INSTITUTO DE ENSINO SUPERIOR DE GOIÁS – IESGO

CURSO: BACHAREL EM SISTEMAS DE INFORMAÇÃO E TECNÓLOGO EM REDES

RENATO FERNANDES PEREIRA
IVAN LOPES

SIMULADOR BÁSICO DE ARQUITETURA BASEADA NO PROCESSADOR MIPS

PLANALTINA – GO
2012

INSTITUTO DE ENSINO SUPERIOR DE GOIÁS - IESGO
CURSO: BACHAREL EM SISTEMAS DE INFORMAÇÃO E TECNÓLOGO EM REDES

DOCUMENTAÇÃO DO PROJETO

PLANALTINA – GO
2012

DESCRIÇÃO DO SIMULADOR

Segue detalhadamente os métodos utilizados no processo de implementação desse software:

1° Método LOAD

```
void load()
{
	unsigned long int instruction;
	int cont=1;
	char end[400],palavra[33];//Aqui é a palavra que está no arquivo;
	PC=1;
	ciclos=0;
	ifstream leitura; //objeto de leitura de arquivo.

	cout<<"Entre com o endereco onde esta localizado o arquivo que será
		lido.\nExemplo:\"C:/exemplo/teste.txt\""<<endl;
	cin>>end;

	leitura.open(end); // abertura de arquivo.
 if(!leitura.is_open( ))  // Saber se o arquivo foi aberto.
 {
   system("cls");
   cout<<"\t\t\tERRO!! CODE:0x001";
   leitura.clear( );
   _getch();
   exit(0);
 }

 system("cls");
 cout<<"\t\t\tArquivo aberto com sucesso!"<<endl;

 cout<<"\nPressione qualquer tecla para que seu codigo seja processado";
 getch();
 while(leitura.getline(palavra,33))
 {
      instruction=strtoul(palavra,NULL,2);
      Memoria.write(cont,instruction);
      cont+=4;
      ciclos++;
  }
  leitura.close();
 }
```

O Método Load foi desenvolvido para carregar um conjunto de instruções previamente estabelecidos pelo programador, ou por um compilador, em memória. Primeiramente é solicitado que o usuário entre com o endereço onde está localizado o arquivo que será processado, caso não haja nenhum problema, todas as instruções serão carregadas em memória, e dará início ao ciclo de busca, decodifica e executa, caso haja algum problema consulte a tabela (Code-Errors).

2°Método FETCH

```
void fetch()
{
	unsigned long int instruction;
	instruction=Memoria.read(PC);
	itoa(instruction,IR,2);
	PC+=4;
}
```

Nesta etapa do processo, a instrução que foi previamente carregada em memória é armazenada no IR (Instruction register) e o PC (program counter) é incrementado em 4 bytes.

3°Método COMPLEMENTAR

```
void complementar()
{
	int i,qtd;
	char comp_zeros[33];
	qtd=strlen(IR);

	for(i=0;i<=32;i++)
	{
	  comp_zeros[i]=NULL;
	}

	i=0;
	while(qtd<32)
	{
	   comp_zeros[i]='0';
	   qtd++;
	   i++;
	}

	strcat(comp_zeros,IR);

	for(i=0;i<=32;i++)
	{
	   IR[i]=comp_zeros[i];
	}
}
```

Este método foi implementado, para fazer um preenchimento de 0's caso seja insuficiente para a perfeita execução do sistema, Este método fará uma análise do conteúdo do IR e contará quantos bits foram previamente desenvolvidos pelo programador, caso seja menor que 32 bits, este método completará com 0's a esquerda até que todos os bits sejam preenchidos.

