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; montador GAS; inclusão no código C; processamento de chamadas.

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 ARM *Thumb*.

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

Entender em termos de códigos de máquina o processamento de uma chamada de função.

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]).

O roteiro 2 [12] mostra que o núcleo Cortex-M0+ é projetado com base na **arquitetura de von Neumann**, isto é, os dados e as instruções dos programas compartilham o mesmo espaço de endereços de memória e os mesmos barramentos de conexão com o núcleo. A sua **arquitetura de E/S** é **mapeada na memória**, ou seja, são as mesmas as instruções de acessos aos periféricos e à memória. O espaço de endereçamento de $2^{32}=2^2\ 2^{10}\ 2^{10}\ 2^{10}$= 4 Gb é segmentado em múltiplos blocos onde são mapeadas periféricos e memórias de diferentes tecnologias (Tabela 4-1/página 105 em [7]). Todos os módulos do sistema são conectados segundo a especificação da arquitetura de barramento de microcontroladores avançada (*Advanced Microcontroller Bus Architecture*, AMBA) usando o protocolo de barramento de alto desempenho avançado (*Advanced High Performance Bus*,) para comunicação de alto desempenho em 8, 16 e 32 *bits* (Capítulo 21/página 331 em [7]), e o protocolo de barramento de periféricos avançado (*Advanced Peripheral Bus*) para comunicação com os periféricos (Capítulo 21/página 335 em [7]).

Sendo **um processador RISC**, o seu repertório de instruções é bem menor e bem mais eficiente que o de um processador CISC. A sua arquitetura do conjunto de instruções (*Instruction Set Architecture*, ISA), de 32 *bits*, é **ARM**, Com o intuito de otimizar o uso da memória, foi proposto o repertório de instruções *Thumb* de 16 *bits* (estado *Thumb*) como uma alternativa para as instruções de 32 *bits* da **arquitetura ARM** (estado ARM). Porém, o processasdor Cortex-M0+ só opera no estado *Thumb* com 8 registradores de trabalho R0—R7 de 32 *bits*, além dos registradores de funções específicas, também de 32 *bits* – contador de programa (*program counter*, PC, cujo código binário de identificação é 0b1111), registrador de Link (*link register*, LR), ponteiro de pilha (*stack pointer*, SP) e registradores de estado (*program status register*, PSR) (Seção A2.3/página 35 em [1]). Ele tem poucas instruções de 32 *bits*, reservadas para desvio às rotinas (Seção A6.7.13/página 123 em [1]), barreiras de sincronismo (Seção A6.7.21/página 133, A6.7.22/página 134, A6.7.24/página 136 em [1]), transferência de dados de registradores de funções específicas (Seção A6.7.42, A6.7.43/página 158 em [1]).

Formato de Instruções Thumb

Um programa em linguagem de montagem (*assembly*) é um arquivo texto, com uma instrução por linha. Cada linha pode ter até três campos: um rótulo, uma instrução, e um comentário.

O campo de instrução Thumb tem dois subcampos codificados em 16 *bits* (Seção A5.2/página 84 em [1]): código de operação representado por mnemônicos e operandos. Tipicamente, dois endereços são envolvidos numa operação, um endereço é o destino e o outro, a fonte. Por concisão, quando há dois operandos numa operação, como a soma, é comum omitir o endereço de um dos operandos, assumindo que ele seja o mesmo do destino. Por exemplo, para a instrução

adds r3, #1

um dos operandos da soma é 1 e o outro está armazenado em R3 que recebe o resultado da soma [R3] + 1, isto é, R3 := [R3] + 1 (Seção A6.7.2/página 107 em [1]). São reservados 3 *bits* para identificar os registradores de trabalho R0--R7. Porém, em algumas instruções, como MOV (Seção A6.7.40/página 155 em [1]), usa-se os códigos binários 0b1101 e 0b1111 para designar, respectivamente, o ponteiro de pilha SP e o contador de programa PC. O *bit* mais significativo é armazenado num *bit* separado e concatenado automaticamente na fase de decodificação da instrução.

