LABORATORIO DI ARCHITETTURA DEGLI ELABORATORI

PROGETTO 6

CONVERSIONE DI CODICE BINARIO NELLA SUA ISTRUZIONE MIPS

- **Cinalli Gian Marco – 178627**
- **D'Ettorre Marco – 178925**

INDICE

PREMESSA

All'interno dei calcolatori elettronici le istruzioni vengono memorizzate come una serie di segnali elettrici e possono essere, quindi, rappresentate da numeri. Nell'**architettura MIPS** da noi trattata ad ogni singolo registro è assegnato un numero, che lo identifica univocamente all'interno di un'istruzione.

Ogni istruzione MIPS può essere rappresentata tramite un codice binario di 32 bit. Un esempio:

"add $s0, $s1, $s2" che corrisponde a → 00000010001100101000000000100000

Come l'architettura MIPS suddivide questi 32 bit tra le varie parti componenti un'istruzione?

I progettisti MIPS hanno deciso di predisporre formati diversi di suddivisione, per tipi diversi di istruzioni da rappresentare. I formati principali sono:

- ✓ **FORMATO R** (Register Adressing)
- ✓ **FORMATO I** (Immediate Adressing)
- ✓ **FORMATO J** (Jump)

Nel progetto a noi assegnato sono stati trattati i primi due formati.

Detti formati suddividono i 32 bit, rappresentanti l'istruzione, in vari campi, ed a ciascun campo associano una determinata parte dell'istruzione MIPS.

Prendiamo ora l'esempio sopracitato:

L'istruzione "add $s0, $s1, $s2" viene suddivisa così:

OP	RS	RT	RD	SHAMT	FUNCT
000000	10001	10010	10000	00000	100000
Inesistente	**"$s1"**	**"$s2"**	**"$s0"**	Inesistente	**"add"**

Il tipo di suddivisione sopra descritto viene effettuato dal FORMATO R ed è composto dai campi:

- ✓ **OP** "*Codice operativo*" – Operazione base dell'istruzione
- ✓ **RS** "*Primo operando sorgente*" – Registro
- ✓ **RT** "*Secondo operando sorgente*" – Registro
- ✓ **RD** "*Registro destinazione*" – Registro
- ✓ **SHAMT** "*Valore dello scalamento*" – Utilizzato per alcune operazioni di scalamento dei bit
- ✓ **FUNCT** "*Funzione*" – Specifica il tipo di funzione

Un esempio di istruzione in FORMATO I è, invece, il seguente:

"addi $s0, $s1, 128" che corrisponde a → 00100010001100000000000010000000

Come è possibile notare al posto dell'ultimo registro operando vi è una costante! Ciò fa sorgere un problema:

- ✓ Come possiamo rappresentare la costante "128" all'interno di un campo da 5 o 6 bit?

Ciò è praticamente impossibile in quanto con 5 bit il massimo numero rappresentabile è **31** e con 6 bit è **63**!!!

Per ovviare a questo problema i progettisti MIPS hanno deciso di mantenere fissa la lunghezza di 32 bit delle istruzioni, fondendo, però, gli ultimi tre campi in un unico grande campo da **16 bit.**

L'istruzione "addi $s0, $s1, 128" viene quindi rappresentata così:

OP	RS	RT	IMMEDIATE
001000	10001	10000	0000000010000000
"addi"	**"$s1"**	**"$s0"**	**"128"**

Il campo **IMMEDIATE** permette così la rappresentazione di **±32767**, massimo numero rappresentabile in 15 bit, e non 16, in quanto il primo bit viene dedicato al segno. Nel caso in cui quest'ultimo bit sia pari a 1, i restanti 15 bit verranno complementati a 2 restituendo il numero negativo corrispondente; nel caso in cui, invece, la costante faccia parte di un istruzione "Unsigned" (es. addiu, sltiu, ecc..) il massimo numero rappresentabile sarà **+32767** in quanto il bit di segno non verrà considerato.

