Universidade Federal
do Espírito Santo

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Arquitetura e Organização de Computadores – Turmas 01 e 02 – 2022/2

Prof. Rodolfo da Silva Villaça – rodolfo.villaca@ufes.br

Laboratório III – Aritmética Digital

1. Objetivo

- Entender como funcionam as operações aritméticas com números inteiros no MIPS;
- Conhecer as operações de ponto flutuante e sua representação numérica;
- Usar as instruções de ponto flutuante.

Adição e Subtração

Adição e subtração são realizadas em números de 32 bits armazenados em registradores de propósito geral (cada um com 32 bits, numerados de ($0 a $31). O resultado da execução da instrução é sempre um número de 32 *bits*. Dois tipos de instruções estão incluídas no conjunto de instruções MIPS para fazer a adição e subtração:

- Instruções para aritmética "com sinal": os números de 32 *bits* são representados sob a forma de valores em complemento de 2. Neste caso, a execução da instrução de adição ou subtração pode gerar indicação de *overflow*, isto é, o resultado não pode ser representado em 32 *bits*;
- Instruções para aritmética "sem sinal": os números de 32 *bits* são representados sob a forma de valores padrão em binário. Neste caso, a execução deste tipo de instrução **nunca gerará indicação de *overflow***.

A multiplicação pode gerar resultados maiores que a faixa de valores representáveis em 32 *bits* [-2^{31} .. $2^{31}-1$]. A divisão inteira retorna dois valores de 32 *bits*, resto e quociente, que exigem 32 *bits*, cada, para sua representação. A arquitetura MIPS fornece 2 registradores especiais de 32 *bits* que são destino para as operações de multiplicação e divisão. Estes registradores extras, chamados *hi* e *lo*, são destinos implícitos nestas instruções *mult* e *div* do conjunto de instruções do MIPS. O registrador *hi* armazena os *bits* de mais alta ordem do resultado da multiplicação (*bits* 63 a 32) e, no caso da divisão, o resto da divisão de dois inteiros. O registrador *lo* armazena os *bits* de mais baixa ordem (*bits* 31 a 0) do resultado da multiplicação e, no caso da divisão, o quociente da divisão de dois inteiros. Duas instruções especiais são oferecidas pelo conjunto de instruções MIPS para movimentar os dados dos registradores *hi* e *lo* para registradores de propósito geral.

Instrução MIPS	Semântica
mfhi Rdest #move from	$hi → Rdest
mflo Rdest	$lo → Rdest
mthi Rsrc #move to	$hi ← Rsrc
mtlo Rsrc	$lo ← Rsrc

Dois tipos de instruções estão incluídas no conjunto de instruções MIPS para fazer a multiplicação e divisão:

- Instruções para aritmética "com sinal" (*signed numbers*): os números de 32 *bits* são representados sob a forma de valores em complemento de 2. A execução da instrução de multiplicação nunca gera *overflow*, mas a divisão pode gerar *overflow*. Note que a CPU não gera um sinal de *overflow* nesse último caso. O *overflow* deve ser resolvido por software (lembre-se de que o programador deve evitar, por exemplo, a divisão por zero);
- Instruções para aritmética "sem sinal" (*unsigned numbers*): os números de 32 *bits* são representados sob a forma de valores padrão em binário. O *bit* 31 é o bit mais significativo do número armazenado. Neste caso, a execução deste tipo de instrução **nunca gerará um sinal de *overflow***.

Um *overflow* ocorrerá quando o resultado de uma operação aritmética não puder ser representado na quantidade de *bits* disponíveis para armazená-lo. No caso MIPS, o resultado com sinal deve poder ser representado em 32 *bits.*

Um tipo inteiro "sem sinal" identifica um dado que sempre terá um valor inteiro e positivo. Com *n bits* para representação, o menor número sem sinal representável com este tipo é 0 e o maior, 2^n-1. Em programas C/C++, as

variáveis inteiras que nunca receberão valores negativos poderiam ser declaradas como *unsigned int*. Inteiros "com sinal" podem ser negativos ou positivos. Com *n bits* para representação e com os negativos representados em complemento de 2, o menor número sem signal representável é -2^{n-1} e o maior, $2^{n-1}-1$. No nível de programação C/C++ este tipo de diferença é normalmente ignorado, em especial porque os programadores não precisam utilizar valores próximos do início e do fim do intervalo de representação (por exemplo, contadores de interação normalmente armazenam valores muito menores do que $2^{n-1}-1$).