Todos os rótulos devem ser seguidos de “:”, enquanto o(s) caracteres que deve(m) preceder os comentários são dependentes do processador. Usualmente é “;”. Porém, para arquitetura i386 e x86_64, ele é “#” e para ARM é “@”. No ambiente IDE CodeWarrior, podemos ainda utilizar a sintaxe de comentários de C /* */. Por exemplo, são equivalentes as seguintes linhas de instrução:

Rótulo	*Código de Operação*	*Operandos*	*Comentários*
INC:	adds	r3,#1	@ R3 := [R3] + 1
INC:	adds	r3,#1	# R3 := [R3] + 1
INC:	adds	r3,#1	/* *R3 := [R3] + 1* */

Para aumentar a faixa de unidades de memória endereçáveis a partir de uma quantidade limitada de *bits*, são explorados dois fatos

- os endereços das instruções de 32 *bits* são sempre múltiplos de 4 com os dois *bits* menos significativos em 0 - esses dois *bits* são omitidos na codificação do deslocamento em instruções que envolvem modo de endereçamento relativo a SP (Seção A6.7.4/página 111, Seção A6.7.67/página 188 em [1]) ou a PC (Seção A6.7.6/página 115, Seção A6.7.27/página 141 em [1]) ou a um registrador-base (Seção A6.7.26/página 139, Seção A6.7.59/página 177 em [1]),
- os endereços das instruções *Thumb* de 16 *bits* são sempre múltiplos de 2 com o *bit* menos significativo em 0 – esse *bit* é omitido na codificação do deslocamento em instruções que envolvem desvios do fluxo de controle no estado *Thumb* (Seção A6.7.10/página 119, Seção A6.7.31/página 146, Seção A6.7.63/página 182 em [1]).

As poucas instruções de 32 *bits* são acessadas numa mesma transferência via o barramento de 32 *bits. S*eus dados são automaticamente extraídos e concatenados (indicado por ":" nas descrições detalhadas das instruções em [1]) para reconstruir um dado de tamanho maior do que uma instrução de 16 *bits* conseguiria conter. Ainda, todas as instruções (Seção A6.7.49/página 165 em [1]) que modifiquem o conteúdo do LR automaticamente alteram o *bit* menos significativo desse registrador para diferenciar o tamanho das instruções a serem processadas: *bit* em 0 (ARM, 32 *bits*) e *bit* em 1 (Thumb, 16 *bits*).

Arquitetura *Load-Store*

A **arquitetura** ARM é uma arquitetura ***load-store***, ou seja, as instruções são divididas em duas classes: (1) as que não envolvem a memória na sua execuçãoo, como todas as operações lógico-aritméticas, e (2) as mais complexas e custosas que fazem acessos à memória (*load* e *store* entre memória e registradores). Esses acessos podem ser em *byte* (8 *bits*, `b`), *halfword* (16 *bits*, `h`) e *word* (32 *bits*), com (`s`) ou sem sinal (Seção A2.2/página 31 em [1]), através de instruções dedicadas de leitura (`ldr`) e escrita (`str`): `ldrh` (Seção A6.7.31/página 146, Seção A6.7.32/página 147 em [1]), `strh` (Seção A6.7.63/página 182, Seção A6.7.64/página 183 em [1]), `ldrsh` (Seção A6.7.34/página 149 em [1]), `ldrb` (Seção A6.7.29/página 144, Seção A6.7.30/página 145 em [1]), `strb` (Seção A6.7.61/página 180, Seção A6.7.62/página 181 em [1]), `ldrsb` (Seção A6.7.33/página 148 em [1]). Essas instruções estendem automaticamente o tamanho dos dados de 8 *bits* e 16 *bits* ao tamanho de 32 *bits* dos registradores, levando em conta o *bit* de sinal, antes de carregá-los nos registradores. Para truncar os *bits* mais significativos dos 32 *bits* dos registradores de trabalho em 16 ou 8 *bits* antes de armazená-los na memória usando `strh` e `strb`, aplica-se as instruções `uxth` (Seção A6.7.74/página 196 em [1]) e `uxtb` (Seção A6.7.73/página 195 em [1]), respectivamente.

