Atividade 02

Escreva um código em assembly que permita a entrada de dois operandos sinalizados em ponto-flutuante de precisão simples e uma operação aritmética. O programa deve retornar a resposta do cálculo de todas as execuções em um arquivo acumulativo.

Casos não relatados neste documento são decisões dos alunos e devem ser relatados.

Especificações

"Escreva um código em assembly"

- Linguagem de Montagem para arquitetura AMD/Intel x86_64, preferencialmente, com sintaxe Intel.
- Deve ser possível montar, ligar (*linkar*) e executar o código sem erros ou avisos (warnings)
- Chamadas de Sistemas utilizando syscall
- Número inferior à 400 linhas de código.

"que permita a entrada de dois operandos inteiros sinalizados em ponto-flutuante de precisão simples e uma operação aritmética"

- A entrada do usuário se dará em uma única linha
  - exemplo de comando em C é
    `scanf("%f %c %f")`
  - Sugestão: use printf e scanf para a interação com o usuário

"O programa deve retornar a resposta do cálculo de todas as execuções em um arquivo acumulativo"

- Para cada execução do código, a entrada do usuário, seguido da resposta, deve ser armazenado em um arquivo de saída.
  - O arquivo deve ser criado se não existe
  - O conteúdo do arquivo não deve ser perdido entre cada execução
    - Toda nova execução deve adicionar no final do arquivo as informações:
      execução correta: `"%lf %c %lf = %lf\n"`
      execução incorreta: `"%lf %c %lf = funcionalidade não disponível\n"`
      sendo: operando1 operador operando2 = resposta (ficará mais claro no exemplo)
- Importante: poderá ser usada a função fprintf, porém, o descritor de arquivo do C é diferente daquele usado pela chamada de sistema sys_open. Assim, o arquivo deve ser aberto e fechado pela funções também em C.

"Casos não relatados neste documento são decisões dos alunos e devem ser relatados"

- Toda e qualquer decisão por parte da equipe deve ser informada no relatório.

Funcionamento

- O código é de execução única
- As operações suportadas são:

Operação	Caractere utilizado para representação*	Descrição
Adição	a	Adição entre op1 e op2
Subtração	s	Subtração do op2 em op1
Multiplicação	m	Multiplicação entre op1 (multiplicando) e op2 (multiplicador)
Divisão	d	Divisão entre op1 (dividendo) e op2 (divisor)
~~Resto da Divisão~~	~~r~~	~~Resto da divisão inteira entre op1 (dividendo) e op2 (divisor)~~
Exponenciação	e	op1 (base) elevado à op2 (expoente)

*optou-se por utilizar caracteres não especiais para evitar possíveis problemas com o terminal linux.

- Todas as operações são diretas, com exceção da exponenciação
  - *"diretas"*: devem ser utilizadas as instruções assembly que as executam.
  - Divisão por zero tem como resposta "funcionalidade não disponível".
  - A exponenciação deverá ser realizada por multiplicações sucessivas e não deve ser utilizado a função pow da biblioteca matemática do C. O op2 (expoente) deve ser positivo, caso contrário, a resposta deve ser "funcionalidade não disponível".
- A resposta deve ser adicionada ao final de um arquivo de saída
  - Os operadores devem ser trocados para:

Operador de Entrada:	a	s	m	d	e
Operador de Saída:	+	-	*	/	^

- Exemplo de execuções:
  - considere <texto> entrada do usuário sem "<" e ">"

Ordem	Terminal Linux	Arquivo de saída
0		Vazio
1	$: ./exercicio2 equação: <95.0 a 1050.0>	(arquivo criado) 95.0 + 1050.0 = 1145.0
2	$: ./exercicio2 equação: <95.0 s 1050.0>	(arquivo existente) 95.0 + 1050.0 = 1145.0 95.0 – 1050.0 = -955.0
3	$: ./exercicio2 equação: <13.0 m 8.0>	(arquivo existente) 95.0 + 1050.0 = 1145.0 95.0 – 1050.0 = -955.0 13.0 * 8.0 = 104.0
4	$: ./exercicio2 equação: <106.0 d 8.0>	(arquivo existente) 95.0 + 1050.0 = 1145.0 95.0 – 1050.0 = -955.0 13.0 * 8.0 = 104.0 106.0 / 8.0 = 13.25
...	...	...

