PMR3309 - 2022
Data Lab: Manipulando Bits
Entregar até: fim do 31 de Agosto de 2023

Adaptado do Material de Bryant e O'Hallaron para o curso 15-213 da CMU

1 Introdução

O propósito deste exercício é familarizar-se com representações binárias de números inteiros e de ponto flutuantes. Para tanto serão resolvidos uma série de "puzzles". A maior parte deles impõe restrições artificais com – admitidamente – pouca aplicabilidade prática, mas elas irão expor aspectos importantes da representação binária.

2 Entrega

Este é um projeto individual. As entregas devem ser feitas eletronicamente pelo moodle da disciplina.

3 Instruções

Copie o arquivo `datalab.tar` em um diretório em uma máquina linux na qual você pretende trabalhar. O arquivo pode ser descompactado com o comando

```
unix> tar xvf datalab-handout.tar.
```

Atenção: Para este exercício, o *único* arquivo a ser modificado e entregue é o `bits.c`.

O arquivo `bits.c` contém um esqueleto para cada um dos 15 puzzles de programação. Sua tarefa é completar cada função. Para os inteiros, você deve usar apenas código *direto* (ou seja, não use desvio de fluxo como loops ou condições) e um conjunto *limitado* de operadores aritiméticos e lógicos em C. Especificamente, você pode usar apenas os seguintes operadores:

```
! ~ & ^ | + << >>
```

Atenção: Algumas funções impõe mais restrições a esta lista. Além disso, você não pode usar nenhuma constante maior do que 8 bits (de `0x00` a `0xFF`). Veja os comentários em `bits.c` para regras mais detalhadas.

3.1 Nota para usuários do MS-Windows WSL

O código deste exercício requer compilar e executar binários de 32 bits. No Microsoft Windows 10, tal recurso só é suportado no WSL 2, que acompanha versões 2004 ou posteriores do Windows.

Verifique a versão do wsl com o comando:

```
wsl -l -v
```

3.2 Nota para usuários de Ubuntu em 64 bits

Para compilar código de 32 bits é necessário instalar o pacote `gcc-multilib`. Isso vale também para usuários do Ubuntu sob WSL.

A máquina virtual da disciplina já tem os pacotes necessários.

4 Os Puzzles

4.1 Manipulação de bits

A tabela 1 descreve funções que manipulam e testam conjuntos de bits. A "Dificuldade" dá a quantidade de pontos que o puzzle vale e o "Max ops" dá o número máximo de operadores que você pode usar para implementar cada função. Veja comentários em `bits.c` para mais detalhes sobre o comportamento desejado de cada função. Você pode também verificar as funções de teste em `tests.c` (note que estas funções *violam* as restrições do problema!).

Nome	Descrição	Dificuldade	Max Ops
`bitAnd(x,y)`	`x & y` usando apenas `|` e `~`	1	8
`getByte(x,n)`	Recupera o byte `n` de `x`.	2	6
`logicalShift(x,n)`	Deslocamento à direita *lógico*.	3	20
`bitCount(x)`	Conta a quantidade de 1's em `x`.	4	40
`bang(x)`	Encontra `!n` sem o operador `!`.	4	12

Table 1: Funções de Manipulação de bits.

4.2 Aritimética de complemento de dois

A tabela 2 descreve funções que exploram a representação em complemento de dois de inteiros. Novamente, veja comentários em `bits.c` e as implementações de referência em `tests.c`.

Nome	Descrição	Dificuldade	Max Ops
`tmin()`	Menor inteiro negativo em complemento de dois	1	4
`fitsBits(x,n)`	O número `x` pode ser representado com `n` bits?	2	15
`divpwr2(x,n)`	Calcula $x/2^n$	2	15
`negate(x)`	`-x` sem o operador -	2	5
`isPositive(x)`	`x > 0`?	3	8
`isLessOrEqual(x,y)`	`x <= y`?	3	24
`ilog2(x)`	Calcula $\lfloor \log_2(x) \rfloor$	4	90

Table 2: Funções Aritiméticas

4.3 Operações com Ponto Flutuante

Nesta parte você implementará algumas operações comuns de ponto flutuante. Aqui você poderá empregar estruturas de controle convencionais (condições, loops) e pode usar variáveis do tipo `int` e `unsigned`, incluindo constantes arbitrárias. Você não pode usar nenhum tipo com unions, structs, ou vetores. Mais importante, você *não* pode usar variáveis de ponto flutuante, operações ou constantes. Todos os argumentos em ponto flutuante serão passados à função como `unsigned` e quaisquer valores de retorno em ponto flutuante serão retornados como `unsigned`. Seu código deve fazer as manipulações de bits que implementam as operações específicas de ponto flutuante.

A tabela 3 descreve o conjunto de funções que operam na representação de bits de números de ponto flutuante. Veja os comentários em `bits.c` e as implementações de referência em `tests.c` para mais informação.