Detto campo assume, invece, il nome **ADDRESS** nel caso in cui il valore in esso scritto rappresenti una costante da sommare ad un indirizzo di memoria, come nel caso di "lw $s0, 128($s1)" che andrà a selezionare la 128esima parola (word) puntata da $s1 (che dovrà, chiaramente, contenere un indirizzo di memoria e non un valore).

OP	RS	RT	ADDRESS
001000	10001	10000	0000 0000 1000 0000
"lw"	**"$s1"**	**"$s0"**	**"128"**

Resta ora da descrivere come i formati vengano riconosciuti dall'hardware, l'uno dall'altro. Questa distinzione risiede nel valore del campo **OP**:

- ✓ Se OP è = 0 → l'istruzione verrà codificata dal calcolatore secondo la divisione dei campi imposta dal FORMATO R
- ✓ Se OP è ≠ da 0 → L'istruzione verrà codificata dal calcolatore secondo la divisione dei campi imposta dal FORMATO I

ANALISI DEL PROBLEMA

Conversione di codice binario nella sua istruzione MIPS

Testo: *"Dato un codice binario m immesso da console, il programma deve produrre la relativa istruzione (assumendo che tale codice corrisponda ad una istruzione MIPS). Si prenda in esame una lista limitata di istruzioni. Ad esempio, se immettiamo 0x 0100 1111 1011 0100 0000 0000 0000 0000 e 0x 0010 0111 1010 0101 0000 0000 0000 0100 il programma dovra' ritornare "lw $a0, 0($sp)" ed "addiu $a1, $sp, 4".*

L'architettura MIPS converte ogni istruzione in un codice binario di 32 bit; quello che invece viene richiesto dal problema è l'inverso.
Per semplificarne la spiegazione procederemo per passi:

PASSO 1 – Prima di tutto bisogna controllare che tutti i bit immessi siano binari e che siano esattamente 32; nel caso in cui questo controllo fallisca si deve restituire un errore all'utente e si deve dare la possibilità a quest'ultimo di poter reinserire il codice.

PASSO 2 – Una volta passato positivamente il controllo precedente, il programma passa alla conversione in decimale dei primi 6 bit del codice binario inserito, il campo OP. Se il risultato di questa conversione è 0 il programma interpreterà la restante parte del codice secondo il FORMATO R, altrimenti secondo il FORMATO I.

