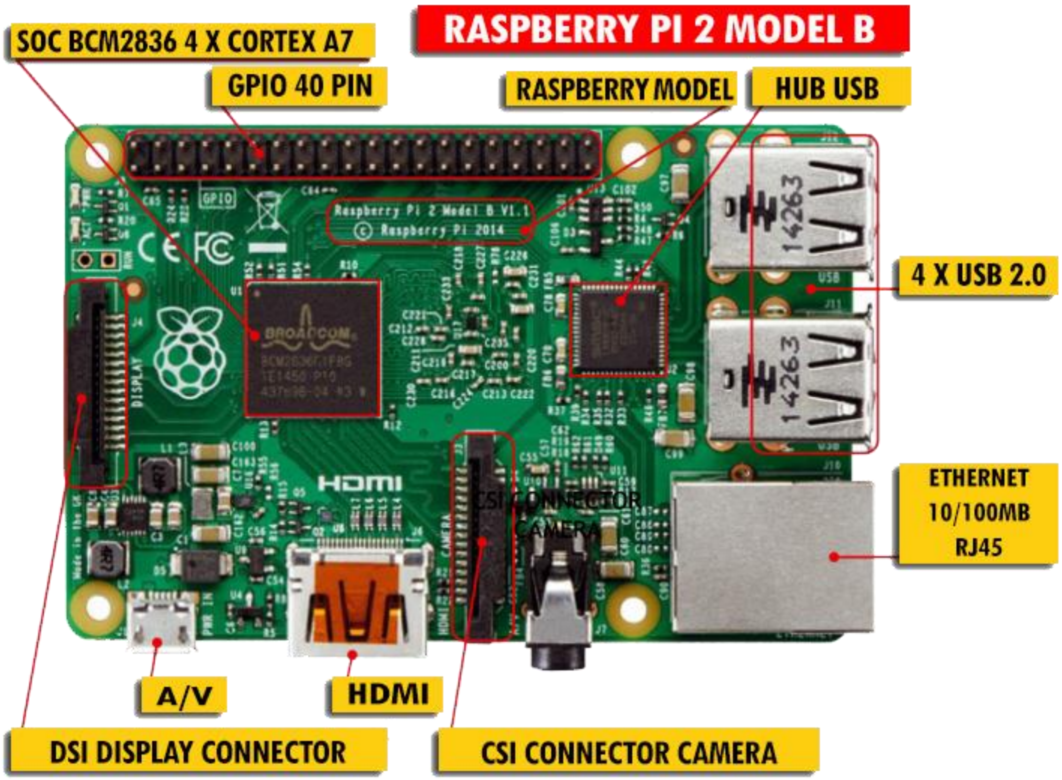
Lab Rasperry Pi

Computer Technology

Raspberry Pi 2

- Processor ARM Cortex-A7 (ARMv7) Quad core Broadcom BCM2836
  - Specific for Raspberry Pi 2
  - Cortex-A7 aims at low energy consumption like Cortex-A15 in mobiles



