Esercitazione 10: "Analizzatore di forme d'onda"

(Esame di Calcolatori Elettronici)

di Silvio Moioli per l'Università degli Studi di Bergamo

Relazione

(documento CalEl_01)

v. 1.0 del 4 Maggio 2005

Testo dell'esercitazione

"Una forma d'onda con componente continua nulla viene fornita in ingresso ad un convertitore analogico-digitale collegato ad un microcalcolatore basato sul microprocessore MIPS R2000 (clock pari a 10 MHz).
L'inizio della conversione viene comandato mediante l'invio del comando 80 esadecimale sulla cella denominata ADCR. Dopo 10 microsecondi dall'inizio della conversione il microprocessore potrà leggere l'informazione digitale corrispondente alla tensione analogica in ingresso. L'informazione digitale sarà rappresentata, in modulo e segno, tramite 12 bit (i numeri positivi hanno il bit 11 uguale a 0; i numeri negativi hanno il bit 11 uguale a 1): in particolare, il dato potrà essere letto attraverso la cella a 16 bit denominata ADC.
Il programma di cui si chiede la scrittura deve essere in grado di rilevare gli attraversamenti dello zero in salita o discesa della forma d'onda di ingresso e di valutare il tempo durante il quale la forma d'onda di ingresso si è mantenuta negativa o positiva.
Lo stesso programma deve inoltre pilotare un display esadecimale con il tempo durante il quale la forma d'onda di ingresso si è mantenuta negativa, mediante l'invio di digit esadecimali alla cella NEG. Discorso analogo vale per un secondo display esadecimale da pilotare con il tempo durante il quale la forma d'onda si è mantenuta positiva: in tal caso la cella da pilotare sia POS.
Si tenga presente che gli intervalli di tempo sono variabili fra 12 millisecondi e 5 secondi.
La lettura del valore digitale corrispondente alla forma d'onda in ingresso deve essere eseguita una volta ogni 100 microsecondi.
Alle celle di memoria sopra menzionate si assegnino indirizzi arbitrari che cadano, però, nell'area dei dati dell'architettura MIPS.
Il programma deve essere assemblato, linkato e sottoposto a simulazione. Si faccia una stampa del sorgente del programma realizzato e si produca una breve relazione che descriva le modalità seguite nelle simulazioni effettuate ed i risultati raggiunti."

Specifiche aggiuntive

Durante la realizzazione del programma e in base ai colloqui e alle spiegazioni aggiuntive del docente, si è ritenuto necessario o opportuno stabilire le seguenti specifiche progettuali aggiuntive:

- Qualsiasi istruzione MIPS viene eseguita dal microcalcolatore in esame in un tempo uguale all'inverso della frequenza di lavoro del microprocessore. Si suppone, in altre parole, che ogni istruzione venga eseguita esattamente in un tempo di clock. Dai dati del problema si deduce quindi T=1/*f*=1/10Mhz=100ns=0,1µs;
- Si suppone tollerabile un (relativamente piccolo) errore di temporizzazione dei rilevamenti dei dati dall'ADC (1 o 2 microsecondi su 100 richiesti);
- Si suppone che i due display siano in grado di visualizzare quattro cifre esadecimali ciascuno, e che siano provvisti di appositi decoder-driver in modo tale che se nella cella POS (o NEG) viene scritto, ad esempio, il valore esadecimale a65a il display corrispondente visualizzi direttamente "a65a";
- Si suppone che il dato venga visualizzato immediatamente (entro 100 microsecondi) sul display una volta scritto il dato nella cella relativa;
- Si suppone che l'operatore che dovrà usare il sistema sia a conoscenza del fatto che il numero esadecimale visualizzato sul display "POS" abbia il significato seguente: tempo in cui l'onda si è mantenuta positiva (s) = valore fornito (convertito in decimale) * 100 (µs). Lo stesso vale per il display "NEG";
- Si suppone che il programma debba funzionare in maniera continuativa per un tempo indeterminato;
- Si suppone che la prima misurazione (all'accensione del dispositivo) possa essere scartata, per via dei tempi di inizializzazione e del fatto che non è possibile misurare il

tempo totale di permanenza positiva (o negativa) dell'onda prima del primo attraversamento dello zero rilevato;
- Si suppone che l'ADC inizi la conversione sul fronte di salita del bit 7 della cella ADCR;
- Si suppone che non si verifichino (o non siano rilevanti) tutti gli attraversamenti dello zero che avvengono in tempi inferiori a 100 μs l'uno dall'altro.

