3
A Biblioteca para Implementação de Máquinas Virtuais

O processo de criação e experimentação de uma máquina virtual no escopo deste trabalho é ilustrada na Figura 3-1. O projetista da arquitetura (*Usuário que projeta uma arquitetura* na Figura), utilizando-se das classes disponíveis na biblioteca, especifica as características da arquitetura que deseja emular, tais como os códigos das operações (opcodes) da CPU, número de ciclos gastos por instrução, número e tamanho dos registradores, tamanho e tipo da memória, etc.

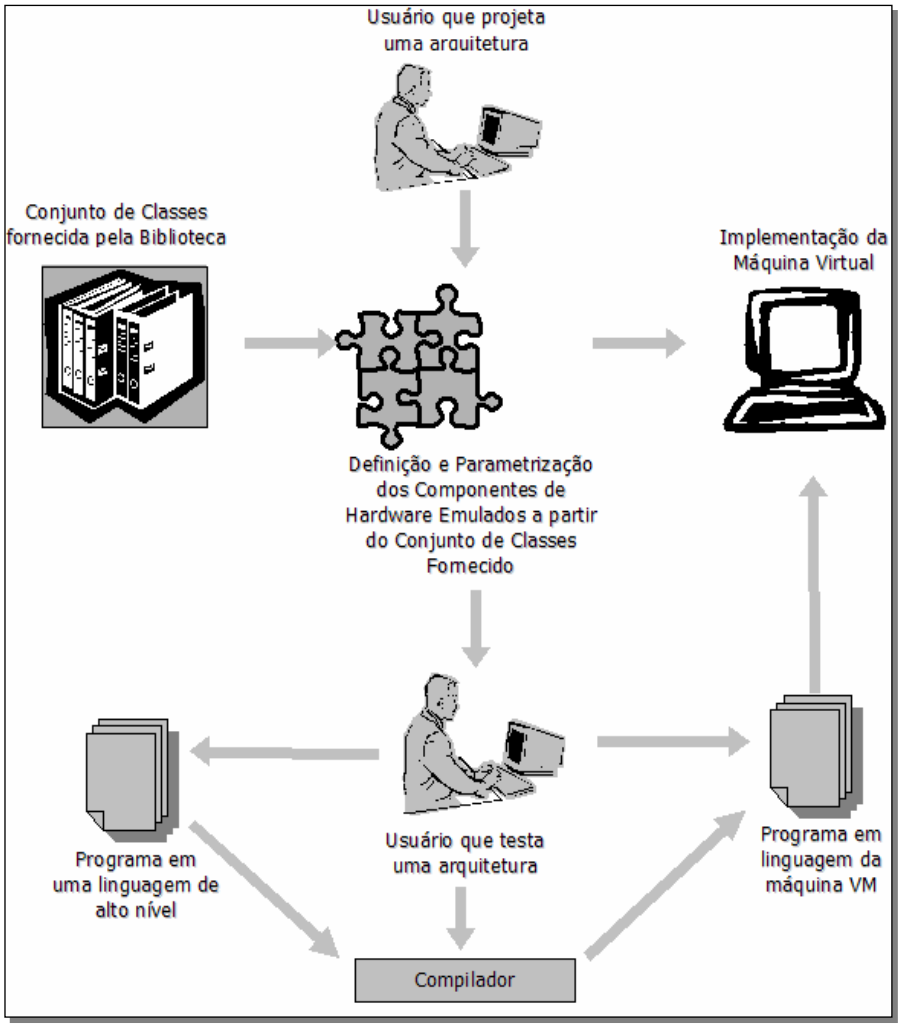

Figura 3-1 - Relacionamentos da Biblioteca com os Projetistas e Funcionamento da Máquina Virtual

Com essas informações, será gerada a implementação da arquitetura desejada, correspondendo à *Implementação da Máquina Virtual*. O *Usuário que testa uma arquitetura*, de posse das especificações da máquina virtual, pode escrever o código que será nela executado.

A Biblioteca necessária à implementação da VM é composta de cinco classes desenvolvidas em Microsoft Visual C++ 6.0. Três delas implementam, cada uma, um objeto genérico que representa um dos principais componentes envolvidos: CPU, memória e registradores. Uma quarta classe implementa a máquina de estados que controla a CPU. A partir dessas quatro classes principais foi definida uma quinta classe que encapsula a cpu, memória e registradores, que é, formalmente, a máquina virtual implementada (VM). Os arquivos de cabeçalhos dessas classes estão detalhados no Apêndice A. O usuário que projeta arquitetura é então o responsável por fornecer parte da implementação da classe CPU e da classe da VM segundo as regras definidas na Seção 3.2.

Ao testar a arquitetura, o usuário será responsável por fazer com que o código executável seja carregado na memória RAM emulada e que a execução seja iniciada a partir do endereço apontado por um registrador contador de programa na inicialização da máquina. A implementação da classe CPU dispõe dos mecanismos para auxiliar nessa tarefa. A partir daí, a VM se comporta como uma máquina real convencional, recuperando sucessivamente instruções armazenadas no endereço apontado pelo registrador contador de programa, efetuando sua decodificação e posterior execução. Toda essa sistemática de controle da máquina virtual corresponde, de forma geral, ao funcionamento de um interpretador genérico, cujo funcionamento pode ser completamente especificado e adaptado pelo usuário que projeta a arquitetura. Para que isso seja feito, as classes da biblioteca e o ambiente proposto foram criados de forma a serem utilizados em conjunto com ferramentas automatizadas de análise léxica e sintática do tipo Lex e Yacc. Com ferramentas desse tipo, auxiliadas pelas classes da biblioteca e estruturas de programas semi-prontos, é possível criar e testar os interpretadores específicos que corresponderão às máquinas virtuais desejadas.

