PCS2031
Laboratório de Microprocessadores

Código C + Código Assembly

Felipe de Castro Santana
Felipe Zequeto Bozoni
Marylia Gutierrez

Objetivo

Nesta experiência faremos a junção de um código assembly com um código em C. Para isso utilizaremos dois arquivos:

segment.c

```
#include <stdlib.h>
#include <stdio.h>
#include "segment.h"

// Mapeamento do microcontrolador

#define SYSCFG      0x03ff0000
#define IOPMOD          ((volatile unsigned *)(SYSCFG+0x5000))
#define IOPDATA         ((volatile unsigned *)(SYSCFG+0x5008))

static unsigned int numeric_display [16] =
{
        DISP_0, DISP_1, DISP_2, DISP_3, DISP_4, DISP_5, DISP_6, DISP_7, DISP_8, DISP_9,
        DISP_A, DISP_B,  DISP_C,        DISP_D, DISP_E, DISP_F
};

int main(void) {
        // inicalização

        *IOPMOD         |= SEG_MASK;
        *IOPDATA        |= SEG_MASK;

        unsigned numero = 0x6;

        if ( numero >= 0 & numero <= 0xf )  {
        *IOPDATA        &= ~SEG_MASK;
        *IOPDATA        |= (unsigned) numeric_display[numero];
        }
        return 0;
}
```

segment.h

```
/*****************************************************************
 *
 * ARM Strategic Support Group
 *
 *****************************************************************/

/*****************************************************************
 *
 *      Module          : segment.h
 *      Description     : Segment Display header File
 *
 *  Tool Chain : ARM Developer Suite 1.0
 *      Platform        : Evaluator7T
 *      Status          : Complete
 *      History         :
 *
 *                              2000-04-04 Andrew N. Sloss
 *                              - implemented
 *
 *      Notes           :
 *
 *                              This program never ends. To terminate the user
 *                              has to break in with the debugger or reset the
 *                              board.
 *
 *****************************************************************/

/*****************************************************************
 * IMPORT
```

```
 ***************************************************************/

// none....

/*****************************************************************
 * MACROS
 *****************************************************************/

/* The bits taken up by the display in IODATA register */

#define SEG_MASK       (0x1fc00)

/* define segments in terms of IO lines */

#define SEG_A          (1<<10)
#define SEG_B          (1<<11)
#define SEG_C          (1<<12)
#define SEG_D          (1<<13)
#define SEG_E          (1<<14)
#define SEG_F          (1<<16)
#define SEG_G          (1<<15)

#define DISP_0         (SEG_A|SEG_B|SEG_C|SEG_D|SEG_E|SEG_F)
#define DISP_1         (SEG_B|SEG_C)
#define DISP_2         (SEG_A|SEG_B|SEG_D|SEG_E|SEG_G)
#define DISP_3         (SEG_A|SEG_B|SEG_C|SEG_D|SEG_G)
#define DISP_4         (SEG_B|SEG_C|SEG_F|SEG_G)
#define DISP_5         (SEG_A|SEG_C|SEG_D|SEG_F|SEG_G)
#define DISP_6         (SEG_A|SEG_C|SEG_D|SEG_E|SEG_F|SEG_G)
#define DISP_7         (SEG_A|SEG_B|SEG_C)
#define DISP_8         (SEG_A|SEG_B|SEG_C|SEG_D|SEG_E|SEG_F|SEG_G)
#define DISP_9         (SEG_A|SEG_B|SEG_C|SEG_D|SEG_F|SEG_G)

#define DISP_A         (SEG_A|SEG_B|SEG_C|SEG_E|SEG_F|SEG_G)
#define DISP_B         (SEG_C|SEG_D|SEG_E|SEG_F|SEG_G)
#define DISP_C         (SEG_A|SEG_D|SEG_E|SEG_F)
#define DISP_D         (SEG_B|SEG_C|SEG_D|SEG_E|SEG_G)
#define DISP_E         (SEG_A|SEG_D|SEG_E|SEG_F|SEG_G)
#define DISP_F         (SEG_A|SEG_E|SEG_F|SEG_G)

/*****************************************************************
 * DATATYPES
 *****************************************************************/

// none...

/*****************************************************************
 * ROUTINES
 *****************************************************************/

// none...
/*****************************************************************
 * END OF SEGMENT.H
 *****************************************************************/
```

Passo 1

Com os arquivos segment.c e segment.h, podemos gerar o segment.o (`arm-elf-gcc segment.c –o segment.o`).

Um arquivo .o é um código objeto, próximo da linguagem de máquina, ele é resultante da compilação do código fonte. Para cada arquivo de código fonte é gerado um arquivo com código objeto, que posteriormente é 'ligado' aos outros, através de um linker, resultando num arquivo executável. Um código objeto ainda não possui as labels definidas.

Agora com o segment.o podemos executá-lo no kit, lembrando que é necessário criar o link antes (`sudo ln ttyUSB0 ttyS0`). Com o kit aberto resetar a placa e executar os comandos:

```
Target rdi /dev/ttyS0
load
c
```

Com isso o display da placa exibirá o valor 6

Passo 2

Vamos criar a função imprime em segment.c, para isso basta retirar do main a parte pertinente, colocar a função antes do main, que irá receber o valor passado no main e imprimir no display. Com o arquivo segment.c alterado, mais uma vez gerar o arquivo segment.o, abrir no kit e observar no display da placa mais uma vez o número 6, se desejar altere o valor passado no main e verificar que este novo valor também deverá ser exibido.