Dettagli implementativi: filosofia del codice e uso dei registri

Durante la progettazione e la stesura del codice, si è fatto riferimento ad alcune linee-guida, che si possono riassumere nei seguenti punti:
- Funzionalità: il codice è stato scritto e rivisto più volte su carta prima della prova sul simulatore, in modo da minimizzare il numero di errori da scoprire in quest'ultima fase (dal momento che non sempre sono di facile individuazione). Naturalmente sono comunque state svolte diverse prove sul simulatore controllando “passo-passo” che tutto funzionasse come previsto;
- Chiarezza: il codice prodotto dovrebbe essere facilmente comprensibile da chiunque abbia esperienza di un linguaggio assemblativo, e dovrebbe essere di immediata comprensione da chi conosca specificamente il MIPS. Particolare cura in questo senso è stata posta nella documentazione interna ed esterna;
- Semplicità: mantenendo il codice più semplice possibile è più facile spiegarlo, testarlo e mantenerlo. Per questo diverse porzioni del programma sono state riscritte dopo la prima stesura;

Questi criteri sono anche stati alla base dell'organizzazione dei registri (che è stata decisa, ovviamente, prima dell'inizio della stesura del codice, per evitare inutili sprechi di tempo). In questo senso ho infatti deciso la seguente lista di priorità:
1. Quando possibile, usare i registri secondo le convenzioni in uso dai programmatori MIPS, ovvero:
   - Usare i registri “$a” (e solo quelli) per il passaggio dei parametri;
   - Usare i registri “$v” per i valori di ritorno;
   - Usare i registri “$t” prevalentemente nelle subroutine, in modo tale che il loro valore non debba essere preservato;
   - Usare i registri “$s” prevalentemente nel main, in modo tale che il loro valore debba essere preservato;
   - Usare i registri “speciali” $ra,$sp,$gp,$fp solo ed esclusivamente nel loro ruolo;
   - Evitare l'uso di $at e dei registri “$k”.
2. Se l'approccio del punto precedente non fosse risultato sufficientemente flessibile (ad esempio, in caso di un numero di registri superiore), salvare e ripristinare in memoria (preferibilmente sullo stack) i registri $t nelle subroutine e i registri $s nel main;
3. Per le subroutine nidificate o ricorsive, salvare e ripristinare $ra usando lo stack;
4. Nel caso in cui la complessità crescesse ulteriormente da rendere troppo scomoda la scrittura del codice, salvare e ripristinare sia i registri $s che i registri $t in ogni subroutine, in modo da avere ben 18 registri a disposizione;
5. Se fosse necessario passare più di due parametri a una subroutine, usare lo stack (push dei parametri da parte del chiamante, pop da parte del chiamato, come fanno spesso i compilatori per l'architettura x86). Lo stesso vale (al contrario) per i valori di ritorno.

In realtà (fortunatamente) è stato applicata solo una piccola parte dei cinque punti appena descritti, dal momento che è stato possibile mantenere il livello di complessità del programma piuttosto basso. Alla versione finale del programma risultano infatti applicate solo le seguenti convenzioni sull'uso dei registri:
- Uso dei registri $a,$v,$t,$s ed $ra come da convenzioni MIPS, in particolare vengono usati solo i primi quattro dei registri $t (esclusivamente nelle subroutine) e solo i primi cinque degli $s (esclusivamente nel main);

- $ra viene salvato e ripristinato usando lo stack dalle routine che necessitano di chiamarne altre al loro interno;
- Il ruolo specifico di ogni registro è chiarito nel codice, prima della subroutine (o del main) che lo usa.