O Lex e o Yacc (Levine et al., 1992), ajudam a escrever programas que transformam ou interpretam entradas estruturadas. Compiladores e interpretadores são exemplos de programas que processam entradas estruturadas. Em programas desse tipo, duas tarefas ocorrem o tempo todo: (i) a divisão da entrada em unidades inteligíveis e (ii) o estabelecimento de

relacionamentos entre essas unidades. Por exemplo, para uma entrada correspondendo a um programa escrito em uma linguagem de programação qualquer, as unidades seriam nomes de variáveis, constantes, strings, operadores e assim por diante. Essa divisão em unidades (que são comumente chamadas de *tokens*) é conhecida como *análise léxica*, ou *lexing*. O Lex auxilia o programador na tarefa de geração de um analisador léxico, pegando um conjunto de descrições dos tokens e produzindo a rotina (geralmente em C) que corresponde a esse analisador, ou seja, a rotina que é capaz de identificar os tokens dada uma seqüência de bits de entrada (que pode estar contida em um arquivo, por exemplo).

A forma da descrição sobre a formação para o reconhecimento dos tokens que é utilizada pelo Lex é conhecida como um conjunto de *expressões regulares*. O Lex converte essas expressões em uma forma que o analisador léxico pode utilizar paralelamente ao texto de entrada de forma extremamente rápida, independente do número de expressões que ele tenta casar.

Depois da divisão em tokens é, em geral, necessário estabelecer as *relações* entre esses tokens. Um compilador C, por exemplo, precisa encontrar as expressões, declarações, blocos e funções no programa de entrada. Essa tarefa é conhecida pelo nome de *análise sintática* ou *parsing* e a lista de regras que define seus relacionamentos é o que se chama de *gramática*. Uma ferramenta como o Yacc pega uma descrição da gramática e produz uma rotina (geralmente em C) capaz de efetuar a análise sintática. A essa rotina dá-se o nome de *parser* ou *analisador sintático*. O parser gerado pelo Yacc detecta automaticamente quando uma seqüência de tokens de entrada é casada com uma das regras da gramática e também detecta um erro de sintaxe sempre que uma entrada não casa com nenhuma das regras.

No caso da biblioteca implementada nesse trabalho, foram utilizadas as versões do Lex e Yacc chamadas de Flex (Lex) e Bison (Yacc). Foram escolhidas versões especificamente compatíveis com o Microsoft Visual C++ 6.0, no qual todo o projeto foi desenvolvido.

O Flex e o Bison permitem criar um projeto com um nome definido pelo usuário e que será utilizado como referência ao nome dos arquivos gerados em toda a implementação.

Assim, esses arquivos fornecidos têm a forma <nome-do-projeto> na formação dos seus respectivos nomes, conforme mostrado na Figura 3-2. O projetista da arquitetura não altera o primeiro bloco de código contendo as definições do analisador léxico, já que elas são um padrão gerado pelo Flex. No

campo das regras (segundo bloco de código) define-se essas regras, para informar ao Flex o significado de cada *opcode* da arquitetura emulada. O bloco de sub-rotinas opcionais não é utilizado.

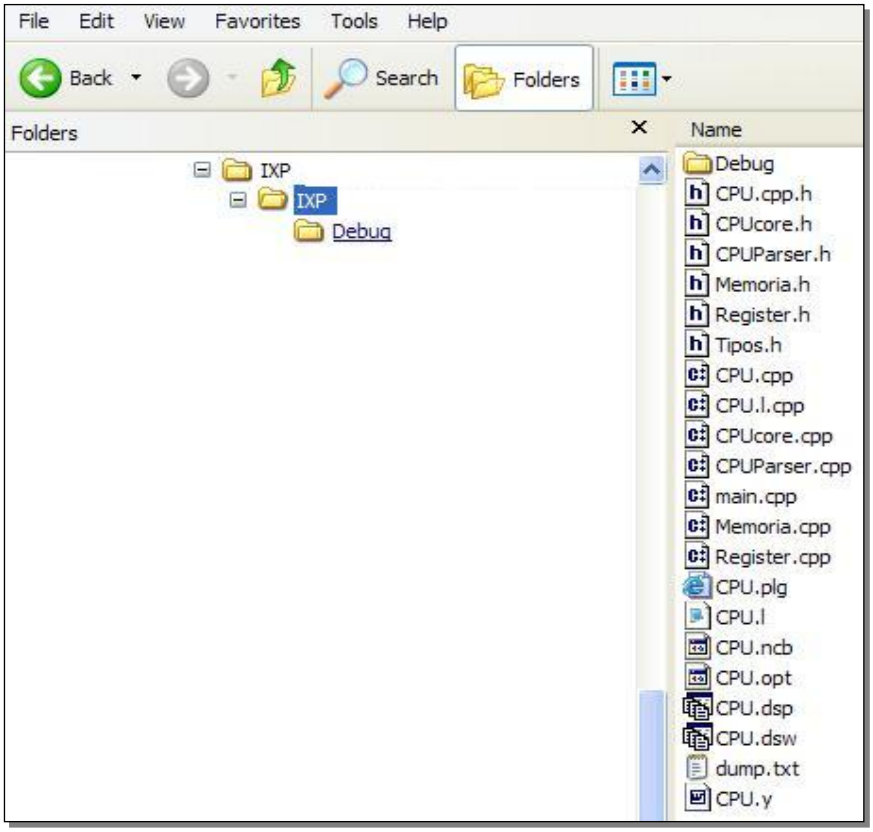

Figura 3-2 – Exemplo da Árvore de Diretório da Biblioteca da Arquitetura IXP criada pelo Flex e Bison

Os arquivos que fornecem as especificações para os analisadores léxico e sintático são estruturados apresentando três blocos distintos de código separados pelos caracteres %%. Esses blocos apresentam respectivamente as definições do programa, as regras utilizadas e opcionalmente sub-rotinas de código inseridas, conforme a Figura 3-3. Os trechos de código delimitados por %{ e %} são trechos de códigos em C os quais são copiados sem modificações para os arquivos .cpp dos analisadores léxico e sintático respectivamente.