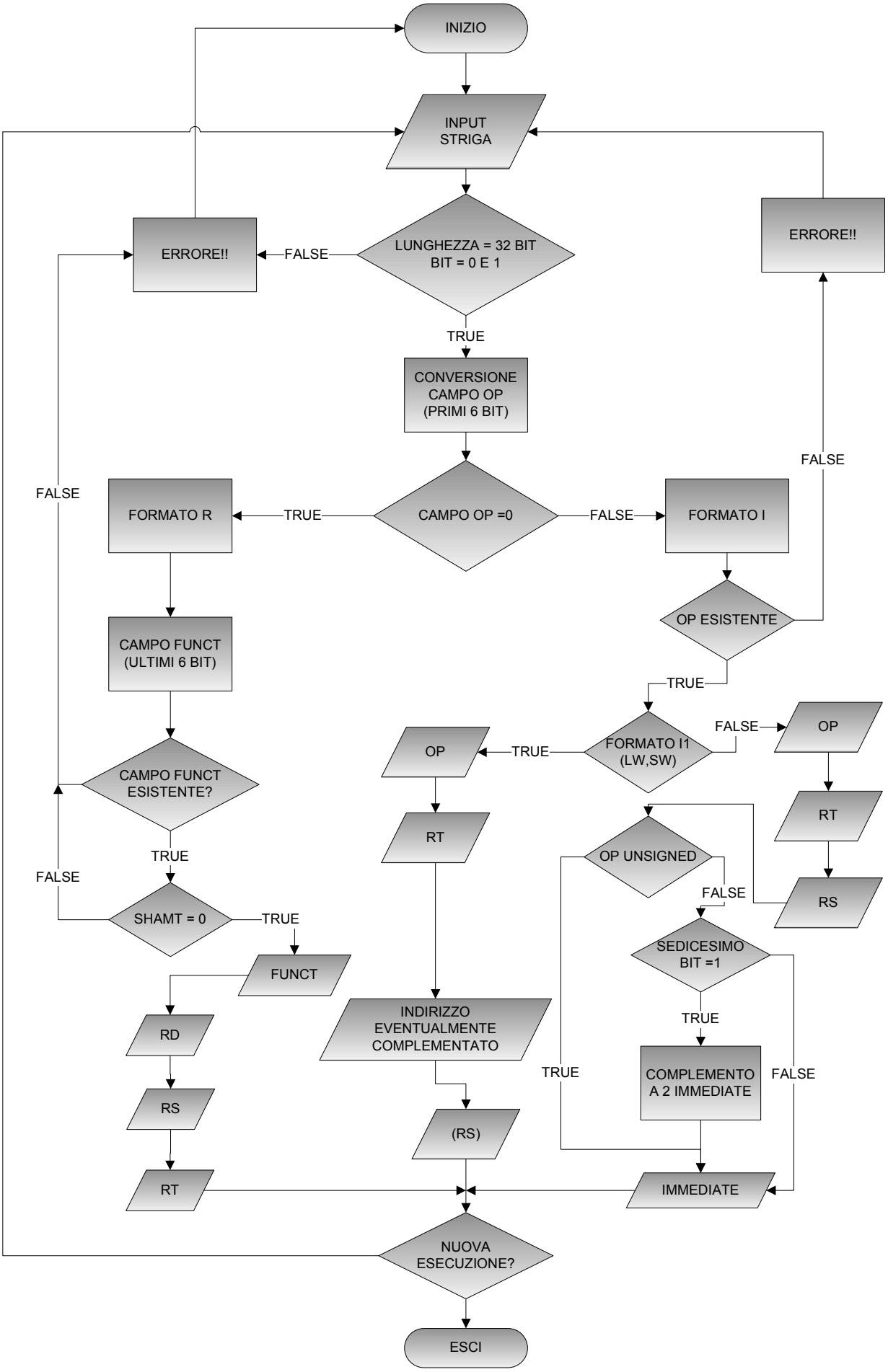
PASSO 3 – Nel caso del FORMATO R si andrà prima a convertire in decimale il campo FUNCT (Ultimi 6 bit), poi si procederà con la stampa dell'istruzione corrispondente al valore ottenuto. In seguito si convertiranno e stamperanno, allo stesso modo, i bit componenti il campo RD (Registro di destinazione), poi quelli per il campo RS (Primo registro sorgente), ed infine quelli del campo RT (Secondo registro sorgente).

Nel caso del FORMATO I, invece, si andrà prima a stampare il l'istruzione corrispondente al valore ottenuto dalla conversione in decimale del campo OP; se questo campo indica un'istruzione del tipo:

- ✓ "lw $s0, 10($s1)" allora si convertirà in decimale i campi RT, ADDRESS e RS, e poi si stamperà ciò a cui corrispondono (registri per i campi RT e RS, ed un numero per il campo ADDRESS)
- ✓ "addi $s0, $s1, 100" allora si convertirà in decimale i campi RT, RS ed IMMEDIATE, e poi si stamperà ciò a cui corrispondono (registri per i campi RT e RS, ed un numero, per il campo IMMEDIATE).
- ✓ Nel caso dei campi IMMEDIATE e ADDRESS il programma controllerà se il bit più significativo di questo campo è 1, se il controllo avrà esito positivo la restante parte del campo verrà complementata a 2, in quanto rappresenterà un numero negativo che poi verrà normalmente stampato.

PASSO 4 – Dopo aver stampato l'intera istruzione corrispondente al codice binario inserito, il programma chiederà all'utente se vuole effettuare un'altra conversione, nel caso in cui l'utente accetti, il programma verrà rieseguito nuovamente, in caso contrario terminerà l'esecuzione.

