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**. Nessa configuração, tanto os dados quanto as instruções dos programas compartilham o mesmo espaço de endereços de memória e os mesmos barramentos de conexão com o núcleo. Sua **arquitetura de Entrada/Saída** (E/S) é mapeada na memória, implicando que as instruções para acessar periféricos e memória são idênticas.

O espaço de endereçamento, com capacidade de $2^{32}=2^2\ 2^{10}\ 2^{10}\ 2^{10}$ (equivalente a 4 Gb), é dividido em múltiplos blocos, cada um destinado ao mapeamento de periféricos e memórias de diferentes tecnologias (Tabela 4-1/página 105 em [7]). Todos os módulos do sistema são interligados de acordo com a especificação da arquitetura de barramento de microcontroladores avançada (*Advanced Microcontroller Bus Architecture*, **AMBA**), utilizando o protocolo de barramento de alto desempenho avançado (*Advanced High-Performance Bus*) para comunicação eficiente em 8, 16 e 32 *bits* (Capítulo 21/página 331 em [7]). Além disso, emprega o protocolo de barramento de

periféricos avançado (*Advanced Peripheral Bus*) para comunicação eficaz com os periféricos (consulte Capítulo 21/página 335 em [7]).

Como um **processador RISC**, o Cortex-M0+ apresenta um repertório de instruções significativamente mais compacto e eficiente em comparação com processadores CISC. Sua arquitetura de conjunto de instruções (*Instruction Set Architecture*, ISA) é de 32 *bits* e segue a plataforma ARM. Para otimizar o uso da memória, foi introduzido o repertório de instruções *Thumb* de 16 *bits* (**estado *Thumb***) como uma alternativa às instruções de 32 *bits* no **estado ARM**.

O processador Cortex-M0+ opera exclusivamente no estado *Thumb* e possui 8 registradores de trabalho R0—R7 de 32 *bits*, além dos registradores de funções específicas, também de 32 *bits*. Estes incluem o contador de programa (PC) com código binário de identificação 0b1111, o registrador de Link (LR), o ponteiro de pilha (SP) e os registradores de estado (PSR) (consulte Seção A2.3/página 35 em [1]).

Destaca-se que o Cortex-M0+ possui um conjunto limitado de instruções de 32 *bits*, geralmente reservadas para desvios às rotinas (consulte Seção A6.7.13/página 123 em [1]), barreiras de sincronismo (consulte Seção A6.7.21/página 133, A6.7.22/página 134, A6.7.24/página 136 em [1]) e transferência de dados entre registradores de funções específicas (consulte 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 de texto com uma instrução por linha, podendo cada linha conter até três campos: um rótulo, uma instrução e um comentário. O campo de instrução *Thumb* é composto por dois subcampos codificados em 16 *bits* (Seção A5.2/página 84 em [1]): o código de operação representado por mnemônicos e os operandos. Normalmente, operações envolvem dois endereços, um como destino e outro como fonte. Por uma questão de concisão, é comum omitir o endereço de um dos operandos em operações que utilizam dois, como é o caso da soma, presumindo-se que 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 LDR (Seção A6.7.28/página 143 em [1])/STR (Seção A6.7.0/página 179 em [1]) que adotam o **modo de endereçamento relativo a um registrador basal** para especificar o endereço de um dos operandos, os códigos binários 0b1101 e 0b1111 são utilizados para designar, respectivamente, o ponteiro de pilha SP e o contador de programa PC como o registrador basal. O *bit* mais significativo é armazenado num *bit* separado e automaticamente concatenado 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 basal (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*, como a MRS (Seção A6.7.42/página 158 em [1]), são acessadas numa mesma transferência via o barramento de 32 *bits*. *S*eus dados são extraídos e concatenados automaticamente para reconstruir um dado de tamanho maior do que uma instrução de 16 *bits* conseguiria conter. Adicionalmente, todas as instruções (Seção A6.7.49/página 165 em [1]) que modificam o conteúdo do LR automaticamente ajustam o *bit* menos significativo desse registrador. Esse ajuste permite diferenciar o tamanho das instruções a serem processadas como vimos no roteiro 2 [12]: *bit* em 0 (ARM, 32 *bits*) e *bit* em 1 (Thumb, 16 *bits*).

Arquitetura *Load-Store*

A **arquitetura ARM** segue o paradigma ***load-store***, onde as instruções são divididas em duas classes: (1) aquelas que não envolvem diretamente a memória em sua execução, como operações lógico-aritméticas, e (2) instruções mais complexas e custosas que acessam a 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 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`, são aplicadas 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

Na classe de instruções que não envolvem diretamente a memória no processamento, os **modos de endereçamento** dos operandos podem ser o **modo imediato** (operandos codificados diretamente em instruções) ou o **modo por registrador** (operandos armazenados num dos 8 registradores de trabalho). Já na classe de instruções que incluem acessos à memória, o segundo operando requer um endereço da memória de 32 *bits* a ser acessado.

