Similarity assembly MIPS and ARM

`lw $1, dato($0)`	**`ldr`** r2, =dato ldr r1, [r2]	// load register r2 with mem addressed by dato // load register r1 with mem addressed by content r2
`sw $1, dato($0)`	**`str`** r2, =dato str r1, [r2]	// store memory address dato with r2 // store memory addressed by content r2 with 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`**	// shift left logical 4 bits of r2 put in 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
		// shift right arithmetical 2 bits r2 put in r1
		// shift right logical 2 bits r2 put in r1
		// r1 ← r1 + (r1 << 1) = **3*r1** (multiply by 3)

`j dir`	**`b`** dir	// branch to address dir
`jal fun`	**`bl`** fun	// jump and link to fun and save next instr in reg. lr
`jr $ra`	**`bx lr`**	// jump to return address in lr

`beq $1, $2, dir`
`(bne)`
	`cmp` r1, r2
	`beq` dir
	(**`bne`**)
		// compare r1 and r2 (r1-r2) loading flag Z.
		// check flag Z, on Z=1 branch
		// (check flag Z, on Z=0 branch)

Stack management		
`addi $sp, $sp, -4` `sw $ra, 0($sp)`	**`push {lr}`**	// save register content (lr) on stack. Several can be pushed simultaneously {r1, r2, r3, r4}
`lw $ra, 0($sp)` `addi $sp, $sp, 4`	**`pop {lr}`**	// pop stack content and put into register (lr). Several can be popped simultaneously {r1, r2, r3, r4}
Terminating a loop when the counter reaches zero using flags		
`addi $8, $8, -1` `beq $8, $0, exit`	add**`s`** r8, r8, #-1 beq exit	// decrease r8 and save flag (Z) value // check flag Z, on Z=1 branch (Z valued by adds)
Direccionamiento con autoincremento para recorrer estructuras en memoria		
`lw $8, 0($9)` `add $9, $9, 4`	ldr r8, [r9], **`#4`**	// post-auto-increment

AAPCS (Procedure Call Standard for the ARM Architecture)

Register	alias	special	Task in procedure call
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

Argument handling by procedure:

- 4 arguments via a1, a2, a3 and a4
- Rest via stack

procedure result

- Up to two results via a1 and a2

On procedure call a1, a2, a3 y a4 are not saved. Registers v1 to a v8 are saved on call.

Register code flags and conditional execution

Condition code flags	meaning
N	Enabled on Negative result from ALU
Z	Enabled on zero result from ALU
C	ALU operation Carried out
V	ALU operation oVerflow. 1 if result not representable in 2C for 32 bits

Condition field {cond}	meaning
EQ	Equal; Z is enabled (Z is 1)
NE	not equal; Z is disabled (Z is 0)
GE	Greater than or equal to in 2C; V and N are both enabled or disabled (V is N)
LT	less than in 2C, opposed to GE; V and N are not both enabled or disabled (V is not N)
GT	greater than in 2C; Z is disabled and N and V are both enabled or disabled; Z is 0, N is V
LE	less than or equal to in 2C; 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; N is enabled (N is 1)
PL	PLus/positive or zero; N is disabled (N is 0)
VS	oVerflow set; V is enabled (V is 1)
VC	oVerflow Clear; V is disabled (V is 0)
HI	Higher; C is enabled and Z is disabled (C is 1 and Z is 0)
LS	Less or Same; C is disabled or Z is enabled (C is 0 or Z is 1) CS/HS
CS/HS	carry set/higher or same; C is enabled (C is 1)
CC/LO	carry clear/lower; C is disabled (C is 0)

Write condition flags	cmp inst{s}: add**s**, sub**s**, and**s**, ...
Conditional branch (b)	b{cond}: b**eq**, b**ne**, b**gt**, b**le** ...
Conditional execution	inst{cond}: add**eq**, sub**ne**, ldr**gt**, ...