Entretanto, é simples verificar que declarar uma variável "sempre positiva" como *int* (quando ela de fato deveria ter sido representada como *unsigned int*) reduz a faixa de valores representáveis à metade da escala. Por exemplo, considere o código C a seguir:

```
// Assuma inteiros em 32 bits
unsigned int ex1;        // ex1 entre [0..4294967295]
int ex2;                 // ex2 entre [-2147483648..2147483647]
```

Se a variável *ex2* for sempre positiva, então a faixa de valores possíveis para esta variável ficará entre 0 e 2147483647, ou seja, a metade dos valores possíveis que um inteiro sem sinal (*unsigned int*) teria.

Atividade 1

Um número negativo é representado como um padrão de *bits* no qual o *bit* mais significativo é 1 (considerando a representação em complemento de 2). O mesmo padrão de *bits* estaria associado à representação de um valor inteiro elevado se este padrão fosse tratado como a representação de um valor "sem sinal". Preencha a tabela a seguir com os valores inteiros representados em 16 *bits* a seguir.

Padrão de bits	Inteiro com sinal	Inteiro sem sinal
0x0000	0	0
0x7fff		
0x8000		
0xffff		

Atividade 2

Note que a extensão da representação de 16 para 32 *bits* pode ser feita de maneira simples, copiando o *bit* mais significativo da representação com menos *bits* para todos os *bits* de mais alta ordem incluídos na representação estendida. Reescreva a tabela anterior, considerando que a representação dos padrões de *bits* anteriores estariam agora representadas em 32 *bits*, conforme o exemplo da primeira linha da tabela.

Padrão de bits	Inteiro com sinal	Inteiro sem sinal
0x00000000	0	0

No nível da linguagem de montagem (*assembly* MIPS) a diferença entre um número com sinal e sem sinal é mais sutil. Muitas instruções aritméticas com números com sinal podem gerar *overflow* e este *overflow* será sinalizado ao "controle da máquina" (o sistema operacional). As instruções que fazem aritmética com inteiros "sem sinal" por outro lado, ignorarão um *overflow* mesmo se a operação gerar um resultado não representável na quantidade de bits disponível para armazená-lo.

Atividade 3

Escreva um programa usando MIPS e o simulador MARS, denominado *lab3.2.asm*, que:

- Declare 2 variáveis inteiras, a e b. Estas variáveis serão usadas para armazenar 2 inteiros digitados pelo usuário, os quais serão armazenados em *$t0* e *$t1*;
- Some *$t0* e *$t1* usando uma instrução nativa para inteiros com sinal e insira o resultado em *$t2*;
- Imprima o resultado da soma.

Rode o programa e preencha a tabela a seguir com seu "plano de testes". Escolha valores que gerem *overflow*. Na coluna "Overflow" use "S/N" para indicar que houve (não houve) overflow. Na coluna "Comentários" descreva o motivo da ocorrência do *overflow*.

Importante: escolha os números cuidadosamente para seus testes serem efetivos. A chamada de sistema que lê um inteiro do usuário truncará um valor muito grande que ultrapassaria o limite do registrador (32 *bits*). Tente colocar um número inteiro muito grande (exemplo 8.589.934.593) e observe o resultado da execução.

Oper_1	Oper_2	Resultado Impresso	Overflow	Justificativa
+2000000000	+2000000000		Y	
+	–			
–	+			
–	–			

Atividade 4

Escreva um programa usando MIPS e o simulador MARS, denominado *lab3.3.asm* (muito similar ao anterior), que:

- Declare 2 variáveis inteiras, a e b. Estas variáveis serão usadas para armazenar 2 inteiros digitados pelo usuário, os quais serão armazenados em *$t0* e *$t1*;
- Subtraia *$t0* e *$t1* usando uma instrução nativa para inteiros com sinal e insira o resultado em *$t3*;
- Imprima o resultado da subtração.