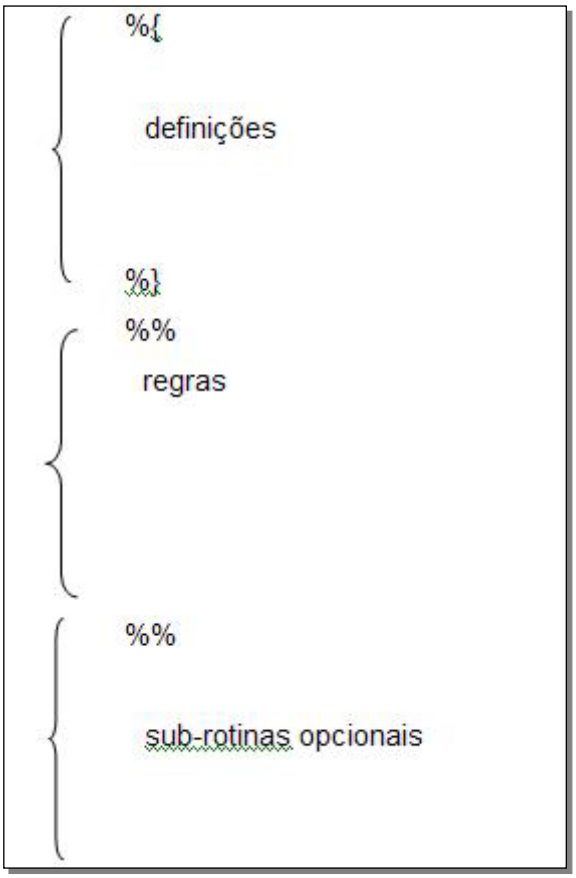

Figura 3-3 – A Construção dos Arquivos do Lexer e Parser

Para a utilização da biblioteca fornecida nesse trabalho, faz-se necessário à utilização de vários arquivos de interesse, pelo projetista da arquitetura, a seguir discriminados: os arquivos de cabeçalho, que descreverão os protótipos das funções declaradas (Tipos.h, Register.h, Memoria.h, CPUcore.h e CPUparser.h); os arquivos que contêm a implementação das classes fornecidas pela biblioteca para implementar os objetos genéricos (Register.cpp, CPUparser.cpp e Memoria.cpp); os arquivos que contêm a implementação das classes escritas pelo usuário para implementar a CPU (CPUcore.cpp e) e a classe que implementa a VM propriamente dita (VM.h); os arquivos usados pelo Flex e Bison (CPU.y, CPU.l) e, o arquivo que contém a função main() do programa, a qual chama as funções da biblioteca para executar a emulação.

A rigor, main.ccp não faz parte da biblioteca, de forma que o projetista pode colocar a função main() em um outro arquivo sem afetar a funcionalidade do seu projeto A finalidade do arquivo tipos.h é apenas conter as definições dos tipos empregados nas classes da biblioteca, de modo a permitir o emprego de formas mais curtas que as normais para variáveis inteiras de 8, 16, 32 e 64 bits.

Os arquivos Register.h e Register.cpp contêm a declaração e implementação da classe registrador, empregada para implementar registradores genéricos de até 64 bits de largura. Em Memoria.h e Memoria.cpp encontram-se a declaração e implementação da classe de memória, que permite ao projetista criar a estrutura que simula o comportamento da memória RAM.

Os arquivos CPUcore.h e CPUcore.cpp definem a implementação da CPU emulada e o arquivo VM.h declara as variáveis da classe VirtualMachine que contém as definições da estrutura interna da VM sendo emulada, tais como nomes e tamanhos dos registradores, tamanho dos registradores utilizados na arquitetura e instância da CPU.

Os arquivos CPUcore (.h e .c) e VM deverão ser fornecidos, pelo menos parcialmente, pelo projetista para se implementar uma arquitetura. Modelos desses arquivos são fornecidos pela biblioteca e podem ser alterados pelo projetista seguindo as características apresentadas na Seção 3.2. As demais classes (Register, Memory e CPUParser), fazem parte do conjunto básico fornecido pela biblioteca e não precisam, a princípio, ser alteradas pelo projetista. Dessa forma, as duas próximas seções apresentam, respectivamente, as classes básicas (fornecidas como subsídio pela biblioteca – Seção 3.1) e as classes que o projetista deverá fornecer ou gerar (Seção 3.2).

3.1. As Classes Fornecidas pela Biblioteca

As classes implementadas pela biblioteca oferecem ao programador objetos que simulam a funcionalidade dos diversos componentes genéricos que compõem uma arquitetura a ser emulada pela VM. As classes fornecidas são as *BasicRegister, Register<SIZE>, CPUparser* e *Memory*. A partir dessas classes o projetista implementa as demais classes, CPU e VirtualMachine.