Devices with Cortex A-7 y A-15

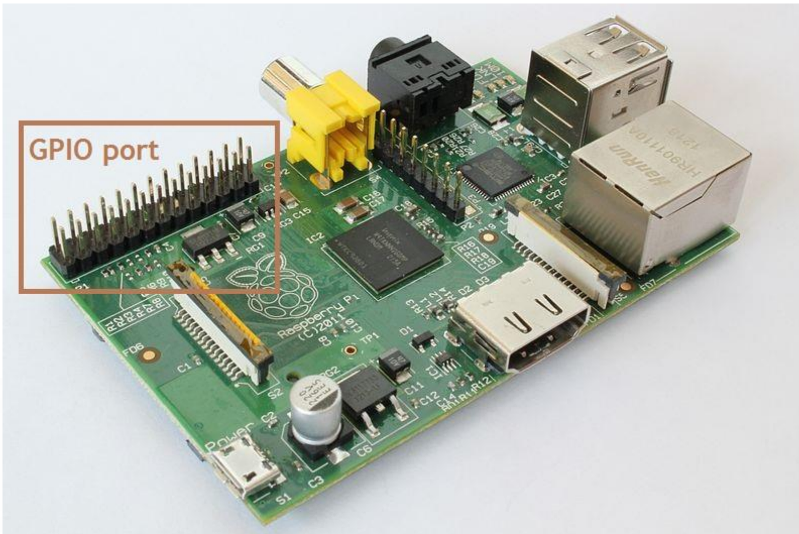
2013

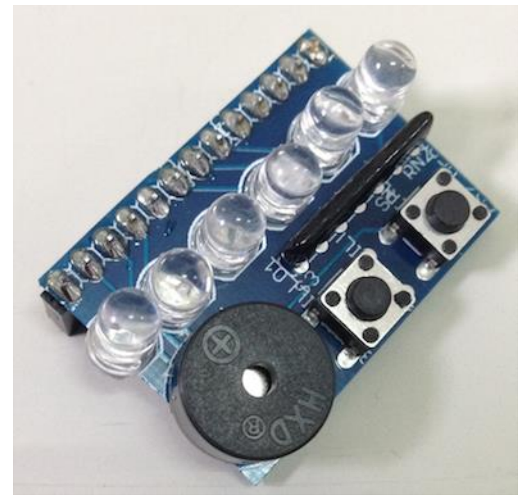
2013

2014

2014

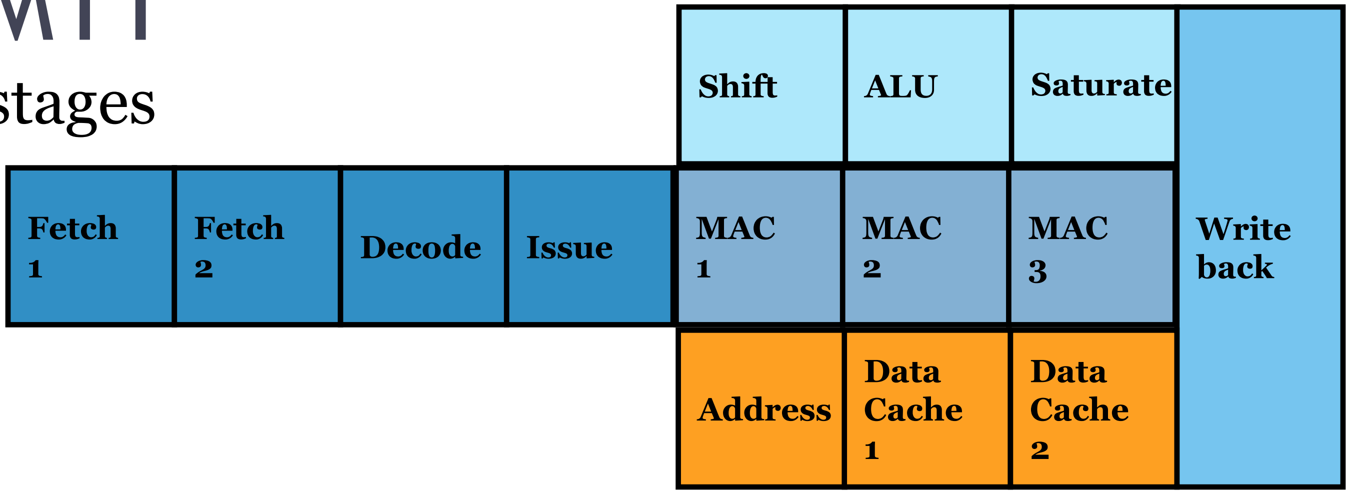
Raspberry Pi: Interrupts and GPIO





ARM11

- 8 stages



- 1. Fe1 – Send address and receive instruction
- 2. Fe2 – jump prediction
- 3. De – instruction decoding
- 4. Iss – read registers
- 5. Sh – offset and shift operations
- 6. ALU – integer arithmetic operations
- 7. Sat – result saturation
- 8. WB – write results into registers