Rode o programa e preencha a tabela a seguir com seu "plano de testes". Sempre que possível, escolha valores que gerem *overflow*. Na coluna "Overflow" use "S/N" para indicar que houve (não houve) *overflow*. Na coluna "Comentários" descreva o motivo da ocorrência do *overflow*.

Oper_1	Oper_2	Resultado Impresso	Overflow	Justificativa
+2000000000	+2000000000		Y	
+	–			
–	+			
–	–			

Atividade 5

Agora, vamos testar o uso de instruções que desconsideram o *overflow*. O simulador MARS sempre imprime o conteúdo dos registradores de propósito geral como sendo um valor inteiro com sinal. O fato de que um valor válido "muito grande" sendo representado nos 32 *bits* e sendo impresso como um valor negativo poderia confundir um programador menos avisado. Os valores tratados como "*unsigned numbers*" podem usar o *bit* mais significativo como sendo um valor e não a representação do sinal. Entretanto, eles serão interpretados como negativos pelo serviço de impressão do sistema operacional.

Escreva agora um programa usando MIPS e o simulador MARS, denominado *lab3.4.asm* (também similar aos anteriores) para:

- Declarar 2 variáveis inteiras, a e b. Estas variáveis serão usadas para armazenar 2 inteiros digitados pelo usuário, os quais serão armazenados em *$t0* e *$t1*;
- Somar *$t0* e *$t1* usando uma instrução nativa para inteiros sem sinal (*unsigned*) e armazenar o resultado em *$t2*;
- Imprima o resultado da adição;
- Subtrair *$t0* e *$t1* usando uma instrução nativa para inteiros sem sinal (*unsigned*) e armazenar o resultado em $t3;
- Imprima o resultado da subtração.

Rode o programa e preencha a tabela a seguir com seu "plano de testes". Sempre que possível, escolha valores que gerem *overflow*. Na coluna "Overflow" use "S/N" para indicar que houve (não houve) *overflow*. Na coluna "Comentários" descreva o motivo da ocorrência do *overflow*.

Oper_1	Oper_2	Resultado Impresso	Overflow	Comentários (Justificativas)
Adição				
+	+			
+	–			
–	+			
–	–			

Subtração				
+	+			
+	–			
–	+			
–	–			

Multiplicação e Divisão

A multiplicação de inteiros pode ser feita usando números com e sem sinal. A arquitetura especifica 2 registradores especiais, *hi* and *lo* (32 *bits* cada), os quais são os destinos dos resultados de divisão e multiplicação de inteiros. Como visto, a multiplicação inteira pode gerar resultados maiores que a faixa de valores representáveis em 32 *bits* [-2^{31} .. $2^{31}-1$]. A divisão inteira retorna dois valores de 32 *bits*, resto e quociente, que exigem 32 *bits*, cada, para sua representação.

Multiplicar 2 valores inteiros "*unsigned*" de *n bits* pode resultar em valores que requerem 2*n bits* para serem representados. Dividir 2 valores inteiros "*unsigned*" de *n bits* resulta em 2 valores de *n bits*, um para o quociente e outro para o resto da divisão. Para multiplicação com valores inteiros "*signed*", os valores negativos estarão representados em complemento de 2. O resultado, se for negativo, também estará representado em complemento de 2. Assim, o resultado de uma multiplicação "com sinal" requer 2(*n* – 1) *bits* para a representação da magnitude do valor e mais 1 *bit* de sinal.

A multiplicação inteira com ou sem sinal nunca requererá mais que 64 *bits* para o resultado e, portanto, nunca ocorrerá um *overflow* quando instruções nativas de multiplicação inteira forem utilizadas. As instruções nativas para a multiplicação são as seguintes:

Instruction	Comment
`mult Rsrc1, Rsrc2`	Multiply the signed numbers in `Rsrc1` and `Rsrc2`. The higher 32 bits of the result go in register **`hi`**. The lower 32 bits of the result go in register **`lo`**
`multu Rsrc1, Rsrc2`	Multiply the unsigned numbers in `Rsrc1` and `Rsrc2`. The higher 32 bits of the result go in register **`hi`**. The lower 32 bits of the result go in register **`lo`**

Por outro lado, a máquina virtual do simulador oferece várias instruções de multiplicação que produzem um resultado em 32 *bits*, com o registrador destino especificado como sendo um registrador de uso geral (e não os registradores especiais *hi* e *lo*).