4°MÉTODO DECODE

```
void decode()
{
     unsigned int qtd;
     qtd=strlen(IR);
     if(qtd<32)
     {
    complementar();
  }
     if(qtd>32)
     {
            system("cls");
                   printf("ERRO CODE:0x004");
                   _getch();
                   exit(0);
     }
     string s;
  s=IR;


     enum opcodes
     {
    LW=35,
    SW=43,
     };

     enum funct
     {
    ADD=32,
    MUL=24,
    DIV=26,
    SUB=34,
     };
  unsigned int opcodeDEC;
     s.copy(opcode,6,0);
     opcode[6]=NULL;
     opcodeDEC=strtoul(opcode,NULL,2);

     if(opcodeDEC!=35 && opcodeDEC!=43 && opcodeDEC!=0)
     {
            system("cls");
            cout<<"ERRO CODE:0x005";
            _getch();
            exit(0);
     }

     unsigned int functDEC;
  s.copy(Funct,6,26);
     Funct[6]=NULL;
  functDEC=strtoul(Funct,NULL,2);
```

```
switch (opcodeDEC)
{
   case LW:
           s.copy(rs,5,6);
              rs[5]=NULL;

              s.copy(rt,5,11);
              rt[5]=NULL;

              s.copy(rd,5,16);
              rd[5]=NULL;

              break;
   case SW:

           s.copy(rs,5,6);
              rs[5]=NULL;

              s.copy(rt,5,11);
              rt[5]=NULL;

              s.copy(rd,16,16);
              rd[16]=NULL;


           break;

   case 0:

          if(functDEC==32)
          {
                    s.copy(rs,5,6);
                    rs[5]=NULL;

                    s.copy(rt,5,11);
                    rt[5]=NULL;

                    s.copy(rd,5,16);
                    rd[5]=NULL;

                    s.copy(sa,5,21);
                    sa[5]=NULL;
          }
          if(functDEC==24)
          {
                  s.copy(rs,5,6);
                    rs[5]=NULL;

                    s.copy(rt,5,11);
                    rt[5]=NULL;

                    s.copy(rd,5,16);
                    rd[5]=NULL;

                    s.copy(sa,5,21);
                    sa[5]=NULL;
          }
```

```
            if(functDEC==26)
            {
                    s.copy(rs,5,6);
                       rs[5]=NULL;

                       s.copy(rt,5,11);
                       rt[5]=NULL;

                       s.copy(rd,5,16);
                       rd[5]=NULL;

                       s.copy(sa,5,21);
                       sa[5]=NULL;
            }
            if(functDEC==34)
            {
                    s.copy(rs,5,6);
                       rs[5]=NULL;

                       s.copy(rt,5,11);
                       rt[5]=NULL;

                       s.copy(rd,5,16);
                       rd[5]=NULL;

                       s.copy(sa,5,21);
                       sa[5]=NULL;
            }
            if(functDEC != 32 && functDEC!=24 && functDEC!=26 &&
               functDEC!=34)
            {
                       system("cls");
                       cout<<"ERRO CODE:0x006";
                       _getch();
                       exit(0);
                }
            break;
          }
    }
```

Este método tem como objetivo pegar a instrução que foi previamente colocada no IR pele método FETCH( ) e fazer uma separação de todos os campos da instrução, as instruções da arquitetura MIPS são divididas em 3 categorias: instruções do tipo R, instruções do tipo I e instruções do tipo J, este simulador utilizará apenas de instruções do tipo I e R.

As instruções do tipo R são divididas em 6 registradores que são eles:

- OPCODE – 6 bits
- RS – 5 bits

- RT- 5 bits
- RD- 5 bits
- SH- 5 bits
- FUNCT – 6 bits

As instruções do tipo I são divididas em apenas 4 registradores sendo eles:

- OPCODE – 6 bits
- RS – 5 bits
- RT- 5 bits
- RD- 16 bits

Para uma facilitação do desenvolvimento desse software, as bases numéricas que foram utilizadas no decorrer do código foram as bases 2(binária) e 10(decimal).

O Método DECODE ( ) irá fazer uma acesso ao registrador de instruções, e de acordo com o opcode, realizar a decodificação colocando os bits em seus respectivos registradores.

5° MÉTODO EXECUTE

```
{
       enum opcodes
       {
      LW=35,
      SW=43,
       };
       enum funct
       {
      ADD=32,
      MUL=24,
      DIV=26,
      SUB=34,
       };

       unsigned long int end;

       unsigned long int opcodeDEC;
       opcodeDEC=strtoul(opcode,NULL,2);

       unsigned long int FunctDEC;
    FunctDEC=strtoul(Funct,NULL,2);

       switch (opcodeDEC)
       {
          case SW:

                   end=strtoul(rd,NULL,2);
                      Memoria.write(end,$t0);
```

```
            break;

    case 0:

        if(FunctDEC==32)// ADD
        {
                  $s0=strtoul(rs,NULL,2);
                  $s1=strtoul(rt,NULL,2);

                  $t0=$s0+$s1;
                  _ltoa($t0,rd,2);
        }
        if(FunctDEC==24)//MUL
        {
                $s1=strtoul(rs,NULL,2);
                  $s2=strtoul(rt,NULL,2);

                  for(int i=1;i<=$s2;i++)
                {
                     $t0+=$s1;
                }
                  _itoa($t0,rd,2);
        }

        if(FunctDEC==26)//DIV
        {
                $s1=strtoul(rs,NULL,2);
                  $s2=strtoul(rt,NULL,2);
                HI=0;
                  if($s1<$s2)
                  {
                         system("cls");
                         printf("ERRO CODE:0x002");
                      _getch();
                         exit(0);
                  }
                  $t0=$s1;
                  for(;;)
                  {
                         $t0-=$s2;
                         HI++;
                         if($t0<=1)
                         break;
                  }
                  itoa(HI,rd,2);
                  $t0=HI;
        }
        if(FunctDEC==34)//SUB
        {
                $s0=strtoul(rs,NULL,2);
                  $s1=strtoul(rt,NULL,2);
                  $t0=$s0-$s1;
                  if($t0<=0)
                  {
                         system("cls");
                         printf("ERRO CODE:0x003");
                      _getch();
                         exit(0);
                  }

                  _itoa($t0,rd,2);
```