Modos de Endereçamento

Para a classe de instruções que não envolvem a memória no seu processamento, os **modos de endereçamento** dos operandos são **modo imediato** (operandos codificados em instruções) ou **modo registrador** (operandos armazenados num dos 8 registradores de trabalho). Para a classe de instruções que envolvem acessos à memória, o segundo operando contém o endereço da memória a ser acessado. Este pode ser especificado pelo **modo de endereçamento relativo a PC** (endereço efetivo é a soma do conteúdo de PC e o deslocamento codificado na instrução), **relativo a SP** (endereço efetivo é a soma do conteúdo de SP e o deslocamento codificado na instrução), **por registrador-base** (endereço efetivo é a soma do conteúdo do registrador-base e o deslocamento codificado na instrução) ou **indexado (**endereço efetivo é a soma dos conteúdos do registrador-base e do registrador de índice). Vale lembra que o conteúdo do PC é sempre múltiplo de 4 e é atualizado para o próximo endereço assim que concluir a fase de busca de uma instrução.

Pipeline de 2 Estágios

Por desempenho, é implementado no núcleo Cortex-M0+ um *pipeline* de 2 estágios, em que um ciclo de instrução é dividido em 2 estágios paralelizáveis, como ilustra a Figura 1. Um estágio corresponde à fase de busca e à fase de pré-decodificação/classificação da instrução acessada, e outro estágio, à fase de decodificação propriamente dita e a execução. Enquanto não houver uma instrução de desvio ou de acessos à memória, a paralelização dos dois estágios faz com que o tempo médio de um ciclo de instrução seja 1 ciclo de relógio ($t = (10^{-6}/20.97512)$ s) ao invés de 2. Além disso, operando no estado *Thumb*, o núcleo Cortex-M0+ acessa em cada busca duas instruções por um barramento de 32 *bits*.

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 bloco de instruções. Na Seção 3.3/página 26 em [2], encontra-se uma lista de ciclos de instrução de todas as instruções do núcleo Cortex-M0+ considerando que o tempo de espera seja 0. Note que o ciclo de instrução de todas as instruções que não envolvem acessos à memória na sua execução é, em média, 1 ciclo de relógio (*clock tick*) e o ciclo de instrução de outras varia conforme a quantidade de acessos à memória.

Neste experimento, usaremos o analisador lógico Saleae para certificarmos a proximidade entre os tempos estimados com base nos ciclos de instrução das instruções e os tempos medidos. No sítio [3] há vídeos-tutorial demonstrando o uso desse analisador.

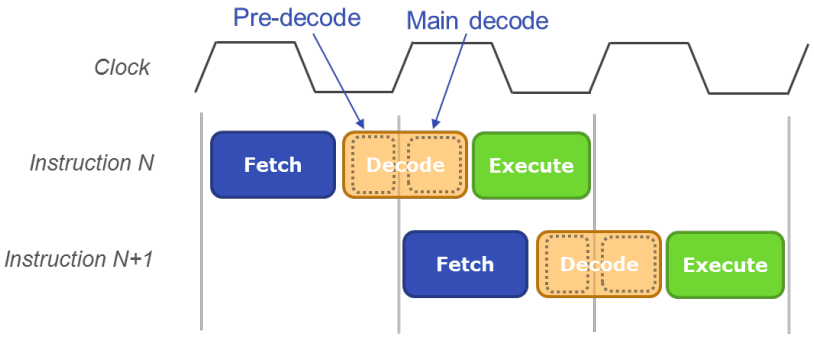

Figura 1: *Pipeline* de 2 estágios (Fonte: [9])

Montador

Mesmo sendo considerada uma linguagem de baixo nível, um processador não consegue gerar sinais de controle a partir das instruções em mnemônicos [1]. 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 denominado **montagem** (*assembling*). Da mesma forma que existe uma grande variedade de linguagens *assembly*, há diversos montadores. Neste curso, usaremos o montador ***GNU Assembler*** (GAS) que faz parte do *toolchain* do *IDE CodeWarrior* [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. Na Seção 3.2/página 13 em [8] é apresentada uma lista de diretivas mais aplicadas, como `.section`, `.align` e `.word`.

Assembler Embutido

Assembler embutido (*Inline Assembler*) é um recurso disponível em alguns compiladores. Ele permite que códigos-fonte em *assembly* sejam embutidos em linguagem de médio nível, como C, melhorando o desempenho e facilitando acessos às instruções específicas de um processador quando não são visíveis nos comandos de linguagem de médio nível. Em linguagem C a palavra `asm`, ou `__asm__`, é reservada para embutir as instruções em *assembly* nos programas em C [10]. As diretivas não são suportadas. O formato do comando é:

__asm__ <qualificadores> (“instrução em *assembly* 1 \n\t”
“instrução em *assembly* ... \n\t”
“instrução em *assembly* n \n\t”

```
   : lista de argumentos de saída separados por vírgula
 [ : lista de argumentos de entrada separados pela vírgula
 [ : lista de recursos modificáveis pelos mnemônicos separados pela vírgula ] ])
```

Cada instrução em *assembly* deve ser seguido de um caracter de controle de quebra de linha (\n) e de tabulação (\t) e transferido para o montador como uma *string* (entre aspas duplas). Para que este interprete as sequências de *strings* como uma sequência de instruções separadas por linhas (sintaxe da linguagem de montagem). As outras 3 listas separadas por “:” constituem interface entre C e montador. Por essas listas é especificada a passagem de argumentos de entrada e saída e, opcionalmente, são adicionados os recursos requeridos pelas instruções em *assembly* na sua execução.

Veja no trecho do código abaixo como uma variável `counter` em C pode ser incrementada (`counter := counter + 1`) por instruções em *assembly* representadas como *strings* em C. A variável `counter` é tanto a variável de entrada quanto a de saída. Portanto, ela consta na lista de argumentos de saída e de entrada. Dentro dos mnemônicos, os operandos podem ser referenciados pela sua posição na sequência de operandos declarados, precedida de %. Neste caso, a sua posição é

0 (primeiro e único argumento da lista). Podemos ter restrições acompanhadas a cada variável. No exemplo abaixo temos duas restrições entre aspas:
= : o conteúdo da variável é sobrescrito, substituído por um novo valor;
r : qualquer registrador pode ser usado para representar `counter` no bloco de códigos de máquina

```
int main(void)
 {
        int counter = 0;

        for (;;) {
        asm (
                "mov r0, %0 \n\t"
                "add r0, r0, #1 \n\t"
                "mov %0, r0 \n\t"
                : "=r" (counter)
                : "r" (counter)
                );
   }
        return 0;
}
```

De acordo com uma convenção, os registradores R0-R3 são automaticamente utilizados para passar os valores de até 4 argumentos. Quando se trata de um trecho de códigos em *assembly* dentro de uma sub-rotina, podemos assumir que os primeiros quatro argumentos da função têm os seus valores armazenados nesses registradores.

Na Seção 3.4/página 16 em [8] há vários exemplos de uso do *assembler* embutido de GNU em códigos C.

Processamento de Chamada de Função

Uma chamada de função é um comando que inclui o nome da função que está sendo chamada ou o endereço da função e, opcionalmente, os argumentos que devem ser compartilhados entre ela e o contexto em que ela é chamada. O processamento dessa chamada implica no desvio do fluxo de controle do contexto em que a função é chamada para o fluxo de controle definida pelas instruções da função propriamente dita. Para que a função chamada possa usar todos os recursos (registradores, ponteiro de pilha e contador de programa) necessários para execução das suas instruções, algumas medidas são tomadas.

Antes de carregar no PC (contador de programa) o endereço da função, salva-se na pilha, com a instrução `push` (Seção A6.7.50/página 167 em [1]), o endereço da instrução a ser executada após a execução da função e os valores dos registradores em uso pelo contexto em que a função é chamada. Usa-se ainda a pilha, através do ponteiro da pilha SP, para passagem dos argumentos entre a função e o contexto em que ela é chamada e para armazenamento das variáveis locais da função. Ao final da execução da função, os dados armazenados na pilha são desempilhados, através da instrução `pop` (Seção A6.7.49/página 165 em [1]) para restaurar o estado em que se encontrava

o contexto em que ela foi chamada e retomar o fluxo de execução carregando no PC o endereço da instrução salvo.

Na arquitetura ARM é disponível o registrador de *Link* (LR) com a finalidade específica de salvar o endereço de retorno de uma função chamada. Sendo LR um registrador do processador, pode-se retomar ao fluxo de controle do contexto em que uma função foi chamda sem fazer acessos à memória onde está armazenada a pilha.

Em C, esse processamento de chamadas fica transparente para o programador. Quando compila um comando de chamada de função, o compilador C gera automaticamente as instruções de salvar e restaurar o contexto de uma função antes de desviar para a outra. Quando compila um comando de retorno de uma função ou detecta o fim de uma função, o compilador gera automaticamente instruções de restauro. É de responsabilidade do programador diferenciar os argumentos que são passados para uma função.

