CET-088, I Semestre de 2017
Data Lab: Manipulando Bits
Enviado: Quarta, 17 de Maio, Entrega: Quarta, 31 de Maio, 23:59

Leard Fernandes (`lofernandes@uesc.br`) é o responsável por esta atividade

1 Introdução

A proposta desta tarefa é torná-lo familiarizado com representações de números inteiros e ponto flutuante ao nível de bit. Você irá fazer isso resolvendo uma série de "Quebra-Cabeças." Muitos desses quebra-cabeças são artificialmente produzidos, contudo, você se encontrará pensando muito mais sobre os bits durante o seu caminho por este trabalho.

2 Logistica

Este é um projeto individual. Todas as tarefas são eletrônicas. Esclarecimentos e correções serão postados na página do Curso/Grupo.

3 Instruções da Tarefa

Inicie copiando o arquivo `datalab-handout.tar` para um diretório protegido numa máquina Linux no qual você planeje realizar o seu trabalho. Então dê o comando

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

Isso irá causar o desempacotamento dos arquivos no diretório. O único arquivo que você irá trabalhar e modificar será `bits.c`.

O arquivo `bits.c` contêm o esqueleto para cada um dos 15 quebra-cabeças de programação. A sua tarefa será completar cada esqueleto de função utilizando apenas um conjunto restrito de *regras de codificação*. Você deverá utilizar apenas código sequêncial (*straightline code*) para o quebra-cabeças de inteiros (i.e., sem laços de repetição e estruturas condicionais) e um número limitado de operadores aritméticos e lógicos do C. Especificamente, você *somente* poderá utilizar os seguintes oito operadores:

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

Algumas das funções restringem ainda mais esta lista. Além disso, você não tem permissão para usar quaisquer constantes maiores que 8 bits. Veja os comentários em `bits.c` para regras detalhadas e uma discussão do estilo de codificação desejada.

4 O Quebra-cabeças

Está seção descreve os quebra-cabeças que você irá resolver em `bits.c`.

4.1 Manipulação de Bits

A Tabela 1 descreve um conjunto de funções que manipulam e testam os conjuntos de bits. O campo "Nível" apresenta o nível de dificuldade (o número de pontos) para o quebra-cabeça, e o campo "Max ops" apresenta o número máximo de operadores que você possui permissão de utilizar para implementar cada função. Veja os comentários em `bits.c` para mais detalhes sobre o comportamento desejado das funções. Você pode consultar as funções de teste em `tests.c`. Os testes são utilizados como funções de referência para expressar o comportamento correto de suas funções, embora eles não satisfaçam as regras de codificação para suas funções.

Nome	Descrição	Nível	Max Ops
`bitAnd(x,y)`	`x & y` utilizando apenas `|` e `~`	1	8
`getByte(x,n)`	Retorna o byte `n` de `x`.	2	6
`logicalShift(x,n)`	Deslocamento lógico à direita.	3	20
`bitCount(x)`	Conta o número de 1's em `x`.	4	40
`bang(x)`	Computa `!n` sem utilizar o operador `!`.	4	12

Tabela 1: Funções de Manipulação ao Nível de Bit.

4.2 Aritmética em Complemento de Dois

A Tabela 2 descreve um conjunto de funções que utilizam a representam de inteiros em complemento de dois. Novamente, consultem os comentários em `bits.c` e as versões de referência em `tests.c` para mais informações.

4.3 Operações em Ponto Flutuante

Para esta parte da terefa, você deverá implmementar algumas operações comuns de ponto flutuante de precisão simples. Nesta seção, você têm permissão para utilizar as estrutura de controle padrão (condicionais e repetição), e poderá utilizar ambos os tipos de dados `int` e `unsigned`, incluindo constantes arbitrárias unsigned e inteiras. Você não poderá utilizar uniões (unions), estruturas (structs) ou vetores. O importante

Nome	Descrição	Nível	Max Ops
`tmin()`	Retorna o menor inteiro em complemento de dois	1	4
`fitsBits(x,n)`	`x` cabe em `n` bits?	2	15
`divpwr2(x,n)`	Compute $x/2^n$	2	15
`negate(x)`	`-x` sem negação	2	5
`isPositive(x)`	`x > 0`?	3	8
`isLessOrEqual(x,y)`	`x <= y`?	3	24
`ilog2(x)`	Compute $\lfloor \log_2(x) \rfloor$	4	90

Tabela 2: Funções Aritméticas

é que você não poderá utilizar qualquer tipo de dado ponto flutuante, operações, ou constantes. Ao invés disso, qualquer operando ponto flutuante será passado para a função como um tipo `unsigned`, e qualquer valor ponto flutuante retornado será do tipo `unsigned`. O seu código deverá realizar a manipulação dos bits que implementam as operações em ponto flutuante especificadas.

A Tabela 3 descreve um conjunto de funções que operam ao nível de bit representações de números em ponto flutuante. Consulte os comentários em `bits.c` e as versões de referência em `tests.c` para mais informações.

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

Tabela 3: Funções Ponto Flutuante. Valor `f` é o número ponto flutuante com a mesma representação de bit como um inteiro unsigned `uf`.