Pseudoinstruções Nativas para Multiplicação Inteira em MIPS

Instruction	Comment
`mul Rdest, Rsrc1, Rsrc2`	Multiply the signed numbers in `Rsrc1` and `Rsrc2`. The lower 32 bits of the result go in register `Rdest`.
`mulo Rdest, Rsrc1, Rsrc2`	Multiply the signed numbers in `Rsrc1` and `Rsrc2`. The lower 32 bits of the result go in register `Rdest`. Signal overflow
`mulou Rdest, Rsrc1, Rsrc2`	Multiply the unsigned numbers in `Rsrc1` and `Rsrc2`. The lower 32 bits of the result go in register `Rdest`. Signal overflow

O resultado de uma divisão inteira é composto por um quociente (armazenado no registrador *lo*) e um resto (armazenado no registrador *hi*), sendo ambos números inteiros de 32 *bits*. Existe uma complicação extra, relacionada com a divisão inteira: o sinal do resto.

Existem duas abordagens para tratar o problema:

1) Seguir o teorema de divisão da matemática; ou
2) usar uma solução baseada na visão da ciência da computação.

Dividir um inteiro *DD* (dividendo) por um *valor positivo DR* (divisor), as seguintes condições devem ser sempre verdadeiras (teorema da divisão):

$$DD = DR*Q + RM \qquad (1)$$
$$0 <= RM < DR \qquad (2)$$

Observe que a divisão gera um *Q* (quociente) e um *RM* (resto – *remainder*) que são inteiros únicos e que o resto é sempre positivo. Assim, se o dividendo e o divisor tiverem o mesmo sinal, então o quociente é positivo; do contrário, será negativo. Um resto diferente de zero sempre terá o mesmo sinal do dividendo.

Atividade 6

Quais são os resultados das seguintes divisões inteiras?

Dividendo	Divisor	Quociente	Resto
22	7		
–22	7		
22	–7		
–22	–7		

No MIPS, se um dos operandos numa divisão inteira for negativo, então o sinal do resto é “não especificado”. Para se obter o resultado correto da divisão, alguns passos extras devem ser executados:

- Converter ambos os operandos em valores positivos;
- Realizar a divisão;
- Ajustar o resultado para a representação correta, tendo como base os sinais iniciais do dividendo e do divisor.

Uma operação de divisão com sinal pode gerar *overflow*, uma vez que a representação em complemento de 2 de inteiros é assimétrica (a representação do 0 fica do lado das representações positivas e, com isso existe um número negativo a mais do que os valores positivos). Note entretanto, que um *overflow* numa divisão inteira não gera um sinal de *overflow* como no caso da adição/subtração. Ou seja, fica a cargo do programador o tratamento deste erro por meio de um código próprio para tratar o *overflow.*

Dividir um valor inteiro por zero é uma operação ilegal! Entretanto, mesmo que o divisor seja zero, a operação de divisão será feita **sem reportar um erro de execução**. Novamente, cabe ao compilador ou ao programador a tarefa de gerar um código que detecta e trata este tipo de operação ilegal antes que ela aconteça.

Instruções Nativas para Divisão Inteira em MIPS

Instruction	Comment
`div Rsrc1, Rsrc2`	Divide the signed integer in `Rsrc1` by the signed integer in `Rsrc2`. The quotient (32 bits) goes in register **`lo`**. The remainder goes in register **`hi`**. Can overflow.

Instruction	Comment
`divu Rsrc1, Rsrc2`	Divide the unsigned integer in `Rsrc1` by the unsigned integer in `Rsrc2`. The quotient (32 bits) goes in register **`lo`**. The remainder goes in register **`hi`**. Does not overflow.