3.1.1. As Classes Register e BasicRegister

Essas classes possibilitam ao programador criar objetos que simulam a funcionalidade dos registradores de uma CPU. A classe *Register<SIZE>,*

implementada como um template,[1] possibilita implementar registradores de qualquer tamanho até o limite 64 bits.

O usuário da biblioteca, ao declarar uma instância dessa classe, deve informar o parâmetro *SIZE* que definirá o tamanho do registrador que será usado por instâncias dessa classe.

Uma das características de generalidade e facilidade de uso da biblioteca está na flexibilidade na manipulação dos objetos que representam os registradores. A esse respeito, ressalta-se a capacidade de manipulação de registradores genéricos de qualquer tamanho, de maneira uniforme, permitindo, inclusive a definição e execução de operações que envolvem registradores de tamanhos diferentes de forma automática. Essa característica tornou-se possível com a implementação da classe básica *BasicRegister*, da qual a classe *Register<SIZE>* é uma subclasse.

A classe *BasicRegister* fornece uma área genérica de armazenamento, implementada por uma variável (*Bits*), que é herdada por suas subclasses. Todas as operações básicas sobre registradores são definidas sobre parâmetros da classe *BasicRegister* e implementadas como métodos na Classe *Register<SIZE>.* Como essas operações manipulam parâmetros da superclasse *BasicRegister*, torna-se possível, tomando os devidos cuidados na verificação do espaço disponível em tempo de execução, implementá-las de forma independente do tamanho. Com isso, pode-se, por exemplo, somar um registrador de 8 bits com outro de 16 bits e armazenar o resultado em um terceiro registrador de 32 bits diretamente e com segurança.

Apesar de todas as funções da classe Register <SIZE> serem declaradas com parâmetros do tipo *BasicRegister*, em tempo de execução, as instâncias passadas a essas funções serão de fato instâncias da sua subclasse Register<SIZE>, isto é, registradores de qualquer tamanho (definido por <SIZE>). Isso permitirá que, em todas as operações da classe, o dado lido ou escrito no registrador tenha o tamanho adequado, em relação ao que foi declarado. Isso é garantido com a utilização de uma máscara de bits (MASK), que serve para manter apenas os bits que de fato são utilizados.

Dessa forma, todos métodos de *Register<SIZE>* têm como sua ultima linha o comando "Bits &= MASK", ou seja, é feito um AND binário entre a máscara e o

[1] Templates em C++ permitem a definição de classes parametrizadas, isto é, é possível definir toda a implementação em função de parâmetros que serão informados apenas quando uma instância da classe for criada.

conteúdo do registrador , que está armazenado na variável Bits. A Figura 3-4 ilustra uma linha de código da implementação genérica da operação ADD no registrador.

```
template<uint16 SIZE>
void Register<SIZE>::Add(BasicRegister a, BasicRegister b)
{
        Bits = a.Bits + b.Bits;
        Bits &= MASK;
}
}
```

Figura 3-4 – Exemplo de Uso de Parâmetros da Classe BasicRegister e da Máscara de Bits

Todas as funções da classe empregam a máscara para garantir que o dado ou escrito ou lido no registrador tenha o tamanho adequado ao que foi declarado para o registrador.

As operações implementadas na classe *Register<SIZE>* servem ao propósito de permitir ao usuário da classe manipular objetos *Register<SIZE>* da mesma forma que se manipula os tipos primitivos do C++ como inteiros ou caracteres. Se não fosse assim, para cada registrador possível no escopo da biblioteca, seria necessário especificar uma versão de cada operação implementada com o tamanho adequado. Também foram tomadas precauções no sentido simplificar a notação de quem usa a biblioteca por meio da sobrecarga de operadores, sempre que possível. Na classe Register <SIZE>, por exemplo, essa sobrecarga foi realizada sobre o operador de igual (=), permitindo, por exemplo, atribuir um valor numérico diretamente a um registrador ou copiar o conteúdo de um registrador para outro diretamente. Dessa forma, tornaram-se possíveis atribuições do tipo AX=constante ou AX=BX. A Figura 3-5 ilustra essa sobrecarga do operador igual.

```
template<uint16 SIZE> class Register : public BasicRegister
{
        public:
                Register();
                void   Write(uint64);
                uint64 Read(void);
                uint64 MASK;
                void   Copy(BasicRegister);
                void   Add(BasicRegister, BasicRegister);
                void   Add(BasicRegister);
                void   Sub(BasicRegister, BasicRegister);
                void   Sub(BasicRegister);
              :
              :
                BasicRegister& operator=(const uint64);
                BasicRegister& operator=(const BasicRegister);
              :
};
```

Figura 3-5– Exemplo da Sobrecarga de Operadores

3.1.2. A Classe Memory

A classe *Memory* tem por objetivo fornecer um objeto que simula a funcionalidade da memória RAM. Por ser um objeto genérico, essa classe deve ser capaz de aceitar quaisquer tamanhos de palavra e endereçamento especificados pelo usuário que projeta a arquitetura.