Instruction and data size

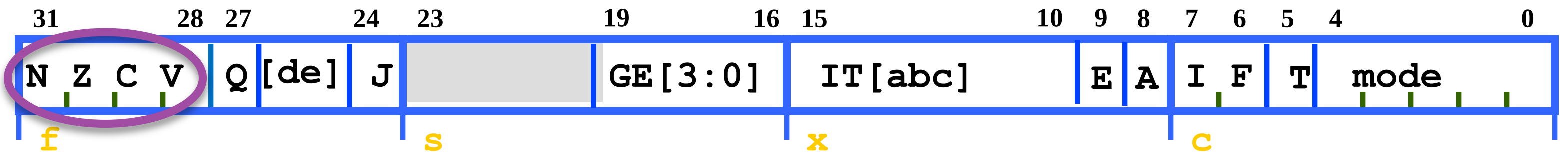
- ARM is a 32-bit RISC architecture, Load-Store like MIPS
- In ARM (= MIPS):
  - **Byte** is 8 bits
  - **Halfword** is 16 bits (2 bytes)
  - **Word** is 32 bits (4 bytes)

- Most ARM implement two instruction sets:
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set

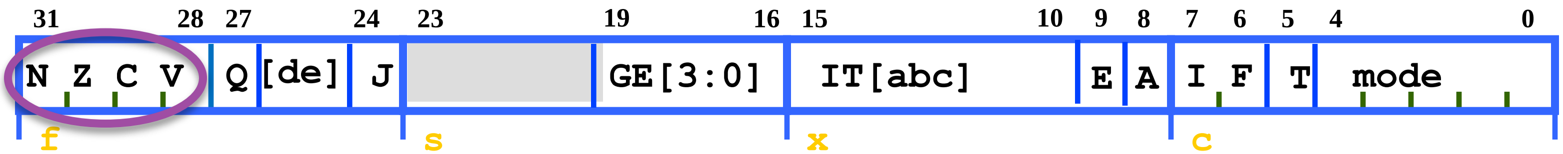
    (Jazelle cores can also execute Java bytecode)

- When running ARM code, the processor has:
  - All instructions at size 32 bits
  - All instructions are made towards word level
  - The PC value is stored in bits [31:2], such that bits [1:0] are undefined

Condition flags



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed, i.e. result cannot be represented in 32 bits 2C



- **Condition code flags**
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed
- **Sticky Overflow flag - Q flag**
  - Indicates if saturation has occurred
- **SIMD Condition code bits – GE[3:0]**
  - Used by some SIMD instructions
- **IF THEN status bits – IT[abcde]**
  - Controls conditional execution of Thumb instructions
- **T bit**
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **J bit**
  - J = 1: Processor in Jazelle state
- **Mode bits**
  - Specify the processor mode
- **Interrupt Disable bits**
  - I = 1: Disables IRQ
  - F = 1: Disables FIQ
- **E bit**
  - E = 0: Data load/store is little endian
  - E = 1: Data load/store is bigendian
- **A bit**
  - A = 1: Disable imprecise data aborts

Conditional execution

- Using the right suffix, an ARM instruction can be run conditionally. Core compares condition field in instruction against NZCV flags to determine if instruction should be executed.
  - 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
```

- Data operation instructions by default do not affect condition flag values. This can be done by using a suffix "S". Instructions comparing register values like CMP, do not require "S".

