EA871 – LAB. DE PROGRAMAÇÃO BÁSICA DE SISTEMAS DIGITAIS
EXPERIMENTO 3 – Linguagem de Montagem (*Assembly)*

Profa. Wu Shin-Ting

OBJETIVO: Apresentação do modelo de programação do Kinetis KL25Z128 e o seu repertório de instruções em linguagem de montagem Thumb-16

ASSUNTOS: Arquitetura de conjunto de instruções (ISA); instruções Thumb-16; ciclos de instrução; inclusão no código C; montador GAS.

O que você deve ser capaz ao final deste experimento?

Ter uma noção dos conceitos de arquitetura envolvidos na caracterização de um sistema computacional.

Saber como as instruções e os dados de um programa são organizados na memória e manipulados por um processador.

Ter uma noção do repertório de instruções ARM Thumb e as diretivas do montador GAS.

Ter uma noção dos conceitos envolvidos com a segmentação e o processamento paralelo das instruções.

Saber estimar a complexidade temporal de um código em *assembly* para uma arquitetura RISC.

Saber integrar os códigos em linguagem de montagem nos códigos em linguagem C.

INTRODUÇÃO

Neste experimento vamos introduzir o modelo de programação do núcleo Cortex-M0+ utilizando os mnemônicos dos seus códigos de máquina, ou seja, a sua linguagem de montagem (*assembly*) (Seção A6.7 em [1][2]).

Sendo o núcleo Cortex-M0+ projetado com base na **arquitetura de von Neumann**, os dados e as instruções dos programas compartilham o mesmo espaço de endereços de memória e os mesmos barramentos de conexão com o núcleo. Sendo de **arquitetura de E/S mapeada na memória**, são as mesmas as instruções de acessos aos periféricos e à memória. Sendo **um processador RISC**, o núcleo apresenta um repertório de instruções bem menor e bem mais eficiente que o de um processador CISC. Por outro lado, ele requer uma quantidade maior de instruções para executar uma mesma tarefa, demandando um espaço maior de memória. Como memória é um quesito crítico para os microcontroladores, foi proposto o repertório de instruções *Thumb* de 16 *bits* como uma alternativa para as instruções de 32 *bits* da **arquitetura ARM**. Porém, algumas instruções da arquitetura ARM não podem ser codificadas com 16 *bits*. Foram então adicionadas ao repertório *Thumb* algumas instruções de 32 *bits* (Thumb-2). O núcleo Cortex M0+ suporta o repertório de instruções Thumb e algumas instruções codificadas em 32 *bits* [1]. Há processadores ARM que podem ser chaveados entre os dois estados, ARM (32 *bits*) e Thumb (16 *bits*), usando o *bit* 0 do endereço (0, estado ARM; e 1, estado Thumb). Porém, o núcleo Cortex-M0+ só opera no estado Thumb.

A **arquitetura** de conjunto de instruções (ISA) do Cortex-M0+ é ***load-store***, ou seja, as instruções são divididas em duas classes: as que fazem acessos à memória (*load* e *store*), e as que só operam

sobre os registradores, como todas as instruções lógico-aritméticas. Adicionalmente, um *pipeline* de 2 estágios é implementado no núcleo. Essas otimizações permitiram que a grande maioria das instruções seja executada em 1 ciclo de relógio ($t = (10^{-6}/20.97512)$ s)[3], sem falar que a homogeneidade no tamanho e no ciclo de instruções facilita, por parte de um desenvolvedor, a estimativa do espaço e do tempo demandado por um segmento de instruções.

Mesmo sendo considerada uma linguagem de baixo nível, um processador não consegue gerar sinais de controle a partir das instruções codificadas em mnemônico. Estes mnemônicos precisam ser traduzidos para os códigos binários da máquina. A ferramenta que faz essa tradução é o **montador** (*assembler*) e o processo de tradução é conhecido como **montagem** (*assembling*). Da mesma forma que existe uma grande variedade de linguagens *assembly*, há diversos montadores. Neste curso, usaremos o montador ***GNU Assembler*** (GAS) disponível no ambiente *IDE CodeWarrion* [4][5]. Vale ressaltar que, embora o repertório de instruções, que são traduzidas para códigos binários, dependa do processador-alvo, as diretivas que orientam o montador na tradução são as mesmas para todos os processadores.

Escrever códigos em *assembly* é uma tarefa árdua, propensa a erros. Porém, é ainda a única forma de acesso direto às funções implementadas no processador e ainda não disponiveis em linguagens de alto nível. Para amenizar o árduo trabalho de codificação em *assembly*, o GNU disponibiliza uma função em C, **asm**, através da qual pode-se inserir trechos de códigos em *assembly* num programa C e assegura a interoperabilidade entre os dois conjuntos de instruções. Como o compilador C não consegue interpretar as instruções em *assembly*, é necessário declarar junto com as instruções em assembly, a sua interface com o restante do programa em C (variáveis de entrada e de saída), os recursos que são utilizados pelas instruções (*clobbers*) e os endereços de desvios presentes nas instruções.

