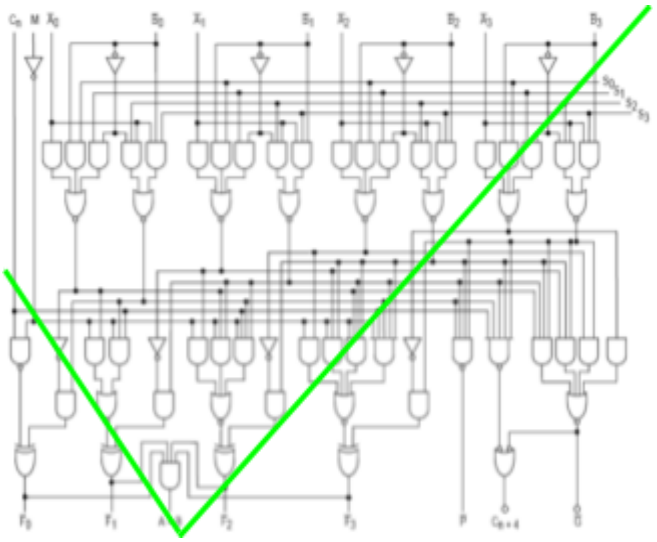
Estructura de Computadors



Tema 1: Introducció

1.1 - Descripció jeràrquica del computador:

- Hardware (Portes lògiques)
- Llenguatge màquina (MIPS, ARM)
- Llenguatge alt nivell(C++, Java)
- Llenguatge usuari (User's UI)

1.1.1 - Traducció: Compilació vs Interpretació

La traducció facilita la comoditat per a canviar de plataformes i permet que evolucionin independentment, però adaptant-se mutuament.

Compilació: Un programa traductor converteix el llenguatge a més baix nivell. No executa, emmagatzema.

- **Pros:** Sols s'ha de traduir un cop i permet la optimització i millora de rendiment.
- **Cons:** S'ha de re-compilar cada cop que es vol adaptar a un sistema diferent.
- **Exemple:** C++

Interpretació: Llegeix una per una les accions del programa i executa una seqüència equivalent d'accions en un llenguatge de més baix nivell . Només executa.

- **Pros:** Es poden executar en tots els sistemes sense necessitat d'adaptar-lo.
- **Contras:** El programa s'ha de traduir cada cop que es vol executar.
- **Exemple:** Java

1.2 - Mesura del rendiment

1.2.1 - Definició del Rendiment

Sempre s'aspira a millorar el rendiment. Pero què s'entén per rendiment:

- Temps d'execució o productivitat:
  - **Temps d'execució:** temps transcorregut des de l'inici fins el final de la tasca. Com menys temps, més rendiment. $Rendiment = 1/t_{execució}$
  - **Productivitat:** Nombre de tasques completades per unitat de temps.
- Temps de CPU: Es la suma dels diversos components (Temps CPU + Temps d'espera + Temps acomulat mentre s'executen altres programens concurrents.

1.2.2 - Factors que influeixen en el temps d'execució

Podem desglosar el temps d'execució en:

$$t_{exe} = n_{cicles} \cdot t_c = n_{cicles} / f_{clock}$$

Direm que t_{exe} es el temps d'execució de la CPU, n_{cicles} es el número de cicles del rellotge en l'execució del programa i t_c i f_{clock} són el període i la freqüència del rellotge, respectivament.

Reduir el n_{cicles}:

El més intuïtiu és fer que el programa s'executi en menys instruccions. Sent CPI el nombre de cicles promig per instrucció del programa,

$$n_{cicles} = n_{ins} \cdot CPI$$

Per tant, el temps d'execució és:

$$t_{exe} = n_{ins} \cdot CPI \cdot t_c$$

Per tant, sabent que a cada instrucció li correspon un CPI, l'ideal és reduir el programa a utilitzar instruccions amb CPI més baix.

Cal anar amb compte amb el concepte CPI, perquè pot significar:

- CPI promig d'un programa o conjunt de programes en un cert computador
- CPI fix d'un tipus d'instruccions en un cert computador: Donat un determinat computador, cada tipus d'instrucció tarda un cert nombre de cicles, i en molt casos és un nombre fix per a cada tipus

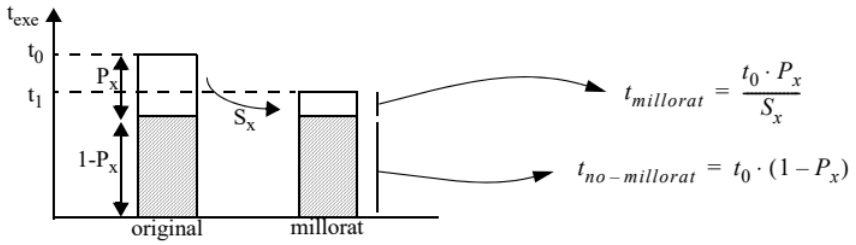
Augmentar la freqüència de rellotge:

Suposem un circuit combinacional "α" que llegeix dades d'entrada d'un biestable i escriu dades de sortida a un altre biestable. En principi, reduint el temps de cicle podem aconseguir que calculi més resultats per unitat de temps. Pero, si reduïm excessivament el temps de cicle, pot ser que les dades encara no estiguin calculades, i que el resultat correcte no s'obtingui fins al següent cicle de rellotge.

1.3 - Llei d'Amdahl

Cal concentrar els esforços d'optimització en aquelles parts del programa que comporten el major cost en el temps.

El rendiment total de S:



$$S = \frac{t_0}{t_1} = \frac{t_0}{\frac{t_0 \cdot Px}{Sx} + t_0 \cdot (1 - Px)} = \frac{1}{\frac{Px}{Sx} + (1 - Px)}$$

Quin guany total màxim:

$$\lim_{Sx \to \infty} S = S_{max} = \frac{1}{1 - Px}$$

1.4 - Mesures de dissipació de potència

A mesura que s'ha anat optimitzant i la freqüència del rellotge s'ha anat reduint, la dissipació de potencia ha crescut i la calor dissipada amb ella. Per tant cal aplicar sistemes de refrigeració per a reduir-ne l'esmentada dissipació.
La potència és l'energia dissipada per corrents elèctrics en el circuit, i es pot dividir en dos tipus: Potencia dinàmica (P_d), causada pels corrents de càrrega i descàrrega (durant la commutació de portes); i la Potencia estàtica (P_s), fruit dels corrents paràsits que circulen pels transistors (tant si commuten, com si no).

$$P = P_d + P_s$$

1.4.1 - Càlcul de la potencia dinàmica

La càrrega desplaçada (q_i) en un cicle complet és de:

$$q_i = C_i \cdot V$$

On C_i és la capacitat equivalent en farads del circuit connectat a la sortida de la porta. Depèn del nombre, grandària i material dels transistors i connectors a la sortida de la porta.
On V és la diferència de tensió entre l'estat inicial i final dels elements capacitatius.

El corrent elèctric generat en tot el circuit durant un cicle de rellotge és:

$$I = \Sigma(q_i / t_c) = \Sigma(C_i \cdot V \cdot f_{clock})$$

On t_c és el temps de cicle del rellotge.

Donat que no totes les portes commuten en cada cicle, es defineix el factor d'activitat (α) com a promig de portes per a un determinat programa. Sol tenir un valor pròxim a 0,1.
Per tant, la potencia dinàmica dissipada pel circuit és:

$$P = \alpha \cdot C \cdot V^2 \cdot f_{clock}$$

o cuándo les apetezca:

$$P = C \cdot V^2 \cdot f_{clock}$$

L'energia dissipada en un temps, en Joules, és:

$$E = P_d \cdot t$$

1.4.2 - Càlcul de la potència estàtica

Hem suposat que pels transistors en circuit - obert no hi circula corrent, però en realitat no estan mai completament en circuit - obert sinó que sempre hi circula de manera constant:

$$P_s = I_{leak} \cdot V$$

1.4.2.1 - Conseqüències derivades de la miniaturització

- C**:** transistors cada vegada més petits i amb menys capacitat, pero com caben més, la diferència no es nota.
- f**:** la reducció del temps de commutació ha permès augmentar la freqüència de rellotge (s'ha multiplicat per 300 en 25 anys). Això comporta un increment de la potència dinàmica.
- V**:** per compensar l'increment de potència dinàmica, s'ha anat reduint V (de 5 a 1V) i també V_t (de 0.7 a 0.4V). La reducció de V_t provoca un augment exponencial de corrents de fuga i potència estàtica. D'aquesta manera, no s'ha pogut mantenir estàtica la potència dissipada (multiplicada per 30 en 25 anys).