DIAGRAMMA DI FLUSSO



DESCRIZIONE DELLE VARIE PROCEDURE

✓ "Main"

- Prende in input la stringa immessa e passa i relativi parametri alle varie procedure.

✓ "Control" – Procedura per il controllo errori di input (Lunghezza – Coerenza bit)

- Controlla se tutti i bit del codice binario inserito sono 0 o 1 e se esso è di 32 bit.

✓ "FormatoR" - Procedura che stampa le istruzioni codificate in FORMATO R

Codifica la stringa in input secondo il FORMATO R:

- legge il campo FUNCT codificando la relativa istruzione e la stampa. Se la stringa immessa nel campo FUNCT non trova riscontri nella tabella delle istruzioni FORMATO R, presente nel ".data", stampa un messaggio di errore, altrimenti:
- Legge i bit del campo RD, li codifica in decimale e stampa il corrispondente registro.
- Legge i bit del campo RS, li codifica in decimale e stampa il corrispondente registro.
- Legge i bit del campo RT, li codifica in decimale e stampa il corrispondente registro.

✓ "FindFunct" - Procedura che cerca l'istruzione corrispondente al valore binario immesso nel campo FUNCT e la stampa

✓ "FormatoI" - Procedura che stampa le istruzioni codificate in FORMATO I

- Chiama la procedura "FindOp" per trovare e stampare l'OP corrispondente ai primi 6 bit inseriti nel codice binario.
- Legge i bit del campo RT, li codifica in decimale e stampa il corrispondente registro.
- Legge i bit del campo ADDRESS, li codifica in decimale (eventualmente complementando i bit) e stampa il corrispondente valore.
- Legge i bit del campo RS, li codifica in decimale e stampa il corrispondente registro.

✓ **"Print16" - Procedura che stampa il valore convertito in decimale del campo IMMEDIATE/ADDRESS**

✓ **"PrintRegister" - Procedura che stampa i registri corrispondenti ai valori inseriti nei campi RS, RT, e RD**

✓ **"FindOp" - Procedura che individua l'OP corrispondente al codice binario inserito nei primi 6 bit**

- Controlla se, al codice binario inserito nel campo OP, corrisponde un'istruzione inserita nella tabella degli OP nel ".data"; se il controllo è negativo, stampa un messaggio di errore, altrimenti stampa l'OP trovato.

✓ **"FormatoI2" - Procedura per le istruzioni con segno in FORMATO I**

- Legge i bit del campo RT, li codifica in decimale e stampa il corrispondente registro.
- Legge i bit del campo RS, li codifica in decimale e stampa il corrispondente registro.
- Chiama la procedura "SedicEquals1" per controllare se il primo bit del campo IMMEDIATE è 1; se lo è, complementa a 2 i bit e stampa il valore corrispondente in decimale.

✓ **"SediciEquals1" - Procedura che controlla, prima se il primo bit del campo IMMEDIATE/ADDRESS è 1, e se lo è complementa a 2 e poi stampa il campo in decimale**

- Controlla se il primo bit del campo IMMEDIATE/ADDRESS (bit di segno) è uguale a 1; se è così complementa a 2 il campo, lo converte in decimale, e stampa il valore ottenuto. Se invece, detto bit è pari a 0, converte in decimale i restanti bit del campo e stampa il valore ottenuto dalla conversione.

✓ **"Conta1" - Procedura che conta il numero degli 1 presenti negli ultimi 16 bit**

- Legge gli 1 presenti nel campo IMMEDIATE/ADDRESS così da sapere fino a che punto complementare il campo se il numero è negativo.

✓ **"Complemento2" - Procedura che complementa a 2 il campo IMMEDIATE/ADDRESS**

✓ **"FormatoI3" - Procedura per le istruzioni unsigned in FORMATO I**

- Legge i bit del campo RT, li codifica in decimale e stampa il corrispondente registro.
- Legge i bit del campo RS, li codifica in decimale e stampa il corrispondente registro.
- Legge i bit del campo IMMEDIATE, li codifica in decimale (non complementanto i bit ed escludendo il bit di segno) e stampa il valore ottenuto dalla conversione.

