Atividade de Laboratório 6

Objetivos

O objetivo desta atividade de laboratório é dar continuidade no aprendizado da linguagem de montagem da arquitetura ARM. Neste exercício você se familiarizará com a construção e invocação de rotinas.

Descrição

Em alguns casos, você não pode confiar totalmente na memória de dados do sistema. Por exemplo, em situações em que os dados foram transmitidos por um meio sujeito a erros. Ainda, mesmo estando inicialmente corretos em uma memória do tipo DRAM, os *bits* podem mudar aleatoriamente de estado devido ao decaimento de partículas alfa (*soft errors* (http://en.wikipedia.org/wiki/Soft_error#Alpha_particles_from_package_decay)) proveniente de isótopos radioativos presentes no encapsulamento do *chip* (*). Para contornar esse problema e usando seus conhecimentos de correção de erros, você decide implementar duas funções em linguagem de montagem ARM. A primeira recebe 4 *bits* de dados e os transforma em 7 *bits*, sendo 3 deles destinados à verificação da consistência dos 4 *bits* de dados. A segunda recebe os 7 *bits* codificados e recupera os 4 *bits* de dados. A codificação é conhecida como código de Hamming(7,4) (http://en.wikipedia.org/wiki/Hamming%287,4%29) e irá aumentar a confiabilidade da memória de dados, uma vez que, checando os 7 *bits*, você conseguirá determinar se um dos 4 *bits* originais foi alterado por uma partícula alfa (ou outros tipos de partículas que perturbam o estado da DRAM) ou por erros na transmissão dos dados.

Codificação

Escreva uma função folha, que é uma função que não faz outras chamadas, com o rótulo "`encode`". No registrador `R0`, sua função deve receber o valor a ser codificado. Lembre-se que `R0` possui 32 *bits*, então suponha que os 4 *bits* menos significativos de `R0` serão a entrada.

Para construir a saída, você deve entender como o código de Hamming(7,4) codifica 4 *bits*. Assuma que o vetor de 4 *bits* da entrada contenha:

$d_1\ d_2\ d_3\ d_4$

A saída então será

$p_1\ p_2\ d_1\ p_3\ d_2\ d_3\ d_4$

Os novos *bits* introduzidos com o radical **p** são *bits* de **paridade**. Cada um dos 3 *bits* de paridade é responsável por refletir a paridade de um determinado subconjunto de *bits* (um subconjunto de 3 elementos dos 4 *bits* de entrada disponíveis). Um *bit* de paridade contém **1** caso o conjunto de *bits* avaliado tenha um número ímpar de **1**s, ou **0** caso contrário. Siga a tabela:

Bit de paridade	Subconjunto de *bits* testados
p_1	$d_1\ d_2\ d_4$
p_2	$d_1\ d_3\ d_4$
p_3	$d_2\ d_3\ d_4$

Coloque a saída em `R0` e retorne.

Dicas:

1. O "ou-exclusivo" (XOR) é um operador lógico que facilita o cálculo de *bits* de paridade.
2. A instrucão AND é útil para deixar apenas uma parte de um conjunto de *bits* setado.

Decodificação

Se você for um aficcionado por matemática ou simplesmente ler o artigo sobre o código de Hamming, eventualmente irá perceber que a mágica está em calcular a paridade de subconjuntos que se sobrepõem. Isso não só torna possível detectar quando um *bit* foi invertido, mas também determinar exatamente qual *bit* foi invertido e, se necessário, corrigi-lo. Infelizmente, se dois *bits* forem invertidos, não há como corrigir. Três ou mais erros podem gerar um dado válido e causar um erro indetectável. Neste exercício, porém, você deve assumir que no máximo 2 *bits* podem ser alterados. Sua função "`decode`" deve receber em `R0` o código de Hamming(7,4) de um dado qualquer. Como esse código possui apenas 7 *bits* e o registrador `R0` possui 32, assuma que a entrada está nos 7 *bits* menos significativos do registrador. Como saída de sua função, você deverá retornar o dado original (de 4 *bits*) no registrador `R0`. Utilize os 4 *bits* menos significativos de `R0` e garanta que todos os outros *bits* estejam zerados. Também retorne em R1 o número 1 caso você tenha detectado algum erro, ou 0 caso contrário. Não é necessário corrigir o valor caso seja detectado erro.

Extrair o campo de dados é trivial. Contudo, para detectar se o campo de dados extraído está correto, isto é, se não contém nenhum erro devido a inversões de *bits* na memória, você deve *verificar a paridade* para cada um dos 3 subconjuntos. Você pode utilizar o operador *ou-exclusivo*, ou XOR, nos *bits* de um determinado subconjunto. Por exemplo, para verificar a paridade pela qual o bit p_1 está responsável, p_1 XOR d_1 XOR d_2 XOR d_4 deve ser igual a **0**. Caso contrário, há um erro no dado codificado. Faça isso para os três subconjuntos a fim de atestar se você deve confiar no dado codificado com Hamming(7,4).

Testes

Utilize o código modelo abaixo para desenvolver seu programa.

- Código modelo: hamming-template.s (./hamming-template.s)

Neste modelo, a função *_start* é a primeira função a ser chamada. Esta função chama outras funções para ler os dados da entrada padrão, codificar o dado e imprimi-lo na saida padrão. Note que:

- a função de codificação e decodificação estão incompletas. Você deve implementá-las!
- a função `_start` contém código que lê uma cadeia de caracteres da entrada padrão, converte-a para um número binário, invoca a função `encode` para codificar o valor, converte o resultado para uma cadeia de caracteres e finalmente imprime esta cadeia na saída padrão. Você deve estender o código para que ele também leia uma cadeia de caracteres que representa um valor codificado e realize o processo de decodificação do valor. O valor decodificado, bem como um valor que indica se algum erro foi detectado (1, caso um erro seja detectado, ou 0, caso contrário) devem ser impressos na saída padrão como resultado do processo de decodificação.

Os casos de teste abertos podem ser encontrados em: testes_abertos.txt (./testes_abertos.txt).

(*) O problema de isótopos radiotivos do encapsulamento é - parcialmente - resolvido exigindo-se um certo nível de purificação (remoção dos isótopos) do material próximo ao *chip* DRAM, que o reveste. Na prática, um ambiente com radiação acima do comum (e.g. Fukushima) poderia facilmente bombardear um *chip* DRAM com erros, tornando impraticável o funcionamento do sistema. Isto é um problema cada vez maior conforme os *chips* diminuem de tamanho e aproximam-se de escalas atômicas e, por esse motivo, uma partícula alfa é suficiente para perturbar um *chip*. Este problema foi descoberto por volta da década de 70.

Entrega e avaliação

O arquivo referente à atividade deve ser submetido para avaliação utilizando-se o sistema Susy em: https://susy.ic.unicamp.br:9999/mc404abef/06ab (https://susy.ic.unicamp.br:9999/mc404abef/06ab) ou https://susy.ic.unicamp.br:9999/mc404abef/06ef (https://susy.ic.unicamp.br:9999/mc404abef/06ef).

É necessário que você envie apenas o arquivo com o código em linguagem de montagem.

Note que o Susy não fará a correção automática do seu programa.