A implementação da classe *Memory* foi concebida a partir da funcionalidade da classe *Register<SIZE>*. A classe foi implementada como um *template*. e, dessa forma, em termos de armazenamento, a memória nada mais é do que um vetor de tamanho definido pelo projetista, cujos elementos (palavras da memória) nada mais são do que registradores (de qualquer tamanho, implementados pela classe *Register<SIZE>*).

```
template <class type, uint64 _MEMLENGTH_> class Memory
{
        public:
                uint16 accessResult;
                type   Array[_MEMLENGTH_];
                type&  operator[](uint64);
                void   dump();
                void   Write(uint64, BasicRegister);   // Escreve na memória
                BasicRegister   Read(uint64, BasicRegister);  // Lê da memória
};
```

Figura 3-6 – O Protótipo da Classe Memória

Os parâmetros da classe Memory são *class type* e *uint64 _MEMLENGTH_*. A Figura 3-6 ilustra parte do código do protótipo da classe *Memory*:

O parâmetro *type* deve sempre ser especificado como sendo da classe *Register<SIZE>*, onde *SIZE* é o tamanho da palavra de memória utilizada. É muito comum que palavras de memória sejam formatadas com a unidade básica de 1 byte, o que, freqüentemente, levará o parâmetro *type* a ser definido como *Register<8>.* Nada impede, porém, que qualquer valor diferente seja usado.

O parâmetro _MEMLENGTH_ é o tamanho do banco de memória, isto é, o número de endereços de memória disponível que, na implementação atual, foi definido como um inteiro sem sinal de 64 bits, o que é muito maior do que os computadores de 32 bits atuais podem endereçar. Com isso, a classe se torna extensível, podendo ser usada em implementações de computadores de 64 bits.

Como a classe *Memory* foi construída como um vetor de *Register <SIZE>*, as sobrecargas de operadores implementadas na classe *Register,<SIZE>* podem ser utilizadas nessa classe da mesma forma, e todas as operações que se pode fazer sobre um registrador também podem ser realizadas sobre uma posição de memória. Assim pode-se escrever diretamente no registrador fazendo uso da sobrecarga do operador “=” (igual) a partir de um endereço do vetor de memória. Um exemplo dessa atribuição seria RAM.Array[0xABCD] = AX onde AX é um registrador.

Adicionalmente, foi implementada na classe Memory, a sobrecarga do operador [], usada para permitir acessos ao banco de memória da mesma forma que se faz acesso a um vetor, como ilustrado na Figura 3-6. Por exemplo, se a memória declarada tiver o nome RAM, as sobrecargas dos operadores “=” e “[ ] “ permitem atribuições com a notação RAM [0xABCD] = 255 para acessar o endereço “0xABCD” e atribuir diretamente o valor “255”.

A função para escrita genérica no banco de memória (*void Write(uint64, BasicRegister)*, recebe como parâmetros o *endereço,* onde será realizada a escrita, e o *registrador,* de onde se origina o dado sendo escrito na memória. Na ilustração da Figura 3-7, a função emprega a variável _MEMLENGTH_ para validar se o endereço solicitado é válido. Em caso positivo, a execução prossegue para o cálculo do número de elementos de memória que serão escritos. Por exemplo, se a fonte for um registrador de 32 bits e a unidade básica da memória for de 8 bits, então 4 unidades de memória serão escritas nessa operação.

```
template <class type, uint64 _MEMLENGTH_>
void Memory<type, _MEMLENGTH_>::Write(uint64 address, BasicRegister x)
{
        // Verifica se o endereço base é válido
        if ((address < _MEMLENGTH_) && (address >= 0))
        {
                uint16 RegWidth, Quotient, MemWidth, num_elem;
                RegWidth = x.RegSize;                           // Largura do registrador
                MemWidth = Array[0].RegSize;            // Largura da unidade de memória
                Quotient = RegWidth / MemWidth;
                // Cálculo do número de elementos de memória a serem acessados
                if((RegWidth % MemWidth) == 0)
                        num_elem = Quotient;
                else
                        num_elem = Quotient + 1;
                // Verifica se o último endereço a ser acessado é válido
                if((address + num_elem - 1) < _MEMLENGTH_)
                {
                        for(int i = 0;i < num_elem; i++)
                        {
                                this->Array[address] = x;
                                x.Bits = (x.Bits >> MemWidth);
                                address++;
```

Figura 3-7 – A Função de Escrita Genérica

Conseqüentemente, deve-se validar a escrita de todas as unidades, verificando se o último elemento sendo escrito está no espaço de endereçamento válido, para evitar estouros de buffer (*buffer overflow*). O estado da operação é indicado pela variável *accessResult* que é alterada caso todos os elementos sendo escritos estejam dentro do espaço de endereçamento válido. Assim, a operação de escrita é realizada em todos os elementos necessários para acomodar o tamanho do registrador de origem.

A função para leitura genérica no banco de memória *(BasicRegister Read(uint64, BasicRegister))*, recebe como parâmetros o *endereço,* de onde será feita a leitura, e o *registrador,* no qual o dado lido da memória será escrito (para saber o tamanho do destino). Após a leitura, a função escreve e retorna o resultado como um *BasicRegister*, que é o tipo de registrador genérico independente de tamanho no qual se baseiam todos os demais registradores.

```
template <class type, uint64 _MEMLENGTH_>
BasicRegister Memory<type, _MEMLENGTH_>::Read(uint64 address, BasicRegister x)
{
        // Verifica se o endereço base é válido
        if ((address < _MEMLENGTH_) && (address >= 0))
        {
                uint16 num_shifts, RegWidth, Quotient, MemWidth, num_elem;
                uint64 MASK;
                RegWidth = x.RegSize;                           // Largura do registrador
                MemWidth = Array[0].RegSize;          // Largura da unidade de memória
                Quotient = RegWidth / MemWidth;
                // Cálculo do número de elementos de memória a serem acessados
                if((RegWidth % MemWidth) == 0)
                        num_elem = Quotient;
                else
                        num_elem = Quotient + 1;
                // Verifica se o último endereço a ser acessado é válido
                if((address + num_elem - 1) < _MEMLENGTH_)
                {
                        num_shifts = 64 - RegWidth;
                        MASK = 0xFFFFFFFFFFFFFFFF;
                        MASK >>= num_shifts;
                        x = this->Array[address++];
                        for(int i = 1;i < num_elem; i++)
                        {
                                x.Bits += (this->Array[address].Bits << (MemWidth*i));
                                address++;
                        }
                        x.Bits &= MASK;
```

