit-by-bit OR
	not	`MVN r1, r2`	r1 = ~ r2	Two reg. operands; bit-by-bit NOT
	logical shift left (optional operation)	`LSL r1, r2, #10`	r1 = r2 << 10	Shift left by constant
	logical shift right (optional operation)	`LSR r1, r2, #10`	r1 = r2 >> 10	Shift right by constant
Conditional Branch	compare	`CMP r1, r2`	cond. flag = r1 – r2	Compare for conditional branch
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	`BEQ 25`	if (r1 == r2) go to PC + 8 + 100	Conditional Test; PC-relative
Unconditional Branch	branch (always)	`B 2500`	go to PC + 8 + 10000	Branch
	branch and link	`BL 2500`	r14 = PC + 4; go to PC + 8 + 10000	For procedure call

ARM assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD r1,r2,r3	r1 = r2 + r3	3 register operands
	subtract	SUB r1,r2,r3	r1 = r2 – r3	3 register operands
	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	Word from memory to register
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	Word from register to memory
	mov	MOV r1, r2	r1 = r2	Copy value into register
Logical	and	AND r1, r2, r3	r1 = r2 & r3	Three reg. operands; bit-by-bit AND
	or	ORR r1, r2, r3	r1 = r2 | r3	Three reg. operands; bit-by-bit OR
	not	MVN r1, r2	r1 = ~ r2	Two reg. operands; bit-by-bit NOT
	logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	Shift left by constant
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	Shift right by constant
Conditional Branch	compare	CMP r1, r2	cond. flag = r1 – r2	Compare for conditional branch
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	Conditional Test; PC-relative
Unconditional Branch	branch (always)	B 2500	go to PC + 8 + 10000	Branch
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	For procedure call

Las instrucciones de procesado de datos sólo trabajan con registros (=MIPS!!)

Transferencia de datos a reg.

- Operando fuente inmediato:
  - `mov r0, #500 @tamaño del inmediato limitado`
    - Pseudo-instrucción para la carga de constantes grandes:
      - `LDR rd, =const`

- Desde/hacia memoria (con diferentes tamaños de datos):

LDR	**STR**	Word
LDRB	**STRB**	Byte
LDRH	**STRH**	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

  - Sintaxis:
    - **LDR**{<cond>}{<size>} Rd, <address>
    - **STR**{<cond>}{<size>} Rd, <address>

    e.g. **LDREQB**

Dirección efectiva

- La dirección efectiva accedida por un LDR/STR se especica mediante un reg. base y un desplazamiento (offset)
- El offset puede ser (for word and unsigned byte accesses):
  - An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes)

    ```
    LDR r0, [r1, #8]
    ```
  - A register, optionally shifted by an immediate value

    ```
    LDR r0, [r1, r2]
    LDR r0, [r1, r2, LSL#2]
    ```
- ... y puede ser sumado o restado al registro base:

  ```
  LDR r0, [r1, #-8]
  LDR r0, [r1, -r2, LSL#2]
  ```
- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (i.e. 0 - 255 bytes)
  - A register (unshifted)

Dirección efectiva con pre/post indexado

- Si no queremos cambiar el registro base, utilizábamos:
  - `str r10, [r9, #4] @[r9+4]<- r10 (el reg. índice r9 no cambia)`
  - `ldr r8, [r9, +r10, LSL #3]  @r8<-[r9+(r10*8)] (tampoco cambia el índice, r9)`

- Pero puede optarse por dejar actualizado el resgistro base del direccionamiento, ya sea con pre-indexamiento o post-indexamiento:

Ejemplos de *pre-indexed* addressing:

- `ldr r2, [r1, #4]!  @sería:  r1<- r1+4 y después r2<-[r1]`
- `ldr r2, [r0, r1]!  @sería:  r0<- r0+r1  y después r2<-[r0]`
- `ldr r2, [r0, r1,  lsl #2]! @r0<-r0 + r1*4  then  r2<-[r0]`

Ejemplos *de post-indexed* addressing:

- `str r10, [r9], #4  @[r9]<- r10 y después r9<- r9+4`
- `str r0, [r1], r2  @[r1]<-r0  then  r1<-r1 + r2`
- `ldr r0, [r1], r2,  lsl #3  @ r0<-[r1]  then r1<-r1 + r2*8`

AAPCS (Procedure Call Standard for the ARM Architecture)

Registro	Sinónimo	Especial	Papel en el estándar de llamada a procedimiento
r15		PC	Program Counter
r14		LR	Link Register
r13		SP	Stack Pointer
r12		IP	Intra-Procedure-call scratch register
r11	v8		Variable register 8
r10	v7		Variable register 7
r9		v6	Platform register (meaning defined by platform)
r8	v5		Variable register 5
r7	v4		Variable register 4
r6	v3		Variable register 3
r5	v2		Variable register 2
r4	v1		Variable register 1
r3	a4		Argument / scratch register 4
r2	a3		Argument / scratch register 3
r1	a2		Argument / result / scratch register 2
r0	a1		Argument / result / scratch register 1