Alcuni dettagli implementativi rilevanti

Rappresentazione dei dati

Il primo problema posto dal testo è stato quello di decidere come manipolare i dati dall'ADC. Essendo rappresentati in modulo e segno su 12 bit, infatti, non sono manipolabili direttamente dalle istruzioni MIPS a patto di particolari accorgimenti. La questione è stata però facilmente superata dal fatto che, non dovendo memorizzare o compiere operazioni sul modulo dei dati (è infatti sufficiente tenere conto del segno per rilevare gli attraversamenti dello zero), il problema era riconducibile all'isolamento del bit di segno e a qualche semplice test su di esso. Naturalmente, però, nella memoria del MIPS i 12 bit vengono memorizzati in halfword (16 bit) i cui 4 bit più significativi sono sempre 0, oppure in registri i cui 20 bit più significativi sono a 0.
Altro problema è stato quello di decidere quanti display pilotare, e come pilotarli. Dato che il testo parlava di “digit esadecimali” (e dato che era il modo più semplice per visualizzare i dati), si è subito deciso di stampare sui display il numero di letture consecutive per cui il segno si fosse mantenuto costante. Ciò significa che il tempo in secondi per cui l'onda si mantiene positiva è data dal numero sui diplay per il periodo di campionamento, ossia 100 microsecondi. Sempre per semplicità, si è supposto che non fosse necessario convertire il dato in un formato specifico per il display, ossia che esso fosse dotato di un appropriato decoder-driver e che il nibble meno significativo per il MIPS fosse quello corrispondente al digit più a destra e così via.
Stabilito questo, si è trattato di calcolare il numero di display necessari. Dal momento che la massima permanenza del segnale a valori positivi o negativi è 5 secondi, significa che al massimo si faranno 50000 rilevazioni consecutive con lo stesso segno. Perciò, saranno necessari almeno 16 bit (considerati senza segno), ossia 4 nibble (quindi 4 cifre esadecimali a display). Dal punto di vista dell'assembly MIPS, questo significa una halfword per POS e un'altra per NEG, avendo cura di usare l'istruzione sh per scrivervi dati.

Temporizzazione

Le richieste di temporizzazione del testo sono state risolte fondamentalmente con l'assunzione che ogni istruzione venga eseguita in un tempo pari all'inverso del periodo, ovvero 0,1 µs. Su questo principio è stata costruita la subroutine delay che fondamentalmente si occupa della temporizzazione di tutto il programma. C'è da dire che né questa né il resto del programma sono perfettamente temporizzati, si è infatti supposto che piccoli errori siano tollerabili anche se sarebbe possibile creare un programma senza difetti in questo senso. Ad esempio, la subroutine delay nella sua implementazione attuale aspetta 0,1 µs in più di quello che dovrebbe, ossia dall'istante della chiamata al ritorno al chiamante trascorrono 0,1 µs in più di quanto specificato nel parametro:

```
delay:
    # Aspetto 0.8 microsecondi
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
```

```
    # Decrementa a0 e...
    addiu $a0,$a0,-1
    bne $a0,$zero,delay
    # ...se è 0 ho finito, torno al chiamante
    jr $ra
```

Come si può facilmente osservare la routine attende esattamente per 1 µs ogni ciclo, e fa $a0 cicli. Alla fine però si deve eseguire l'istruzione jr, che impiega altri 0,1 µs per tornare al chiamante. Ciò è inevitabile (a meno dell'uso delle macro anziché delle subroutine), e “bilanciare” questo ritardo potrebbe non essere semplice, pertanto si accetta un errore di 0,1 µs.
Un ragionamento simile è stato applicato per il fatto che le letture avvengono ogni 100 µs: le istruzioni del main e dei sottoprogrammi, ogni ciclo, prendono da 1 a 2 µs circa, più 10 µs di attesa per il dato dall'ADC. Per questa ragione, all'inizio di ogni ciclo, si attende per 88 µs (in altre parole si accetta un errore di circa 1 µs su 100).

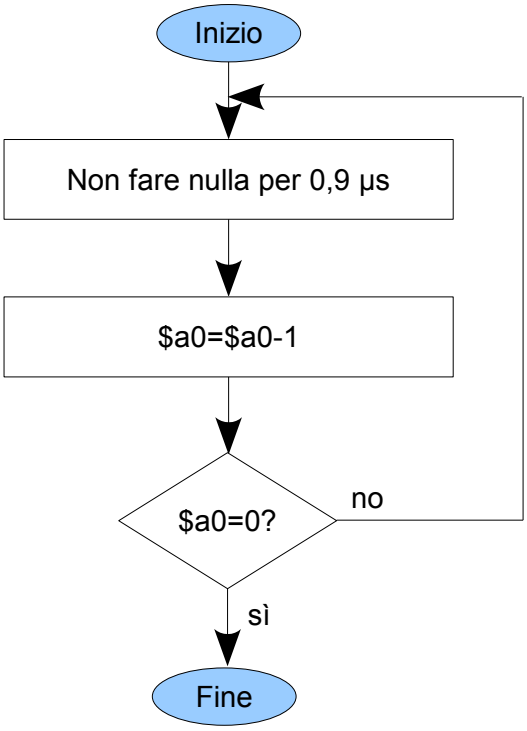
```
infloop:
    # Ciclo principale
    #
    # Aspetto 88 microsecondi e leggo dall'ADC
    li $a0,88
    jal delay
    jal read_adc_data
    ...
```