Figura 3-8 – A Função de Leitura Genérica

A função de leitura, na Figura 3-8, através da informação do tamanho do registrador de destino, calcula quantos elementos serão lidos e retornando esse valor. Caberá ao chamador da função indicar onde o dado retornado será armazenado (normalmente, o mesmo registrador que foi passado como parâmetro à função, ou qualquer um de mesmo tamanho). Todas as validações de acesso são idênticas às da função de escrita. Como a função retorna o valor lido em uma variável local BasicRegister que não é previamente inicializada, se o acesso falhar, o valor retornado será imprevisível, e o chamador da classe deve se certificar que o acesso foi bem sucedido, lendo o valor de accessResult.

3.1.3.
A Classe CPUParser e a Construção dos Arquivos do Lexer e Parser

A biblioteca disponibiliza ainda a classe CPUparser, criada para funcionar como uma interface dos analisadores léxico e sintático para o mundo externo. Esta classe está definida no arquivo CPUParser.h e implementada em CPUParser.cpp.

Durante a execução do programa, após ler um código de operação (*opcode*), o emulador chama o analisador léxico que interpreta quem é a instrução a ser executada, como parte do ciclo de busca da VM emulada. Ao decidir o que fazer, o analisador léxico retorna um *token* para o analisador sintático sendo que cada instrução tem um *token* único, o que as diferencia. Com a informação desse *token*, o analisador sintático salta para o trecho de código correspondente à instrução atual, e inicia sua execução. Ao término da execução desta instrução, ele retorna para a função que o chamou, na classe CPUcore, e o ciclo de busca se reinicia, processando, desta forma, instrução por instrução do código da RAM emulada.

As descrições para geração dos analisadores léxico e sintático pelas ferramentas Flex e Bison estão contidas, respectivamente, nos arquivos CPU.l e CPU.y. A partir da compilação desses arquivos, são gerados os seguintes arquivos fonte C++: CPU.l.cpp (a partir do.l), CPU.cpp.h e CPU.cpp (a partir do.y).

O arquivo CPU.l é a especificação do que o usuário deseja em termos de funcionalidade para seu analisador léxico, tendo o código fonte para o Flex, que produzirá a implementação do analisador léxico correspondente às instruções de máquina definidas pelo projetista da CPU (no arquivo CPU.l.cpp).

A biblioteca implementada neste trabalho já fornece modelos prontos para a especificação do arquivo CPU.l, sobre os quais o projetista de uma arquitetura pode efetuar as modificações que julgar necessárias, em geral, o projetista não precisa alterar o primeiro bloco de código contendo as definições do analisador léxico, já que elas são um padrão já definido pelo próprio ambiente (implementação do Flex para o Microsoft Visual C++ 6.0) no momento em que se cria um projeto que usa o Flex. No campo das regras (segundo bloco de código) do arquivo CPU.l, o projetista deverá definir as regras para informar ao Flex como reconhecer e particionar as operações e seus operandos na arquitetura emulada. O bloco de sub-rotinas opcionais não é utilizado. Os códigos das operações (*opcodes)* deverão ser definidos de acordo com o manual do

processador, pois são esses os valores armazenados na memória RAM e lidos pelo processador emulado para posterior execução. Assim, no arquivo CPU.l, a maior modificação a ser feita é a escrita dos códigos de operação (*opcodes*) da arquitetura emulada.

Os *tokens* usados pelo analisador sintático também devem ser declarados no arquivo *CPU.y*, conforme as especificações do Bison. A seguir, na Figura 3-9, encontra-se um exemplo genérico de declaração dos *opcodes*:

```
%%
[ \n] ;
[\r] ;
10 {return WRITE_AX;}
11 {return WRITE_BX;}
12 {return WRITE_MEM;}
13 {return LOAD_AX;}
14 {return LOAD_BX;}
15 {return STORE_AX;}
16 {return STORE_BX;}
    :
    :
    :
[a-zA-Z]+ {printf("\nInvalid OpCode: %s", yytext);}
[0-9]     {printf("\nInvalid OpCode: %s", yytext);}
59        {printf("\nInvalid OpCode: %s", yytext);}
[6-9][0-9] {printf("\nInvalid OpCode: %s", yytext);}

%%
```

Figura 3-9 – A Declaração de Opcodes em CPU.l

O arquivo CPU.y corresponde à descrição da gramática que permitirá a implementação do analisador sintático com as instruções de máquina definidas pelo projetista da CPU. Ao ser processado pelo Bison, o arquivo CPU.y gera os arquivos CPU.cpp.h e CPU.cpp.

