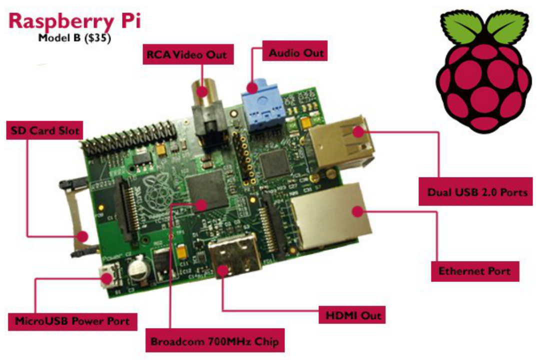
Raspberry Pi

ARM ISA review

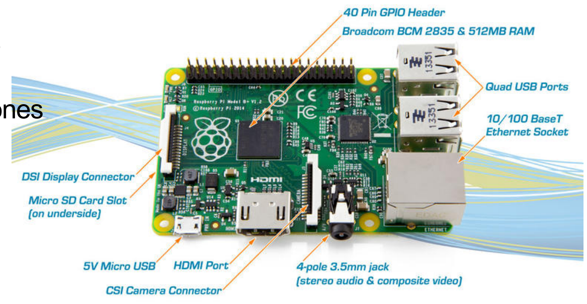
Raspberry Pi

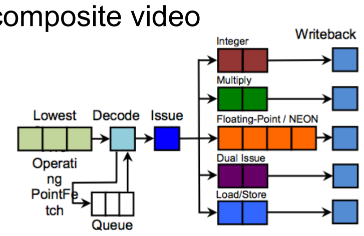
- Based on ARM11 (ARMv6) Broadcom BCM2835/BCM2836.
- Low Power ARM1176JZ-F Applications Processor
- ARM11 is the processor integrated in the first iPhone (also in iPhone 3G).



Raspberry Pi 2

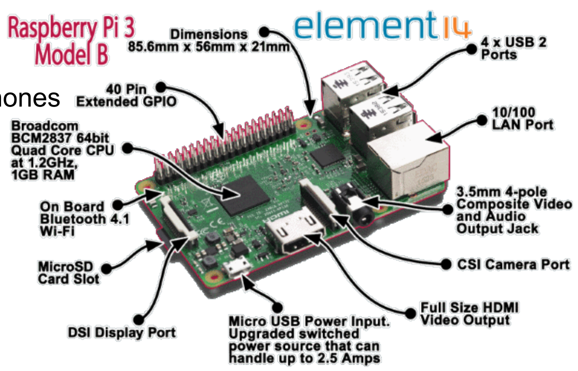
- ARM Cortex A7
  - 900 MHZ quad-core
  - 1GB RAM
  - mid-range smartphones
- 4 USB ports
- 40 GPIO pins
- Full HDMI port
- Ethernet port
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)
- Micro SD card slot
- VideoCore IV 3D graphics core


40 Pin GPIO Header
Broadcom BCM 2835 & 512MB RAM
Quad USB Ports
10/100 BaseT Ethernet Socket
DSI Display Connector
Micro SD Card Slot (on underside)
5V Micro USB
HDMI Port
CSI Camera Connector
4-pole 3.5mm jack (stereo audio & composite video)



Raspberry Pi 3

- ARMv8 Cortex A-53 (**64 bits**)
  - **1.2 GHZ** quad-core
  - 1GB RAM
  - high-range smartphones
- 4 USB ports
- 40 GPIO pins
- Full HDMI port
- ….
- 10/100 Ethernet port
- **Wifi 802.11n**
- **Bluetooth 4.1 (Low Energy)**