Testing e debugging

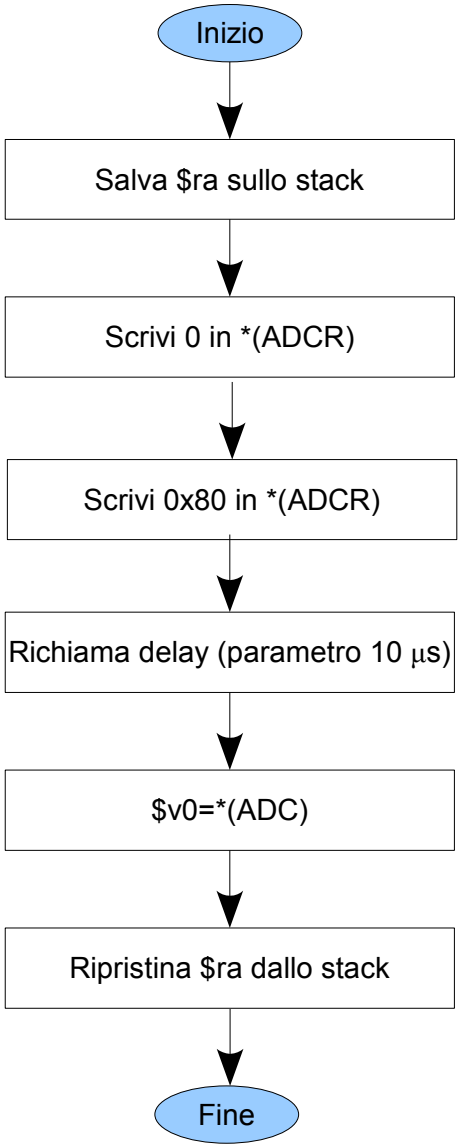
Il programma è stato più volte testato (e sono stati corretti bug) tramite il simulatore SPIM di James Larus. In particolare è stata usata la versione un*x del programma in ambiente Linux. Dal momento che ho avuto problemi nella ricompilazione dell'interfaccia grafica xspim (che comunque è ben lungi dall'essere gradevole o produttivamente utilizzabile) ho utilizzato la versione a linea di comando, che si è dimostrata all'altezza del compito (ed anzi molto comoda in certi aspetti).
Durante l'uso ho avuto modo di scoprire alcune particolarità di SPIM, tra cui una che mi è sembrata comoda e degna di menzione: il simulatore cerca automaticamente un'etichetta “main” e se la trova setta automaticamente $pc all'indirizzo di questa etichetta.