O arquivo CPU.y é o que sofre a maior parte das modificações ao se implementar uma nova arquitetura, já que nele são fornecidos os trechos de código que corresponderão à implementação das instruções propriamente ditas. A Figura 3-10 ilustra um trecho de código de CPU.y da arquitetura hipotética de testes, com a declaração dos tokens do analisador sintático, ainda na Seção de definições do arquivo.

```
%token WRITE_AX WRITE_BX WRITE_MEM
%token LOAD_AX LOAD_BX
%token STORE_AX STORE_BX
%token ADD_AX_MEM ADD_BX_MEM ADD_AX_BX ADD_MEM_MEM
```

Figura 3-10 – A Declaração de Tokens em CPU.y

Da mesma forma que no lexer, no bloco de código inicial de CPU.y (definições) só há alterações se for necessária a implementação de alguma funcionalidade que não esteja já incluída como padrão. As referências às variáveis externas não devem ser modificadas e ainda podem ser incluídas algumas variáveis auxiliares, se for o caso, que serão dependentes da implementação. Ainda no campo de definições são inseridas as definições dos tokens que o usuário deseja implementar e que serão utilizados pelos analisadores léxico e sintático. Os *tokens* são os valores que o analisador léxico retorna ao analisador sintático e esse, por sua vez, os emprega para decidir qual instrução executar. No trecho de código ilustrado na Figura 3-10, retirado da arquitetura de testes (que será detalhada no Capítulo 4), alguns *tokens* que representam as instruções de entrada e saída daquela implementação são ilustrados. Nesse ponto, são declarados todos os *tokens* referentes a todas as instruções que foram efetivamente implementadas, lembrando apenas que esses *tokens* serão usados como valores de retorno pelo analisador léxico e como referências às instruções no bloco de regras do arquivo CPU.y.

No segundo bloco de código (regras) do arquivo CPU.y será fornecido, especificamente, o código para a implementação das instruções emuladas. Nesse bloco, chamadas aos métodos definidos pelas demais classes da biblioteca podem ser inseridas de forma a fornecer a semântica das operações definidas pelo processador. Finalmente, no campo de sub-rotinas (terceiro bloco de código) há uma função de tratamento de erro gerada pelo parser que precisa ser alterada.

Na Figura 3-11, ilustra-se um trecho do segundo bloco contido em um arquivo CPU.y. A barra vertical "|" indica "*ou*" o que significa que "*instruction*" pode ser qualquer uma das opções abaixo.

```
%%
instruction:
        | WRITE_AX    {printf("\n O valor lido foi %X\n", CPU8.CPU.ax);
                CPU8.CPU.cycle_count = CPU8.CPU.cycle_count - CPU8.CPU.writeRegCycles;
                YYACCEPT;}
        | WRITE_BX    {printf("\n O valor lido foi %X\n", CPU8.CPU.bx);
                CPU8.CPU.cycle_count = CPU8.CPU.cycle_count - CPU8.CPU.writeRegCycles;
                YYACCEPT;}
        | WRITE_MEM   {address = CPU8.RAM.Read(CPU8.CPU.program_counter, address);
                printf("\n O valor lido foi %X\n", CPU8.RAM[address.Bits]);
                CPU8.CPU.program_counter += 4;
                CPU8.CPU.cycle_count = CPU8.CPU.cycle_count - PU8.CPU.writeMemCycles;
                YYACCEPT;}
        | LOAD_AX     {address = CPU8.RAM.Read(CPU8.CPU.program_counter, address);
                CPU8.CPU.ax = CPU8.RAM[address.Bits];
                CPU8.CPU.program_counter += 4;
            .   CPU8.CPU.cycle_count = CPU8.CPU.cycle_count - CPU8.CPU.loadCycles;
            .    YYACCEPT;}
            .
            .
        | HALT          {printf("\n O programa terminou e a CPU parou!\n\n");
                        exit(0);
                        YYACCEPT;};
%%
```

Figura 3-11 – A Implementação dos Opcodes em CPU.y

Os nomes depois da barra são os tokens do parser declarados logo acima, e o código da instrução é colocado entre os “{}”. A última linha de cada instrução YYACCEPT é um apelido do Yacc para return(1). Essa linha faz o parser retornar, encerrando a execução da instrução atual e não pode ser removida de nenhuma instrução sob pena de o parser funcionar de forma incorreta.

3.2. As Classes Fornecidas pelo Projetista

Conforme foi mencionado na Seção 3.1, o ambiente aqui proposto especifica a existência de dois conjuntos de classes: as *fornecidas* pela biblioteca (*BasicRegister, Register<SIZE>, CPUparser e Memory* – tratadas na Seção 3.1*) e* as *que são de responsabilidade do* usuário projetista implementar. Nessa Seção, serão tratadas as classes CPU e VirtualMachine, que pertencem ao segundo conjunto

3.2.1.
A Classe CPUCore

Esta classe é a implementação, pelo projetista da arquitetura, da CPU para a VM. Nessa classe é realizado todo o controle de execução da CPU e também contém a declaração da coleção de registradores utilizados, O controle de execução da CPU é definido pelo método “execute”, que utiliza o analisador sintático para implementar o ciclo de busca executado na VM. Existe ainda a necessidade de se definir o tamanho do opcode que ser quer implementar em função da arquitetura emulada.

Todos os registradores utilizados na implementação são dependentes da arquitetura emulada e são declarados como instâncias da classe *Register*, herdando assim toda a sua funcionalidade, conforme detalhado na Seção 3.1.1.

No próximo trecho de código, contido na Figura 3-12, tirado da arquitetura hipotética de teste, as declarações *Register<16>* e *Register<32>* ilustram que os registradores dessa arquitetura hipotética

```
class CPUcore
{
   public:
                CPUcore();                  // Construtor da Classe
                long execute(long);         // Controla a execução da CPU
                void reset_cpu(void);       // Reseta a CPU
                uint64 program_counter;     // Contador de programa
                uint64 stack_pointer;       // Ponteiro da pilha
                uint32 flags;               // Registrador de flags
                uint8 opcode;               // Registrador de Instruções
                long cycle_count;           //Conta o número de ciclos executados
                Register<16> ax;            // Cria os registradores da CPU
                Register<32> bx;            // Cria os registradores da CPU
                CPUParser parser;           // Cria o Parser Yacc

                // Variáveis para armazenar o número de ciclos de cada instrução
};
#endif // CPUCORE_H
```

Figura 3-12 – O Protótipo da Classe CPUcore

3.2.2.
A Classe VirtualMachine

A implementação da classe VirtualMachine deve ser fornecida pelo programador de forma a conter toda a funcionalidade da Máquina Virtual (VM) emulada.