É no método execute que todo o processo aritmético e de registro de dados em memória é feito, após a instrução passar por um processo de decodificação chega o momento de fazer uso de alguns outros registradores que serão responsáveis por toda aritmética necessitada pela instrução, lembrando que nos momentos dos cálculos aritméticos que aparecem a instrução FOR( ) representa um jump, pois o processador não é capaz de processar cálculos aritméticos como multiplicação (*) ou divisão (/), cabendo ao programador, elaborar uma lógica utilizando jumps para que seja concretizada uma operação dessas respectivas propriedades.

Mais uma vez de acordo com o Opcode a instrução será executada de uma forma diferente,

- quando o opcode representar um SW(Store Word), este método pegará o dado que está guardado dentro de um dos registradores temporários aos quais armazena os resultados dos cálculos, e guardará esse valor em memória,
- caso o opcode seja de um ADD, o valor guardado no registrador RS será passado ao registrador $s0 e o valor armazenado em RT será passado ao registrador $s1, que após ser feito uma soma, o valor será retornado para $t0.
- Caso o opcode seja de um SUB, o valor guardado no registrador RS será passado ao registrador $s0 e o valor armazenado em RT será passado ao registrador $s1, que após ser feito uma subtração, o valor será retornado para $t0.
- Caso o opcode seja de um MUL, o valor armazenado no registrador RS será passado ao registrador $s0, e o valor armazenado em RT ao registrador $s1, após isso ser feito, haverá um teste condicional, e o valor correspondente a quantidade de vezes que estiver dentro de $s1, será a quantidade de vezes que $s0 será armazenado em $t0.
- Caso o Opcode seja de uma divisão, o valor armazenado no registrador RS será passado ao registrador $s0, e o valor armazenado em RT ao registrador $s1,feito isso, haverá um teste condicional, e a quantidade de vezes que o valor de $s1 “couber” em $s0, será incrementado o registrador HI, que o resultado será a parte inteira de uma divisão.

6° Método RUN e STEP

```
void run()
{
	while(ciclos>0)
	{
		step();
		ciclos--;
	}
}



void step()
{
	fetch();
	decode();
	execute();
}
```

Estes 2 métodos são responsáveis por chamar todos os outro métodos, exceto LOAD ( ), que é chamado pela função principal (main), o método run, significa que ele vai fazer os ciclos de FETCH, DECODE e EXECUTE proporcionalmente a quantidade de instruções que o programa tiver, ou seja, 10 instruções, 10 ciclos.