El resultat dels tres factors és que la freqüència ja no es pot augmentar més ja que això faria incrementar la potència i la temperatura posant en risc que sobrepassi els marges de tolerància.

1.4.2.1 - Algunes tècniques de reducció del consum

- **Clock gating:** Permet reduir α. Consisteix en inhabilitar selectivament la senyal del rellotge de determinats circuits durant períodes de temps en què sabem que els biestables que no han de canviar l'estat
- **Dynamic voltage and frequency scaling (DVFS):** Permet reduir V i f. Consisteix a reduir temporalment la tensió i la freqüència d'una part del circuit quan aquest no ha d'operar a màxima velocitat.
- **Power gating:** Redueix el consum tallant temporalment l'alimentació de determinats circuits quan no s'han d'usar durant un llarg període de temps

1.6 - Exercicis

1.6.1 - Exercici 1.3:

Traduïm un programa en alt nivell a llenguatge assemblador i en fem 2 versions: Una en MIPS, que s'executarà en els processadors P1 i P2. L'altra, en x86, que s'executarà en P3. Cada processador té les següents característiques:

Processador	Freqüència	CPI mitja	ISA	#Instruccions
P1	**2 Ghz**	**1.5**	**MIPS**	**2*10^6**
P2	**2.5 Ghz**	**1**	**MIPS**	**2*10^6**
P3	**3 Ghz**	**0.75**	**Intel x86**	**10^6**

Contesta les següents preguntes raonant les respostes:

a) Quin dels tres processadors executa més ràpidament el programa? Quantes vegades és més ràpid que els altres dos?

- Per a calcular quin és més ràpid, calculem el temps d'execució de cadascun d'ells:

  $t.P_1 = 1.5\,x10^{-3}\,s$

  $t.P_2 = 0.8\,x10^{-3}\,s$

  $t.P_3 = 0.25\,x10^{-3}\,s$

  Per tant podem veure que el P_3 és el processador més ràpid. Per a saber quantes vegades més ràpid és, operem:

  $\beta = t.P_1 / t.P_3 = 6$

  $\gamma = t.P_2 / t.P_3 = 3.2$

  On β és el nombre de vegades més ràpid que és P_3 respecte P_1 i γ el nombre de vegades més ràpid que és P_3 respecte P_2.

b) Sense conèixer el nombre d'instruccions, podem calcular si el processador P1 executa el programa més ràpid que el processador P2? I més ràpid que el processador P3?

- En teoría no podem saber-ho ja que si en un processador ⅓ més lent (en quant a freqüència i PCI) executa ½ d'instruccions (per exemple) seria més ràpid. Així que és necessari saber quantes instruccions s'executen en cada processador per poder comparar la velocitat del mateix programa en diferents processadors.

c) Amb aquestes dades, podem comparar el rendiment de P1 i P2 en general (no només per aquest programa)? I el rendiment de P1 i P3? I si el CPI fos el CPI mitjà per a tots els programes?

- Podem comparar els processadors P1 i P2, ja que el seu codi ISA es el mateix, però no els podem comparar amb el P3, ja que uns tenen un ISA MIPS i l'altre un ISA Intel x86. En canvi, si tinguéssim el CPI mitjà de cada processador, amb la freqüència i el CPI podem saber quin de tots té més rendiment.

1.6.2 - Exercici 1.9:

La següent taula mostra la freqüència de rellotge (F), voltatge (V) i potència dinàmica (P) de dos processadors.

Processador	F	V	P	Càrrega capacitiva (C)
A	10 MHz	5V	2W	
B	3GHz	1V	100W	

a) Calcula la càrrega capacitiva dels processadors A i B.

$P = C \cdot V^2 \cdot F \rightarrow C = \frac{P}{V^2 \cdot F} \rightarrow C_A = 8 \cdot 10^{-9};\ C_B = 3,3 \cdot 10^{-8}$

b) Quina seria la potència del processador A si, sense canviar-ne el voltatge ni la capacitància, volguèssim aconseguir la mateixa freqüència de rellotge que el processador B?

$P_1 = C \cdot V^2 \cdot F = 8 \cdot 10^{-9} \cdot 5^2 \cdot 3 \cdot 10^9 = 120W$

Tema 2: Ensamblador MIPS i tipus de dades bàsics

2.1 - Perspectiva històrica

2.1.1 - CISC vs RISC

- Complex Instruction Set Computer:
    - Joc d'instruccions grans i complexes
    - Instruccions de longitud variable
    - Cada instrucció es descodifica en múltiples microoperacions
    - Exemples: x86
- Reduced Instruction Set Computer:
    - Jocs d'instruccions petites i senzilles
    - Instruccions de longitud fixa
    - Formats d'instruccions i mètodes de direccionament senzills
    - Executades directament per el hardware
    - Exemples: MIPS, ARM, RISC-V

2.1.2 - MIPS

També conegut com a Microprocessor without Interlocked Pipeline Statges. Va ser dissenyat entre els anys 1981 i 1985 per Henessy i Patterson. Conté un jocs d'instruccions tipus RISC i és utilitzat per multiples implementacions comercials:

- Routers de Cisco i Linksys
- Módems ADSL
- Controladores de impresora láser
- Playstation (PSX, PS2 i PSP)
- Nintendo 64

2.2 - Memòria i variables de MIPS

2.2.1 - La Memòria

2.2.1.1 - Vector de bytes

Cada byte s'identifica una direcció.

ADRECES	DADES
0x00000000	1 byte
0x00000001	1 byte
0x00000002	1 byte
0x00000003	1 byte
…….	…….
0xFFFFFFFF	1 byte

2.2.1.2 - Palabra (word)

- Normalment el tamany de la paraula és igual al tamany del registre.
- MIPS32: 32 bits (4 bytes) (1 word)
- Exemples: `0xAABBCCDD, 0x44332211`

2.2.2.3 - Endianness

Little - endian

El byte de menor pes en la direcció més baixa de memoria:
word: [`0x44, 0x33, 0x22, 0x11`]

0x10010000	0x11
0x10010001	0x22
0x10010002	0x33
0x10010003	0x44

Big - endian

El byte de major pes en la direcció més baixa de memoria:
word: [`0x44, 0x33, 0x22, 0x11`]

0x10010000	0x44
0x10010001	0x33
0x10010002	0x22
0x10010003	0x11

2.2.2.4 - Alineació en memoria

C:

```
unsigned char a;
short b = 13;
char c = -1, d = 10;
int e = 0x10AA00FF
long long f = 0x7766554433221100
```

MIPS:

```
    .data
a:     .byte 0
b:     .half 13
c:     .byte -1
d:     .byte 10
e:     .word 0x10AA00FF
f:      .dword 0x7766554433221100
```

a:	00
b:	0D
	00
c:	FF
d:	0A
e:	FF
	00
	AA
	10
f:	00
	11
	22
	33
	44
	55
	66
	77

2.2.2 - Declaració de variables

2.2.2.1 - Variables globals o locals

- Variables **globals**
  - Poden ser accedides des de qualsevol funció
  - Es mantenen en memoria durant tota l'execució del programa
  - S'emmagatzemen en una direcció de memòria fixa (dedicada)

- Variables **locals**
  - Només es pot accedir des d'on es declaren
  - Es creen a l'inici del bloc d'execució i s'eliminen quan acaba
  - Es reserva espai a la memòria de forma dinàmica

Llenguatge C

```
int a = 0x44332211

   int main (void) {
         int i = 7;
   }
```

Ensamblador MIPS

```
.data
   a: .word 0x44332211

   .text
   main:
         li $t0, 7
```

2.2.2.2 - Tipus de variables

Tamany	C	MIPS
1 byte	char / unsigned char	`.byte`
2 bytes	short / unsigned short	`.half`
4 bytes	int / unsigned int	`.word`
8 bytes	long long / unsigned long long	`.dword`

2.2.2.3 - Declaració d'un vector

2.2.2.3.1 - Declaració amb inicialització

C:

```
short vec [5] = {2, -1, 3, 5, 0};
```

MIPS:

```
      .data
vec:  .half 2, 1, 3, 5, 0
```

2.2.2.3.2 - Declaració sense inicialització (alineació explícita)

C:

```
char a;
int v[100];
```

MIPS:

```
      .data
a:    .byte 0
      .align 2
# Alinear a multiple de 4

v:    .space 400
#Vector de 100 ints (100 ints * 4 espais per
int)
```