Raspberry Pi 3 Model B
Dimensions 85.6mm x 56mm x 21mm
element14
4 x USB 2 Ports
40 Pin Extended GPIO
10/100 LAN Port
Broadcom BCM2837 64bit Quad Core CPU at 1.2GHz, 1GB RAM
On Board Bluetooth 4.1 Wi-Fi
3.5mm 4-pole Composite Video and Audio Output Jack
MicroSD Card Slot
CSI Camera Port
DSI Display Port
Micro USB Power Input. Upgraded switched power source that can handle up to 2.5 Amps
Full Size HDMI Video Output

Devices with Cortex-A53

Samsung Galaxy Tab A 10.1

Huawei Nova Plus

Egolggo S95X smart TV set of box

Lenovo Yoga Tab 3 10

Samsung Galaxy C7

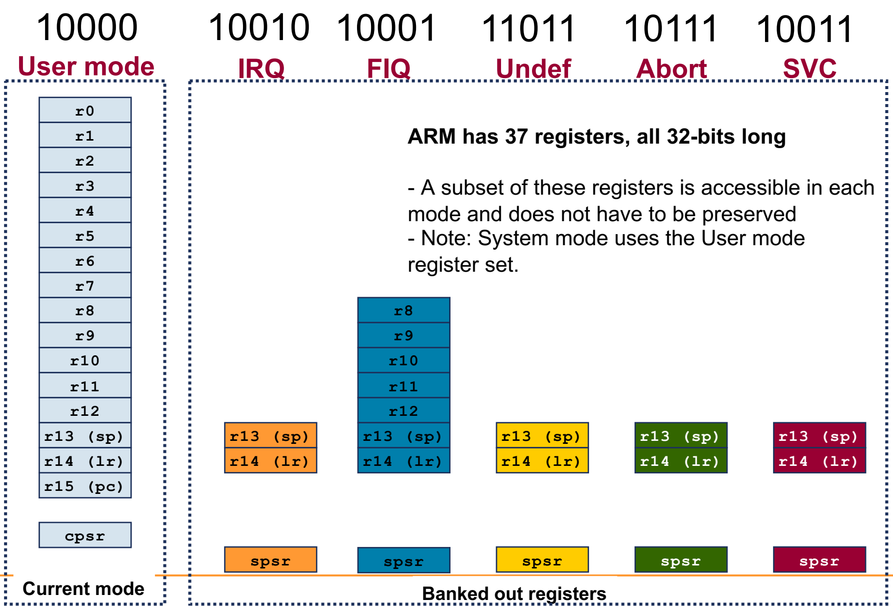
Data Sizes and Instruction Sets

- The ARM is a 32-bit, Load-Store RISC architecture.
- When used in relation to the ARM:
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
- Most ARM’s implement two instruction sets
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set
- Jazelle cores can also execute Java bytecode

Processor Modes

- The ARM has seven basic operating modes, used to run user tasks, an operating system, and to efficiently handle exceptions such as interrupts:
  - **User** : unprivileged mode under which most tasks run
  - **FIQ** : entered when a high priority (fast) interrupt is raised
  - **IRQ** : entered when a low priority (normal) interrupt is raised
  - **Supervisor** : entered on start up or reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations as a result of fetching instructions or accessing data.
  - **Undef** : used to handle unknown or illegal instructions
  - **System** : privileged mode using the same registers as user mode

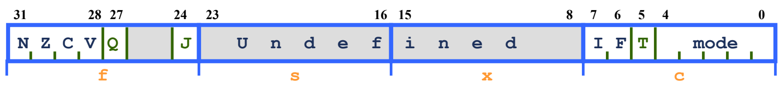
The ARM Register Set



ARM has 37 registers, all 32-bits long

- A subset of these registers is accessible in each mode and does not have to be preserved
- Note: System mode uses the User mode register set.

Program Status Registers



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed (the result is not representable in 32 bits (C2))
- **Sticky Overflow flag - Q flag**
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- **J bit**
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state
- **Interrupt Disable bits.**
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- **T Bit**
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **Mode bits**
  - Specify the processor mode

Program Counter (r15)

- **When the processor is executing in ARM state:**
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the pc value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned)
- **When the processor is executing in Thumb state:**
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the pc value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned)
- **When the processor is executing in Jazelle state:**
  - All instructions are 8 bits wide
  - Processor performs a word access to read 4 instructions at once

Conditional Execution (predicated inst.)