As Funções `float_neg` e `float_twice` devem manipular o maior intervalo possível dos valores de argumento, incluindo not-a-number (NaN) e infinito. O padrão IEEE não especifica precisamente como manipular NaN's e o comportamento IA32 é um pouco obscuro. Nós iremos seguir uma conveção que para qualquer função que retorne uma valor NaN, ela irá retornar uma representação de bit com valor `0x7FC00000`.

O programa `fshow` ajudará você entender a estrutura dos números ponto flutuantes. Para compilar `fshow`, vá para o diretório da tarefa e digite:

```
unix> make
```

Você pode utilizar `fshow` para ver o que um padrão arbitrário representa como um número 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 também pode passar para `fshow` valores em hexadecimal e ponto flutuante, e ele irá decifrar sua strutura de bit.

5 Avaliação

Sua nota será computada de um máximo de 76 pontos baseados na seguinte distribuição:

41 Pontos de correção.

30 Pontos de performance.

5 Pontos de estilo.

Pontos de correção. Os 15 quebra-cabeças devem ser resolvidos por você num nível de dificuldade entre 1 e 4, tal que as somas dos pesos totalize 41. Suas funções serão avaliadas utilizando o programa `btest`, que será descrito na próxima seção. Você irá obter crédito total para um quebra-cabeça se ele passar por todos os teste realizados por `btest`, e nenhum crédito caso contrário.

Pontos de performance. A principal preocupação neste ponto do curso é que você pode obter a resposta correta. Entretanto, é objetivo instigar em você uma sensação de manter as coisas tão curtas e simples o quanto você puder. Além disso, alguns dos quebra-cabeças podem ser resolvidos pela força bruta, mas espera-se que você seja mais inteligente. Assim, para cada função foi estabelecido um número máximo de operadores que você poderá usar para cada função. Este limite é muito generoso e destina-se apenas a capturar soluções ineficazes. Você receberá dois pontos para cada função correta que satisfaça o limite de operador.

Pontos de estilo. Finalmente, foi reservado 5 pontos para uma avaliação subjetiva para o estilo de sua solução e comentários. Suas soluções deverão ser limpas e diretas o quanto possível. Seus comentários deverão ser informativos, mas não precisam ser extensos.

Autoavaliando o seu trabalho

Foi incluido algumas ferramentas de autoavaliação no diretório da atividade `btest`, `dlc`, e `driver.pl` — para ajudá-lo checar a correção do seu trabalho.

- **`btest:`** Este programa checa a correção funcional das funções em `bits.c`. Para construir (build) e usá-lo digite os seguintes comandos:

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

Observer que você deverá reconstruir `btest` cada vez que você modificar o seu arquivo `bits.c`.

Você achará útil trabalhar com as funções uma a uma, testando cada uma como quiser. Você pode usar o flag `-f` para instruir `btest` testar apenas uma única função:

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

Você pode passar argumentos para uma função especifica utilizando a opção flags `-1`, `-2`, and `-3`:

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

Confira o arquivo `README` para documentação sobre executar o programa `b`test.

- **`dlc`:** Esta é uma versão modificada do compilador ANSI C do grupo MIT CILK, você poderá checar a conformidade das regras dos códigos de cada quebra-cabeças. O uso pode ser:

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

  O programa roda silenciosamente, ao menos que detecte um problema, tais como: operador ilegal, muitos operadores, ou código não sequêncial no quebra-cabeças de inteiros. Rodando com o parâmetro `-e` switch:

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

  o `dlc` irá imprimir a contagem de operadores utilizados em cada função Digite `./dlc -help` para uma lista de comandos de linha.

- **`driver.pl`:** Este é um programa guia que utiliza `btest` e `dlc` para computar os pontos de performance e correção para sua solução. Não possui argumentos:

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

  Seu instrutor irá utilizar `driver.pl` para avaliar sua solução.

6 Instruções de Entrega

Você deverá enviar até a data da entrega para o email do instrutor o seu arquivo `bits.c`

Notas:

- Tenha certeza da correção de seu arquivo, somente o primeiro envio será avaliado.
- Você deve enviar o seu arquivo sem qualquer impressão extra (utilizada por você durante um debug, etc).

7 Aviso

- Não incluam a biblioteca `<stdio.h>` em seu arquivo `bits.c`, ele confude o `dlc` e os resultados em algumas mensagens de erro não são intuitivas. Você poderá utilizar `printf` em seu arquivo de programa `bits.c` para depuração sem utilizar a biblioteca `<stdio.h>`, apesar do `gcc` imprimir uma mensagem que você pode ignorar.

- O programa `dlc` emprega uma forma rígida de declarações do que no caso do C++, Java ou mesmo que é aplicada pelo `gcc`. Em particular, qualquer declaração deve aparece num bloco (o que você colocar em entre chaves) antes de qualquer declaração que não seja uma declaração. Por exemplo, ele irá reclamar sobre o seguinte código:

```
int foo(int x)
{
  int a = x;
  a *= 3;     /* Statement that is not a declaration */
  int b = a;  /* ERROR: Declaration not allowed here */
}
```