Esse endereço pode ser especificado pelo **modo de endereçamento indexado imediato relativo a um registrador basal** (endereço efetivo é a soma do índice codificado no campo de endereçamento da instrução e o conteúdo do registrador basal especificado) e o **modo de endereçamento indexado relativo a um registrador basal** (endereço efetivo é a soma do índice armazenado no registrador de índice codificado no campo de endereçamento e o conteúdo do registrador basal especificado). Em particular, quando o registrador basal é PC ou SP, é comum referir-se ao modo de endereçamento como **relativo a PC** e **relativo a SP**, respectivamente**.** Vale ainda ressaltar 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

Para otimizar o desempenho, o núcleo Cortex-M0+ implementa um *pipeline* de 2 estágios, onde um **ciclo de instrução** é dividido em 2 estágios paralelizáveis, como ilustra a Figura 1. Um estágio abrange a fase de busca e a pré-decodificação/classificação da instrução acessada, enquanto o outro estágio compreende a fase de decodificação propriamente dita e a execução. A paralelização dos dois estágios, quando não há instruções de desvio ou acessos à memória, resulta em um tempo médio de ciclo de instrução de 1 ciclo de relógio ($t = (10^{-6}/20.97512)$ s), ao invés de 2. Além disso, por operar no estado *Thumb*, o núcleo Cortex-M0+ acessa em cada busca duas instruções por um barramento de 32 *bits*.

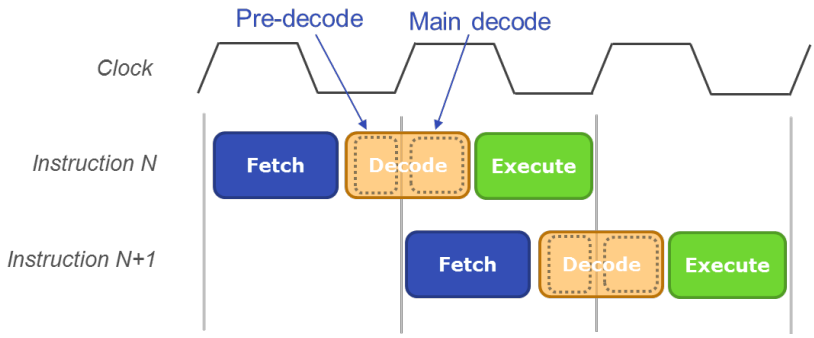

Figura 1: *Pipeline* de 2 estágios (Fonte: [9])

A uniformidade no tamanho e no ciclo de instruções simplifica a tarefa do desenvolvedor ao estimar o espaço e o 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 um tempo de espera de 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*), enquanto o ciclo de instrução de outras varia conforme a quantidade de acessos à memória.

Neste experimento, usaremos o analisador lógico Saleae para certificar a proximidade entre os tempos estimados, com base nos ciclos de instrução das instruções, e os tempos medidos. Para uma orientação detalhada sobre a utilização desse analisador, é possível encontrar vídeos tutoriais em [3].

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 destacar 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

O *Assembler* embutido (*Inline Assembler*) é um recurso disponível em alguns compiladores que permite a incorporação de códigos-fonte em *assembly* em linguagem de médio nível, como C. Essa abordagem visa aprimorar o desempenho e facilitar o acesso a instruções específicas de um processador, especialmente quando essas instruções não são diretamente visíveis nos comandos de linguagem de médio nível. Na linguagem C, a palavra-chave asm ou `__asm__`, é reservada para embutir instruções *assembly* nos programas em C [10]. Vale notar que diretivas não são suportadas nesse contexto.

