UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

RELAZIONE ELABORATO ASM
ARCHITETTURA DEGLI ELABORATORI

Studenti:

MARZARI LUCA VR421483

PINTANI DEBORAH VR422805

Docente: SETTI FRANCESCO

ANNO ACCADEMICO 2017/2018

INDICE

1. Specifiche

Si ottimizzi un codice in linguaggio C che controlla un dispositivo per la gestione intelligente del consumo di energia elettrica all'interno di un sistema domotico mediante l'uso di Assembly inline. Il dispositivo riceve in ingresso lo stato acceso/spento di un numero finito di dispositivi di cui è noto il consumo istantaneo a priori, e fornisce in uscita la fascia di consumo ad ogni ciclo di clock. Qualora l'assorbimento istantaneo sia superiore al limite di 4.5kW per più di 5 cicli di clock consecutivi, il sistema deve disattivare l'interruttore generale. Al fine di prevenire questa situazione, il dispositivo può disattivare la lavatrice e la lavastoviglie (in questo ordine di priorità).

Inizialmente il sistema è sempre spento (INT_GEN=0) e gli interruttori della lavatrice (INT_WM) e della lavastoviglie (INT_DW) sono sempre non-armati (entrambi posti a 0).
Fintanto che INT_GEN=0, il sistema rimane spento ed il dispositivo deve restituire 0 per tutti i bit di output. Dal momento in cui RES_GEN assume il valore 1: INT_GEN, INT_DW e INT_WM commutano a 1, il sistema si accende ed il dispositivo inizia a leggere lo stato acceso/spento di tutti i carichi (LOAD).

Il dispositivo deve restituire in uscita la fascia di consumo istantaneo: F1<=1.5kW, 1.5kW<F2<=3kW, 3kW<F3<=4.5kW, OL>4.5kW. Nel caso in cui il sistema rimanga in overload (OL) per almeno 4 cicli di clock, il dispositivo commuta INT_DW a 0 e, fintanto che non viene riarmato (RES_DW=1), il carico relativo alla lavastoviglie deve essere ignorato; qualora ciò non sia sufficiente a uscire dallo stato OL, al ciclo di clock successivo (5° ciclo in OL) il dispositivo commuta INT_WM a 0 e, fintanto che non viene riarmato (RES_WM=1), il carico relativo alla lavatrice deve essere ignorato.
Nel caso in cui il sistema permanga in OL, al successivo ciclo di clock (6° ciclo in OL) il dispositivo commuta INT_GEN a 0 ed il sistema si spegne.

Il programma deve leggere il contenuto di *testin.txt* contenente in ogni riga i seguenti valori: *RESET-LOAD*

• RESET [3]: contiene la sequenza dei comandi di riarmo degli interruttori; nell'ordine RES_GEN, RES_DW e RES_WM (senza spazi).
• LOAD [10]: stato di accensione (1=ON, 0=OFF) dei carichi elettrici. Ogni carico ha un suo consumo istantaneo associato. Il carico complessivo del circuito è dato dalla somma di tutti i carichi accesi

Il programma deve restituire i risultati del calcolo in *testout.txt* in cui ogni riga contiene: *INT-TH*

• INT [3]: indica lo stato di attivazione (1=ON, 0=OFF) degli interruttori; in ordine INT_GEN, INT_DW e INT_WM (senza spazi).
• TH [2]: indica la fascia di consumo istantanea secondo la seguente codifica:
  o F1 <= 1.5kW
  o 1.5kW < F2 <= 3kW
  o 3kW < F3 <= 4.5kW
  o OL > 4.5kW

Nel caso in cui la macchina sia spenta, il controllore deve restituire la stringa "000-00".

2. Progettazione iniziale

In un primo momento per la realizzazione delle specifiche presentate nel capitolo precedente avevamo pensato di utilizzare una funzione esterna (*extern*). Tuttavia la gestione finale dello Stack risultava complessa e quindi abbiamo preferito utilizzare *ASM inline*.

Inizialmente avevamo pensato di utilizzare più variabili separate per raccogliere i bit di load, successivamente abbiamo ottimizzato il codice attraverso l'utilizzo di un array.

Il controller presenta quindi delle sezioni ben distinte fra loro:

- Ciclo esterno che itera sulla stringa di input finchè questa non presenta un '*\0*'
- Parte di raccolta dei RES nelle opportune variabili prima di trovare il trattino
- Parte di raccolta dei load in un array dopo aver trovato il trattino
- Inizio elaborazione vera e propria dove si esaminano i bit di RES
- Ciclo di conteggio del consumo totale
- Controllo e conteggio cicli di overload
- Sistemazione valore consumo totale in base allo spegnimento di DW e WM in OL
- Inserimento dei valori calcolati nell'array di output

3. Variabili utilizzate e il loro scopo

La dimensione della variabili e degli array è stata scelta concordemente al loro utilizzo.