5	`$: ./exercicio2`
	`equação: <2.0 e 4.0>`
		`(arquivo existente)`
		
		`95.0 + 1050.0 = 1145.0`
		`95.0 – 1050.0 = -955.0`
		`13.0 * 8.0 = 104.0`
		`106.0 / 8.0 = 13.25`
		`2.0 ^ 4.0 = 16.0`

6	`$: ./exercicio2`
	`equação: <2.0 e -5.0>`
		`(arquivo existente)`
		
		`95.0 + 1050.0 = 1145.0`
		`95.0 – 1050.0 = -955.0`
		`13.0 * 8.0 = 104.0`
		`106.0 / 8.0 = 13.25`
		`2.0 ^ 4.0 = 16.0`
		`2.0 ^ -5.0 = funcionalidade não disponível`

7	`$: ./exercicio2`
	`equação: <106.0 d -8.0>`
		`(arquivo existente)`
		
		`95.0 + 1050.0 = 1145.0`
		`95.0 – 1050.0 = -955.0`
		`13.0 * 8.0 = 104.0`
		`106.0 / 8.0 = 13.25`
		`2.0 ^ 4.0 = 16.0`
		`2.0 ^ -5.0 = funcionalidade não disponível`
		`106.0 / -8.0 = -13.25`

8	`$: ./exercicio2`
	`equação: <106.0 d 0.0>`
		`(arquivo existente)`
		
		`95.0 + 1050.0 = 1145.0`
		`95.0 – 1050.0 = -955.0`
		`13.0 * 8.0 = 104.0`
		`106.0 / 8.0 = 13.25`
		`2.0 ^ 4.0 = 16.0`
		`2.0 ^ -5.0 = funcionalidade não disponível`
		`106.0 / -8.0 = -13.25`
		`106.0 / 0.0 = funcionalidade não disponível`

Restrições

- Não é permitido o uso de funções prontas além de scanf, printf, fprintf, fopen e fclose.
- As funções próprias dos alunos podem ser otimizadas, mas devem respeitar o protocolo
  - isto é, não é necessário salvar registradores não utilizados
  - porém, o *stack frame* é obrigatório,
  - as variáveis locais alocadas na pilha são opcionais
- Variáveis do tipo string (vetor de caracteres) podem ser globais
- O código deve conter as seguintes funções:
  - `float` **`adicao`**`(float op1, float op2)`
    Retorna op1 + op2
  - `float` **`subtracao`**`( float op1, float op2)`
    Retorna op1 – op2
  - `float` **`multiplicacao`**`( float op1, float op2)`
    Retorna op1 * op2
  - `float` **`divisao`**`( float op1, float op2)`
    Retorna quociente de op1 / op2
  - `float` **`exponenciacao`**`( float op1, float op2)`
    Retorna $op1^{(int\_truncated)\ op2}$

- `void escrevesolucaoOK(float op1, char op, float op2, float resposta, int ponteiroArquivo)`
  Escreve no arquivo apontando por ponteiroArquivo o texto:
  `"%lf %c %lf = %lf\n"`
- `void escrevesolucaoNOTOK(float op1, char op, float op2, float ponteiroArquivo)`
  Escreve no arquivo apontando por ponteiroArquivo o texto:
  `"%lf %c %lf = funcionalidade não disponível\n"`

Tips and Tricks (não ordenados)

- fprintf
  - int fprintf(FILE *restrict stream, const char *restrict format, ...);
    - parâmetro FILE *restrict stream deve ser retorno de fopen, e não de kernel OPEN
  - utiliza, assim como o printf, double para escrever no arquivo, portanto, a string de controle deve conter "%lf" e não somente "%f".
    - use cvtss2sd no próprio registrador xmmi
  - primeiro parâmetro FILE é o descritor de arquivo recuperado via fopen()
  - segundo parâmetro é a string de controle, a do "%lf"!
  - Terceiro e demais parâmetros são os argumentos da string de controle
- passagens de parâmetros
  - ponto-flutuante em geral são alocados nos registradores xmmi, na ordem de 0 até 15
  - inteiro e caracter são passados por cópia, e não por referência
  - strings são passadas por referência
  - ordem dos parâmetros
    - rdi, rsi, rdx, rcx, r8 e r9
    - xmm0, xmm1, xmm2 ...
- a escolha da operação pode ser resolvida como um switch-case, em assembly, claro!
- Potência? Uhm, acho que já vi algo na atividade AT1102 com inteiros
  - A função de exponenciação poderá ficar complicada muito rápida, pois para calcular x^y usa-se $2^{y*log2(x)}$, que usa instrução FYL2X da FPU, que não é estudada na disciplina, portanto, a solução "preguiçosa" é truncar y para inteiro usando a instrução:
    - CVTTSS2SI - Convert with Truncation Scalar Single-Precision F.P. Value to Integer
- Não esqueça de verificar as condições especiais para funcionalidade não disponível
- SIGDEV? SIGFAULT? Segmentation Fault?
  - Cuide da pilha
  - Cuide dos parâmetros
  - scanf, printf, fprintf com ponto-flutuante gostam de pilha alinhada em 16 bytes!

Entrega

Via Teams
Arquivo: [LM pratica 02] NomeSobrenome1 NomeSobrenome2.zip
contendo:
- fonte em formato .asm
adicione a linha de montagem e ligação como comentário no início da solução.
- relatório em .pdf com as decisões tomadas pela dupla.

Prazo: verificar na Tarefas na Equipe de LM no Teams

Composição da Nota da Etapa 02

Será observado o Plano de Ensino da ano letivo.