Pseudoinstruções para Divisão Inteira em MIPS

Instruction	Comment
`div Rdest, Rsrc1, Rsrc2`	Divide the signed integer in `Rsrc1` by the signed integer in `Rsrc2`. Store the quotient (32 bits) in register `Rdest`. Can overflow.
`divu Rdest, Rsrc1, Rsrc2`	Divide the unsigned integer in `Rsrc1` by the unsigned integer in `Rsrc2`. Store the quotient in Rdest. Does not overflow.
`rem Rdest, Rsrc1, Rsrc2`	Divide the signed integer in `Rsrc1` by the signed integer in `Rsrc2`. Store the remainder[a] (32 bits) in register `Rdest`. Can overflow.
`remu Rdest, Rsrc1, Rsrc2`	Divide the signed integer in `Rsrc1` by the signed integer in `Rsrc2`. Store the remainder[a] (32 bits) in register `Rdest`. Does not overflow.

Se um dos operandos for negativo, o resto terá sinal indefinido. Assim como na multiplicação, a máquina virtual do MARS oferece pseudoinstruções de divisão que podem ser armazenadas em registradores de propósito geral (e não os registradores especiais *hi* e *lo*).

Atividade 7

Escreva um programa MIPS, usando o simulador MARS, chamado *lab3.4.asm*, no qual você:

- Declara 2 variáveis inteiras, a e b. Estas variáveis serão usadas para armazenar 2 inteiros digitados pelo usuário, os quais serão armazenados em *$t0* e *$t1*;
- Multiplica *$t0* e *$t1* usando uma instrução nativa para inteiros com sinal;
- Imprime o resultado da multiplicação.

Execute o programa e preencha a tabela a seguir com seu plano de testes. Use as ferramentas do simulador MARS para verificar o conteúdo dos registradores hi e lo. Inclua o maior e menor inteiro representável em 32 *bits* para os valores as duas variáveis a e b nas duas últimas linhas da tabela.

Multiplicação com sinal

a	b	Registrador *hi* (hexadecimal)	Registrador *lo* (hexadecimal)
2	1		
2	–1		

2^{18}	2^{14}		
2^{18}	-2^{14}		

Preencha a tabela usando os mesmos valores anteriores com os resultados esperados e obtidos nesta execução.

Multiplicação com sinal

a	b	Resultado Esperado	Resultado Obtido
2	1		
2	–1		
2^{18}	2^{14}		
2^{18}	-2^{14}		

Alguns dos resultados impressos não estão corretos. Por quê?

Atividade 8

Escreva um programa, chamado *lab3.5.asm*, no qual você:

- Declara 2 variáveis inteiras, a e b. Estas variáveis serão usadas para armazenar 2 inteiros digitados pelo usuário, os quais serão armazenados em *$t0* e *$t1*;
- Multiplica *$t0* e *$t1* usando uma instrução nativa para inteiros sem sinal;
- Imprime o resultado da multiplicação.

De forma similar ao que foi feito anteriormente, execute o programa e preencha a tabela a seguir com seu plano de testes. Inclua o maior e menor inteiro representável em 32 *bits* para os valores as duas variáveis a e b nas duas últimas linhas da tabela.

Multiplicação sem sinal

a	b	Registrador *hi* (hexadecimal)	Registrador *lo* (hexadecimal)
2	1		
2	–1		
2^{18}	2^{14}		
2^{18}	-2^{14}		

Repita a execução do programa *lab3.5.asm* usando os mesmos valores anteriores e preencha a tabela a seguir com os resultados esperados e obtidos nesta execução.

Multiplicação sem sinal

a	b	Resultado Esperado	Resultado Obtido
2	1		
2	–1		
2^{18}	2^{14}		
2^{18}	-2^{14}		

Atividade 9

Escreva um programa usando MIPS e o simulador MARS, chamado *lab3.6.asm*, no qual você:

- Declara 2 variáveis inteiras, a e b. Estas variáveis serão usadas para armazenar 2 inteiros digitados pelo usuário, os quais serão armazenados em *$t0* e *$t1*;
- Divida *$t0* e *$t1* usando uma instrução nativa para inteiros com sinal;
- Imprime o quociente e o resto da divisão.