7°DUMP_REGS e DUMP_MEMORY

```
int i;
unsigned long int data;
i=inic;
while (i<=fim)
{
	data=Memoria.read(inic);
	if(data==3435973836)
	{
		Memoria.write(inic,0);
		data=Memoria.read(inic);
		cout<<"\n"<<inic<<" - "<<data<<"\n";
	}
	if(data>1000 && data<3435973836)
	{
		cout<<"\n"<<inic<<" - "<<data<<" [instrucao em
		memoria]"<<endl;
	}
	if(data>0 && data<1000)
	{
		cout<<"\n"<<inic<<" - "<<data<<" [dado]\n";
	}
	inic++;
	i++;

}
```

```
	_getch();
	menu();
	}
	void dump_regs()
	{
		int opc;
		system("cls");
		unsigned int opcodeDEC;
		opcodeDEC=strtoul(opcode,NULL,2);

		if(opcodeDEC==43)
		{
			std::cout<<"\nOpcode= "<<opcode
			<<"\nrd[address]= "<<rd
			<<endl;
			cout<<"\nSelecione a opcao desejada:\n1-Para Ver o endereco
			 em decimal\n2-Voltar ao menu principal\n3-Sair do programa”;
			cin>>opc;
			if(opc==1)
			{
				unsigned int rdDEC;
				rdDEC=strtoul(rd,NULL,2);
				cout<<"\nrd[address]= "<<rdDEC
					<<endl;
				getch();
				menu();
			}
			if(opc==2)
			{
				menu();
			}
			if(opc==3)
			{
				system("cls");
				cout<<"\t\tObrigado por usar o simulador de
arquitetura Mips Basic\n\t\t    Developed by: Renato Fernandes e Ivan Lopes";
				_getch();
				exit(0);
			}
		}
		else
		{
			std::cout<<"\nOpcode= "<<opcode
			<<"\n$s0= "<<rs
			<<"\n$s1= "<<rt
			<<"\n$t0= "<<rd
			<<"\nFunct= "<<Funct
			<<endl;
		}

		cout<<"\nSelecione a opcao desejada:\n1-Para Ver o resultado em
decimal\n2-Voltar ao menu principal\n3-Sair do programa ";
		cin>>opc;
		if(opc==1)
		{
			unsigned int rsDEC,rtDEC,rdDEC,saDEC,functDEC,opcodeDEC;
			rsDEC=strtoul(rs,NULL,2);
			rtDEC=strtoul(rt,NULL,2);
			rdDEC=strtoul(rd,NULL,2);
			functDEC=strtoul(Funct,NULL,2);
			opcodeDEC=strtoul(opcode,NULL,2);
```

```
                    cout<<"\nOpcode= "<<opcodeDEC
                    <<"\n$s0= "<<rsDEC

                    <<"\n$s1= "<<rtDEC
                    <<"\n$st= "<<rdDEC
                    <<"\nFunct= "<<functDEC
                    <<endl;
                    _getch();
                    menu();
             }
                    if(opc==2)
                    {
                           menu();
                    }
                    if(opc==3)
                    {
                           system("cls");
                           cout<<"\t\tObrigado por usar o simulador de
arquitetura Mips Basic\n\t\t    Developed by: Renato Fernandes e Ivan Lopes";
                           _getch();
                           exit(0);
                    }

}
```

São 2 métodos responsáveis por fazer uma impressão na tela daquilo que estava ocorrendo internamente no simulador, o método DUMP_REGS ( ), mostrará na tela do usuário, o estado final dos registradores após a execução da ultima instrução digitada, dando como opção ao usuário a visualização daqueles mesmos dados em um formato décima, já o método DUMP_MEMORY ( ) fará uma impressão do que existe em memória na posição solicitada pelo usuário, ao solicitar este método é requerido a posição que você deseja ver na memória, este método já fará automaticamente uma impressão do que é dado e o que é instrução em memória.

8° A FASE DE TESTES

No decorrer do processo de desenvolvimento deste software, vários testes foram feitos para que fossem melhorados a funcionalidade e a usabilidade do mesmo, sendo que foi dado como concluído a fase de testes após:

1. Conferir todos os itens de menu,
2. Saber se estava correspondendo com a necessidade do usuário

3. Um bloco de instruções foi feito, contendo 8 instruções sendo cada uma de uma operação aritmética, acompanhada de um SW, segue:

00000010100010100000000000100000//instrução de ADD (20+10)
10101100000000000000000000011110//instrução de SW, armazene em Mem[30]
00000010100010000000000000100010// instrução de MULT (7*9)
10101100000000000000000000100010// instrução de SW, armazene em Mem[34]
00000000111010010000000000011000// instrução de DIV (26/2)
10101100000000000000000000100110// instrução de SW, armazene em Mem[38]
00000011010000100000000000011010// instrução de SUB (20-8)
10101100000000000000000000101010// instrução de SW, armazene em Mem[42]

Após te sido executada essas instruções, foi utilizado o método DUMP_MEMORY ( ), e os valores estavam corretos sendo que os cálculos aritméticos haviam todos sido feitos com precisão, e armazenados na posição solicitada pelo programador.

MANUAL DO PROGRAMADOR

Este programa foi elaborado para realizar as operações de ADD, SUB, MULT, DIV, SW e LW, segue algumas dicas e instruções para uma melhor utilização do programa.

1° instruções com 32 bits

0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Este programa tem nele um método para completar a quantidade de bits que for necessária para o seu funcionamento, porém é recomendado que sempre o programador coloque exatamente os 32 bits, para ter um maior controle e para ter a garantia de que a saída do programa seja condizente com o esperado,

Caso uma instrução tenha mais de 32 bits, o programa acusará um erro e será fechado imediatamente.