2.2.2.4 - Pseudointruccions o macros

- Simplifiquen operacions comunes per les que no existeix una instrucció en MIPS.
- Faciliten el desenvolupament, la lectura i la depuració del codi
- En el moment de ser ensamblat es tradueix a una o diverses instruccions MIPS

```
move $t1, $t2           #addu $t0, $t2, $zero

li $t1, 100             #addiu $t1, $zero, 100

li $t1, 0x0030D900      #lui $at, 0x0030
                        #ori $t1, $at, 0xD900
```

```
    .data
y:  .word 42            #Direcció de y = 0x10010024

    .text
    la $t0, y           #lui $at, 0x1001
                        #ori $t0, $at, 0x0024
```

2.3 - Direccions i direccionament immediat MIPS per 32 bits

2.3.1 - Modes de direccionament:

- Forma en la que s'especifica un operand en una instrucció
- MIPS suporta cinc modes de direccionament:
    - Mode registre
    - Mode immediat
    - Mode memoria
    - Mode pseudo-directe
    - Mode relatiu al PC

2.3.2 - Registres

Número	Nombre	Descripció
`$0`	`$zero`	Conte el valor 0
`$1`	`$at`	Registre temporal per a pseudoinstruccions
`$2 - $3`	`$v0 - $v1`	Resultats de subrutinas
`$4 - $7`	`$a0 - $a3`	Paràmetres de subrutines
`$8 - $15`	`$t0 - $t7`	Temporals
`$16 - $23`	`$s0 - $s7`	Segurs (preserven al cridar una subrutina)
`$24 - $25`	`$t8 - $t9`	Temporals
`$26 - $27`	`$k0 - $k1`	Reservats per el SO
`$28`	`$gp`	Global pointer
`$29`	`$sp`	Stack pointer
`$30`	`$fp`	Frame pointer
`$31`	`$ra`	Return address

2.3.3 - Restricció d'alineament

- Les direccions de memòria utilitzades en les instruccions `lw y sw` han de ser múltiples de 4
- Les direccions de memòria utilitzades en les instruccions `lh, lhu i sh` han de ser múltiples de 2

En cas contrari, es produeix una excepció per direcció no alineada i el programa breaks

2.4 - Operands de hardware del computador

2.4.1 - Operands en mode registre

- L'operand resideix en un registre
- La Instrucció especifica l'identificador del registre
- Suma: addu rd, rs, rt
- Resta: subu rd, rs, rt
- MIPS inclou 32 registres de 32 bits

2.4.2 - Operands en mode immediat

L'operant es codifica en la propia instrucció. El valor es de 16 bits en Ca2. Per convertir en valor de 32 bits es fa extensió de signe o extensió de zeros.

`addiu rt, rs, imm16`	`rt = rs + SignExt(imm16)`
`lui rt, imm16`	`rt31..16 = imm16` `rt15..0 = 0x0000`
`ori rt, rs, imm16`	`rt = rs OR ZeroExt(imm16)`

2.4.3 - Operands en mode memòria

Només les instruccions de tipus load i store admeten un operand de memòria.
S'han de carregar les dades de memoria en registres per poder utilitzar-los en instruccions aritmetico-lògiques.
Lectura o escriptura de paraules en memòria:

`lw rt, off16(rs)`	`rt = MW[rs + SignExt(off16)]`	`Load word`
`sw rt, off16(rs)`	`MW[rs + SignExt(off16)] = rt`	`Store word`

2.4.3.1 - Operands en mode memòria (Exemple)

C	MIPS
`g = h + A[8]`	`lw $t0 , 32( $t3 ) # $t0 = A[ 8 ]` `addu $t1 , $t2 , $t0 # g = h + $t0 ;`

2.4.4 - Accés a halfword o byte - Enters amb signe

- **Load halfword:** `lh rt, off16 (rs)`
  - Copia un halfword (2 bytes) de la memoria als 16 bits de menor pes del registre `rt`
  - Realitza extensió de signe (extén el bit 15)
- **Store halfword:** `sh rt, off16 (rs)`
  - Copia a memoria els 2 bytes de menor pes de `rt`
- **Load byte:** `lb rt, off16 (rs)`
  - Copia 1 byte de memoria als 8 bits de menor pes del registre `rt`
  - Realitza extensió de signe (extén el bit 7)
- **Store byte:** `sb rt, off16 (rs)`
  - Copia a memoria el byte de menor pes del registre `rt`

2.4.5 - Accés a halword o byte - Naturals

- **Load halfword unsigned:** `lhu rt, off16( rs )`
  - Copia un halfword (2 bytes) de la memoria als 16 bits de menor pes del registre rt
  - Realitza extensió de zeros
- **Load byte unsigned:** `lbu rt, off16 ( rs )`
  - Copia 1 byte de memoria els 8 bits de menor pes del registre rt
  - Realitza extensió de zeros

2.4.6 - Accés a doble paraula - dword

```
        .data
    x:  .dword 0x7766554433221100

        .text
main:
        #$t2 conte la direcció de memoria de x
        lw $t0, 0 ( $t2 )
        lw $t1, 4 ( $t2 )
```

2.5 - Números amb signe i sense signe

2.5.1 - Naturals en base 2

- X = 0b10011101
- $X_u = \Sigma_{i=0}^{n-1} X_i \cdot 2^i$
- $X_u = 128 \cdot 1 + 64 \cdot 0 + 32 \cdot 0 + 16 \cdot 1 + 8 \cdot 1 + 4 \cdot 1 + 2 \cdot 0 + 1 \cdot 1$
- $X_u = 157$

2.5.2 - Declaració de variables de tipus natural

Enters “unsigned” en C	Enters “unsigned” en MIPS

```
unsigned char      var1 = 0;
unsigned short     var2 = 0;
unsigned int       var4 = 0;
unsigned long long var8 = 0;
```

```
      .data
var1: .byte  0
var2: .half  0
var4: .word  0
var8: .dword 0
```

Enters en C	Enters en MIPS

```
char      var1 = 0;
short     var2 = 0;
int       var4 = 0;
long long var8 = 0;
```

```
      .data
var1: .byte  0
var2: .half  0
var4: .word  0
var8: .dword 0
```

2.5.2 - Enters en Ca1

- Asigna a les representacions que tenen el bit de major pes a 1 els enters del rang $-2^{n-1} + 1$ fins a 0
- Cada una d’aquestes cadenes de bits es correspon a un enter que és una unitat major que en Ca2
- Per a n = 8, s’assignen els naturals del 128 al 255 als enters del -127 al 0

2.5.3 - Enters en Ca1 - Regla d’interpretació

- $X_s = X_u - X_{n-1} \cdot (2^n - 1)$
  - X: Representació en Ca1
  - X_u: Natural (u = unsigned) que correspon a X
  - X_s: Enter (s = signed) que correspon a X

2.5.3.1 - Enters en Ca1 - Regla d'interpretació (Exemple)

- X = 0b10001011
- X_u: 128 + 8 + 2 + 1 = 139
- X_s: 139 - 255 = -116

2.5.4 - Enters en Ca1 - Regla de representació:

- Si $X_s \geq 0$ then $X_u = X_s$
- Si $X_s \leq 0$ then $X_u = X_s + (2^n - 1)$

2.5.3.1 - Enters en Ca1 - Regla representació(Exemple)

- Representar en Ca1 en 8 bits l'enter X_s = 29
  - Com 29 ≥ 0, X_u = 29 = $2^4 + 2^3 + 2^2 + 2^0$
  - X = 0b00011101
- Representar en Ca1 en 8 bits l'enter X_s = -124
  - Com -124 ≤ 0, X_u = -124 + 255 = 131 = $2^7 + 2^1 + 2^0$
  - X = 0b10000011

2.5.5 - Enters en Ca1 - Regla de canvi de signe:

- Sigui Xla representació en Ca1 d'un enter X_s , la representació del seu oposat $-X_s$ s'obté complementant tots els bits de X
- $X_s = 12$, X = 0b00001100
- $-X_s$ = -12, X' = 0b11110011

2.5.6 - Enters "en excés"

- Al número enter se li suma una constant (excés) i es representa el resultat
- Per a 3 bits i excés 3:

X_s	X_u	X
-3	0	000
-2	1	001
-1	2	010
0	3	011

1	4	101
2	5	101
3	6	110

2.5.7 - Enters “en excés” - Interpretació i representació

- Interpretació:
  - $X_s = X_u - excés$
- Representació:
  - $X_u = X_s + excés$
- Propietats:
  - El bit de major pes no indica signe
  - El valor més negatiu té la representació 000...0
  - El valor més positiu té la representació 111...1
  - S’utilitza per representar l’exponent en el format de coma flotant (*Tema 5)

2.6 - ASCII o American Standard Code for Information Interchange

Es un sistema de codificació de caràcters on s'assigna un codi a cada símbol.
Estudiarem l'ASCII de 7 bits:

- Els caràcters s'emmagatzemen utilitzant un byte.
- El bit de major pes sempre val 0
- Codis del 0 al 31 són de control
- La resta de codis son símbols tipogràfis.