Execute o programa e preencha a tabela a seguir com seu plano de testes. Use as ferramentas do simulador MARS para verificar o conteúdo dos registradores *hi* e *lo*. Use números "pequenos" que gerem resto diferente de zero. Para as últimas duas linhas, insira valores em *$t0* e *$t1* que gerem *overflow* na execução da divisão. Note que divisão por 0 (que é uma operação ilegal) e *overflow* são coisas diferentes em termos de execução. Na coluna "Erro" indique quando um erro de execução for observado.

Divisão com sinal

$Oper_1$	$Oper_2$	Quociente impresso	Resto impresso	Erro
> 0	> 0			
> 0	< 0			
< 0	> 0			
< 0	< 0			
–1	2			
	= 0			

Atividade 10

Escreva um programa usando MIPS e o simulador MARS, chamado *lab3.7.asm*, no qual você:

- Declara 2 variáveis inteiras, a e b. Estas variáveis serão usadas para armazenar 2 inteiros digitados pelo usuário, os quais serão armazenados em *$t0* e *$t1*;
- Divida *$t0* e *$t1* usando uma instrução nativa para inteiros sem sinal;
- Imprime o quociente e o resto da divisão.

De maneira similar ao feito para a divisão de inteiros com sinal, execute o programa e preencha a tabela a seguir com seu plano de testes, usando os mesmos valores anteriores de entrada dos operandos.

Divisão sem sinal

$Oper_1$	$Oper_2$	Quociente impresso	Resto impresso	Erro
> 0	> 0			
> 0	< 0			
< 0	> 0			
< 0	< 0			
–1	2			
	= 0			

Representação em Ponto Flutuante

A aritmética de Ponto Flutuante é implementada pelo coprocessador 1 (*Coproc 1* no MARS) da arquitetura MIPS. O coprocessador possui 32 registradores de 32 *bits*, numerados de 0 a 31 (*$f0* a *$f31*). Os valores armazenados nestes registradores seguem o padrão IEEE-754. O padrão IEEE-754 reserva vários padrões de *bits* para ter um significado especial. Em outras palavras, nem todos os padrões de bits representam algum número, conforme quadro a seguir.

Special bit patterns in IEEE-754

Sign bit	Exponent	Significand	Comment
x	0..0	0..0	Zero
x	0..0	not all zeros	Denormalized number
0	1..1	0..0	Plus infinity (+inf)
1	1..1	0..0	Minus infinity (-inf)
x	1..1	not all zeros	Not a Number (NaN)

Infinito significa algo grande demais para ser representado. Um estouro de representação pode retornar um + inf ou um -inf. Algumas operações no infinito retornam outro infinito como resultado. Um resumo dessa representação encontra-se na Tabela 1.

Table 1: Some operations with infinity

Operation	Result	Comment
`x + ( inf)`	`inf`	x finite
`x - (+inf)`	`-inf`	x finite
`(+inf) + (+inf)`	`+inf`	
`(-inf) + (-inf)`	`-inf`	
`x * (+inf)`	`+inf if x>0, -inf otherwise`	x nonzero

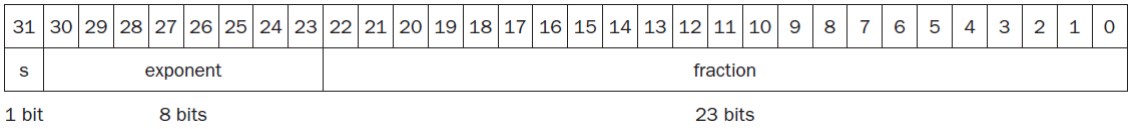
Pode haver um zero positivo se o *bit* de sinal for 0 e um zero negativo (*bit* de sinal é 1). Números não normalizados estão incluídos no padrão IEEE-754 para lidar com casos de estouro negativo de expoente (números muito pequenos). Um NaN (às vezes denotado por nan) é usado para representar um resultado indeterminado. Existem dois tipos de NaNs: sinalização e silêncio.

O padrão de bits no significando é usado para diferenciar entre eles, e é dependente da implementação. Um NaN de sinalização pode ser usado, por exemplo, para variáveis não inicializadas. Observe que qualquer operação em um NaN de sinalização terá como resultado, um NaN silencioso. Operar em um NaN silencioso simplesmente retorna outro NaN sem gerar nenhuma exceção, vide Tabela 2.