Variabili libere:

- **int_gen**: byte, contiene il bit di *INT_GEN*
- **int_dw**: byte, contiene il bit di *INT_DW*
- **int_wm**: byte, contiene il bit di *INT_WM*
- **res_gen**: byte, contiene il bit di *RES_GEN*
- **res_dw**: byte, contiene il bit di *RES_DW*
- **res_wm**: byte, contiene il bit di *RES_WM*
- **flagTurnOn**: byte, indica se la macchina è accesa o meno, utilizzata per alcuni controlli
- **flagTurnOffDW**: byte, indica se in quel ciclo la DW è stata spenta
- **flagTurnOffWM**: byte, indica se in quel ciclo la WM è stata spenta
- **consumoTot**: long, contiene la somma dei Watt accumulati nel ciclo di clock
- **ctOL**: byte, contiene il conteggio dei cicli di overload
- **flagFineStringa**: byte, indica se ci si trova alla fine della stringa

Array:

- **consumiLoad[10]**: long, contiene i consumi per ogni elettrodomestico
- **bitLoad[10]**: byte, contiene i bit di load raccolti dall'input corrente

Registri fissi:

- **esi**: utilizzato per contenere l'array di bufferIn

- **edi**: utilizzato per memorizzare l'array di bufferOut_asm

Registri general purpose:

- **eax**: utilizzato per eseguire moltiplicazioni o come registro di supporto e nella sua variante più piccola
- **ebx**: utilizzato per tenere il conteggio del ciclo che raccoglie i bit di load
- **ecx**: utilizzato come registro di supporto
- **edx**: non utilizzato perché l'operazione di moltiplicazione lo azzerava

4. Modalità di passaggio/restituzione valori

Avendo utilizzato ASM inline, il passaggio di valori tra il controller in C e quello in ASM è stato minimo.
Infatti non ci sono veri e propri output, visto che passiamo in input i puntatori sia dell'array bufferIn che dell'array bufferOut_asm.

La sintassi utilizzata è stata:

```
:                                   /*output*/
: "S"(bufferin), "D"(bufferout_asm)  /*input*/
: "%eax", "%ebx", "%ecx", "%edx"     /*clobbered registers*/
```

Come si può vedere, il puntatore a bufferin è stato copiato nel registro esi, mentre il puntatore a bufferOut_asm in edi.
La riga destinata all'output non è stata utilizzata.

Per quanto riguarda la spiegazione sui clobbered register, si rimanda al capitolo 6.1.

5. Codice ad alto livello

La cartella inviata contiene 3 file in C che sono serviti come base sui cui costruire il programma in ASM:

- *elaborato in C:* contiene il codice ad alto livello dell'elaborazione
- *elaborazioneOutputASM:* contiene il codice ad alto livello che riguarda lo scorrimento e la scrittura su array di output
- *elaborazioneInputASM*: contiene il codice ad alto livello che riguarda lo scorrimento e il raccoglimento dei bit dalla stringa di input

Viene quindi riportato di seguito il codice riferito all'elaborazione di input e output.

Elaborazione input:

```c
#include <stdio.h>

#define LUNG 40

int main(void)
{
    int ct = 0;
    int ctLoad = 0;
    char a[LUNG] = {'0', '0', '1', '-', '0', '1', '0', '1', '0', '0', '0',
    int bitLoad[10] = {0};
    int res_gen = 0;
    int res_dw = 0;
    int res_wm = 0;

    while(a[ct] != '\0') //inizioWhileFineStringa
    {
        if(a[ct] == '-')
        {
            ct++;
            ctLoad = 0;

            while(a[ct] != '\n') //inizioWhileACapo
            {
                if(a[ct] == '0')
                    bitLoad[ctLoad] = 0;
                else    //elseIfTrattino
                    bitLoad[ctLoad] = 1;

                //incrementiWhileACapo
                ctLoad++;
                ct++;
            } //fineWhileACapo

            //elaborazione!

            printf("gen: %d, dw: %d, wm: %d\n", res_gen, res_dw, res_wm);
            printf("Load: ");

            for(int i = 0; i < 10; i++)
            {
                printf("%d ", bitLoad[i]);
            }

            printf("\n");

            ct++;
            ctLoad = 0;
        }
        else //inizioCalcoloRes
        {
            if(ctLoad == 0)
            {
                if(a[ctLoad] == '0')
                    res_gen = 0;
                else
                    res_gen = 1;
            }
            else if(ctLoad == 1) //elseDW
            {
                if(a[ctLoad] == '0')
                    res_dw = 0;
                else
                    res_dw = 1;
            }
            else if(ctLoad == 2) //elseWM
            {
                if(a[ctLoad] == '0')
                    res_wm = 0;
                else
                    res_wm = 1;
            }

            ctLoad++;
        }


        ct++;
    }

    return 0;   //fine
}
```