Linux shell commands

Description	Command
Connect by internet to Raspberry Pi (user: tc, password: tc)	`ssh -X tc@nameRaspPi`
Copy file by internet to Raspberry Pi (password: tc)	`scp file tc@nameRaspPi:`
Copy file from Raspberry Pi (password: tc)	`scp tc@nameRaspPi:file .`
Show directory content	`ls`
Change directory	`cd nameDir`
Make new directory	`mkdir nameDir`
Delete (remove) file	`rm nameFile`
Edit assembly file	`geany program.s &`

Compile assembly file
Link
Link with libraby wiringPi
	`as -o program.o program.s`
	`gcc -o program program.o`
	
	`gcc -o program program.o -lwiringPi`

Compile with Makefile	`make program`

Execute compiled program
Execute program using wiringPi (asks password: tc)
	`./program`
	
	`sudo ./program`

Close Raspberry Pi and delete files	`./deleteAndShutdown.sh`

Copying files to/from the Raspberry Pi in graphical way

1. Open the file explorer of Linux
2. In the address field put
   sftp://tc@nameRaspPi
3. Change directory to:
   /home/tc/practicas
4. Drag and drop files to/from this window

Note: You may use the CV task to store your files temporarily even when not finished.

Use of GDB when compiled with "-g"

Description	Command
launch gdb	`gdb [-tui] executable_name`
List source code	**l**`ist`
Put breakpoint	**b**`reak line-number`
List breakpoints	**i**`nfo` **break**
delete breakpoint	**d**`el num_breakpoint`
Delete all breakpoints	**cl**`ear`
Run program	**r**`un`
Consult register content CPU	**i**`nfo registers`
Disassemble machine code	**disas**`semble [starting_address ending_address]`
Stepwise instruction execution	**s**`tep i [number_steps]`
Stepwise execution (enters in functions in the source)	**n**`ext i`
Continue execution	**c**`ontinue`
Consult memory content	**x**`/nfs starting_address` **Options** / **Possible values** n: Number of ítems — any number f: Format — **o**ctal, he**x**, **d**ecimal,**u**nsigned decimal, bi**t**, **f**loat, **a**ddress, **i**nstruction, **c**har, and **s**tring s: Size — **b**yte, **h**alfword, **w**ord, **g**iant(8B) Examp. 12 words at the top of the stack in hexadecimal.: `x/12xw $sp`
print memory/register	**p**`rint/f label/reg`
exit	**q**`uit`

TUI (Text layout Interface for gdb) commands

Description	Command
Activate/deactivate TUI	`C-x A`
Change active window	`focus next/prev src/asm/split/regs, C-x o`
Change shown windows	`layout next/prev/src/asm/split/regs, C-x 1, C-x 2`
Refresh screen	`refresh, C-L`

Library *wiringPi*

Function calls from assembler:

1. Load function arguments in registers. Up to 4 arguments via registers: 1st argument in r0, 2nd in r1, …)
2. Call function:
   bl *function_name*
3. Catch function result value from register r0

function	Arguments	Description
int **wiringPiSetup** (void)	None	Initializes the library. Call before using a library function
void **pinMode** (int pin, int mode)	**pin**: pin id to configure **mode**: pin operation mode (INPUT, OUTPUT)	Configures pin mode. Argument "pin" can be (INPUT, 0) or (OUTPUT, 1).
void **digitalWrite** (int pin, int value)	**pin**: pin id to write into **value**: value to write	Generates output of argument "value" (0, 1) in the pin argument "pin". Pin should be initialized as (OUTPUT) pin with pinMode.
int **digitalRead** (int pin)	**pin**: pin id	Reads pin state (0, 1) of argument "pin". Value is put into register r0.
void **delay** (unsigned int howLong)	**howLong**: number of milliseconds	Suspends program execution during "howLong" milliseconds
void **delayMicroseconds** (unsigned int howLong)	**howLong**: number of microseconds	Suspends program execution during "howLong" microseconds

Example pin state consult, corresponding to a button on the board connected to the Raspberry Pi.

```
.include "wiringPiPins.s"      // file with pin definitions #BUTTON1, #RLED1, …
                               // and pin modes #INPUT, #OUTPUT

      bl wiringPiSetup         // initialize wiringPi library

                               // configuring button 1 pin as input (pinMode):
      mov r0, #BUTTON1         // button pin in r0 → first function argument
      mov r1, #INPUT           // give pin mode to r1 → second argument
      bl pinMode               // call pinMode

                               // Read button 1 state with digitalRead:
      mov r0, #BUTTON1         // button pin in r0 → first function argument
      bl digitalRead           // call function digitalRead
      cmp r0, #0               // register r0 contains the button pin value (0,1)
```