Passo 3

Neste passo iremos separar a função imprime, para isto criaremos o arquivo imprime.c que irá conter a função gerada no passo anterior. O arquivo segment.c permanecerá praticamente igual, apenas será apagada a função criada nele e as atribuições de numeric_display. No arquivo imprime.c teremos a nova função, as atribuições de numeric_display, SYSCFG, IOPMOD e IOPDATA, além dos includes necessários (não esquecer de incluir segment.h). Com os arquivos modificados criaremos o segment.o, mas para isso o comando deverá ser: `arm-elf-gcc -g imprime.c segment.c -o segment.o` , em seguida faremos como nos passos anteriores, abrindo no kit, carregando o programa na placa e verificando o valor exibido no display.

Passo 4

Com o arquivo imprime.c funcionando, geraremos o seu objeto imprime.s (`arm-elf-gcc -S imprime.c`) e em seguida testaremos mais uma vez o programa em segment.c, mas desta vez usando o imprime.s no lugar de imprime.c, para isso deverá ser dado o comando:

```
arm-elf-gcc -g imprime.s segment.c -o segment.o
```

Note que este último comando dado já cria os objetos e dispara o linker, atribuindo os labels, uma outra forma de fazer é a partir do imprime.s gerar imprime.o (`arm-elf-as imprime.s -o imprime.o`) e a partir de segment.c gerar o segment.o (`arm-elf-gcc segment.c -o segment.o`) e com os objetos em mão criar o link entre eles através de `arm-elf-ld`.

Passo 5

Agora alteraremos o arquivo imprime.s para que o programa imprima na tela ao invés de imprimir no display. Entender o código do imprime.s pode ser difícil, o

que complica para descobrir onde devemos fazer a modificação. Uma forma de entender melhor é modificar o arquivo imprime.c retirando a parte que realmente faz a impressão, ou seja, a parte do 'if', com isso compare os arquivos imprime.s com e sem o 'if' e desta forma ficará mais fácil perceber em que parte deverá ser feita a alteração. A seguir podemos ver o código imprime.s que faz a impressão do número no display.

```
  .file    "imprime.c"
    .data
    .align    2
    .type    numeric_display, %object
    .size    numeric_display, 64
numeric_display:
    .word    97280
    .word    6144
    .word    60416
    .word    48128
    .word    104448
    .word    111616
    .word    128000
    .word    7168
    .word    130048
    .word    113664
    .word    121856
    .word    126976
    .word    91136
    .word    63488
    .word    123904
    .word    115712
    .text
    .align    2
    .global    imprime
    .type    imprime, %function
imprime:
    @ args = 0, pretend = 0, frame = 4
    @ frame_needed = 1, uses_anonymous_args = 0
    mov    ip, sp
    stmfd    sp!, {fp, ip, lr, pc}
    sub    fp, ip, #4
    sub    sp, sp, #4
    str    r0, [fp, #-16]
    ldr    r3, [fp, #-16]
    cmp    r3, #15
    bhi    .L1
    mov    r2, #66846720
    add    r2, r2, #217088
    add    r2, r2, #8
    mov    r3, #66846720
    add    r3, r3, #217088
    add    r3, r3, #8
    ldr    r3, [r3, #0]
    bic    r3, r3, #130048
    str    r3, [r2, #0]
    mov    r2, #66846720
    add    r2, r2, #217088
    add    r2, r2, #8
    mov    r3, #66846720
    add    r3, r3, #217088
    add    r3, r3, #8
    ldr    ip, .L3
    ldr    r1, [fp, #-16]
    ldr    r0, [r3, #0]
    ldr    r3, [ip, r1, asl #2]
    orr    r3, r0, r3
    str    r3, [r2, #0]
```

```
.L1:
    ldmfd    sp, {r3, fp, sp, pc}
.L4:
    .align    2
.L3:
    .word    numeric_display
    .size    imprime, .-imprime
    .ident    "GCC: (GNU) 3.4.3"
```

Passo 6

Agora faremos a junção do código em C com o código em assembly. No arquivo imprime.s já descobrimos qual a parte que faz a impressão do número, então colocaremos esta parte dentro do código imprime.c.
Para colocar um código assembly em um código em C, basta colocá-lo dentro de __asm__ ( ); onde cada linha deverá estar entre aspas e com /n/t no fim, por exemplo:

```
void imprime(unsigned numero){
    __asm__(
        "ldr    r3, [fp, #-16]\n\t"
        "cmp    r3, #15\n\t"
        "bhi    .L1\n\t"
         ...
    );
```

Feita esta alteração poderemos testar da mesma forma que fizemos nos outros passos.

Passo 7

Neste passo voltaremos o código imprime.c para antes, onde não era usado assembly e então faremos com que essa função imprima de 1 a 7 recursivamente (Dica: use delay entre cada impressão para conseguir observar melhor). Em seguida, geraremos novamente o imprime.s para ser analisado. Analisando o código imprime.s estude como o frame pointer é usado para marcar uma posição na pilha empilhando parâmetros e variáveis locais. (Dica: estude o fp e o ip, um deles é ponteiro para variáveis locais e outro para variáveis globais).