Codi	Símbol	En C i MIPS
0x09	TAB	'\t'
0x0A	LF	'\n'
0x30	1	'1'
0x30	A	'A'
0x61	a	'a'

2.6.1 - Propietats de la codificació ASCII

- Ordre alfabètic:
  - 'a' = 97, 'b' = 98, 'c' = 99, …
  - 'A' = 65, 'B' = 66, 'C' = 67, …
  - '0' = 48, '1' = 49, '2' = 50, …

- Conversió minúscula/majúscula
  - de MAY a min → sumant 32: `'a' = 'A' + 32;`
    `'x' = 'X' + ('a' - A') //x caràcter qualsevol`
  - de min a MAY → restant 32: `'A' = 'a' - 32;`
    `'X' = 'x' - ('a' - 'A'); //x caràcter qualsevol`

- Conversió dígit `ASCII`/valor numèric
  - de `ASCII` a val. num. → restant 48: `1 = '1' - 48;`
    `x = 'x' - '0'; //on x és un digit qualsevol`
  - de val. num a `ASCII` → sumant 48: `'1' = 1 + 48;`
    `'x' = x + '0'; //on x és un digit qualsevol`

C	MIPS
`char letra = 'R';`	`letra: .byte 'R'`

2.7 - Format de les instruccions MIPS

Les instruccions es representen amb cadenes de bits i s'emmagatzemen a la memòria.
En MIPS32 les instruccions són dee 32 bits.
Formats d'instruccions de MIPS: R (register), I (immediate) i J (jump)

Format	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt		imm 161	
J	opcode	target				

`opcode`: codi d'operació
`funct`: extensió del codi d'operació
`rs, rt`: operands en mode registre
`imm16`: operand en mode immediat de 16 bits
`shamt`: shift amount, immediat per desplaçaments de bits
`target`: direcció de destí en un salt

2.7.1 Exemples:

		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
addu rd, rs, rt	R	000000	rs	rt	rd	00000	100001
sra rd, rt, shamt	R	000000	rs	rt	rd	shamt	000011
addiu rt, rs, imm16	I	001000	rs	rt	imm16		
lui rt, imm16	I	001111	00000	rt	imm16		
lw rt, offset16(rs)	I	100011	rs	rt	offset16		
j target	J	000010	target				

2.8 Punters

- Variable que conté una direcció de memoria
    - 32 bits en MIPS32
    - Similiar a una variable de tipus enter
- Si p conté la direcció de memòria de la variable v, llavors direm que $p\ apunta\ a\ v$
- Declaració de punters:

C:	**MIPS**
int *p1, *p2; char *p3;	.data p1: .word 0 p2: .word 0 p3: .word 0

2.8.1 - Com utilitzar un punter (en C/C++)?

Imaginem una variable `int var1 = 5;` que està emmagatzemada a la direcció `0xAABBCC` i un punter `int* p1 = &a;` que apunta a `a` i està guardat a la direcció `0xDDEEFF`.

Valor de `a`:
```
cout << a << endl; //imprimeix "5"
```

Direcció de `a`:
```
cout << a << endl; //imprimeix "0xAABBCC"
```

valor de `p`:
```
cout << p << endl; // imprimeix "0xAABBCC", és a dir, la dir. de a
```

Valor que apunta `p`:
```
cout << *p << endl; // imprimeix "5", és a dir, el valor de a
```

Direccio de `p`:
```
cout << &p << endl; // imprimeix "0xDDEEFF"
```

2.8.2 - Inicialització de punters

- Assignant un altre punter del mateix tipus
- Assignant una direcció d'una variable (operador &)

C:	MIPS

char a = 'E';
char b = 'K';
char *p = &a
 	
void foo() {
  p = &b;
}
	.data
	a: .byte 'E'
	b: .byte 'K'
	p: .word a
	
	.text
	foo:
	la $t0, b
	la $t1, p
	sw $t0, 0($t1)

2.8.3 - Desreferènica (indirecció) de punters

Els punters serveixen per accedir a les variables apuntades.
La desreferència consisteix en accedir a la direcció de memòria apuntada pel punter. És a dir, accedir al valor que hi ha a aquella direcció de memòria.
En C s'utilitza l'operador *

En C	En MIPS

`char a = 'E';`
`char *p = &a;`
`//direcció de memoria de a`
 	
`void foo() {`
`char tmp = *p;`
`//valor de a`
`}`
	`.data`
	`a: .byte 'E'`
	`p: .word a`
	
	`.text`
	`foo: la $t0, p`
	`lw $t1, 0($t0)`
	`lb $t2, 0($t1)`

.

2.8.4 - Aritmetica de punters

Suma d'un punter `p` més un enter `N`
Dona com a resultat un altre punter `q` del mateix tipus: `q = p + N;`
`q` apunta a un altre enter situat a `N` elements més endavant.

En C	En MIPS

`char* p1;`
`int* p2;`
`long long* p3;`
`…`
`p1 = p1 + 3;`
`p2 = p2 + 3;`
`p3 = p3 + 3;`
	`p1: .word 0`
	`p2: .word 0`
	`p3: .word 0`
	`…`
	`addiu $t1, $t1, 3`
	`addiu $t2, $t2, 12`
	`addiu $t3, $t3, 24`

2.9 - Vectors

Són agrupacions unidimensionals de elements del mateix tipus, els quals s'identifiquen per un índex. Els elements s'emmagatzemen en posicions consecutives a partir de la direcció inicial del vector. El primer element té index 0. Si treballem en MIPS, els elements del vector han de respectar les regles de alineació:

- Com que tots els tipus tenen un tamany múltiple de 2, si el primer element està alineat, la resta d'elements també

C	MIPS

`short vec1[5] = {0, 1, 2, 3 ,4};`
 	
`int vec2[100];`
	`vec1:  .half 0, 1, 2, 3, 4`
	`  .aling`
	`vec2:  .space 400`

2.9.1 - Accés a un element de un vector

Per accedir a un element i de un vector s'ha de calcular la seva direcció de memòria. Si els seus elements tenen un tamany T bytes, llavors la direcció iessima es:

`@vec[i] = @vec[0] + i * T`

2.9.2 - Vectors i punters

- A C, els vectors tenen el mateix tipus que un vector i sempre apunten al primer element.
  - int vec[100];
  - int *p
- Podem inicialitzar un punter assignant un vector, però NO a l'inrevés
  - p = vec;
- Podem utilitzar l'operador [] amb un punter
  - p[8] = 0;
- Podem utilitzar l'operador d'indirecció * amb un vector

    - *vec = 0;
- Un punter es pot considerar com un vector, on el primer element és al que apunta p
- La següent expressió és perfectament vàlida:
    - *(p + i) = 0;
- I és equivalent a l'expressió:
    - p[i] = 0;

2.9.3 - String

Es un vector amb un número variable de caracters. El vector consta d'un centinela que sempre val ['\0']

2.9.3.1 - Declaració de strings

C	MIPS

`char cadena [] = "Una frase";`	`.data`
	`cadena: .ascii "Una frase"`
	`.byte 0`

2.9.3.2 - Accés a un element de un string

- A C, els caràcters es codifiquen en ASCII (1 byte)
- S'accedeix utilitzant les instruccions `lb i sb`
- Mateix mètode que s'utilitza per accedir als vectors
    - `char cadena[] = " Una frase " ;`
    - `@cadena[i] = @cadena[0] + i;`
    - El tamany dels elements és sempre 1 pels strings

2.10 - Exercicis

2.10.1 - Exercici 2.1

En els següents apartats hauràs de traduir codi C a codi MIPS. Suposa que les variables f, g, h, i i j són variables enteres de 32 bits i han estat assignades als registres $t0, $t1, $t2, $t3 i $t4 respectivament. Tradueix els següents fragments a llenguatge assemblador:

a) C: `f = g + h + i + j`

MIPS:
```
addu $t0, $t1, $t2          #f = g + h
addu $t0, $t0, $t3          #f = f + i
addu $t0, $t0, $t4          #f = f + j
```

b) C: `g = f + ( h + 5 ) - i`

MIPS:
```
addui $t1, $t2, 5           #g = h + 5
addu  $t1, $t1, $t0         #g = g + f
subu  $t1, $t1, $t3         #g = g - i
```