Vimos no roteiro 1 [16] que a passagem de argumentos de uma função é **por valor** em linguagem C, Alterações nos valores dos argumentos dentro de uma função não são refletidas nas variáveis da função que a chamou, porque a função chamada processa uma cópia das variáveis armazenadas num outro endereço temporário (tipicamente numa pilha). No entanto, podemos contornar essa limitação passando um endereço como valor usando o operador “endereço-de” & e acessar o conteúdo do endereço dentro da função com o operador “valor-de” *. Neste caso, a função chamada processa o conteúdo de uma cópia de endereço, assegurando que alterações no conteúdo do endereço sejam visíveis por todas as funções que tenham acesso a tal endereço, seja na versão original ou seja na cópia.

EXPERIMENTO

Neste experimento vamos explorar o código de máquina do projeto `rot2_aula` [13] no ambiente IDE CodeWarrior e desenvolver um projeto em C com *assembler* embutido.

1 **C para *Assembly*:** Importe rot2_aula no IDE CodeWarrior. Use na perspectiva C/C++ o comando Disassemble para obter o código de máquina do arquivo `main.c`. As instruções em *assembly* são agrupadas por blocos. Cada bloco de linhas de instrução *assembly* corresponde a uma linha de comando C que antecede o bloco e o código de máquina de cada instrução é mostrado na segunda coluna do bloco (Seção 2.2.5/página 15 em [15]).

1.a) **Arquitetura Load-Store:** Em quais instruções *assembly* foi traduzida a linha de comando

```
contador++;
```

que equivale a

```
contador = contador + 1;
```

Quais instruções carregam (*load)* o registrador com o valor da variável `contador`? Quais instruções incrementam o valor? E quais instruções salvam (*store*) o valor incrementado na

variável contador? Note que este padrão de acesso se repete para todas as operações lógico-aritméticas.

1.b) **Modos de Endereçamento:** Explique a seguinte afirmação: “No final da função `main` os valores constantes, 0x40048038, 0x4004a04c etc, definidos na função são armazenados no segmento de instruções (`.text`) com a diretiva `.word`. Eles são endereçados pelo modo de endereçamento relativo a PC.” Use como base as instruções *assembly* correspondentes à seguinte linha de comando em C

*(uint32_t volatile *) 0x40048038u |= (1<<10);

1.c) **Instruções Thumb:** Em quais instruções em *assembly* foi traduzida a linha de comando em C

*i = valor * 32;

Note que não foi usada a instrução de multiplicação MUL (Seção A6.7.44/página 159 em [1]). Quais instruções equivalentes foram usadas? Há algum ganho no tempo de execução? Justifique.

1.d) **Processamento de Chamada de Função:** Identifique para a chamada da função `espera`, o bloco de instruções em *assembly*, gerado pelo compilador, que (1) salva o conteúdo dos registradores e armazenar nos registradores r0 – r3 os quatro primeiros argumentos passados para a função antes do desvio para a função `espera`; (2) salva na pilha os registradores, os argumentos passados para a função e as variáveis locais da função; e (3) restaura o estado do sistema após a execução de `espera`.

1.e) **Formato de Instruções *Thumb***: Os códigos de máquina das instruções

ldr r3, [pc,#104]
str r0,[r7,#4]
bne.n 2e
pop {r7,pc}
cpy sp,r7

são, respectivamente, 0x4b1a, 0x6078, 0xd1f9, 0xbd80 e 0x46bd. Explique como são codificados os deslocamentos #104 e #4 e como se obtém os endereços efetivos dos operandos a partir dos códigos binários das instruções. Explicações: (1) # precede um valor decimal; (2) formato b<c>.<q> da instrução b(ranch) é explicado na Seção A6.2/página 98 em [1]. Quando <c>=ne significa que a condição é “*not equal*” (código binário 0001) e quando <q>=n indica que a codificação é em 16 *bits*; (3) 2e é o endereço a ser codificado como endereço relativo ao conteúdo do contador do programa; (4) cpy = mov (Seção A6.7.40/página 1595 em [1]).

