Códigos de correção de erro

Objetivo

Desenvolver um circuito que calcule o código de correção de erro de dados armazenados em memória e também um circuito que corrija o erro de um bit.

Introdução

Nesse laboratório, você vai desenvolver circuito para corrigir erros que acontecem na memória de sistemas computacionais ou em transmissões de dados. Esses erros podem ser do tipo 'Hard Error', que significa uma falha permanente com o bit sempre fixo em 0 ou em 1, ou do tipo 'Soft Error', que significa uma falha temporária que trocou um bit de 0 para 1 ou vice-versa. Estamos interessados no segundo caso. Alguns estudos mostram que os erros são causados, majoritariamente, por radiação cósmica, mas também podem ser causados por falhas de fabricação, problemas de alimentação, entre outros.

Por que é importante?

A memória é um dos componentes mais importantes de um sistema de computação. Ela armazena os dados e instruções que são utilizados pelo processador. A memória é composta por células de armazenamento que podem armazenar um bit de informação. Cada célula de memória é endereçada por um número único que é utilizado para acessar o conteúdo da célula. Se um bit muda de valor, a informação armazenada na célula de memória pode ser corrompida. A correção de erro de memória é um método que permite detectar e corrigir erros de armazenamento de dados em memória.

Tem um artigo muito legal na Wikipedia sobre esse assunto. Dois valores interessantes citados no artigo foram o estudo feito no datacenter do Google publicado em 2009 indicando que erros em memória podem chegar a 1 erro de 1 bit por gigabyte de RAM a cada 1,8 horas.

Definições

Existem inúmeros circuitos que podem detectar e/ou corrigir erros em memória. O mais simples deles é o bit de paridade, que é capaz de detectar se aconteceu um erro, mas não consegue identificar nem corrigir o bit errado. Um mecanismo mais complexo é um código de Hamming, que é capaz não só de detectar um erro quanto corrigir o bit que está errado caso apenas um erro tenha acontecido. Existem códigos ainda mais complexos, como o código de Reed-Solomon, que é capaz de corrigir mais de um erro, mas esses códigos são mais complexos e demandam mais hardware.

Bit de paridade

O bit de paridade é um método de detecção de erros que consiste em adicionar um bit extra ao final de uma palavra de dados. O valor desse bit é escolhido de forma que o número total de bits 1 na palavra de dados, incluindo o bit de paridade, seja par ou ímpar. Para paridade 'par', se o número de bits 1 na palavra de dados for par, o bit de paridade é 0. Se o número de bits 1 na palavra de dados for ímpar, o bit de paridade é 1. Se um bit de dados é alterado, o número de bits 1 na palavra de dados será alterado e o bit de paridade não será mais compatível com o número de bits 1 na palavra de dados. O bit de paridade é capaz de detectar a ocorrência de um erro, mas não é capaz de identificar qual bit de dados está errado. Para paridade 'ímpar', o oposto acontece, adicionando o bit 1 quando o número de bits 1 na palavra de dados for par.

Simplificadamente: Paridade par que um número par de bits 1 no total e paridade ímpar quer um número ímpar de bits 1 no total.

Dados	Paridade par	Paridade ímpar
0000	0	1
0001	1	0
0010	1	0
0011	0	1

Dado com paridade	Paridade par	Paridade ímpar
00000	correto	falha
00001	falha	correto
00110	correto	falha
00111	falha	correto

Tarefa 1

Desenvolva um circuito que calcule o bit de paridade de uma palavra de 8 bits. O circuito deve ter 8 entradas (os bits da palavra) e uma saída (o bit de paridade). O circuito deve ser capaz de calcular o bit de paridade par para qualquer palavra de 8 bits. O circuito deve ser implementado em

submetida junto com a tarefa 5.

Tarefa 2

Desenvolva um circuito que detecte se houve uma falha de paridade em uma palavra de 8+1 bits (8 bits + um bit de paridade). O circuito deve ter 9 entradas (os 8 bits da palavra e o bit de paridade) e uma saída (um bit que indica se houve erro). O circuito deve ser capaz de detectar a ocorrência de um erro em qualquer palavra de 8 bits. O circuito deve ser implementado em Verilog e simulado no DigitalJS e no Icarus Verilog. Essa tarefa só será submetida junto com a tarefa 5.