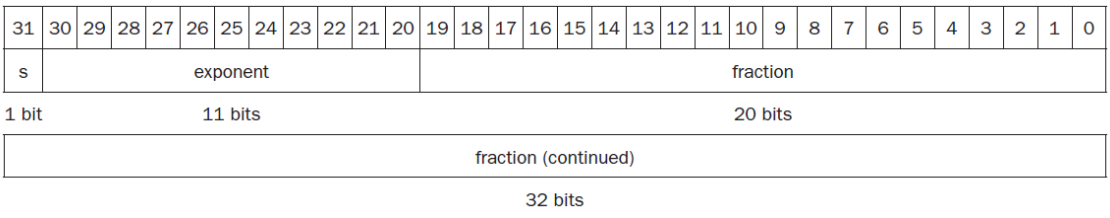
Table 2: Some operations which produce a quiet NaN

Operation	Comment
`x + (NaN)`	Any operation on a quiet NaN (addition in this example)
`(+inf) + (-inf)`	
`0 * ( inf)`	
`0/0`	
`inf/inf`	
`x%0`	The remainder of division by 0
$\sqrt{x}$, `x < 0`	

Para permitir o armazenamento de valores em precisão dupla (64 *bits*), a arquitetura usa o artifício de agrupar pares de registradores subsequentes, isto é, *$f0* e *$f1*, *$f2* e *$f3*, e assim, sucessivamente. Assim, a arquitetura oferece "virtualmente" 16 registradores de 64 *bits*, nos quais o valor armazenado terá sempre seus 32 *bits* de mais alta ordem armazenados num registrador par e os 32 *bits* de mais baixa ordem num registrador ímpar. A representação em IEEE-754 para 32 e 64 *bits* é mostrada a seguir:



Representação IEEE-754 em precisão simples (32 bits)



Representação IEEE-754 em precisão dupla (64 bits)

Para simplificar as coisas, operações em ponto flutuante usam sempre registradores numerados e podem ser executadas em precisão simples (32 bits) ou precisão dupla (64 *bits*). A diferença das instruções MIPS é determinada por um sufixo após o mnemônico da instrução, por exemplo, as instruções *add.s* e *add.d* representam adições em precisão simples (.s) e precisão dupla (.d), respectivamente.

O conjunto de instruções MIPS oferece, além das instruções aritméticas, comparações, desvios, movimentação de dados da (*load*) e para (*store*) a memória, conversões de formatos de ponto flutuante (32 para 64 *bits* e vice-versa) e conversão de inteiro para ponto flutuante e vice-versa. Enquanto nas comparações envolvendo inteiros um dos registradores de propósito geral pode ser definido como destino da execução, no caso de ponto flutuante, uma comparação irá implicitamente determinar ("setar") uma *flag*. Esta *flag* poderá então ser usada para testar se um desvio é realizado ou não numa instrução de desvio condicional.

Instruction	Comment
`mfc1 Rdest, FPsrc`	Move the content of floating-point register `FPsrc` to `Rdest`
`mtc1 Rsrc, FPdest`	Integer register `Rsrc` is moved to floating-point register `FPdest`
`mov.x FPdest, FPsrc`	Move floating-point register `FPsrc` to `FPdest`
`lwc1 FPdest, address`	Load word from `address` in register `FPdest`[a]
`swc1 FPsrc, address`	Store the content of register `FPsrc` at `address`[b]
`add.x FPdest, FPsrc1, FPsrc2`	Add single precision