Diagrammi di flusso

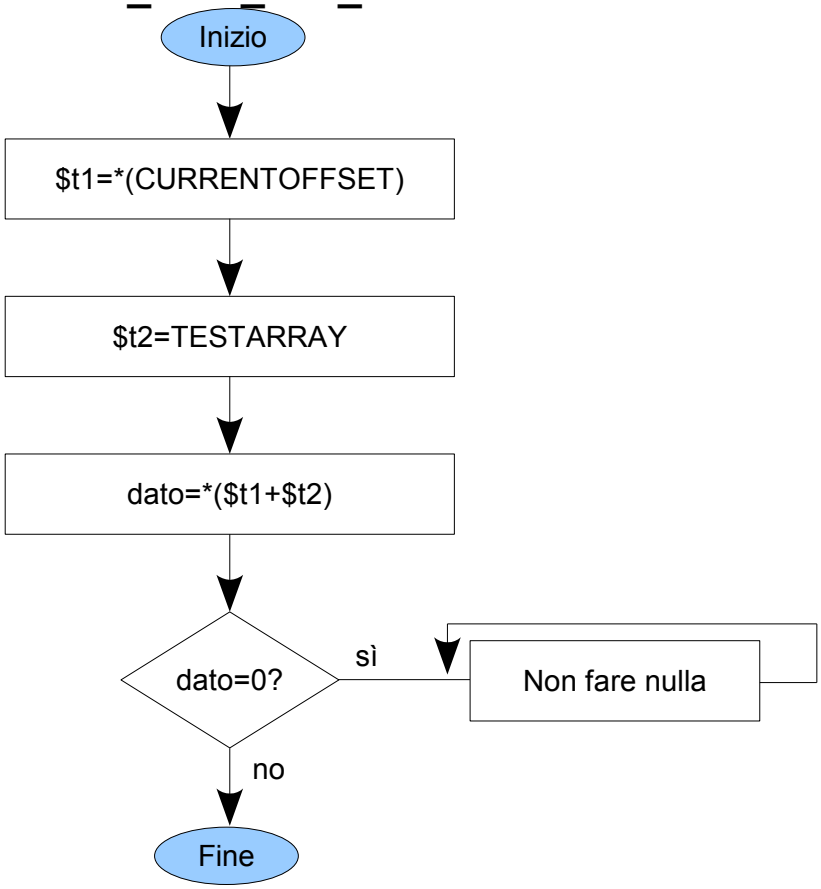
Subroutine delay



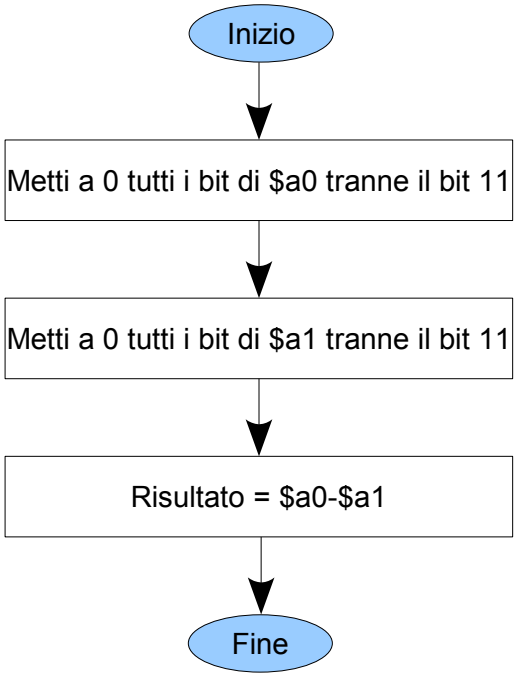
Subroutine read_adc_data



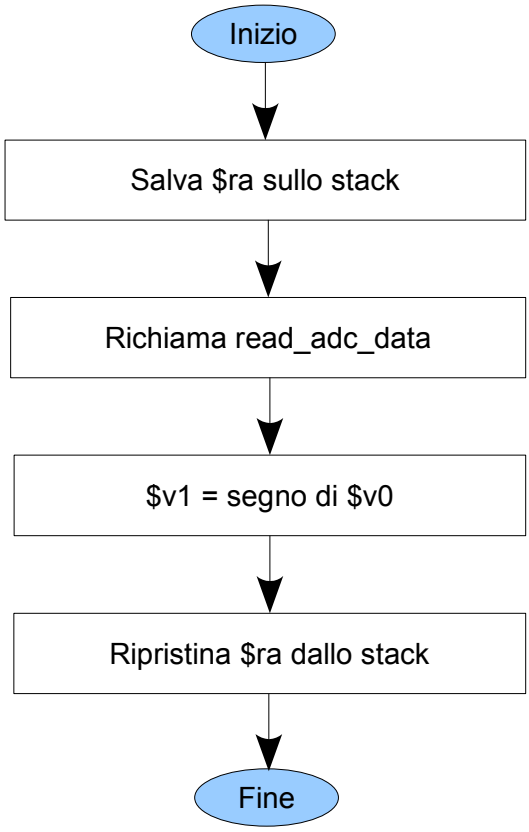
Subroutine fake_read_adc_data



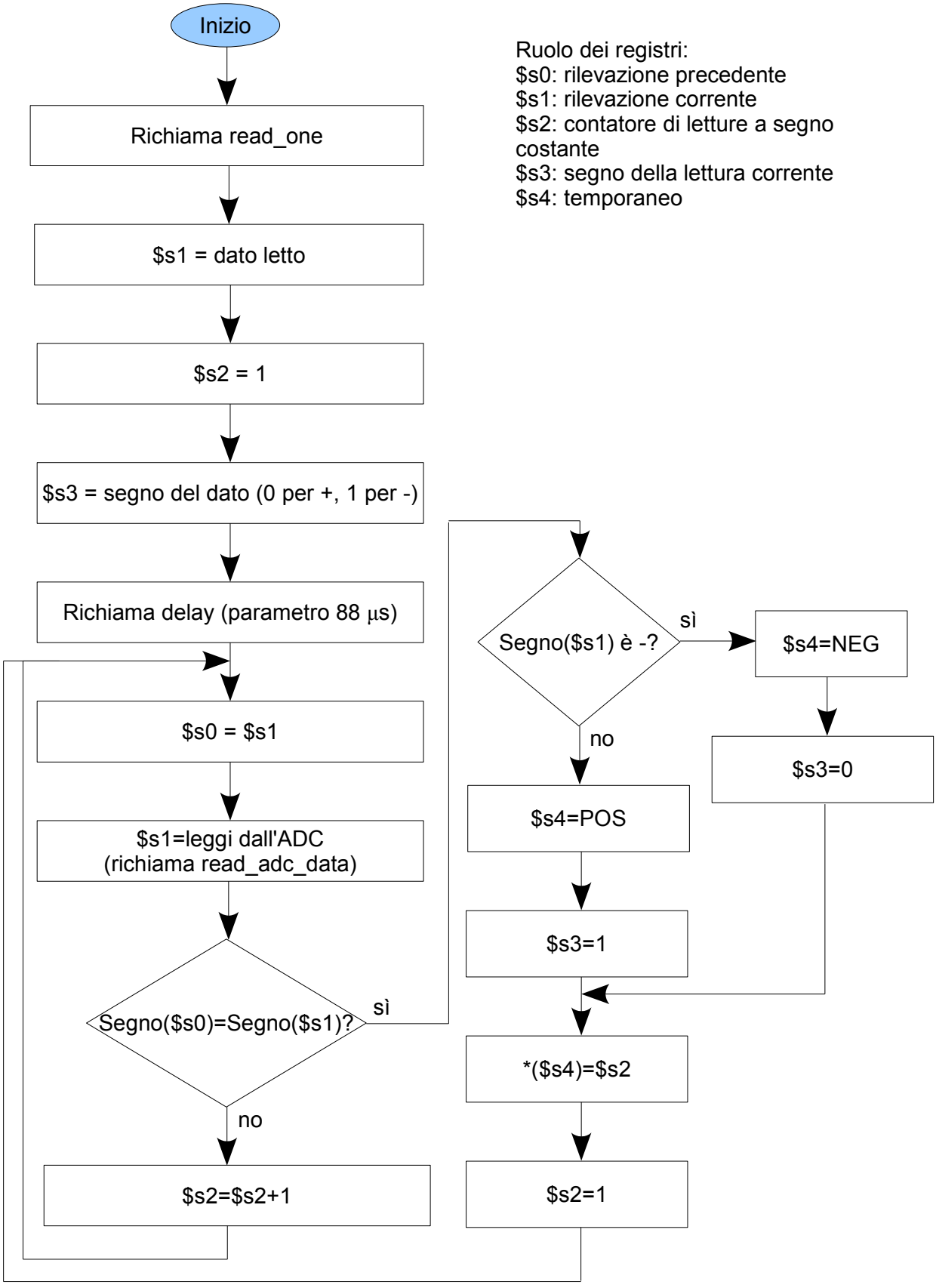
Subroutine test_sign_change



Subroutine read_one



Main routine



Ruolo dei registri:
$s0: rilevazione precedente
$s1: rilevazione corrente
$s2: contatore di letture a segno costante
$s3: segno della lettura corrente
$s4: temporaneo