2.10.2 - Exercici 2.2

En els següents apartats hauràs de traduir codi MIPS a codi C. Suposa que les variables f, g, h, i i j són variables enteres de 32 bits i han estat assignades als registres $t0, $t1, $t2, $t3 i $t4 respectivament. Tradueix els següents fragments a llenguatge C:

a) C: `f = ( g + h ) + ( i + j ) · 2`

MIPS:
```
addu $t0, $t1, $t2          #f = g + h
addu $t1, $t3, $t4          #g = i + j
addu $t1, $t1, $t1          #g = g + g
addu $t0, $t0, $t1          #f = f + g
```

b) C: `f = 2 · ( ( g + h ) + ( i + j ) + 2 )`

MIPS:
```
addu  $t1, $t1, $t2         #g = g + h
addu  $t3, $t3, $t4         #i = i + j
addu  $t0, $t1, $t3         #f = g + i
addiu $t0, $t0, 2           #f = f + 2
addu  $t0, $t0, $t0         #f = f + f
```

2.10.3 - Exercici 2.3

Donada la següent declaració de dades, que s'emmagatzemarà a memòria a partir de l'adreça 0x10010000:

```
.data
.byte 1, 2, 3, 4
.word -1, 1, -2, 2, -3, 3
.word 0x12345678
```

Indiqueu quin serà el valor en hexadecimal dels bytes emmagatzemats a les adreces de memòria 0x100100**0C** i 0x100100**1C** (poseu NA si no es pot determinar a partir de la declaració donada).

0x10010000	0x01	0x1001000A	0x00	0x10010014	0xFD
0x10010001	0x02	0x1001000B	0x00	0x10010015	0xFF
0x10010002	0x03	0x1001000C	0xFE	0x10010016	0xFF
0x10010003	0x04	0x1001000D	0xFF	0x10010017	0xFF
0x10010004	0xFF	0x1001000E	0xFF	0x10010018	0x00
0x10010005	0xFF	0x1001000F	0xFF	0x10010019	0x00
0x10010006	0xFF	0x10010010	0x02	0x1001001A	0x00
0x10010007	0xFF	0x10010011	0x00	0x1001001B	0x03
0x10010008	0x01	0x10010012	0x00	0x1001001C	0x78
0x10010009	0x00	0x10010013	0x00	0x1001001D	0x56

2.10.4 - Exercici 2.4

Les variables enteres A, B, C estan ubicades a les adreces 0x10010000, 0x10010004 i 0x10010008 de memòria. Donats els següents continguts inicials de registres i de memòria, tots ells en hexadecimal:

```
$t0=0x10010000,  $t2=10010008
```

ADRECES	CONTINGUTS (byte per byte)
0x10010000	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x10010010	00 00 00 00 EC FD 0E 0F 00 00 00 00 00 00 00 00

Determina, per a cada instrucció, quin registre o adreça de memòria es modifica, i quin és el seu contingut.

```
.text
lw $t1, 0($t0)        # $t1         ← 0x30201000
lw $t3, 0($t2)        # $t3         ← 0xB0A00908
sw $t3, -8($t2)       # 0x10010000 ← 0xB0A00908
lw $t4, 4($t0)        # $t4         ← 0x70605040
sw $t1, -4($t2)       # 0x10010004 ← 0x30201000
sw $t4, 0($t2)        # 0x10010008 ← 0x70605040
```

2.10.5 - Exercici 2.5

Donada la següent declaració:

```
    .data
A:  .byte 0, 0, 0, 0, 0
```

Escriu un fragment de no més de 8 línies de codi en assemblador tal que guardi el valor de $t1 a l'adreça A+1, però sense produir cap excepció (es poden escriure valors temporals a la posició A de memòria, si cal).

```
    .text
    la $t0, A
    addui $t0, $t0, 1
    sb $t1, 0($t0)
```

2.10.6 - Exercici 2.6

Indica quin és el contingut de memòria a partir de l'adreça 0x10010000 i a nivell de byte si tenim la següent declaració de variables globals.

```
    .data
    .byte 0x10
A:  .half -5
B:  .word -1, 67
C:  .byte -4, '5', 6
D:  .half 66, 67
E:  .dword 0x5799
F:  .byte 'C'
G:  .half 0x66
H:  .dword 579
```

0x10010000	0x10	0x1001000A	0x00	0x10010014	--
0x10010001	--	0x1001000B	0x00	0x10010015	--
0x10010002	0xFB	0x1001000C	0xFC	0x10010016	0x99
0x10010003	0xFF	0x1001000D	0x35	0x10010017	0x57
0x10010004	0xFF	0x1001000E	0x06	0x10010018	0x00
0x10010005	0xFF	0x1001000F	--	0x10010019	0x00
0x10010006	0xFF	0x10010010	0x42	0x1001001A	0x00
0x10010007	0xFF	0x10010011	0x00	0x1001001B	0x00
0x10010008	0x43	0x10010012	0x43	0x1001001C	0x00
0x10010009	0x00	0x10010013	0x00	0x1001001D	0x00

0x1001001E	0x43	0x10010029	0x00	0x10010033	--
0x1001001F	--	0x1001002A	0x00	0x10010034	--
0x10010020	0x66	0x1001002B	0x00	0x10010035	--
0x10010021	0x00	0x1001002C	0x00	0x10010036	--
0x10010022	--	0x1001002D	--	0x10010037	--
0x10010023	--	0x1001002E	--	0x10010038	--
0x10010024	0x43	0x1001002F	--	0x10010039	--
0x10010025	0x02	0x10010030	--	0x1001003A	--
0x10010026	0x00	0x10010031	--	0x1001003B	--
0x10010027	0x00	0x10010032	--	0x1001003C	--

2.10.7 - Exercici 2.7

Donades les següents declaracions de variables, emmagatzemades a memòria a partir de l'adreça 0x10010000:

```
    .data
A:  .word 5, 2
B:  .word 3
C:  .word 4, 0x0FFFF, 0x4180
D:  .word 0x10010008, 0
```

Contesteu els següents apartats, suposant que tots ells parteixen del mateix estat inicial (els càlculs d'un apartat no influeixen en els següents):

a) Quin és el valor final de la variable B després d'executar el següent codi?

```
.text
la $t1, C           # $t1 ← 0x1001000C    # &C
lw $t2, 12($t1)     # $t2 ← 0x10010008
lb $t1, 8($t2)      # $t1 ← 0xFFFFFFFF
la $t3, B           # $t3 ← 0x10010008    # &B
lb $t4, 0($t3)      # $t4 ← 0x00000003
addu $t1, $t1, $t4  # $t1 ← 0x00000002    # (-1) + 03 = 02
sb $t1, 0($t3)      # Ox10010008 ← 0x00000002
```

La variable B acabarà amb el valor 0x0000002

b) Expliqueu textualment i de forma concisa què fa el següent fragment de codi

```
.text
la $t1, A+1         # $t1 ← 0x10010004    # A + (1*4word)
lw $t1, 0($t1)      # $t1 ← 0x00000002
```

Guardem el valor de 0x10010004 al registre $t1

2.10.8 - Exercici 2.8

Indica quin és el contingut de memòria a partir de l'adreça 0x10010000 i a nivell de paraula si tenim la següent declaració de variables globals.

```
     .data
     .byte 0,1,2
A:   .byte 5, -1
B:   .half 0x66
C:   .word 4, 0x0FDF
D:   .half 3
E:   .byte 25
F:   .byte 0x08
G:   .dword 0x10010008
```

0x10010000	0x00	0x1001000A	0x00	0x10010014	--
0x10010001	0x01	0x1001000B	0x00	0x10010015	--
0x10010002	0x02	0x1001000C	0xDF	0x10010016	--
0x10010003	0x05	0x1001000D	0x0F	0x10010017	--
0x10010004	0xFF	0x1001000E	0x00	0x10010018	0x08
0x10010005	--	0x1001000F	0x00	0x10010019	0x00
0x10010006	0x66	0x10010010	0x03	0x1001001A	0x01
0x10010007	0x00	0x10010011	0x00	0x1001001B	0x10
0x10010008	0x04	0x10010012	0x25	0x1001001C	0x00
0x10010009	0x00	0x10010013	0x08	0x1001001D	0x00

2.10.9 - Exercici 2.23

Convertiu les següents instruccions de llenguatge assemblador a llenguatge màquina, seguint el format de les instruccions MIPS:

Ll. Assemblador MIPS	Ll. Màquina MIPS (Bit a Bit)	Ll. Màquina MIPS (Hexadecimal)
addu $t4, $t3, $t5	000000 01011 01101 01100 00000 100001	0x16D6021
addiu $t7, $t6, 25	001001 01110 01111 00000 00000 011001	0x25CF0019
lw $t3, 0($t2)	100011 01010 01011 0000000000000000	0x8D4B0000
sw $t0, 0($t1)	101011 01001 0100 00000000000000000	0xAD280000

2.10.10 - Exercici 2.27

Donada la següent declaració de dades global, en C:

```
int dada = 5;          //0x1234
int *pdada = &dada     //0x5678
```

Tradueix a una única sentència en C el conjunt d'instruccions de cada apartat:

a)
```
la $t0, pdada        # $t0 ← &pdada (0x5678)
lw $t0, 0($t0)       # $t0 ← pdada (0x1234)
lw $t1, 0($t0)       # $t1 ← *pdada (dada)
addiu $t1, $t1, 4    # $t1 ← dada + 4
sw $t1, 0($t0)       # dada ← $t1
```

```
*pdada += 4;
```

b)
```
la $t0, pdada        # $t0 ← &pdada (0x5678)
lw $t1, 0($t0)       # $t1 ← pdada (0x1234)
addiu $t1, $t1, 4    # $t1 ← pdada + 1 (0x1238)
sw $t1, 0($t0)       # &pdada ← pdada + 1
```

```
pdada += 1;
```

c)
```
la $t0, pdada        # $t0 ← &pdada
lw $t0, 0($t0)       # $t0 ← pdada
lw $t1, 0($t0)       # $t1 ← *pdada
addiu $t0, $t0, 4    # &t0 ← pdada + 1
sw $t1, 0($t0)       # pdada ←
```

```
*(pdada + 1) = *pdada
```

Tema 3 - Traducció de programes

3.1 - Desplaçaments lògics

- Shift Left Logical: `sll rd, rt, shamt`
  - Desplaça a l'esquerra el número de bits indicat en l'operand immediat `shamt`.
- Shift Right Logical: `srl rd, rt, shamt`
  - Desplaça `rt` a la dreta el número de bits indicant l'operand immediat `shamt`.
- Les posicions que queden vacants s'omplen amb zeros

```
li $t0, 0x33333333
sll $t1, $t0, 2       # $t1 = 0xCCCCCCCC
srl $t2, $t0, 2       # $t2 = 0x0CCCCCCC
```

3.1.1 - Desplaçament aritmètic a la dreta

- Shift Right Arithmetic: `sra rd, rt, shamt`
  - Desplaça `rt` a la dreta el número de bits indicat a l'operant immediat `shamt`.
- Les posicions que queden vacants s'omplen amb una extensió de signe de `rt`

```
li $t0, 0x88888888
sra $t1, $t0, 1       # $t1 = 0xC4444444
```

3.1.2 - Desplaçament indicat en registre

- `sllv rd, rt, rs`
- `srlv r d, rt, rsm`
- `srav rd,rt,rs`
- El número de bits a desplaçar s'indica en els 5 bits de menor pes del registre `rs`, la resta de bits s'ignora.

3.1.3 - Multiplicació i divisió per potències de 2

- Desplaçar a l'esquerra equival a multiplicar per una potencia de 2
  - `sll rd, rt, shamt`
  - $rd = rt \times 2^{shamt}$
  - Útil per a calcular l'`offset` d'un vector donat l'índex
- Desplaçar a la dreta un natural (amb `srl`) o un enter (amb `sra`) equival a dividir-lo per una potencia de 2
  - `srl rt, rt, shamt`
  - $rd = rt / 2^{shamt}$

3.1.3.1 - Exemple:

Codi en C

```
int vec[100];

void main(){
    int i = 12;
    vec[i] = 0;
}
```

Codi en MIPS

```
        .data
        .align 2
vec:    .space 400

        .text
main:   li $t0, 12              # $t0 = i
        sll $t1, $t0, 2
        la $t2, vec
        addu $t2, $t2, $t1
        sw $zero, 0($t2)
```

3.1.4 - Conversió de Ca2 a Ca1

Es pot convertir de Ca2 a Ca1 restant el bit de signe:

```
# Desplacem el bit de signe a la posició 0
srl $t1, $t0, 31

# Restem el bit de signe
subu $t0 , $t0 , $ t 1
```

3.1.5 - Traducció dels operads de desplaçamient en C

En C es defineixen dos opeadors de desplaçament de bits:

Desplaçament a l'esquerra: `<<`

Es tradueix a la instrucció `sll`

Desplaçament a la dreta: `>>`

Per naturals es tradueix a la instrucció `srl`
Per enters es tradueix a la instrucció `sra`

Traduïr a MIPS la següent sentencia en C (a i b són enters i estan emmagatzemats a $t0 i $t1)

```
a = ( a << b ) >> 2

sllv $t4, $t0, $t1
sra $t0, $t4, 2
```

3.2 - Operaciones lógicas bit a bit en MIPS

- Repertori d'instruccions lògiques [imm16 ha de ser un nombre natural] :

`and rd, rs, rt`	`rd = rs AND rt`
`or rd, rs, rt`	`rd = rs OR rt`
`xor rd, rs, rt`	`rd = rs XOR rt`
`nor rd, rs, rt`	`rd = rs NOR rt = NOT (rs OR rt)`
`andi rd, rs, rt`	`rt = rs AND ZeroExt(imm16)`
`ori rd, rs, rt`	`rt = rs OR ZeroExt(imm16)`
`xori rd, rs, rt`	`rt = rs XOR ZeroExt(imm16)`

3.2.1 Operacions lògiques bit a bit (C vs MIPS)

- Assumint que a, b i c es troben a $t0, $t1 i $t2:

Llenguatge C	**Ensamblador MIPS**
`c = a & b`	`and $t2, $t0, $t1`
`c = a | b`	`or $t2, $t0, $t1`
`c = a ^ b`	`xor $t2, $t0, $t1`
`c = ~a`	`nor $t2, $t0, $zero`
`c = a & 7`	`andi $t2, $t0, 7`
`c = a | 7`	`ori $t2, $t0, 7`
`c = a ^ 7`	`xori $t2, $t0, 7`

Utilitzem el símbol `'~'` per fer la negació bit a bit

3.2.2 - Seleccionar bits

- La instrucció `and` s'utilitza per a seleccionar determinats bits d'un registre, posant la resta a 0.
- La instrucció `andi` s'utilitza per a calcular la divisió per potències de 2.

3.2.2.1 - Exemple

Seleccionar els 16 bits de menor pes del registre $t0

```
andi $t1, $t0, 0xFFFF
```

3.2.2.2 - Exemple

Extreure els bits 0, 2, 4, 6 (posar la resta de bits a 0)

```
andi $t1, $t0, 0x0055
```

3.2.3 - Utilitat de les operacions lògiques bit a bit

Utilitat	**Instrucció exemple**
Posar bits a 1	`ori $t0, $t0, 0xFFFF`
Posar bits a 0	`xor $t0, $t0, $t0`

Complementar bits	`#complementar els bits parells`
	`li $t0, 0x55555555`
	`xor $t0, $t0, $t1`

3.3 - Comparacions i operacions booleanes

En C, podem utilitzar operacions de comparació en expressions enteres:

```
==, !=, <, <=, >, >=
```

Encara que no existeix el booleà, ja que el resultat d'una comparació dóna un valor enter "normalitzat". Sigui 0 si és fals i 1 si és cert.

També existeixen expressions lògiques formades per operadors booleanes:

```
&&, ||, !
```

3.3.1 - Repertori d'instruccions de comparació

`slt rd, rs, rt`	`rd = rs < rt`	enters
`sltu rd, rs, rt`	`rd = rs < rt`	naturals
`slti rd, rs, imm16`	`rd = rs < Sext(imm16)`	enters
`sltiu rd, rs, imm16`	`rd = rs < Sext(imm16)`	naturals

Si la comparació és certa s'escriu 1 en rd.

SI la comparació és falsa s'escriu 0 en rd.