EXPERIMENTO

1. Arquitetura ARM-Thumb

a) Há duas instruções de deslocamento (S) linear para direita (R) no repertório de instruções ARM thumb: ASR e LSR. Carregue num registrador <Rm> o valor 0xFFFF_FFE3 e desloque os seus *bits* de 4 posições para direita, usando o modo de endereçamento imediato de ASR e LSR. Com base nos resultados obtidos, explique a diferença entre as duas instruções.

b) O repertório de instruções contém uma instrução de deslocamento rotativo/circular para direita, ROR, como mostra no projeto asm_ula. Explique o efeito desta instrução sobre o conteúdo do registrador.

c) Um processador ARM pode operar em dois estados, o estado ARM e o estado Thumb. No estado ARM as instruções são de 32 *bits* (4 *bytes*) e no estado Thumb, 16 *bits* (2 *bytes*). Portanto, os endereços das instruções são sempre pares. Os projetistas de repertório de instruções tem explorado este fato para aumentar a faixa de deslocamentos relativos ao endereço armazenado no contador de programa (PC) nas instruções de desvio. Cite duas instruções de desvio que fazem uso desta estratégia.

2. Ligação

a) Qual(is) é(são) o(s) espaço(s) de endereços ocupado(s) pelas instruções e pelos dados do código executável do projeto asm_ula? Compare-os com os endereços dos segmentos de instruções e de

dados especificados no arquivo de configuração MKL25Z128_flash.ld gerado automaticamente pelo IDE.
b) Construa o código executável, transfira-o para o microcontrolador no modo *Debug*. Anote o valor de SP (endereço 0x0000_0000) e o valor inicial de PC (0x0000_0004) registrados na aba **Memory.** Explique o que está codificado no PC. (Dica: Dê um "Reset" (ícone na barra de ferramentas da vista *Debug*: seta envolvida por uma circunferência vermelha) e verifique se o resultado da ação condiz com o conteúdo codificado no endereço 0x0000_0004.)

3) Programação
a) Não há um operador dedicado para deslocamento rotativo/cirsular dos *bits* binários de um operando em C. Em [6] é apresentada uma implementação de deslocamento rotativo/circular dos *bits* de um número usando os operadores OR lógico (|) e deslocamento linear para direita (>>) e deslocamento linear para esquerda (<<):

```
/*Function to right rotate n by d bits*/
#define INT_BITS 32
int rightRotate(int n, unsigned int d)
{
    /* In n>>d, first d bits are 0.
    To put last 3 bits of at
    first, do bitwise or of n>>d
    with n <<(INT_BITS - d) */
    return (n >> d)|(n << (INT_BITS - d));
}
```

Utilize **Disassembler** do CodeWarrior (Perspectiva C/C++ > apertar o botão direito sobre o código-fonte > selecionar "Disassembler" no menu *pop-up*) para traduzir o trecho de código acima em *assembly* Thumb-16. Quantos recursos (quantidade de registradores e quantidade de posições de memória ocupadas pelas instruções) são usados? Quantos ciclos de relógio são necessários para execução da função implementada? Justifique.

b) No projeto asm_ula é usada a instrução ROR disponível no repertŕoio de instruções Thum-16. Quantos recursos (quantidade de registradores e quantidade de posições de memória ocupadas pelas instruções) são usados? Quantos ciclos de relógio são necessários para execução da função implementada? Substitua as instruções em C pelas instruções em *assembly* em que se aplica a função nativa ROR suportada pelo processador Cortex-M0+. Teste a sua função rightRotate implementando um projeto asm_ROR com uso da seguinte função main:

```
int main()
{
    int n = 16;
    int d = 2;
    rightRotate(n, d);
}
```

Dica: Siga o modelo de programação do projeto asm_inline.
c) Estime o tempo de execução das instruções correspondentes ao trecho de instruções correspondentes ao rótulo **delay** (de push até pop inclusive) no projeto asm_FRDMKL25.

Implemente um novo projeto **asm_FRDMKL25_2us** em que as piscadas do *led* azul acontecem na frequência de 2Hz e o atraso gerado pelo **delay** seja aproximadamente 2us. Certifique a frequência das piscadas que você programou com o canal 3 do analisador lógico instalado nas máquinas LE30-5 ou LE30-6. Capture a forma de onda gerada pelo analisador. Em Anexo A são mostradas as fotos como os 4 canais, 0, 1, 2 e 3, são conectados, respectivamente, aos pinos PTE20, PTE21 (*led* vermelho), PTE22 (*led* verde) e PTE23 (*led* azul) do microcontrolador KL25Z de um lado e a uma porta USB do *desktop* no outro lado.