Sorgente del programma

```
# Silvio Moioli, 46598
# Svolgimento dell'esercizio 10 "Analizzatore di forme d'onda"
# Esame di Calcolatori Elettronici anno 2005, prof. G. Coldani

# Convenzioni sui registri (vedi relazione per dettagli)
#
# Riservati ai parametri: $a0, $a1
# Riservati ai valori di ritorno dalle subroutine: $v0, $v1
# Uso main (da preservare nelle subroutine): da $s0 a $s4
# Uso locale (utilizzabili dalle subroutine): da $t0 a $t3

# Direttive all'assemblatore - aree di memoria rilevanti

.data
# Cella comandi dell'ADC
ADCR: .byte 0
# Cella dati dell'ADC
ADC: .half 0x0822
# Celle per il pilotaggio dei display
NEG: .half 0
POS: .half 0
# Dati per il test del programma (vedi fake_read_adc_data)
CURRENTOFFSET: .word 0
TESTDATA: .half 0x0B34,0x0334,0x0364,0x0376,0x0B76,0x0B56
#Segni (bit 11)    -      +      +      +      -      -
          .half 0x025E,0x0334,0x0334,0x0B76,0
#                  +      +      +      -

# Codice MIPS
.text

# Main routine
#
# Scopo: leggere ogni 100 microsecondi circa un dato dall'ADC ed aggiornare
# opportunamente i display (vedi relazione)
# Ruolo dei registri: $s0 contiene la rilevazione precedente (dato dall'ADC),
# $s1 quella corrente, $s2 il contatore di letture consecutive in cui la forma
# d'onda è rimasta di segno costante, $s3 è il suo segno (vale 0 se è positiva
# o 1 se è negativa), $s4 è un registro temporaneo per le operazioni di sh.
# Tutti gli altri registri come da specifiche.
main:
    # Inizializzazione
    #
    # Leggo un dato (v0=dato, v1=segno) e lo copio nei registri locali
    jal read_one
    move $s1,$v0
    move $s3,$v1
    # Inizializzo il contatore di permanenza del segno
    li $s2,1
infloop:
    # Ciclo principale
    #
    # Aspetto 88 microsecondi e leggo dall'ADC
    li $a0,88
    jal delay
    jal read_adc_data
    # Sposto la vecchia rilevazione in $s0 e la nuova in $s1
    move $s0,$s1
    move $s1,$v0
    # Controllo se il segno è cambiato...
    move $a0,$s0
    move $a1,$s1
    jal test_sign_change
    bne $v0,$zero,signchanged
signnotchanged:
```

```
    # ...non lo è, incremento il contatore corrente
    addiu $s2,$2,1
    j infloop
signchanged:
    # ...lo è, controllo se sono passato da valori negativi a
    # positivi o viceversa
    bne $s3,$zero,minustoplus
plustominus:
    # Da + a -: mi preparo ad aggiornare il display POS
    la $s4,POS
    li $s4,1
    j cleanup
minustoplus:
    # Da - a +: mi preparo ad aggiornare il display NEG
    la $s4,NEG
    li $s4,0
cleanup:
    # Aggiorno il display in questione
    sh $s2,0($s4)
    # Reinizializzo il contatore
    li $s2,1
    j infloop

# Subroutine delay
#
# Scopo: attendere un certo numero di microsecondi
# Parametri: $a0 -> numero di microsecondi (>1)
# Ruolo dei registri: $a0 -> contatore, $t0 -> non viene in realtà
# modificato, $ra come da specifiche
# Note implementative: l'implementazione si basa sull'assunto che ogni
# istruzione venga eseguita in 0.1 microsecondi. La procedura, in realtà,
# aspetta sempre 0.1 microsecondi in più rispetto a quanto dichiarato, in
# quanto è necessario (e non facilmente bilanciabile in termini temporali)
# eseguire l'istruzione jr per tornare al chiamante.
delay:
    # Aspetto 0.8 microsecondi
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    addi $t0,$t0,0
    # Decrementa a0 e...
    addiu $a0,$a0,-1
    bne $a0,$zero,delay
    # ...se è 0 ho finito, torno al chiamante
    jr $ra