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 por caracteres de controle de quebra de linha (\n) e tabulação (\t), sendo transferida para o montador como uma *string*, delimitada por aspas duplas. Esse formato permite que o montador interpreta as sequências de *strings* como uma série de

instruções separadas por linhas, conforme a sintaxe da linguagem de montagem. As outras 3 listas, separadas por ":", representam a interface entre C e o montador. Por meio dessas listas, são especificadas 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 a variável `counter` em C pode ser incrementada (`counter := counter + 1`) por meio de instruções em *assembly* representadas como *strings* em C. A variável `counter` desempenha o papel de variável de entrada e de saída, sendo incluída, portanto, na lista de argumentos tanto de saída quanto de entrada. No contexto dos mnemônicos, os operandos podem ser referenciados pela sua posição na sequência de operandos declarados, sendo precedidos por %. Neste caso, a posição é 0, indicando o primeiro (e único) argumento da lista. Além disso, é possível impor restrições específicas a cada variável. No exemplo abaixo, duas restrições são adicionadas 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;
}
```

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

Conforme uma convenção estabelecida, os registradores R0-R3 são automaticamente utilizados para armazenar os valores de até 4 argumentos. No contexto de um trecho de código em *assembly* dentro de uma sub-rotina, podemos inferir que os valores dos primeiros quatro argumentos da função são armazenados nesses registradores.

Processamento de Chamada de Função

Uma **chamada de função** é um comando que envolve o nome ou endereço da função sendo chamada, e opcionalmente, os argumentos a serem compartilhados entre a função e o contexto em que ela é invocada. O processo de realizar essa chamada implica em redirecionar o fluxo de controle do contexto da chamada para o fluxo de controle definido pelas instruções da função em si. Para

garantir que a função chamada tenha acesso a todos os recursos necessários, como registradores, ponteiro de pilha e contador de programa, algumas medidas são adotadas.

Antes de carregar o endereço da função no PC (contador de programa), realiza-se o salvamento na pilha utilizando a instrução `push` (Seção A6.7.50/página 167 em [1]). Nesse processo, são armazenados o endereço da instrução a ser executada após a conclusão da função e os valores dos registradores em uso pelo contexto em que a função foi chamada. A pilha, acessada pelo ponteiro de pilha SP, é também utilizada para a passagem de argumentos entre a função e o contexto chamador, além de servir para o armazenamento de variáveis locais da função.

Ao término da execução da função, os dados previamente armazenados na pilha são desempilhados, por meio da instrução `pop` (Seção A6.7.49/página 165 em [1]). Esse processo visa restaurar o estado original do contexto chamador, permitindo a retomada do fluxo de execução ao carregar no PC o endereço da instrução previamente salvo. Na arquitetura ARM, é disponibilizado o **registrador de Link** (LR) com a finalidade específica de armazenar o endereço de retorno de uma função chamada. Por ser um registrador do processador, a retomada ao fluxo de controle do contexto em que a função foi chamada pode ocorrer sem a necessidade de acessar diretamente a memória onde a pilha está armazenada.

Em C, o processamento de chamadas de função ocorre de maneira transparente para o programador. Ao compilar um comando de chamada de função, o compilador C automaticamente gera as instruções necessárias para salvar e restaurar o contexto da função antes de desviar para outra. Similarmente, ao compilar um comando de retorno de função ou identificar o fim de uma função, o compilador automaticamente produz as instruções correspondentes de restauração. Cabe ao programador distinguir os argumentos que são passados para uma função.

Conforme discutido no roteiro 1 [16], a passagem de argumentos em linguagem C é **por valor**. Alterações nos valores dos argumentos dentro de uma função não são refletidas nas variáveis da função chamadora, pois a função chamada opera sobre cópias das variáveis armazenadas em um endereço temporário, geralmente na pilha. No entanto, é possível contornar essa limitação passando um endereço como valor, utilizando o operador “endereço-de” & para obter o endereço e o operador “valor-de” * para acessar o conteúdo desse endereço dentro da função. Nesse cenário, a função chamada manipula o conteúdo de uma cópia do endereço, garantindo que alterações no conteúdo sejam visíveis para todas as funções que têm acesso a esse endereço, tanto na versão original quanto 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. Embora os conhecimentos específicos sobre a codificação de instruções abordados nos itens (1) e (2) possam não ser diretamente aplicáveis em muitos cenários práticos, eles proporcionam uma base sólida para a compreensão de conceitos mais avançados em arquitetura de computadores e programação de baixo nível. Além disso, esses conhecimentos permitem que vocês ganhem *insights* sobre a engenhosidade por trás de um processador programável.

1 **C para *Assembly*:** Importe `rot2_aula` no IDE CodeWarrior e analise os códigos em *assembly* do arquivo `main.c`. Conforme mostrado no roteiro 2 [13], é possível utilizar o comando `Disassemble` na perspectiva C/C++ para visualizar o código em *assembly* traduzido. Alternativamente, caso prefira, pode-se gerar um executável, transferí-lo para o microcontrolador e examinar o código em *assembly* alocado no endereço físico efetivo na aba `Disassembly`. As instruções em *assembly* são agrupadas por blocos. Cada bloco de linhas de instruções em *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* e modos de endereçamento:** A seguinte soma numa arquitetura *load-store* é realizada em 3 passos distintos: transferir o valor da variável num registrador; somar o valor armazenado no registrador com "1" codificado na instrução; e armazenar o resultado obtido no registrador de volta no endereço de memória associado à variável.