1.f) **Processamento de Chamada de Função:** A função `multiplo_iteracoes` tem dois argumentos, `valor` e `i`. Espera-se que a função retorna com um novo valor `i` a partir do argumento de entrada `valor`. Portanto, foi usado o operador “endereço-de” & na chamada da função `multiplo_iteracoes (valor, &i)`. Para acessar o conteúdo da variável `i` (que é agora um endereço) dentro da função, é aplicado o operador “valor-de” *. Explique sucintamente como isso funciona, analisando no modo de **passo por instrução em *assembly*** (*Instruction Stepping Mode*, Seção 2.5/página 26 em [15]) o que é efetivamente empilhado antes e durante a execução da chamada `multiplo_iteracoes (valor, &i)` e

desempilhado com a execução da linha de instrução `return` dentro da função `multiplo_iteracoes`.

2 **Pipeline de 2 Estágios:** Vamos estimar o tempo de execução de um trecho de códigos a partir dos ciclos de instruções especificados pelo fabricante e da frequência do processador (20.971.520 Hz). Usaremos o analisador lógico Saleae [3], conectado conforme mostra a Figura 1 no pino `PTE20` do microcontolador (um canal do analisador no pino 4 do *header* H5 e o terra do analisador no pino 5 de H5) [6], para validar o tempo estimado. Para isso, é necessário configurar o pino com o projeto `rot3_aula`, em *assembly* [11] (Seção 2.1/página 8 em [15]).

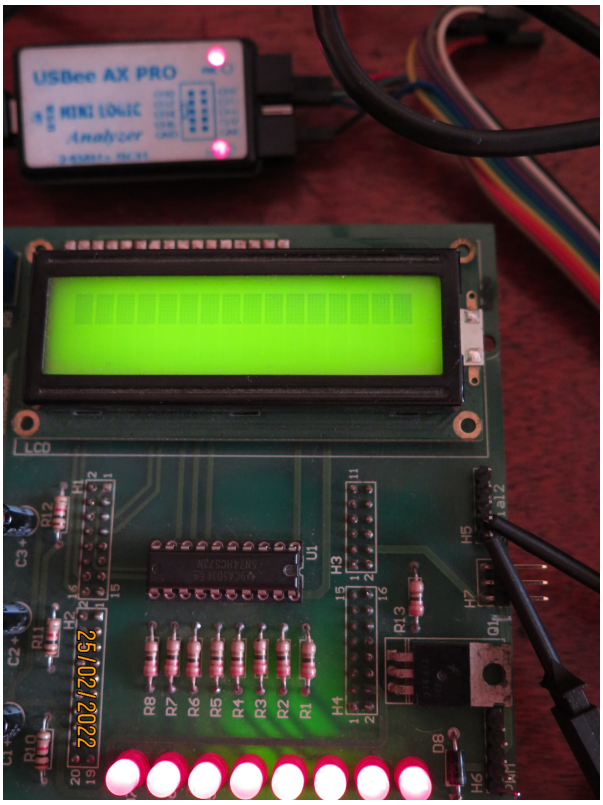

Figura 1: Conexão do analisador com os pinos 4 e 5 (terra) do *header* H5 da placa FEEC871.

2.a) A menos das instruções de controle da quantidade de iterações dentro dos dois laços, `iteracao` e `laco`, as outras instruções foram adicionadas para aumentar o tempo de execução da função `espera`, e consequentemente o tempo de espera, de forma controlada. A tarefa dessas funções é só ocupar o processador para ele aguardar o momento apropriado para retomar o fluxo de execução "útil". É muito comum usar a instrução `NOP` (Seção A6.7.47/página 163 em [1]), cujo ciclo de instrução é considerado 1 ciclo de relógio (1 *clock tick*), no lugar dessas instruções para explicitar o papel de "espera" sem operação "útil". Com base na explicação dada na Seção A6.7.47/página 163 em [1], explique por quê esta segunda solução não foi utilizada nessa implementação.

2.b) Com base nos ciclos de instrução especificados na Seção 3.3/página 26, em [2], associe na Tabela 1, em ciclos de relógio, o ciclo de instrução de cada instrução da função `espera` do programa `main.s` do projeto.

2.c) Estime a quantidade de ciclos de relógio para COUNT=1000 (passados pelo registrador R0) e meça a largura dos pulsos dos sinais gerados no pino `PTE20`. Qual é a diferença, em percentagem, entre o estimado e o valor medido? Faça um *printscreen* da forma de onda

amostrada, contendo os dados da largura de pulso, da frequência e do período do sinal amostrado.