2° LOAD ( )

Ao iniciar o programa é solicitado ao usuário que entre com o caminho onde está o arquivo que será executado:

1. Não se esqueça de colocar a extensão do arquivo que será carregado.
2. É necessário que seja dividido por (/) o endereço em que ele se encontra, não confunda essa (/) com essa (\ )

   Caso não seja encontrado o arquivo, ou ele foi digitado incorretamente, sendo a extensão ou o caminho, o programa acusará um erro e fechará imediatamente.

3°INSTRUÇÕES ARITMÉTICAS

- **ADD**

  Para se escrever uma instrução de soma, utilize o opcode: **000000**

  O RS será o primeiro termo que deseja somar, converta esse valor em binário e coloque como sendo RS.

  O RT será o Segundo termo que deseja somar, converta esse valor em binário e coloque como sendo RS.

O RD Será onde esse valor será armazenado, contando que não deseje usar o resultado de uma instrução anterior, deixe em **00000**

O SH Não é utilizado nessa instrução então deixe **00000**

E o funct do ADD É **100000.**

Exemplo de instrução do tipo ADD:

00000010100010100000000000100000

- **SUB**

  Para se escrever uma instrução de subtração utilize o opcode: **000000**

  O RS será o primeiro termo que deseja subtrair, converta esse valor em binário e coloque como sendo RS.

  O RT será o Segundo termo que deseja subtrair, converta esse valor em binário e coloque como sendo RS.

  O RD Será onde esse valor será armazenado, contando que não deseje usar o resultado de uma instrução anterior, deixe em **00000**

  O SH Não é utilizado nessa instrução então deixe **00000**

  E o funct do SUB É **100010.**

  Exemplo de instrução do tipo SUB:

  00000010100010000000000000100010

  NOTA: Caso o primeiro termo seja menor que o segundo o programa não realizará a subtração e submeterá a um erro e fechará

- **MULT**

  Para se escrever uma instrução de multiplicação utilize o opcode:**000000**

  O RS será o primeiro termo que deseja multiplicar, converta esse valor em binário e coloque como sendo RS.

  O RT será o Segundo termo que deseja multiplicar, converta esse valor em binário e coloque como sendo RS.

  O RD Será onde esse valor será armazenado, contando que não deseje usar o resultado de uma instrução anterior, deixe em **00000**

  O SH Não é utilizado nessa instrução então deixe **00000**

  E o funct da multiplicação é **011000.**

  Exemplo de instrução do tipo MULT:

  00000000111010010000000000011000

- **DIV**

  Para se escrever uma instrução de divisão utilize o opcode:**000000**

  O RS será o primeiro termo que deseja dividir, converta esse valor em binário e coloque como sendo RS.

  O RT será o Segundo termo que deseja Dividir, converta esse valor em binário e coloque como sendo RS.

  O RD Será onde esse valor será armazenado, contando que não deseje usar o resultado de uma instrução anterior, deixe em **00000**

  O SH Não é utilizado nessa instrução então deixe **00000**

  E o funct da divisão é **011010.**

  Exemplo de instrução do tipo DIV:

  00000011010000100000000000011010

  NOTA: O Primeiro termo deve sempre ser maior que o primeiro, caso assim não seja, o programa submeterá a um erro e encerrará.

- **SW**

  Para se escrever uma instrução de Store Word utilize o opcode: **101011**

  O RS será o dado, como a não ser que queria armazenar um dado estipulado por você em memória, deixo em zero que ele pegará o resultado da instrução anterior e guardará no endereço de memória indicado por você quando escrever o RD

  O RT, ficará em zero pois sua utilização só se faz necessária ao trabalhar com números com ponto flutuante.

  O RD guardará o ENDEREÇO onde será armazenado o rs, este diferentemente das outras instruções possui 16 bits.

  Exemplo de instrução SW:

  10101100000000000000000101010

$CODE- ERROR$		
Cód. Erro	**Causa**	**Solução**
CODE:0x001	O Arquivo não pôde ser carregado	Verifique se digitou corretamente o caminho,nome e extensão.
CODE:0x002	1° termo da divisão é menor do que o primeiro	Inverta a ordem no código, ou altere os valores
CODE:0x003	1° termo da subtração é menor do que o primeiro	Inverta a ordem no código, ou altere os valores
CODE:0x004	Alguma instrução contém mais de 32 bits	Reorganize as suas instruções de forma que não ultrapassem 32 bits.
CODE:0x005	O opcode que você utilizou não consta em nossa ISA	Veja se todos os opcodes foram digitados de forma correta.
CODE:0x006	O funct que você utilizou não consta em nossa ISA	Veja se todos os functs foram digitados de forma correta.