CK0195 - ARQUITETURA DE COMPUTADORES
PRIMEIRA AVALIAÇÃO
LÍVIA BELIZÁRIO ROCHA - 418304

**AULA 8 - Junto com esta aula foi disponibilizado um esqueleto em C/C++ da implementação do emulador. A estrutura geral aqui comentada está lá, tendo também comentários detalhados sobre o que cada função deve realizar e como se comportam as variáveis. Você deve, então, completar a implementação para ter um emulador funcional, conforme descrito na tarefa passada no SIGAA.**

Implementação do emulador já foi enviada pelo SIGAA.

**AULA 9 - Usando a implementação do emulador passada como tarefa na aula 8, coloque o microprograma acima no armazenamento de controle. Na memória principal, escreva um programa para instruir o processador a realizar a soma 5 + 4. (Dica: valores binários podem ser atribuídos em C utilizando o prefixo "0b". Por exemplo, o comando "x = 0b0110;" atribui o valor binário 0110 (6 decimal) à variável x.)**

* Feito aqui de acordo com o código do emulador já enviado.

```
bool carregarArmazenamentoDeControle()
{
   ArmazenamentoDeControle[0] = 0b00000000010000110101000001000010001;
   ArmazenamentoDeControle[1] = 0b00000001000000110101000001000010001;
   ArmazenamentoDeControle[2] = 0b00000001100000010100100000000000010;
   ArmazenamentoDeControle[3] = 0b00000010000000110101000001000010001;
   ArmazenamentoDeControle[4] = 0b00000010100000010100010000000000010;
   ArmazenamentoDeControle[5] = 0b00000000000000011110000000100000010;
   return false;
}

bool carregarPrograma(char *prog)
{
   memoria[1] = 0b1;   // colocar o 1 para indicar soma
   memoria[2] = 0b100; // primeiro número a ser somado (4)
   memoria[3] = 0b101; // segundo número a ser somado (5)
   return false;
}
```

**AULA 10 - Comente o programa acima, mostrando quais linhas de memória correspondem a quais instruções (completas, ou seja, identificador da instrução e operando). Use o formato do pseudocódigo apresentado no microprograma para as instruções.**

```
[1]  0010; // OPC = OPC + MEM[10]
[2]  1010; // 10
[3]  0010; // OPC = OPC + MEM[11]
[4]  1011; // 11
[5]  0110; // MEM[12] = OPC
[6]  1100; // 12
// operandos da soma nas posições 40 e 44
[40] 0101; // 5
[44] 0011; // 3
```

**AULA 11 - Reescreva utilizando a linguagem de montagem proposta os programas dos itens "c", "d" e "e" dos exercícios da aula 10.**

* Item C.

```
[1]  ADD OPC, [10]
[2]  ADD OPC, [13]
[3]  MOV OPC, [13]
[4]  SUB OPC, [13]
[5]  ADD OPC, [11]
[6]  ADD OPC, [12]
[7]  MOV OPC, [11]
[8]  JZ 11
[9]  SUB OPC, [11]
[10] GOTO 1
```

**AULA 12 - Escreva em Assembly IJVM o programa para realizar a soma "5+3". Ou seja, o programa deve inserir na pilha os valores 5 e 3 e depois fazer a soma. Espera-se que ao fim de sua execução a pilha contenha apenas o valor 8.**

```
bipush 5
bipush 3
iadd
```

**AULA 13 - Implemente no seu emulador uma função que carregue o arquivo “microprog.rom” no armazenamento de controle.**

```
bool carregarArmazenamentoDeControle()
{
   FILE *armazenamento;
   armazenamento = fopen("microprog.rom", "rb");
   if (armazenamento)
   {
       fread(ArmazenamentoDeControle, 8, 512, armazenamento);
       fclose(armazenamento);
       return false;
   }
   cout << "(ERRO) Microprograma não existe!" << endl;
   return true;
}
```

**AULA 14 - Escreva em Assembly IJVM um programa que insira os valores 7 e 9 na pilha, realize a multiplicação deles e deixe o resultado no topo da pilha. Faça também a montagem completa do seu programa, isto é, mostre os bytes de inicialização e os bytes do programa em si, da mesma forma que é mostrado no final da seção 1 desta nota de aula o resultado da montagem do programa exemplo.**

```
     bipush 7        ; pega os valores 7 e 9 e guarda nas variáveis x e y
     istore x
     bipush 9
     istore y

     bipush 0        ; inicializa o produto como 0
     istore product

loop iload product   ; carrega o produto
     iload x         ; carrega x
     iadd            ; soma os dois
     istore product  ; guarda o produto
     iinc y -1       ; decrementa y
     iload y         ; carrega y
     ifeq end        ; checa se chegou a 0, se sim, acaba o programa
     goto loop       ; senão volta para o loop

end nop
```

* Não foi possível fazer a montagem pois o iinc não está funcionando devido ao erro no “microprog.rom”.

