**Tutoraggio Calcolatori Elettronici – Assembler 2 (C/Assembler)**

**Primo programma**

void main(Void)

{

return;

}

Invoco il **main**, ma non restituisco nessun valore. Cioè restituisco il valore al chiamante. (Statement return)

.text → Dove devo inserire il codice che andrò ad eseguire

.globl main → Dichiaro la funzione Main (etichetta) globale, in modo da poter essere accessibile anche dall’esterno ad altri programmi.

Main:

jr $31 → (Restituisco il valore al chiamante, Statement return)

nop → (Metodo che utilizzo per evitare l’eccezione)

Dopo un’istruzione di salto, c’è un brach di lay slot. Si tratta di un istruzione eseguita dopo un **j\*\*** eseguita a prescindere, quindi dopo l’istruzione di salto vado ad inserire una qualsiasi istruzione “neutrale”. (**Nop** o **ottimizzazioni varie**, cioè posso inserire un’operazione che andrebbe eseguita comunque a prescindere dal salto, in questo modo riesco ad ottimizzare il codice e risparmiare delle linee di codice). L’istruzione dopo il **j\*\*** viene eseguita a prescindere (nell’emulatore MIPS è possibile disattivare questa cosa).

Vado a riutilizzare il codice, cioè utilizzo questo codice per il Main, la parte “finale”, per tutti gli altri programmi che andremo a realizzare.

**Dichiarazione Variabili**

int a=0; → Variabile globale

static b=0; →

inc c=1; →

static int d=2; →

const int e=3; →

void main(void)

{

int b ; → Variabile locale (o automatica)

return;

}

**Tipi di dati MIPS:**

# int a = 1; → Codice C che voglio ottenere in MIPS

# .data → Dichiaro i dati del programma (Variabili, array, liste ecc).

# a: → Direttive dell’assemblatore tramite le quali specificare il tipo di dato (\* dimensione prefissata)

.byte \* → C’è su SPIM

.short

.word \* → C’è su SPIM

.int

.quad \*

.long

.text → Dove devo inserire il codice che andrò ad eseguire

**Dichiarazione variabile inizializzata a 0:**

# int c= 0:

.bss → Su SPIM comando non presente, non riconosce .bss come una sezione valida

# c:

.space 4 → Lascia spazio nella sezione in cui utilizziamo per un numero (4 in questo caso) di byte (Occupo i 4 byte per avere qui i valori inizializzati a 0, in questo modo non devo inizializzarli a zero ogni volta e risparmio spazio) → Però questo c’è su SPIM. Lo spazio che lasciamo, di default viene inizializzato a 0.

**Dichiarazione costante:**

# const int d = 2 → Valore costante, sola lettura

# .rodata → Read Only Data

.rdata → Devo utilizzare questa dicitura su SPIM, o forse no

d:

.word 2

**Dichiarazione vettore:**

#int v[4] = {1,2,3,4} → Dichiarazione vettore C

v:

.word 1, 2, 3, 4 → Dichiarazione vettore SPIM

**Array valori inizializzati a 0:**

#int v[4] = {0, };

w:

.space 16 → Inizializzo n\*4 elementi inizializzati a 0

**Prova codice:**

.data

a:

.word 1

.globl main → Dichiaro la funzione Main (etichetta) globale, in modo da poter essere accessibile anche dall’esterno ad altri programmi.

Main:

jr $31 → (Restituisco il valore al chiamante, Statement return)

nop → (Metodo che utilizzo per evitare l’eccezione)

**Torno a lavorare con il codice base:**

.text

.globl main

Main:

addi $4, $0, 1

ori $4, $0, 1

jr $31

nop

La sezione Data si trova all’interno della memoria. Visto che l’ISA MIPS non prevede che le operazioni provengano direttamente dalla memoria, bisogna prima caricare le variabili dentro i registri.

.data

a:

.word 1

.text

.globl main

Main:

la $2, a

# int \*$2 = &a;

# int $3= a (d+0);

# int $s3 = d[0];

lw $3, 0($2) → Puntatore in C

addi $3, $3, 1

sw $3, 0($2)

jr $31

nop

**Loop Assembly**

**.r(o)data** → Non penso di modificare dimensione Array, quindi lo metto in Read Only Data

dim:

.word 4

**.data**

v:

.word 0x1, 0x2, 0x3, 0x4

res:

.space 4

.text

.globl main

main:

$la $2, dim

$lw $2, 0($2)

$la $3, v

add $4, $0, $0

loop:

lw $5, 0($3)

nop

add $4, $4, $5

addi $3, $3, 4

addi $2, $2, -1

bne $2, $0, loop

nop

la $3, res

sw $4, 0($3)

jr $31

nop