✓ **"PrintSpazio" - Procedura che stampa uno spazio vuoto**

✓ **"PrintVirgola" - Procedura che stampa il carattere ","**

✓ **"PrintFinalStr" - Procedura che stampa il codice binario a fine esecuzione**

✓ **"PrintContinue" - Procedura che chiede se si vuole rieseguire il programma**

✓ **"LetturaCampo" - Procedura che viene chiamata per convertire il contenuto binario di ogni campo in decimale**

✓ **"Esci" - Termina il programma**

PSEUDO-ISTRUZIONI UTILIZZATE

Alcune delle istruzioni messe a disposizione dall'assembler MIPS al programmatore, per maggior comodità non sono reali, ma particolari; esse sono chiamate **pseudo-istruzioni** e la loro funzione è esattamente la stessa di una o più (normalmente poche) istruzioni MIPS reali. Gli unici svantaggi derivanti dal loro utilizzo sono:

- ✓ L'utilizzo del registro "$at"
- ✓ Diminuzione delle prestazioni

Qui di seguito verranno elencate le pseudo-istruzioni utilizzate all'interno del programma.

- ✓ "la $t1, label" → *Load address*
- ✓ "li $t1, 100" → *Load 16 bit immediate into register*
- ✓ "beq $t1, 100, label" → *Branch if equal to immediate*

La pseudo-istruzione "la $t1, label" ha un argomento immediato di 32 bit (l'indirizzo dell'etichetta "label" che corrisponderà, per esempio a 0x100FF300) che nessuno dei tre formati R, I e J è in grado di contenere, per cui è necessario sostituirla con due istruzioni ("*load upper immediate*" e "*or immediate*") capaci di gestire un argomento di 16 bit:

lui $1, 0x100F
ori $9, $1, 0xF300

La pseudo-istruzione "li $t1, 100", è del tutto equivalente a:

ori $9, $zero, 100
oppure
addi $9, $zero, 100

La pseudo-istruzione "beq $t1, 100, label", invece, corrisponde alle due istruzioni:

addi $1, $zero, 100 → che pone nel registro "$1" ($at), il valore immediato "100"
beq $9, $1, label → che confronta i registri "$9" e "$1", e se sono uguali salta all'indirizzo di "label"

ERRORI RISCONTRATI

LF – Line Feed

Nello svolgimento del progetto è stato riscontrato un piccolo problema riguardo all'inserimento di una stringa da console.

Questo problema consisteva nell'inserimento automatico, da parte del compilatore, del carattere "LF – Line Feed (corrispondente a 10 nella tabella ASCII)" a fine stringa. Questo carattere non era visibile alla stampa, ma in fase di esecuzione; infatti quando la procedura "Control" andava a leggere ogni singolo carattere della stringa immessa, arrivando a detto carattere, effettuava un confronto di quest'ultimo con i caratteri "0" ed "1" e, chiaramente, restituiva l'errore: "ERRORE!!! CODICE INSERITO NON VALIDO, INSERIRE 32 BIT BINARI!!!" (Messaggio da noi definito per indicare una non correttezza dei bit inseriti).

Dopo aver effettuato ricerche su internet e vari test, si è arrivati alla conclusione che il suddetto "problema di compilazione" era dovuto alla semplice pressione del tasto "Invio" in seguito all'immissione della stringa da console. Ciò ci è stato anche confermato, dalla prova dell'inserimento della stringa, non da console, ma direttamente nel ".data" a fianco dell'etichetta corrispondente; infatti, cosi facendo il carattere non era più presente.

Per ovviare a questo problema, si era inizialmente modificata la condizione di uscita dal ciclo di lettura della stringa, controllando se, come carattere di fine-stringa, si era letto 10 e non il classico valore 0. Successivamente si è pensato di "eliminare" questo inconveniente alla radice, sostituendo tramite una "sb – Store Byte", il carattere LF con l'intero "0", il reale carattere di fine-stringa.

Così facendo la procedura "Control" non restituiva più il messaggio di errore sopracitato.