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

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

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

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

decrement r1 and set flags

if Z flag clear then branch

Condition Codes

Mnemonic {cond}	Meaning
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**)

Update status register	**cmp** **inst{s}: adds, subs, ands, …**
Brach (b) cond.	b{cond}: b**eq**, b**ne**, b**gt**, b**le** …
Instruction cond.	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

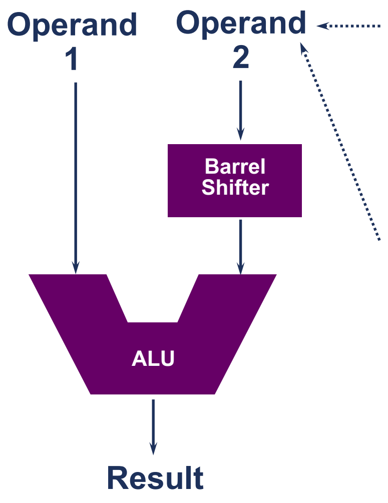
Data Processing Instructions

- Consist of :
  - Arithmetic: `ADD` `ADC` `SUB` `SBC` `RSB` `RSC`
  - Logical: `AND` `ORR` `EOR` `BIC`
  - Comparisons: `CMP` `CMN` `TST` `TEQ`
  - Data movement: `MOV` `MVN`
- These instructions only work on registers, NOT memory.
- Syntax:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn
- Second operand is sent to the ALU via barrel shifter.

Using a Barrel Shifter:The 2nd Operand



Register, optionally with shift operation

- Shift value can either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

Data Processing Solutions

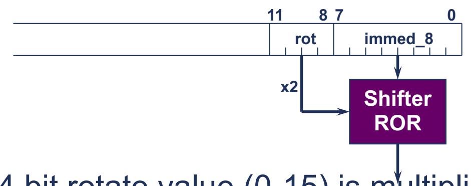
```
1. MOV      r6, #0

2. MOVS     r7,r7          ; set the flags
   RSBMI    r7,r7,#0       ; if neg, r7=0-r7

3. ADD      r9,r8,r8,LSL #2      ; r9=r8*5
   RSB      r10,r9,r9,LSL #3     ; r10=r9*7
```

Immediate constants

- No ARM instruction can contain a 32 bit immediate constant
  - All ARM instructions are fixed as 32 bits long
- The data processing instruction format has 12 bits available for operand2



Quick Quiz:

```
0x3FC
MOV r0, #???
```

- 4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2
- Rule to remember is

  "8-bits rotated right by an even number of bit positions"

Loading 32 bit constants

- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
  - `LDR rd, =const`
- This will either:
  - Produce a **`MOV`** or **`MVN`** instruction to generate the value (if possible).

  or
  - Generate a **`LDR`** instruction with a PC-relative address to read the constant from a *literal pool* (Constant data area embedded in the code).
- For example
  - `LDR r0,=0xFF` => `MOV r0,#0xFF`
  - `LDR r0,=0x55555555` => `LDR r0,[PC,#Imm12]`

```
...
.LTORG
DCD 0x55555555
```

- This is the recommended way of loading constants into a register

Single register data transfer

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

- Memory system must support all access sizes
- Syntax:
  - `LDR`{<cond>}{<size>} Rd, <address>
  - `STR`{<cond>}{<size>} Rd, <address>

e.g. `LDREQB`

Address accessed

- Address accessed by LDR/STR is specified by a base register with an offset
- For word and unsigned byte accesses, offset can be:
  - 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]
    ```
- This can be either added or subtracted from the base register:
  ```
  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)
- Choice of *pre-indexed* or *post-indexed* addressing
- Choice of whether to update the base pointer (pre-indexed only)
  ```
  LDR r0, [r1, #-8]!
  ```

Load/Store Exercise

Assume an array of 25 words. A compiler associates y with r1. Assume that the base address for the array is located in r2. Translate this C statement/assignment using just three instructions:

```
array[10] = array[5] + y;
```

Load/Store Exercise Solution

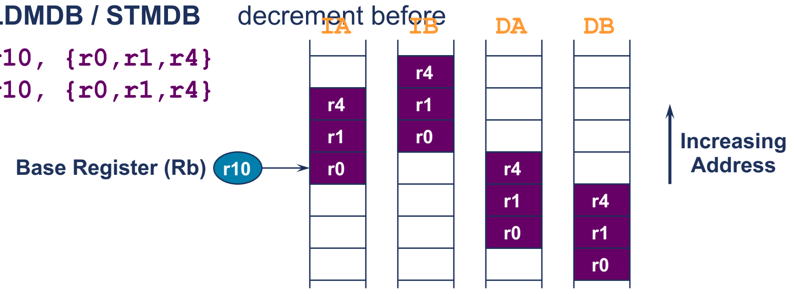
```
array[10] = array[5] + y;
```

```
LDR    r3, [r2, #20]  ; r3 = array[5]
ADD    r3, r3, r1      ; r3 = array[5] + y
STR    r3, [r2, #40]   ; array[5] + y =
array[10]
```

Load and Store Multiples

- Syntax:
  - **<LDM|STM>**{<cond>}<addressing_mode> Rb{!}, <register list>
- 4 addressing modes:
  - **LDMIA / STMIA** increment after
  - **LDMIB / STMIB** increment before
  - **LDMDA / STMDA** decrement after
  - **LDMDB / STMDB** decrement before

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```



Multiply and Divide

- There are 2 classes of multiply - producing 32-bit and 64-bit results
- 32-bit versions on an ARM7TDMI will execute in 2 - 5 cycles

```
MUL r0, r1, r2            ; r0 = r1 * r2
MLA r0, r1, r2, r3        ; r0 = (r1 * r2) + r3
```

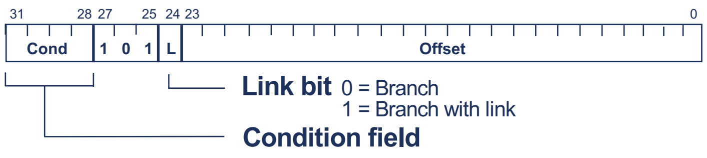
- 64-bit multiply instructions offer both signed and unsigned versions
  - For these instruction there are 2 destination registers

```
[U|S]MULL r4, r5, r2, r3 ; r5:r4 = r2 * r3
[U|S]MLAL r4, r5, r2, r3 ; r5:r4 = (r2 * r3) + r5:r4
```

- Most ARM cores do not offer integer divide instructions
  - Division operations will be performed by C library routines or inline shifts

Branch instructions

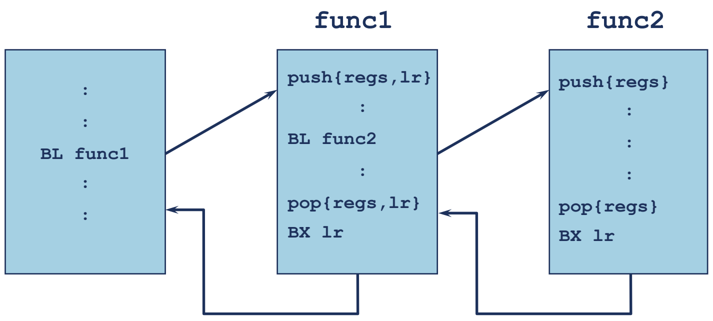
- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`



- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

ARM Branches and Subroutines

- **B <label>**
  - PC relative. ±32 Mbyte range.
- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked



```
:
:
BL func1
:
:
```

func1

```
push{regs,lr}
:
BL func2
:
pop{regs,lr}
BX lr
```

func2

```
push{regs}
:
:
:
pop{regs}
BX lr
```

Register Usage

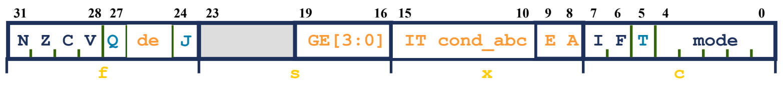
	Register	
Arguments into function Result(s) from function otherwise corruptible (Additional parameters passed on stack)	r0 r1 r2 r3	
Register variables Must be preserved	r4 r5 r6 r7 r8 r9/sb r10/sl r11	 - Stack base - Stack limit if software stack checking selected
Scratch register (corruptible)	r12	
Stack Pointer Link Register Program Counter	r13/sp r14/lr r15/pc	- SP should always be 8-byte (2 word) aligned - R14 can be used as a temporary once value stacked

The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see **AAPCS**)

CPSR flags may be corrupted by function call. Assembler code which links with compiled code must follow the AAPCS at external interfaces

The AAPCS is part of the new ABI for the ARM Architecture

PSR access



- MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register or take an immediate value
  - MSR allows the whole status register, or just parts of it to be updated
- Interrupts can be enable/disabled and modes changed, by writing to the CPSR
  - Typically a read/modify/write strategy should be used:

```
MRS r0,CPSR        ; read CPSR into r0
BIC r0,r0,#0x80    ; clear bit 7 to enable IRQ (BIt Clear)
MSR CPSR_c,r0      ; write modified value to ‘c’ byte only
```

- In User Mode, all bits can be read but only the condition flags (_f) can be modified

Addressing modes

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	—

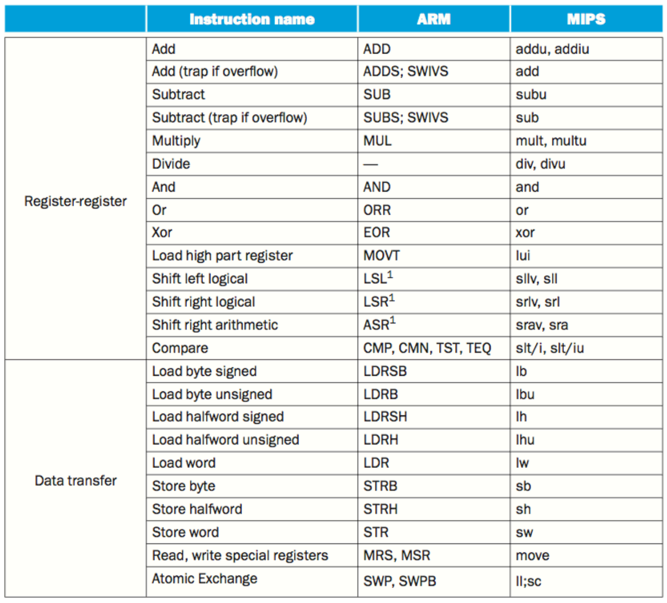
FIGURE 2.31 Summary of data addressing modes in ARM vs. MIPS. MIPS has three basic addressing modes. The remaining six ARM addressing modes would require another instruction to calculate the address in MIPS.

ARM assembly language

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 vs MIPS



	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

Examples: MIPS vs 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)	**ldr 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)

Other examples of ARM vs. MIPS

Stack management		
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}
Loop control when a counter reaches 0		
addi $8, $8, -1 beq $8, $0, exit	adds r8, r8, #-1 beq exit	// decrementa r8 y guarda estado en flags (Z) // mira flag Z, si Z=1 salta (Z cargado en adds)
Auto-increment operation to traverse data structures		
lw $8, 0($9) add $9, $9, 4	ldr r8, [r9], #4	// direccionamiento post-auto-incrementado