**AULA 15 - Está disponível no SIGAA na aula de hoje a tarefa (portanto, será cobrada como parte da avaliação, assim como o emulador) de implementação de um Assembler (montador) IJVM. Ou seja, você deverá implementar na linguagem de sua preferência um programa que recebe de entrada um arquivo contendo um programa escrito em Assembly IJVM e dá de saída um arquivo binário conforme descrito nesta aula com o programa montado.**

Implementação do Assembler já foi enviada pelo SIGAA.

**AULA 16 - Reescreva o microprograma (em pseudocódigo, como exemplificado) considerando que a microarquitetura possui dois barramentos de entrada para a ALU e que qualquer registrador que antes podia ter seu valor colocado no barramento ligado à entrada B da ALU, também pode ter seu valor colocado no novo barramento ligado à entrada A da ALU. Em outras palavras, sub-microinstruções como "PC ← OPC + MDR" agora são válidas. Para quantos ciclos de clock você conseguiu reduzir cada instrução?**

```
//MAIN
[0] PC <- PC + 1; fetch; GOTO MBR;
//OPC = OPC + memory[end_word]
[2] PC <- PC + 1; fetch;
[3] MAR <- MBR; read;
[4] OPC <- OPC + MDR; GOTO MAIN;
//memory[end_word] = OPC
[5] PC <- PC + 1; fetch;
[6] MAR <- MBR;
[7] MDR <- OPC; write; GOTO MAIN;
//goto endereco_comando_programa
[8] PC <- PC + 1; fetch;
[9] PC <- MBR; fetch; GOTO MAIN;
//if OPC = 0 goto endereco_comando_programa else goto proxima_linha
[10] OPC <- OPC; IF ALU = 0 GOTO 268 (100001100)
                 ELSE GOTO 11 (000001011);
[11] PC <- PC + 1; GOTO MAIN;
[268] PC <- PC + 1; fetch;
[269] PC <- MBR; fetch; GOTO MBR;
//OPC = OPC - memory[end_word]
[12] PC <- PC + 1; fetch;
[13] MAR <- MBR; read;
[14] OPC <- OPC - MDR; GOTO MAIN;
```

O programa continua o mesmo com exceção das microinstruções marcadas em verde, que reduz as instruções ***ADD OPC*** e ***SUB OPC*** em 1 clock, totalizando em 2 clocks reduzidos.

**AULA 17 - Pesquise e explique sobre como os múltiplos núcleos na microarquitetura Bulldozer (AMD) estão organizados. Mais especificamente, disserte sobre a organização da Cache e a execução de instruções inteiras e ponto flutuante.**

Introduziu algo chamado "Clustered MultiThreading" (CMT), onde algumas partes do processador são compartilhadas por uma thread e outras partes são únicas para cada thread. Possui até 8MB de cache L2 compartilhado para cada dois núcleos, e entre 4MB e 8MB de cache L3 compartilhado entre todos os núcleos. A arquitetura é compatível com as instruções x86, tem suporte para o conjunto de instruções AVX, que suporta operações de 8 números ponto flutuante de 32 bits de precisão simples ou 4 números ponto flutuante de 64 bits de precisão dupla simultaneamente. Também tem suporte para SSE4.1, SSE4.2, AES, CLMUL, além dos conjuntos de instruções propostos pela AMD (XOP, FMA4, and F16C), que possuem as mesmas funcionalidades do SSE5 mas com compatibilidade com o AVX.

**AULA 18 - Pesquise e responda, para cada um dos videogames a seguir, se seus processadores são RISC ou CISC: XBOX, XBOX 360, XBOX ONE, PS2, PS3 e PS4.**

| | |
|---|---|
| XBOX | CISC |
| XBOX 360 | RISC |
| XBOX ONE | CISC |
| PS2 | RISC |
| PS3 | RISC |
| PS4 | CISC |

**AULA 19 - Existem duas sintaxes principais utilizadas por assemblers para processadores x86. Pesquise e responda:**
**a) Quais são e quais as diferenças?**
**b) Cite o nome de um montador que implemente cada uma das duas sintaxes.**

a) Sintaxe Intel e AT&T.
   Na sintaxe AT&T, as instruções necessitam de sufixos para diferenciar o tamanho do operando (‘l’ para ‘long’, ‘w’ para ‘word’ e ‘b’ para byte);
   A direção dos operandos nas duas sintaxes são opostas;
   Na sintaxe Intel, não são necessários prefixos, na AT&T eles são necessários (movl $1, %eax);
   Os operandos de memória se diferem, onde na sintaxe Intel se utilizam ‘[]’ e na AT&T ‘()’.

b) Montadores Intel: NASM, FASM, MASM, TASM;
   Montador AT&T: GAS.

**AULA 20 - Para cada uma das instruções a seguir, do conjunto IJVM estudado anteriormente, indique o tipo de endereçamento: bipush, iload, goto.**

```
bipush -> endereçamento imediato;
iload  -> endereçamento indexado;
  goto -> endereçamento direto.
```