Instruction	Comment
`sub.x FPdest, FPsrc1, FPsrc2`	Subtract `FPsrc2` from `FPsrc1`
`mul.x FPdest, FPsrc1, FPsrc2`	Multiply
`div.x FPdest, FPsrc1, FPsrc2`	Divide `FPsrc1` by `FPsrc2`
`abs.s FPdest, FPsrc`	Store the absolute value of `FPsrc` in `FPdest`
`neg.x FPdest, FPsrc`	Negate number in `FPsrc` and store result in `FPdest`
`c.eq.x FPsrc1, FPscr2`	Set the floating-point condition flag to true if the two registers are equal
`c.le.x FPsrc1, FPsrc2`	Set the floating-point condition flag to true if `FPsrc1` is less than or equal to `FPsrc2`
`c.lt.x FPsrc1, FPsrc2`	Set the floating-point condition flag to true if `FPsrc1` is less than `FPsrc2`
`bc1t label`	Branch if the floating-point condition flag is true
`bc1f label`	Branch if the floating-point condition flag is false
`cvt.x.w FPdest, FPsrc`	Convert the integer in `FPsrc` to floating-point
`cvt.w.x FPdest, FPsrc`	Convert the floating-point number in `FPsrc` to integer
`cvt.d.s FPdest, FPsrc`	Convert the single precision number in `FPsrc` to double precision and put the result in `FPdest`
`cvt.s.d FPdest, FPsrc`	Convert the double precision number in `FPsrc` to single precision and put the result in `FPdest`

Atividade 11

Usando o simulador MARS:

- Declare as variáveis Zero.s, PlusInf.s, MinusInf.s, PlusNaN.s, MinusNaN, inicializadas com os padrões de bits correspondendo a zero, mais infinito, menos infinito, NaN positivo, NaN negativo em representação de precisão simples;
- Declare as variáveis Zero.d, PlusInf.d, MinusInf.d, NaN.d, inicializadas com os padrões de bits correspondendo a zero, mais infinito, menos infinito, NaN positivo, NaN negativo em representação de precisão dupla;
- Carregue essas variáveis em registradores de ponto flutuante, começando com *$f0*;
- Imprima, começando com *$f0*, o conteúdo dos registradores onde as variáveis foram carregadas adicionando um caractere de nova linha (\n) após cada valor.

Execute o programa e preencha a tabela a seguir com o nome de cada variável e o respectivo valor impresso após a execução:

Variável	Saída impressa

Qual é o padrão de *bits* para o maior número possível de ponto flutuante de precisão única? Escreva em hexadecimal.

Atividade 12

Nesta atividade você deverá criar um programa que calcula o fatorial de um número inteiro, representado usando notação de números de ponto flutuante, conforme passos a seguir:

- Solicite ao usuário que insira um número inteiro;
- Verifique se o número inserido é negativo: se for negativo, imprima uma mensagem de erro e solicite o usuário para entrar novamente com um número inteiro positivo;
- Realizar a operação '*FactorialSingle*', cujo parâmetro será o número lido do usuário convertido em ponto flutuante de precisão simples. A operação deve retornar o fatorial (em PF precisão simples) desse número;
- Imprimir o valor retornado em ponto flutuante de precisão simples.

Execute o '*FactorialSingle*' para completar o plano de testes a seguir. Use notação científica normalizada para representar a saída impressa por '*FactorialSingle*'.

Número	Fatorial (Inteiro)	Fatorial PF (single)
0!		
5!		

10!		
15!		
20!		
40!		

A seguir, responda:

a) Por que alguns dos resultados impressos são negativos?
b) Quais são os valores máximos da entrada para os quais a saída correta ainda é impressa?
c) Usando algum outro método, calculadora ou linguagem de programação, calcule o valor exato de 20! e compare com o valor impresso pelo seu programa no MARS. Quais são esses valores? Por quê os valores são diferentes?

Atividade 13

Na Atividade 2 você usou a conversão de inteiro para ponto flutuante (*cvt.s.w*). Vamos agora tentar uma conversão de PF para inteiro, conforme instruções a seguir:

1) Após imprimir o valor retornado por 'FactorialSingle', converta esse valor em um número inteiro e imprima-o também;
2) Com esse novo programa, conclua o próximo plano de teste. Anote o valor impresso para o fatorial, sem usar notação científica neste momento.

Número	Fatorial (Impressão como Inteiro)	Fatorial (Impressão como PF simples)
0!		
5!		
10!		
12!		
14!		
15!		

Destaque os casos em que a saída de inteiro é um número diferente da saída de ponto flutuante. A operação de conversão produz um inteiro com ou sem sinal? Explique suas conclusões.

3. Execução
- Grupos de até 3 (três) alunos;
- Submissão até 31/10;