2.d) Ajuste as instruções da rotina `espera` de forma que, mantendo COUNT=1000, a largura do pulso do sinal no pino PTE20 fique em torno de 5ms. Dica: Para que a unidade de espera seja 5us (=104,8576 ciclos de relógio), deve-se adicione mais instruções nos laços da função `espera` em *assembly*.

Tabela 1: Ciclos de instrução

Instrução	**Ciclos de relógio**
espera:	
push {r0,r2,r3,r7,lr}	
sub sp, sp, #4	
add r7, sp, #0	
str r0, [r7,#0]	
iteracao:	
mov r2, #NUM_ITERACOES	
laco:	
mov r3, #5	
orr r3, r0	
and r3, r0	
lsr r3, #1	
asr r3, #1	
sub r2, #1	
cmp r2, #0	
bne laco	
rev r3, r3	
lsl r3, #0	
sub r0, #1	
cmp r0, #0	
bne iteracao	
pop {r0,r2,r3,r7,pc}	

3 Desenvolva um projeto `led_piscante` com o LED verde piscando na frequência de 2Hz. Recomenda-se os seguintes passos:

3.a) Crie um novo projeto (Seção 2.1/página 4 em [15]).

3.b) Sobreescreva o arquivo `main.c` do projeto `rot2_aula` sobre `main.c` do novo projeto (Seção 2.2.3/página 14 em [15]).

3.c) Gere um executável com a cópia e carregue-o no microcontrolador para certificar a sua operação correta (Seção 2.3/página 18 em [15]).

3.d) Crie uma nova função `espera_5us` que substitui o simples decremento de i como passo de tempo de espera em `espera` por um bloco de instruções cujo tempo de execução se aproxima de 5us.

3.e) ***Assembler Embutido*** Embute na função `espera_5us` as instruções em *assembly* seguindo o modelo, lembrando que as instruções de salvar e restaurar contextos numa chamada de função são geradas automaticamente pelo compilador e que a diretiva `equ` não é "embutível":

```
void espera_5us (unsigned int i)
{
```

```
    asm(
          <mnemônicos>
          :<Operandos de saída>
          :<Operandos de entrada>
          :<Clobbers>
    );
}
```

3.f) Documente a função `espera_5us` segundo a sintaxe Doxygen [17] .

3.g) Substitua o argumento 16384 da chamada da função `espera` pelo valor 100000, passando as chamadas para `espera_5u (100000)`.

3.h) Habilite *Print Size* para uma simples análise do tamanho de memória ocupado. Gere um executável e verifique se atende a especificação.

3.i) Gere uma documentação do projeto.

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 as questões 1 e 2 do roteiro num arquivo em pdf. Exporte o projeto `led_piscante` devidamente documentado (Seção 2.7/página 39 em [15]) para um arquivo depois de aplicar *Clean* no projeto e apagar a pasta de documentação html e latex. Suba os dois arquivos no sistema *Moodle*.

REFERÊNCIAS

[1] ARM. *ARMv6-M Architecture Reference Manual*.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/ARMv6-M.pdf
[2] Cortex-M0+ Technical Reference Manual
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/Cortex-M0+.pdf
[3] Analisador Lógico Saleae
https://www.saleae.com/pt/
[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] Nova versão do esquemático do shield FEEC
http://www.dca.fee.unicamp.br/cursos/EA871/references/complementos_ea871/Esquematico_EA871-Rev3.pdf
[7] Freescale. KL25 Sub-Family Reference Manual
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KL25P80M48SF0RM.pdf
[8] Wu S.-T. Linguagem de Montagem
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/LinguagemMontagem.pdf
[9] ARM Cortex®-M0+ Pipeline
https://microchipdeveloper.com/32arm:m0-pipeline
[10] How to Use Inline Assembly Language in C Code
https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html
[11] rot3_aula.zip
https://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot3_aula.zip
[12] Roteiro 2

http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/roteiros/roteiro2.pdf
[13] rot2_aula.zip
https://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot2_aula.zip
[14] Thumb 16-bit Instruction Set: Quick Reference Card
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/ARM_QRC0006_UAL16.pdf
[15] Wu, S.T. Ambiente de Desenvolvimento – *Software*
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf
[16] Roteiro 1
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/roteiros/roteiro1.pdf
[17] Doxygen
https://www.doxygen.nl/index.html

Revisado em Janeiro/2023
Revisado em Fevereiro/Agosto de 2022
Revisado em Março/Julho de 2021
Setembro de 2020.