RELATÓRIO

O relatório deve ser devidamente identificado, contendo a identificação do instituto e da disciplina, o experimento realizado, o nome e RA do aluno. Para este experimento, responda os itens 1-3 do roteiro e suba-o, acompanhado dos projetos asm_ROR e asm_FRDMKL25_2us exportados no ambiente IDE CodeWarrior, no sistema *Moodle*.

REFERÊNCIAS

[1] ARM. *ARMv6-M Architecture Reference Manual.*

ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/ARMv6-M.pdf

[2] Wu Shin-Ting. Linguagem de Montagem

ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/apostila_C/LinguagemMontagem.pdf

[3] Cortex-M0+ Technical Reference Manual

ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/Cortex-M0+.pdf

[4] The GNU Assembler

http://tigcc.ticalc.org/doc/gnuasm.html

[5] Using as

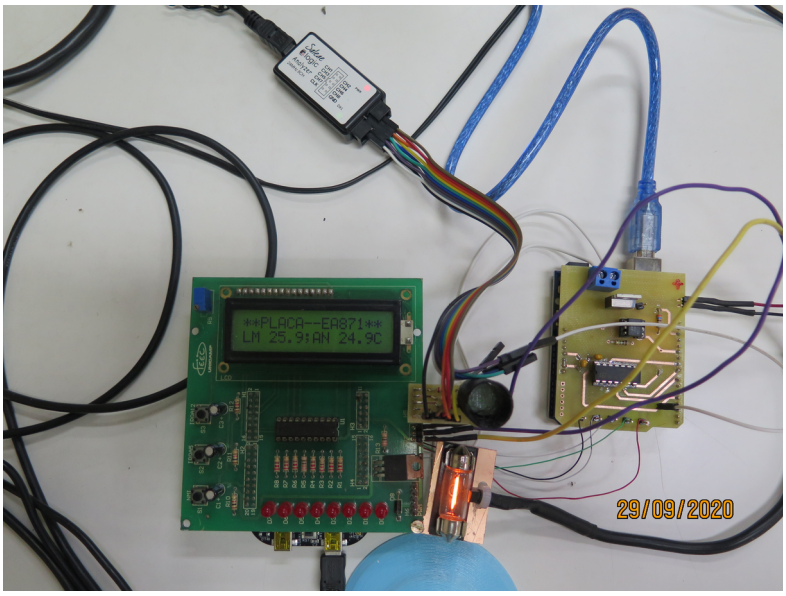
http://www.sourceware.org/binutils/docs-2.12/as.info/

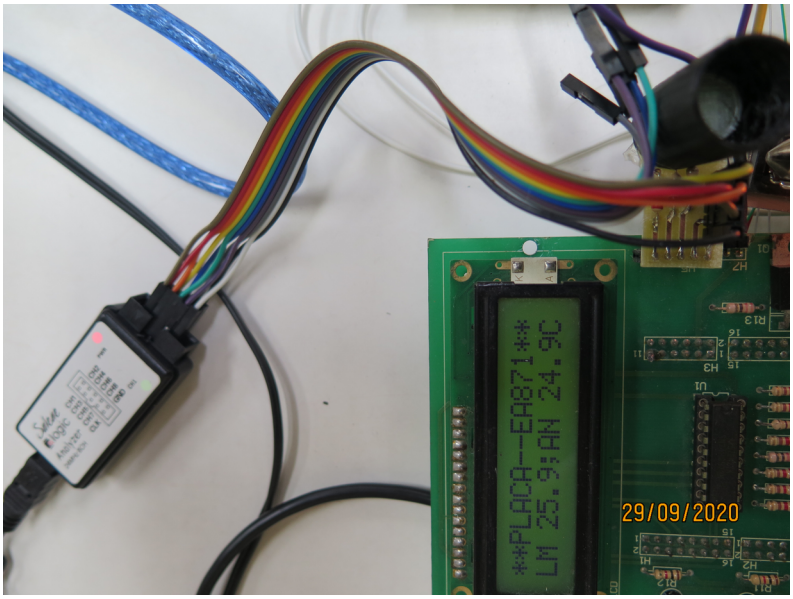
[6] Rotate bits of a number

https://www.geeksforgeeks.org/rotate-bits-of-an-integer/

Agosto de 2016.
Revisado em Fevereiro de 2017.
Revisado em Julho de 2017.
Revisado em Fevereiro de 2018.
Revisado em Setembro de 2020.

ANEXO A


PLACA--EA871
LM 25.9;AN 24.9C
29/09/2020


PLACA--EA871
LM 25.9;AN 24.9C
29/09/2020


29/09/2020