Paso de parámetros

- Paso de parámetros a procedimiento:
  - 4 primeros parámetros por a1, a2, a3 y a4
  - el resto por la pila
- Resultado del procedimiento
  - hasta 2 resultados por a1 y a2
- No se preservan a1, a2, a3 y a4 en la llamada a procedimiento
- Se preservan en la llamada a procedimiento de v1 a v8

Lenguaje ensamblador ARM

Manejo de la pila		
addi $sp, $sp, -4 sw $ra, 0($sp)	**push {lr}**	// salvar contenido de un registro (lr) en pila. Se pueden almacenar varios a la vez {r1, r2, r3, r4}
lw $ra, 0($sp) addi $sp, $sp, 4	**pop {lr}**	// sacar contenido de pila y meterlo en un registro (lr). Se puede hacer con varios rg. {r1, r2, r3, r4}
Finalización de un bucle cuando contador llega a 0 usando flags en op. aritméticas		
addi $8, $8, -1 beq $8, $0, exit	add**s** r8, r8, #-1 beq exit	// decrementa r8 y guarda estado en flags (Z) // mira flag Z, si Z=1 salta (Z cargado en adds)
Direccionamiento con autoincremento para recorrer estructuras en memoria		
lw $8, 0($9) add $9, $9, 4	ldr r8, [r9], **#4**	// direccionamiento post-auto-incrementado

	Instruction name	ARM	MIPS
Register-register	Add	ADD	addu, addiu
	Add (trap if overflow)	ADDS; SWIVS	add
	Subtract	SUB	subu
	Subtract (trap if overflow)	SUBS; SWIVS	sub
	Multiply	MUL	mult, multu
	Divide	—	div, divu
	And	AND	and
	Or	ORR	or
	Xor	EOR	xor
	Load high part register	MOVT	lui
	Shift left logical	LSL[1]	sllv, sll
	Shift right logical	LSR[1]	srlv, srl
	Shift right arithmetic	ASR[1]	srav, sra
	Compare	CMP, CMN, TST, TEQ	slt/i, slt/iu
Data transfer	Load byte signed	LDRSB	lb
	Load byte unsigned	LDRB	lbu
	Load halfword signed	LDRSH	lh
	Load halfword unsigned	LDRH	lhu
	Load word	LDR	lw
	Store byte	STRB	sb
	Store halfword	STRH	sh
	Store word	STR	sw
	Read, write special registers	MRS, MSR	move
	Atomic Exchange	SWP, SWPB	ll;sc

Modos de direccionamiento

Data addressing mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

FIGURE 2.31 Summary of data addressing modes in ARM vs. MIPS. MIPS has three basic addressing modes. The remaining six ARM addressing modes would require another instruction to calculate the address in MIPS.

Lenguaje ensamblador ARM

lw $1, dato($0)	**ldr** r2, =dato ldr r1, [r2]	// cargar dirección de dato en r2 // cargar dir de mem apuntada por r2 en r1
sw $1, dato($0)	**str** r2, =dato str r1, [r2]	// cargar dirección de dato en r2 // guardar en dir de mem apuntada por r2, r1
add $1, $2, $3 (sub …)	**add** r1, r2, r3 **(sub …)**	
addi $1, $2, 1	add r1, r2, #1	
sll $1, $2, 4	mov r1, r2, **LSL #4**	// desplazamiento lógico a izq. 4 bits de r2 a r1

sra $1, $2, 2
srl $1, $2, 2
	mov r1, r2, **ASR #2**
	mov r1, r2, **LSR #2**
	add r1, r1, r1,LSL #1
		// desplazamiento aritmético a drch.2 bits de r2 a r1
		// desplazamiento lógico a drch. 2 bits de r2 a r1
		// r1 ← r1 + (r1 << 1) = **3*r1** (multiplicar por 3)

j dir	**b** dir	// salta a la dirección dir
jal fun	**bl** fun	// salta a fun y guarda dir. sig. instrc. en reg. lr
jr $ra	**bx lr**	// salta a la dir. almacenada en lr (dir. retorno)

beq $1, $2, dir
(bne)
	cmp r1, r2
	beq dir
	(**bne**)
		// compara r1 y r2 (r1-r2) cargando flag Z.
		// mira flag Z, si Z=1 salta
		// (mira flag Z, si Z=0 salta)

Comandos del Shell de Linux

Descripción	Comando
Conectar por red a la Raspberry Pi (usuario: tc, password: tc)	`ssh -X tc@nombreRaspPi`
Copiar fichero por red a la Raspberry Pi (password: tc)	`scp fichero tc@nombreRaspPi:`
Copiar fichero desde la Raspberry Pi (password: tc)	`scp tc@nombreRaspPi:fichero .`
Mostrar contenido directorio	`ls`
Cambiar de directorio	`cd nombreDir`
Crear nuevo directorio	`mkdir nombreDir`
Borrar fichero	`rm nombreFichero`
Editar fichero ensamblador	`geany programa.s &`

Compilar fichero ensamblador
Enlazar
Enlazar con librería wiringPi
	`as -o programa.o programa.s`
	`gcc -o programa programa.o`
	`gcc -o programa programa.o -lwiringPi`

Ejecutar programa compilado Ejecutar programa que usa wiringPi (pide password: tc)	`./programa` `sudo ./programa`