3.3.1.1 - Comparació de tipus '<'

MIPS només inclou instruccions de comparació "menor que" .
Traducció a MIPS, suposant que a, b, c s'emmagatzema en $t0, $t1, $t2

```
c = a < b
```

Si a i b són:

- Naturals: `sltu $t2, $t0, $t1`
- Enters: `slt $t2, $t0, $t1`

3.3.1.2 - Comparació '>'

- a > b ⇔ b < a
- Traduir a c = a > b; suposant que a, b i c es guarden a `$t2, $t0, $t1`, `respectivament:`

```
slt  $t2, $t0, $t1                    # c = b < a
```

3.3.1.3 - Comparació '<'

- a < b ⇔ b > a

3.3.1.4 - Comparació '≥'

- a ≥ b ⇔ $\overline{a < b}$
- Traduir c = a ≥ b; suposant que a, b i c es guarden a `$t0, $t1 i $t2`, respectivament:

```
slt $t4, $t0, $t1                     # aux = (a < b)
sltiu $t2, $t4, 1                     # c = !aux
```

3.3.1.5 - Comparació '≤'

- a ≤ b ⇔ $\overline{a > b}$ ⇔ $\overline{b < a}$
- Traduir c = a ≤ b; suposant que a, b i c es guarden a `$t0, $t1 i $t2`, respectivament:

```
slt $t4, $t1, $t0                     # aux = (b < a)
sltiu $t2, $t4, 1                     # c = !aux
```

3.3.1.6 - Negació lógica '!'

```
unsigned char a = 0x55;
unsigned char b;

b = !a;  // b = 0
```

- Traducció a MIPS amb comparació "menor que 1" com a natural
- Suposant que `$t0 = a` i `$t1 = b` :

```
sltiu $t1, $t0, 1
```

  - Si `$t0` és 0 (fals) : `$t1 = 1` (cert)
  - Si `$t1` és diferent de 0 (cert) : `$t1 = 0` (fals)

3.3.1.6.1 - Negació bit a bit vs Negació lògica

- `La negació bit a bit i la negació lògica pueden dar resultado distinto`

```
unsigned char a = 0x55;
unsigned char b,c;

b = !a;                                    # b = 0;
c = ~a;                                    # c = 0xAA;
```

- Traducció a MIPS (a, b i c estan a `$t0, $t1, $t2,` respectivament):

```
sltiu $t1, $t0, 1                          # b = !a
nor $t2, $t0, $zero                        # c = ~a
```

3.3.1.7 - Comparació '=='

Es tradueix a una resta seguida d'una negació lògica o d'una normalització a 0 o 1
Traduir c = a == b; suposant que a, b i c es guarden a `$t0, $t1, $t2`

```
sub     $t4, $t0, $t1              # $t4 = 0 si $t0 = $t1
sltiu   $t2, $t4, $t1              #negació lògica
```

3.3.1.8 - Comparació '!='

Es tradueix a una resta seguida d'una negació lògica o d'una normalització a 0 o 1
Traduir c = a != b; suposant que a, b i c es guarden a `$t0, $t1, $t2`

```
sub     $t4, $t0, $t1              # $t4 = 0 si $t0 = $t1
sltu    $t2, $zero, $t4            #negació lògica
```

3.3.2 - Conversió a valor lògic normalitzat

	Valors lògics correctes en C	Valores lògics normalitzats en C
Fals	0	0
Cert	diferent de 0	1

Es pot normalitzar el valor lògic amb instrucció `sltu`

```
sltu $t1, $zero, $t0  #si es 0: $t0 = 0, sino, $t1 = 1
```

3.3.3 - Operador booleans

3.3.3.1 - Operador and '&&'

Es tradueix a una resta seguida d'una negació lògica o d'una normalització a 0 o a 1.

`c = (a == b)`	`sub   $t4, $t0, $t1  #val 0 si son iguals` `sltiu $t2, $t4, 1    #negació lògica`
`c = (a != b)`	`sub  $t4, $t0,   $t1 #val 0 si son iguals` `sltu $t2, $zero, $t4 #normalitzar a 0 o 1`

3.3.3.2 And bit a bit "&" vs and lògica "&&"

- L'operador `and bit a bit` i l'operador `and lògic` poden donar un resultat **diferent!**

```
unsigned char a = 0x55, b = 0xAA, c;
c = a & b;                                  #c = 0
c = a && b;                                 #c = 1
```

- Traducció a MIPS (a, b i c estan a `$t0, $t1, $t2`, respectivament

```
and $t2, $t1, $t0                           #c = a & b

sltu $t3, $zero, $t0
sltu $t4, $zero, $t1                        #c = a && b
and  $t2, $t3,   $t4
```

3.3.3.3 - Operador or '||'

Utilitzar instrucció `or` i normalitzar el resultat
Traduir `c = a || b`; suposant que a, b i c es guarden en `$t0, $t1, $t2`

```
or      $t2, $t1, $t0
sltu    $t2, $zero, $t2
```

3.3.4 - Salts condicionals relatius al PC

`beq rs, rt, label`	si `rs == rt` PC = PCup + SignExt(offset16)
`bne rs, rt, label`	si `rs != rt` PC = PCup + SignExt (offset16)
`b label`	`beq $0, $0, label`

- La direcció de destí de salt s'especifica mitjançant una etiqueta en ensamblador
- La instrucció inclou un immediat de 16 bits (offset16)
  - El offset codifica la distància a saltar respecte al PC, que és la diferència entre la direcció de destí i la direcció de la instrucció següent
  - offset16 = (label - PCup) / 4
  - PCup = PC + 4
  - Permet saltar dintre del rang [-2^15, -2^15-1]
  - Mode de direccionament relatiu al PC

3.3.4.1 - Macros de salto

`blt rs, rt, label`	si `rs < rt` salta a label	`slt $at, $rs, $rt` `bne $at, $zero, $label`
`bgt rs, rt, label`	si `rs > rt` salta a label	`slt $at, $rs, $rt` `bne $at, $zero, $label`
`bge rs, rt, label`	si `rs >= rt` salta a label	`slt $at, $rs, $rt` `beq $at, $zero, $label`
`ble rs, rt, label`	si `rs <= rt` salta a label	`slt $at, $rs, $rt` `beq $at, $zero, $label`

- Per a naturals, les macros `bltu, bgtu,bgeu` i `bleu` s'expandeixen de la mateixa forma utilitzant `sltu`

3.3.4.2 - Salts incondicionals

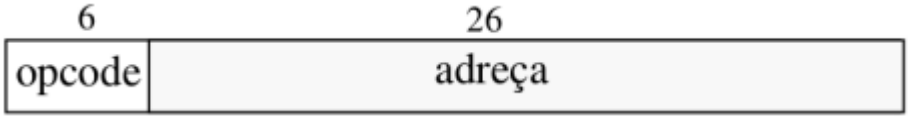
- La macro b està restringida al rang $[-2^{15}, 2^{15}-1]$
- Per salts que superin el rang, s'utilitzen les següents instruccions:

`j target`	`PC = target`
`jr rs`	`PC = rs`
`jal target`	`PC = target; $ra =` PC_{up}
`jalr rs, rd`	`PC = rs; rd =` PC_{up}

- S'utilitzen per implementar subrutines

3.3.4.2.1 - En mode pseudodirecte

- Les instruccions j i jal es codifiquen al format J
  - La direcció de destí es codifica en els 26 bits de menor pes de la instrucció



3.3.4.3 - Modes de direccionament

- Mode registre
  - `addu $t0, $t0, $t1`
- Mode immediato
  - `addiu $t0, $t0, 4`
- Mode memòria
  - `lw $t0, 0($t1)`
- Mode pseudodirecte
  - `j label`
- Mode relatiu al PC
  - `beq $t0, $zero, label`

3.3.5 - Sentencia if - then - else

Traduïr a MIPS el codi en C, suposant que a, b, c, d són enters emmagatzemats a $t0, $t1, $t2, $t3:

```
if ( a >= b ) {
  d = a;
}
else {
  d = b;
}
```

```
        blt $t0, $t1, else
        move $t3, $t0
        b fisi
else:
        move $t3, $t1
endif:
```

3.3.6 - Evaluació ‘lazy’

En C, els operadors lògics `&&` i `||` s’evaluen d’esquerra a dreta de forma “lazy”, és a dir, en el `&&` si la part esquerra es falsa, ja no s'avalua i en el `||` si la part esquerra és certa.