Nome	Descrição	Dificuldade	Max Ops
`float_neg(uf)`	Calcula `-f`	2	10
`float_i2f(x)`	Calcula `(float) x`	4	30
`float_twice(uf)`	Calcula `2*f`	4	30

Table 3: Funções de Ponto Flutuante. O valor `f` é o número de ponto flutuante que tem a mesma representação em bits do inteiro `uf`.

As funções `float_neg` e `float_twice` devem ser capazes de tratar todos os possíveis values de pongo flutuante IEEE, incluindo *not-a-number* (NaN) e infinito. O padrão IEEE não especifica precisamente como tratar um NaN (e os processadores IA32 são "obscuros" neste aspecto). Para este exercício será seguida a convenção de que uma função que precise retornar um NaN deve usar a representação `0x7FC00000`.

O programa `fshow` ajudará a entender a representação de números com ponto flutuante. Para compilar `fshow` use o comando

```
unix> make
```

Você pode usar `fshow` para ver o que um padrão arbitrário de bits representa como número de ponto flutuante:

```
unix> ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized.  1.0000000000 X 2^(121)
```

Você pode também passar a `fshow` valores hexadecimal e fracionários e ele exibirá a sua estrutura de bits.

5 Nota

A nota será baseada em um score de no máximo 76 pontos baseados na seguinte distribuição:

41 Pontos de correção.

30 Pontos de desempenho.

5 Pontos de estilo.

Pontos de correção. Cada um dos 15 puzzles tem valor de dificuldade, entre 1 e 4. O total dos valores de dificuldade é 41. Suas funções serão avaliadas de acordo com o seu `melhor` programa (vide próxima seção). Você ganhara todos os pontos de um puzzle se conseguir passar por todos os testes e zero caso contrário.

Pontos de desempenho. Cada puzzle tem um limite máximo de operadores. Se você resolver o puzzle dentro do limite, ganha dois pontos.

Pontos de estilo. 5 pontos para soluções claras, concisas e bem documentadas.

Auto-avaliando suas soluções

Foram incluidas algumas ferramentas de auto-avaliação no material do exercício — `btest`, `dlc`, and `driver.pl` — para ajudá-lo a verificar a correção do seu trabalho.

- **`btest`:** Este programa verifica a correção das funções em `bits.c`. Para compilá-lo e usá-lo, digite os comandos:

  /
  ```
  unix> make
  unix> ./btest
  ```

  Note que você deve recompilar `btest` cada vez que modificar o arquivo `bits.c`.

  Você provavelmente irá preferir trabalhar nas funções uma por vez. Você pode usar a opção `-f` para instruir `btest` a testar apenas uma função:

```
unix> ./btest -f bitAnd
```

Você pode passar argumentos específicos de função usando as opções `-1`, `-2`, and `-3`:

```
unix> ./btest -f bitAnd -1 7 -2 0xf
```

Verifique o arquivo `README` para a documentação do programa `b`test.

- **`dlc`:** Esta é uma versão modificada de um compilador C ANSI do grupo CILK do MIT. Você pode usar esta ferramenta para verificar a adequação da sua solução às regras de cada puzzle. O uso típico é:

  ```
  unix> ./dlc bits.c
  ```

  Este programa roda sem mensagens, a menos que identifique um problema como um operador ilegal, excesso de operadores ou uso de controle de fluxo em puzzles inteiros.

  A opção `-e`:

  ```
  unix> ./dlc -e bits.c
  ```

  faz com que `dlc` mostre a contagem de operadores em uso em cada função. Digite `./dlc -help` para uma lista de opções de linha de comando.

- **`driver.pl`:** Este é um programa *driver* que usa `btest` e `dlc` para calcular os pontos de correção e desempenho para a sua solução. Ele não usa nenhum argumento:

  ```
  unix> ./driver.pl
  ```

6 Instruções para Entrega

Você deve entregar *somente* o arquivo `bits.c` no moodle da disciplina PMR3309. Certifique-se de que seu código está documentado apropriadamente.

Sugestões

- Não inclua o arquivo `<stdio.h>` em seu arquivo `bits.c`, pois ele interfere no funcionamento de `dlc` e causa mensagens de erro não-intuitivas. Você pode usar a função `printf` dentro do arquivo `bits.c` para depurar seu código mesmo sem incluir `<stdio.h>`. O compilador gcc irá gerar uma mensagem de warning que você pode ignorar..
- O programa `dlc` impõe regras de em declarações da linguagem C mais estritas do que as do compilador `gcc`. Em particular, todas as declarações devem aparecer dentro de um bloco antes de qualquer linha que não seja uma declaração. Por exemplo, ele não aceitará o seguite código:

```
int foo(int x)
{
  int a = x;  /* Declaração */
  a *= 3;     /* Não é uma declaração */
  int b = a;  /* Declaração: ERRO, esta declaração não é permitida aqui */
}
```