```
contador++;
```

que equivale a

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

Identifique as instruções correspondentes a cada passo, detalhando o endereço em que a variável `contador` está armazenado e como ele é codificado nas instruções de transferência do valor da memória para o registrador e do registrador para a memória, e classificando os modos de endereçamento aplicados nos dois operandos na instrução de soma em *assembly* e nas instruções de transferências.

1.b) **Modo de Endereçamento Relativo a PC:** No roteiro 2 [13] mostramos que os endereços dos registradores pré-fixados pelos fabricantes são tratados como constantes pelo compilador C no IDE CodeWarrior e essas constantes são armazenadas no final do segmento de instruções correspondente à função `main`. Analise o bloco de instruções correspondente à linha de comando em C, destacando como o endereço do registrador é acessado nas instruções para modificar o valor armazenado nele:

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

1.c) **Processamento de Chamada de Função – Mudança de Contexto:** O roteiro 2 [13] destacou que ao desviar o fluxo de controle de uma função chamadora para a função chamada, ocorre uma **mudança de contexto**. Para garantir a restauração adequada do contexto da função chamadora antes de retornar para ela, é necessário salvar esse contexto antes de assumir o contexto da função chamada. O compilador de C gera automaticamente as instruções para salvar e recuperar o contexto. Vimos também diferenças entre variáveis locais

e variáveis estáticas. Vamos analisar agora os detalhes dessas instruções. Identifique os blocos de instruções de salvamento e recuperação no desvio do controle de fluxo de `main` para `espera`, destacando os dados (contexto) que foram armazenados, onde estes dados foram armazenados, onde e como as variáveis locais `i` e `valor` da função `espera` são armazenadas, e como as variáveis estáticas são processadas durante o chaveamento de contexto.

1.d) **Processamento de Chamada de Função – Passagem de Valor:** A função `multiplo_iteracoes` tem dois argumentos, `valor` e `i`. Espera-se que a função retorna com um novo valor `i` com base no argumento de entrada `valor`. Portanto, o endereço de i foi passado como o valor do argumento. Foi usado o operador "endereço-de" & na chamada da função `multiplo_iteracoes (valor, &i)`. Para acessar o conteúdo da variável `i`, cujo valor é um endereço, dentro de `multiplo_iteracoes`, é aplicado o operador "valor-de" *. Usando o modo de **passo por instrução em *assembly*** (*Instruction Stepping Mode*, Seção 2.5/página 26 em [15]), explique o que é efetivamente empilhado durante o chaveamento de contexto de `espera` para `multiplo_iteracoes` e qual endereço é acessado para armazenar o resultado `valor * 32`.

1.e) **Compilador:** O compilador C é, de fato, um programa que traduz o código-fonte em C para o código de máquina. A forma como o compilador realiza essa tradução pode variar entre diferentes implementações e desenvolvedores. Um mesmo operador em C pode ser traduzido em diferentes sequências equivalentes de códigos de máquina, e a eficiência dessas traduções pode depender de otimizações específicas aplicadas pelo compilador. Note que não foi usada a instrução de multiplicação MUL (Seção A6.7.44/página 159 em [1]) na tradução para *assembly* da seguinte linha de comando em C:

```
*i = valor * 32;
```

Identifique as instruções em *assembly* usadas, destacando a eficiência em relação a uma solução convencional sob o ponto de vista de ciclo de instrução.

2 **Formato de Instruções *Thumb***: Explique a codificação binária feita para cada menmônico mostrado na tabela abaixo, incluindo com se obtém o endereço efetivo dos operandos envolvidos em cada instrução.

Mnemônico	Código de máquina	Detalhamento da codificação
ldr r3, [pc,#104]	0x4b1a	
str r0,[r7,#4]	0x6078	
bne.n 2e	0xd1f9	
pop {r7,pc}	0xbd80	
cpy sp,r7	0x46bd	

Cabe esclarecer que a sintaxe do mnemônico envolve diversas convençoes, como: (1) # antecede 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` equivale a `mov` (Seção A6.7.40/página 1595 em [1]).

3 **Pipeline de 2 estágios - Ciclo de Instruções:** Vamos estimar o tempo de execução de um trecho de códigos com base nos ciclos de instruções especificados pelo fabricante e na frequência do processador (20.971.520 Hz). Para validar a estimativa, usaremos o analisador lógico Saleae [3], conectado conforme mostra a Figura 1, onde o pino `PTE20` do microcontolador é conectado a um canal do analisador no pino 4 do *header* H5 e o terra do analisador no pino 5 de H5 [6]. Para realizar essa validação, é necessário configurar o pino com o projeto `rot3_aula`, escrito em *assembly* [11] (Seção 2.1/página 8 em [15]).

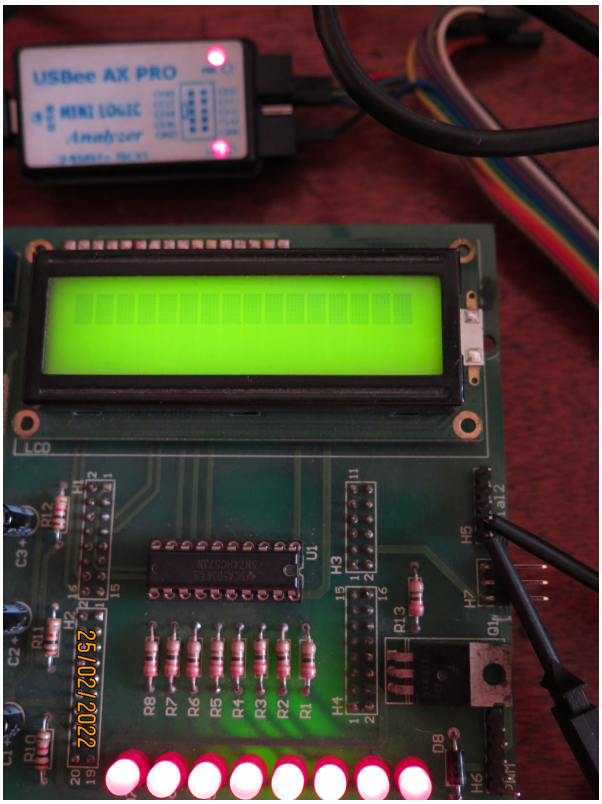

Figura 1: Conexão do analisador com os pinos 4 e 5 (terra) do *header* H5 da placa FEEC871.

3.a) Exceto pelas instruções que controlam a quantidade de iterações nos laços `iteracao` e `laco`, as demais instruções foram adicionadas com o intuito de prolongar o tempo de execução da função `espera` de maneira controlada, consequentemente aumentando o tempo de espera. O propósito dessas instruções adicionais é ocupar o processador, permitindo que ele aguarde o momento adequado para retomar o fluxo de execução "útil". Embora seja comum utilizar a instrução NOP (Seção A6.7.47/página 163 em [1]), cujo ciclo de instrução é considerado 1 ciclo de relógio (1 *clock tick*), para indicar esse período de "espera" sem realizar uma operação "útil", essa segunda solução não foi empregada nesta implementação. A decisão se baseia na explicação fornecida na Seção A6.7.47/página 163 em [1]. Qual é a explicação?

3.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.

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:	
movr2, #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.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.

3.d) Ajuste as instruções da rotina `espera`, por exemplo o valor de `NUM_ITERACOES`, de forma que, mantendo COUNT=1000, a largura do pulso do sinal no pino PTE20 fique em torno de 2ms.

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

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

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

4.c) Gere um executável com o arquivo `main.c` atualizado e carregue-o no microcontrolador para certificar a sua operação (Seção 2.3/página 18 em [15]).

4.d) Crie uma nova função `espera_2us` 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 2us.

4.e) ***Assembler Embutido*** Embute na função `espera_2us` 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_2us (unsigned int i)
```

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

4.f) Documente a função `espera_2us` segundo a sintaxe Doxygen [17] .
4.g) Substitua a chamada `espera (1384)` pela chamada `espera_2us (250000)`.
4.h) Habilite *Print Size* para uma simples análise do tamanho de memória ocupado. Gere um executável e verifique se atende a especificação.
4.i) Escreva um *script* para gerar uma documentação do projeto incluindo gráficos de dependência entre as funções.

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 a 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_ticks.zip
https://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot3_aula.zip

[12] Roteiro 2
http://www.dca.fee.unicamp.br/cursos/EA871/1s2024/roteiros/roteiro2.pdf
[13] rot2_aula.zip
https://www.dca.fee.unicamp.br/cursos/EA871/1s2024/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/EA871/1s2024/roteiros/roteiro1.pdf
[17] Doxygen
https://www.doxygen.nl/index.html

Revisado em Janeiro/2024
Revisado em Janeiro/2023
Revisado em Fevereiro/Agosto de 2022
Revisado em Março/Julho de 2021
Setembro de 2020.