Codi en C

```
if (a >= b && a < c)
    d = a;
else
    d = b;
```

Codi en MIPS

```
                    blt  $t0, $t1, else
                    bge $t0, $t2, else
                    move $t3, $t0
                    b endif
          else:
                    move
          endif:
```

3.3.7 - Sentencia switch

```
switch (expressio) {
  case const1:
     sentencies1;
     break ;
  case const2:
     sentencies2;
     break;
  default:
     setencies_default:
}
```

```
main:          li $t1, 2

               li $t2, 0x01
               beq $t1, $t2, firstState

               li $t2, 0x02
               beq $t1, $t2,
               secondState

               li $t2, 0x03
               beq $t1, $t2, thirdState

               b defaultState

firstState:    li $t4, 1
               b endSwitch

secondState:   li $t4, 2
               b endSwitch

thirdState:    li $t4, 3
               b endSwitch

defaultState   li $t4, 4
:              b endSwitch

endSwitch:
```

3.3.8 - Sentencia while

Traduïr a MIPS el codi en C, suposant que dd, dr i q són enters emmagatzemats a $t1, $t2, $t3:

```
q = 0;
while ( dd >= dr ) {
   dd = dd - dr;
   q++;
}
```

```
          move $t3, $zero
          blt $t1, $t2, fiwhile
while:
          subu $t1, $t1, $t2
          addiu $t3, $t3, 1
          bge $t1, $t2, while
fiwhile:
```

```
          move $t3, $zero

while:    blt $t1m $t2, fiwhile
          subu $t1, $t1, $t2
          addiu $t3, $t3, 1
          b while
fiwhile:
```

3.3.9 - Sentencia for

```
for(s1; condició; s2)
     s3;
```

Equivalent a:

```
while(condició) {
     s3;
     s2;
}
```

3.3.10 - Sentencia do - while

`do {`
`  sentencia_cos_do`
`} while(condició);`		

3.4 - Subrutines

Codi tot en main **Codi amb subrutina**

```
void main(){
    int x,y,z,m;
    …
    if (x > y) m = x;
    else       m = y;
    …
    if(y > z)  m = y;
    else       m = z;
    …
}
```

```
int max(int a, int b) {
    if(a > b)  return a;
    else       return b;
}

void main(){
    int x,y,z,m;
    …
    m = max(x,y);
    …
    m = max(y,z);
    …
}
```

3.4.1 - Crida a subrutina i retorn

- Podem utilitzar la macro b? NO LO ENTIENDO, ACABÁDLO

- S'ha de guardar la direcció de retorn a un registre
    - `jal` guarda la direcció de retorn i salta a una etiqueta
    - La direcció de retorn es guarda a `$ra`
    - `jr` permet saltar a la direcció guardada a un registre

```
max:
    …
    jr $ra

main:
    …
    jal max
    …
    jal max
```

3.4.2 - Paràmetres i resultats

Els paràmetres es passen als registres `$a0 - $a3`
El resultat es passa al registre `$v0`

Codi en C:	`void main(){`
	`  // en $t0, $t1, $t2`
	`  int x,y,z,m;`
	`  …`
	`  z = suma2 (x, y);`
	`}`
		`int suma2(int a, int b) {`
		`  return a+b;`
		`}`

Codi en MIPS:	`main:`
	`  move $a0, $t0`
	`  move $a1, $t1`
	`  jal suma2`
	`  move $t2, $v0`
		`suma2:`
		`  addu $v0, $a0, $a1`
		`  jr $ra`

- Els paràmetres de menys de 32 bits (char o short) s'expandeixen a 32 bits
  - Extensió de zeros (unsigned) o de signe (signed)
- Vectors: Es passa la direcció base

.data

```
short vec [3] = {5, 7, 9};
short sumv (short v[]);

void main(){
    short res;
    res = sumv (vec);
}
```

```
vec : .half 5, 7, 9
sumv:
    …
main:
    la $a0, vec
    jal sumv
    move $t0, $v0
```

- Per valor o per referència? C només admet el pas de paràmetres per valor. Aquest valor es pot passar mitjançant un punter

```
int a, b;

void main(){
    int *p = &a;
    sub(p);
```

```
void sub (int *p) {
    *p = *p + 10;
}
```

```
    sub(&b);
}
```

3.4.3 - Variables locals

- Es creen cada cop que s'invoca la funció
    - Valor indeterminat si no s'inicialitzen de forma explícita
    - Només son visibles dins la funció
    - Es guarden a registres temporals o a la pila

```
int foo(int x, int y){
    int a,b = 0;
    short vecs[8]
    …
}
```

3.4.3.1 Bloc d'activació i la pila

- Algunes funcions requereixen guardar variables locals a memòria

- Aquestes variables locals s'emmagatzemen a la pila del programa
    - La pila està inicialment situada a la direcció `0x7FFFFFFC`
    - Creix des d'una direcció alta fins direccions més baixes
    - El registre `$sp` (Stack Pointer) apunta al cap de la pila

- Cada funció manté el seu bloc d'activació a la pila
    - A l'inici es decrementa `$sp` per a reservar memòria a la pila
    - Al final s'incrementa `$sp` per alliberar memòria a la pila
    - `$sp` sempre ha de ser multiple de 4

3.4.3.2 Regles de la ABI per a varibles locals

- Les variables escalars es guarden als registres `$t0-$t9, $s0-$s7 o $v0-$v1`
    - Excepte si al cos de la funció apareix la variable local precedida per l'operador `&`
    - Si no hi ha suficients registres, les que no caben es guarden a la pila

- Les variables estructurades (vectors, matrius…) es guarden a la pila

- El bloc d'activació ha de respectar les següents normes:
    - Les variables locals es col·loquen a la pila seguint l'ordre en que apareixen a la declaració, començant des del cap de la pila
    - S'han de respectar les normes d'alineació (padding)
    - La direcció inicial i el tamany del bloc d'activació han de ser múltiples de 4

3.4.4 - Subrutines multinivell

- Subrutines que truquen a altres subrutines
- Context d'una subrutina:
    - Paràmetres ($a0 - $a3)
    - Direcció de retorn ($ra)
    - Punter de pila ($sp)
    - Càlculs intermedis

- Problema:
    - Com preservem el context de la rutina que ens ha trucat?
    - Com sabem quins registres podem modificar?

funcA:

```
addu  $t1, $a0, $a1
move  $a0, $a2
jal   funcC
addu  $v0, $v0, $t1
```

funcB:

```
addiu $t0, $t0, $t1
jal   $t0, $t0, $t1
subu  $t0, $t0, $t1
```

funcC:

…

3.4.4.1 Guardar i restaurar registres

- L'ABI permet preservar el context guardant a la pila el mínim número de registres
- Requereix dos passos:
    - 1.- Determinar els registres segurs
        - Identificar quines dades emmagatzemades a registres es generen ABANS d'una crida a subrutina i s'utilitzen DESPRÉS de la crida
    - 2.- Guardar i restaurar els registres segurs
        - Guardar el valor anterior dels registres segurs al bloc d'activació (pila) a l'inici de la subrutina
        - Restaurar el valor dels registres segurs al final de la subrutina

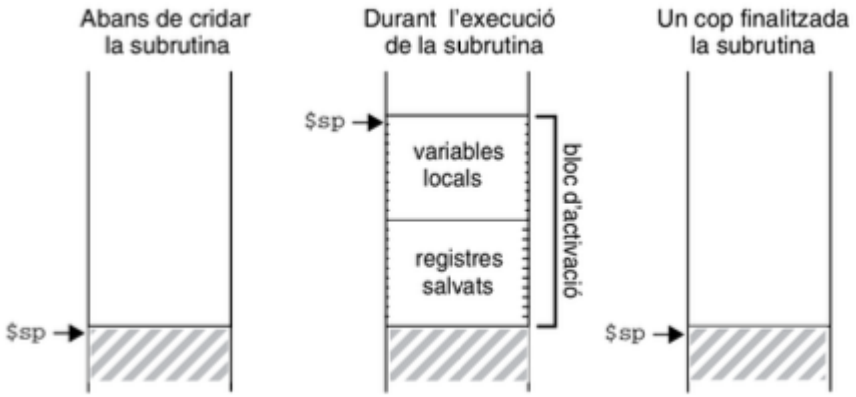
3.4.4.2 Estructura del bloc d'activació

El bloc d'activació està ubicat en la pila i inclou la següent informació:

- Variables locals:
    - De tipus estructurat
    - Escalars si els hi posem l'operador `&`

- Valors inicials dels registres segurs:
    - Només es guarden els que es modifiquen durant la subrutina
    - S'assigna un segur a cada dada emmagatzemada en un registre temporal que ha de sobreviure la trucada de subrutina
- El bloc d'activació ha de respectar les seguents normes:
    - Posició:
        - Les variables locals van al començament, seguides dels registres segurs
    - Ordenació:
        - Les variables locals s'ubiquen en l'ordre en que es declaren al codi
    - Alineament:
        - Les variables locals han de respectar les normes d'alineament
        - Els dos registres segurs han d'anar alineats a direccions múltiples de 4
        - El tamany del bloc d'activació ha de ser múltiple de 4

3.4.4.2.1 Imatge:



3.5 Exercicis