# Subroutine read_adc_data
#
# Scopo: leggere un dato dal convertitore analogico-digitale
# Ritorna: $v0 -> il valore letto dall'ADC (nel formato dell'ADC: 11+1 bit)
# Note implementative: comprende il ritardo della lettura dovuta all'attesa
# dell'input da parte dell'ADC.
# Ruolo dei registri: $t0 -> dato da scrivere o leggere per le sb e lb,
# $t1 -> indirizzo, $a0, $v0, $sp ed $ra come da specifiche
# Note implementative: in quanto richiama un'altra funzione fa il
# salvataggio/ripristino di $ra sullo stack
read_adc_data:
    #Salva $ra nello stack
    addi $sp,$sp,-4
    sw $ra,0($sp)
    #Manda il fronte all'ADC
    la $t1,ADCR
    li $t0,0
```

```
    sb $t0,0($t1)
    li $t0,0x80
    sb $t0,0($t1)
    #Aspetta 10 ms
    li $a0,10
    jal delay
    la $t1,ADC
    #Torna il risultato
    lhu $v0,0($t1)
    #Ripristina $ra dallo stack
    lw $ra,0($sp)
    addi $sp,$sp,4
    jr $ra

# Subroutine fake_read_adc_data
#
# Scopo: simulare la lettura di un dato dal convertitore analogico-digitale
# leggendo un dato da un array predefinito
# Ritorna: $v0 -> il valore letto dall'ADC (nel formato a 11+1 bit dell'ADC)
# Ruolo dei registri: $t0 -> indirizzo dell'offset, $t1 -> offset,
# $t2 -> indirizzo del primo elemento dell'array, $t3 -> indirizzo
# dell'elemento da leggere, $v0 ed $ra come da specifiche
# Note implementative: questa è la routine utilizzata nei test. Semplicemente
# ritorna (uno alla volta) una serie di valori prefissati nel vettore
# TESTDATA. Noti i dati in ingresso rende quindi possibile un trace/debug
# accurato dal momento che è possibile sapere in anticipo i risultati.
fake_read_adc_data:
    # Metti in $t1 lo scostamento del dato da leggere dal primo dato in
    # TESTDATA (in byte)
    la $t0,CURRENTOFFSET
    lw $t1,0($t0)
    # Metti in $t2 l'indirizzo del primo elemento di TESTDATA e calcola
    # l'indirizzo del dato corrente
    la $t2,TESTDATA
    add $t3,$t2,$t1
    # Carica il dato. Se è zero ho finito, altrimenti torna al main
    lhu $v0,0($t3)
    beq $v0,$zero,end
    addi $t1,$t1,2
    sw $t1,0($t0)
    jr $ra
end:
   # Ho finito i dati (speriamo anche il debug), me ne sto qui
   j end

# Subroutine test_sign_change
#
# Scopo: testare se due dati dall'ADC hanno lo stesso segno
# Parametri: $a0 -> primo dato, $a1 -> secondo dato. Entrambi devono essere
# nel formato 11+1 bit dell'ADC
# Ritorna: $v0 -> 0 se i segni sono uguali, un numero diverso da 0 se sono
# diversi
# Ruolo dei registri: $a0 e $a1 -> temporanei per le operazioni aritmetiche
# intermedie, $v0 ed $ra come da specifiche
test_sign_change:
    # Isolo il bit 11 (di segno). Metto a 0 tutti gli altri.
    andi $a0,$a0,0x800
    andi $a1,$a1,0x800
    # Sottraggo: otterrò zero se e solo se i dati sono concordi
    subu $v0,$a0,$a1
    jr $ra

# Subroutine read_one
#
# Scopo: fare una lettura singola del dato dall'ADC e determinare il segno
# del dato letto
# Ritorna: $v0 -> il dato letto, $v1 -> il suo segno (0 se il dato è positivo
```

```
# o 1 se è negativo)
# Ruolo dei registri: come da specifiche
# Note implementative: comprende il ritardo della lettura dovuta all'attesa
# dell'input da parte dell'ADC. In quanto richiama un'altra funzione fa il
# salvataggio/ripristino di $ra sullo stack
read_one:
    # Salva $ra nello stack
    addi $sp,$sp,-4
    sw $ra,0($sp)
    # Leggo dall'ADC
    jal read_adc_data
    # Estraggo separatamente il segno
    andi $v1,$v0,0x800
    srl $v1,$v1,11
    # Ripristina $ra dallo stack
    lw $ra,0($sp)
    addi $sp,$sp,4
    jr $ra
```

Indice