Código de Hamming

O código de Hamming é um método de detecção e correção de erros que consiste em adicionar bits extras a uma palavra de dados. O código de Hamming é capaz de detectar a ocorrência de um erro e identificar qual bit de dados está errado. O número de bits extras é escolhido baseado na fórmula

$$2^m \geq m + k + 1$$

onde ***k*** é o número de bits de informação, chamados de ***d1***, ***d2***, etc e ***m*** é o de bits de correção, chamados de ***p1***, ***p2***, ... Por praticidade de geração dos valores, esses bits ***p*** ficam nas potências de 2 (isto significa que a representação em binário da posição desses bits só tem um bit com valor 1). A tabela abaixo, extraída da referência acima, ilustra melhor:

bit codificado	p1	p2	d1	p4	d2	d3	d4
p1	✅		✅		✅		✅
p2		✅	✅			✅	✅
p4				✅	✅	✅	✅

essa codificação é chamada de H(7,4) pois tem 7 bits no total e 4 bits de dados. Note que ***p1*** está ativo para todas as posições com o bit menos significativo ativo, ***p2*** para todas as posições com o segundo bit menos significativo ativo e ***p4*** para todas as posições com o terceiro bit menos significativo ativo. A fórmula para o cálculo desses bits, se considerarmos paridade par, é a seguinte:

$$p_i = \bigoplus_{j=1}^{n} d_j$$

$$p_1 = d_1 \oplus d_2 \oplus d_4$$

$$p_2 = d_1 \oplus d_3 \oplus d_4$$

$$p_4 = d_2 \oplus d_3 \oplus d_4$$

O cálculo da volta segue a mesma regra recalculando os bits de paridade. Os que tiverem o valor diferente do esperado são utilizados para calcular a posição do bit errado. Se somente um bit de paridade estiver errado, o erro aconteceu exatamente nele. Se mais de um bit de paridade estiver errado, a posição deles é somada e o bit errado é o que está nessa posição. Ex.: Se ***p1*** e ***p4*** estiverem errados, o bit errado é o da posição 5 (***d2*** conforme tabela acima).

Desenvolva um circuito que calcule o código de Hamming para uma palavra de 11 bits. O circuito deve ter 11 entradas (os bits da palavra) e 15 saídas (os bits da palavra codificada). O circuito deve ser capaz de calcular o código de Hamming para qualquer palavra de 11 bits (utilize paridade par). O circuito deve ser implementado em Verilog e simulado no DigitalJS e no Icarus Verilog. Essa tarefa só será submetida junto com a tarefa 6.

Tarefa 4

Desenvolva um circuito que detecte se houve uma falha de paridade em uma palavra de 15 bits (11 bits de dados e 4 bits de paridade). O circuito deve ter 15 entradas (os 11 bits da palavra e os 4 bits de paridade) e uma saída de 11 bits correta (com o erro corrigido). O circuito deve ser capaz de detectar e corrigir a ocorrência de um erro em qualquer palavra de 15 bits. O circuito deve ser implementado em Verilog e simulado no DigitalJS e no Icarus Verilog. Essa tarefa só será submetida junto com a tarefa 6.

Colocando tudo junto

Agora vamos utilizar os componentes que você fez para testar todo o ambiente. Aqui vamos instanciar os componentes de dois em dois, o circuito que calcula o código e o que verifica. Entre eles teremos um módulo que injetará erros nos dados (inverter um bit específico do dado). Assim, se esse módulo injetar erro no circuito de paridade, o seu detector deve ser capaz de indicar que houve erro. Se o módulo injetar erro no circuito de Hamming, o seu detector deve ser capaz de

vez.

Tarefa 5

Desenvolva um circuito que injete um erro em uma palavra de 9 bits (8 bits de dados e 1 de paridade). O módulo de injeção de erro deve receber os mesmos 9 bits de entrada e gerar 9 bits de saída. Além de ter outras duas entradas, ***n*** de 4 bits que indica o bit onde injetar o erro e ***erro*** que quando tiver valor 1 injeta o erro, quando tiver o valor 0, não injeta erro. O circuito deve ser implementado em Verilog e simulado no Icarus Verilog. Veja a atividade no GitHub Classroom para submissão.

Tarefa 6

Faça algo similar para o par de codificação/decodificação de Hamming. O circuito deve ser implementado em Verilog e simulado no Icarus Verilog. Veja a atividade no GitHub Classroom para submissão. Veja a atividade no GitHub Classroom para submissão.

Conclusão

Nesse laboratório, você desenvolveu circuitos que calculam códigos de detecção e correção de erros. Note que seu circuito, apesar da descrição mais complexa, foi desenvolvido com lógica combinacional simples e que pode ser implementado em hardware real. Você pode, posteriormente, utilizar esse circuito como parte e algo maior, como um sistema de memória.

A data de entrega será até dia 13/03.

Previous
Atividade 1

Next
Atividade 3