Elaborazione output:

```c
#include <stdio.h>

#define LUNG 10

int main(void)
{
    char int_gen = '1';
    char int_dw = '0';
    char int_wm = '1';
    int ct = 0;
    int flag = 0;

    int consumoTot = 250;

    char output[LUNG] = {0};

    output[ct] = int_gen;
    output[++ct] = int_dw;
    output[++ct] = int_wm;
    output[++ct] = '-';

    if(consumoTot == 0)
    {
        output[++ct] = '0';
        output[++ct] = '0';
    }
    if(consumoTot > 0 && consumoTot <= 150)
    {
        output[++ct] = 'F';
        output[++ct] = '1';
    }
    if(consumoTot > 150 && consumoTot <= 300)
    {
        output[++ct] = 'F';
        output[++ct] = '2';
    }
    if(consumoTot > 300 && consumoTot <= 450)
    {
        output[++ct] = 'F';
        output[++ct] = '3';
    }
    if(consumoTot > 450)
    {
        output[++ct] = 'O';
    }
    if(consumoTot > 450)
    {
        output[++ct] = 'O';
        output[++ct] = 'L';
    }

    output[++ct] = '\n';

    if(flag == 1)
        output[++ct] = '\0';

    for(int i = 0; i<LUNG; i++)
        printf("%c", output[i]);

    return 0;
}
```

6. Descrizione delle scelte progettuali

Il progetto è stato realizzato interamente usando Assembly Inline con la sintassi AT&T. All'interno del codice da noi scritto, sono state inizializzate delle variabili e degli array utilizzati solamente all'interno del codice ASM e pertanto il codice C originale non è stato modificato in nessun modo.

Durante la progettazione del codice abbiamo incluso il caso nel quale i file di test non abbiano il '\n' alla fine della stringa ma direttamente '\0'.
È stato sufficiente inserire una semplice "cmp" nell'operazione di raccoglimento di bit di load.

Al fine di migliorare l'efficienza e la pulizia del codice, abbiamo deciso di utilizzare degli array nella zona di conteggio dei consumi. Infatti risulta molto più semplice ciclare su array che contiene tutti i consumi degli elettrodomestici piuttosto che usare variabili singole.

Inoltre abbiamo usato l'operazione di 'xor' quando possibile per azzerare i registri e aumentare le prestazioni del codice.

6.1. Criticità e utilizzo del debugger

Nella stesura del codice ASM ci siamo spesso imbattuti in errori di 'segmentation fault' che abbiamo risolto nei seguenti modi.

- **Aggiunta dei clobbered registers**: sono i registri che il programmatore deve indicare di aver modificato nella parte di codice ASM, così che il compilatore riesca a ripristinarli correttamente una volta arrivati alla parte in C.
  All'inizio non li avevamo considerati perché sapevamo che il compilatore li inseriva automaticamente, tuttavia è stato necessario scriverli manualmente.

- **Scorrimento di array contenenti variabili "long"**: è stato necessario utilizzare una sintassi particolare per poter scorrere correttamente un array contenente variabili "long" (vedi consumiLoad).

  Esempio:
  ```
  mulb consumiLoad(, %ecx, 4)
  ```

  In questo esempio, si omette il primo argomento delle parentesi perché indicato dal nome dell'array, mentre ecx è moltiplicato per il fattore 4 (che indica i byte presenti in un "long"). In questo modo si riesce agevolmente a scorrere l'array.

- **Utilizzo del debugger (GDB)**: trovare il modo per riuscire a debuggare un programma scritto in ASM inline non è stato per nulla semplice.
  Alla fine siamo giunti ad una soluzione di seguito riportata:

La compilazione è divisa in 3 fasi:

1. Generazione listato assembly dal sorgente C
2. Compilazione del sorgente assembly in file oggetto
3. Linking del file oggetto e generazione dell'eseguibile

Di seguito il significato dei vari flag:

- -Wall -Wextra -Wpedantic: abilita molti warning sul codice
- -O0: disabilita tutte le ottimizzazioni del compilatore
- -fPIC: *Position Independent Code*, il codice può essere caricato ovunque in memoria
- -ggdb3: abilita le informazioni di debug al livello 3 (massimo, GDB)
- -gstabs+: genera diverse informazioni di debug per assembly (solo GCC)
- -m32: compilazione forzata a 32 bit
- -masm=att: forza sintassi generata AT&T
- -S: genera sorgente assembly in controller.s
- -c: compilazione
- -o: file oggetto di destinazione

Possiamo concatenare i vari step in una sola linea:

gcc -Wall -Wextra -Wpedantic -O0 -fPIC -ggdb3 -gstabs+ -m32 -masm=att -S controller.c && gcc -c -m32 -ggdb3 -gstabs+ controller.s -o controller.o && gcc -m32 -ggdb3 -gstabs+ controller.o -o controller

Ora è possibile avviare GDB e inserendo il comando "*layout asm*" è possibile vedere a schemo le istruzioni ASM di tutto il programma (anche di quello inline).
Per scorrere una a una le istruzioni si utilizza il comando "*ni*".

- **Visualizzazione errata delle variabili "byte"**: utilizzando il debugger e stampando a schermo specifici indirizzi di memoria riferiti a variabili "byte" il valore visualizzato è errato, in quanto il debugger stampa l'intero "long" e non solo la porzione "byte".

Il nostro codice testato con il file "*testin.txt*" genera esattamente gli stessi outputs contenuti in "*trueout.txt*".