```
loop
    …
    SUBS r1,r1,#1     ; decrement r1 and set flags
    BNE loop          ; if Z flag clear then branch
```

Condition field and conditional execution

Condition field {cond}	means
EQ	(equal) **Z** is enabled (**Z is 1**)
NE	(not equal). **Z** is disabled. (**Z is 0**)
GE	(greater than or equal to, in two's complement). When both **V** and **N** are enabled or disabled (**V is N**)
LT	(less than, in 2C). This is the opposite of GE, so when **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 alternatively **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 than or same) **C** is disabled or **Z** is enabled (**C is 0 or Z is 1**)
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}: adds, subs, ands, …
Conditional branch (b)	b{cond}: b**eq**, b**ne**, b**gt**, b**le** …
Conditional execution	inst{cond}: add**eq**, sub**ne**, ldr**gt**, …

Conditional execution examples

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 memory to register
				lfword memory to register
				lfword memory to register
				lfword register to memory
				te from memory to register
				te from memory to register
				te from register to memory
			Memory[r2 + 20] = r1	omic 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

Data process operations work only with registers like MIPS

Data transfer to registers

- Immediate operand:
  - `mov r0, #500 @immediate size limited`
    - Pseudo-instruction for large values
      - `LDR rd, =const`

- From and to memory with various sizes:

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

  - Syntax:
    - `LDR`{<cond>}{<size>} Rd, <address>
    - `STR`{<cond>}{<size>} Rd, <address>

    e.g. `LDREQB`

Effective address

- Effective address used by LDR/STR given by base register and offset
- offset can be (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]
    ```
- … and added or subtracted from a base register value:

    ```
    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)

Effective address on pre/post indexed

- For not changing the base register can be used:
  - `str r10, [r9, #4] @[r9+4]<- r10 (register index r9 not changed)`
  - `ldr r8, [r9, +r10, LSL #3]    @r8<-[r9+(r10*8)] (index r9 not changed)`
- Base register can be updated via pre-indexing or post-indexing:

  pre-indexed addressing examples:
  - `ldr r2, [r1, #4]!   @r1<- r1+4 then r2<-[r1]`
  - `ldr r2, [r0, r1]!   @r0<- r0+r1  then r2<-[r0]`
  - `ldr r2, [r0, r1,  lsl #2]! @r0<-r0 + r1*4   then  r2<-[r0]`

  post-indexed addressing examples:
  - `str r10, [r9], #4  @[r9]<- r10 then 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 (ARM Architecture Procedure Call Standard)

Register	Synonym	special	Task in function 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 transfer

- Argument transfer:
  - First 4 arguments via a1, a2, a3 and a4
  - Rest via stack
- Function result
  - Up to 2 results via a1 and a2
- Not saved on call a1, a2, a3 and a4
- Saved on function call v1 to v8

Assembly language ARM

Stack management		
addi $sp, $sp, -4 sw $ra, 0($sp)	push {lr}	// load register content (lr) in stack. Several are saved simultaneously {r1, r2, r3, r4}
lw $ra, 0($sp) addi $sp, $sp, 4	pop {lr}	// get stack content and put in register (lr). Several can be popped simultaneously {r1, r2, r3, r4}
Ending a loop on counter value 0 using flags in arithmetic operations		
addi $8, $8, -1 beq $8, $0, exit	adds r8, r8, #-1 beq exit	// decreases r8 and saves condition flags (Z) // checks flag Z, on Z=1 branch (Z set by adds)
Using auto-incrementing for going through a memory structure		
lw $8, 0($9) add $9, $9, 4	ldr r8, [r9], #4	// post-auto-incrementing addressing

	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

Addressing mode

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.

Assembly language ARM

lw $1, dato($0)	ldr r2, =dato ldr r1, [r2]	// load content memory address dato into r2 // load content memory addressed by r2 into r1
sw $1, dato($0)	str r2, =dato str r1, [r2]	// store content r2 in memory address dato // sore content r1 in memory directory in r2
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**	// 4 bits logical shift left of r2 to 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
		// 2 bits arithmetic shift right of r2 to r1
		// 2 bits logical shift right of r2 to r1
		// r1 ← r1 + (r1 << 1) = **3*r1** (multiply by 3)

j dir	b dir	// jump to address dir
jal fun	bl fun	// jump to fun and save address next instruction in lr
jr $ra	bx lr	// jump to address in lr (return address)

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

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