Essa classe fornece o encapsulamento para a CPU e memória criadas para a VM. A Figura 3-13 ilustra a criação dessa classe. No código são mostradas as instâncias para a CPU emulada a partir da classe CPUcore, e da RAM emulada, a partir das classes Memory e Register <SIZE>.

```
// Esta classe constrói a Máquina Virtual
class VirtualMachine
{
        public:
                CPUcore CPU;                                    // Cria a CPU
                Memory<Register<8>,MEM_SIZE> RAM;       // Cria a memória principal
};
#endif // VM_H
```

Figura 3-13 – A Declaração da Classe VirtualMachine

Finalmente, a instância da Máquina Virtual projetada é feita a partir dessa classe no programa principal. Como exemplo, é exibida a instância da VM implementada para a arquitetura hipotética de testes, chamada de CPU8, conforme ilustrado na Figura 3-14.

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "Tipos.h"                  // Inclui meus Typedefs
#include "CPUcore.h"                // Inclui a classe CPUcore
#include "VM.h"                     // Inclui a Máquina Virtual
#include "Register.cpp"
#include "Memoria.cpp"

VirtualMachine CPU8;

void
main(int argc, char** argv)
{
        long num_ciclos;
        Register<32> r32;

        /* Inicializa os valores dos ciclos de cada instrução
        Se o projetista ignorar este passo, o programa automaticamente
        designará valores default para cada instrução */
        CPU8.CPU.writeRegCycles = 10;
        CPU8.CPU.writeMemCycles = 10;
      :
      :
      :
        num_ciclos = CPU8.CPU.execute(101);
        cout << "\n****Executando o Push e o Pop****\n";
        cout << "\nArmazena um valor em AX, empurra (PUSH) pra pilha,";
        cout << "\nsalva novo valor em AX, e restaura (POP) o valor anterior";
        cout << "\na partir da pilha!\n";
        num_ciclos = CPU8.CPU.execute(69);
}
```

Figura 3-14 – A Instância da VM Emulada a partir da Classe VirtualMachine

3.3. A Contagem de Ciclos Executados pelo Emulador

Em termos de utilidade prática do uso do emulador gerado pela ferramenta, uma das facilidades oferecidas pela biblioteca é a contagem do número de ciclos executados enquanto a simulação estiver rodando. Isso também é necessário para sincronizar a saída do programa na tela, se o projetista desejar implementar uma emulação completa de uma arquitetura, fazendo uso de uma interface gráfica.

Por exemplo, a CPU executa um certo número de instruções (que vai depender da freqüência da CPU emulada), então pára, gera uma interrupção, escreve o conteúdo da memória que será usado para atualizar e volta a executar o mesmo número pré-definido de instruções; então pára novamente e assim por diante. Para que isso funcione corretamente é crucial saber quantos ciclos o

núcleo foi instruído a executar e quantos ele efetivamente executou (alguns programas são severamente dependentes dessa temporização).

Não é incomum que o número de ciclos efetivamente executados seja diferente do que aquele que foi solicitado. Por exemplo, o núcleo recebe uma ordem de executar dez ciclos e executa algumas instruções que consumiram nove desses ciclos. Ao iniciar a execução da próxima instrução, ainda existe um ciclo disponível; então o núcleo inicia a execução dessa instrução; no entanto, se essa consumir mais do que um ciclo, digamos quatro, o núcleo terá executado três ciclos a mais do que o ordenado. Esse tipo de situação deve ser levada em consideração ao temporizar as atualizações de saída do programa do emulador.

O núcleo recebe instruções através da função *execute()* da classe *CPUcore*. No fragmento de código abaixo, a instância do núcleo é denominada de CPU simplesmente, e a chamada como segue:

```
num_ciclos = CPU8.CPU.execute(10);
```

Figura 3-15 – A Definição do Número de Ciclos em CPU.y

A variável *num_ciclos* armazena a diferença entre o número de ciclos solicitados (no caso da Figura 3-15, 10 ciclos) e o número de ciclos efetivamente executados (no caso do exemplo relatado acima, 13). Assim, nesse exemplo, *num_ciclos* tem o valor -3, suficiente para o emulador ter controle de que foram executados mais ciclos do que foram ordenados.

A contagem é feita ao final da execução de cada instrução e o arquivo CPU.y, onde a semântica das instruções foi especificada, possui, na penúltima linha de cada instrução, um comando para atualizar a contagem de ciclos, conforme ilustrado na Figura 3-11. Naquela Figura, CPU8 é o nome da instância da classe *VirtualMachine* que representa a máquina virtual emulada. *CPU8.CPU.cycle_count* é uma variável da classe *CPUcore* na qual é armazenado o valor de ciclos executados (cycle_count é inicializada com o valor de ciclos que o núcleo deve executar e vai sendo decrementada pelo número de ciclos que cada instrução gasta). No fragmento de código é mostrada a contagem de ciclos para a instrução *LOAD* da arquitetura hipotética de testes. Os valores das variáveis de contagem de ciclos, tais como *CPU8.CPU.loadCycles* são inicializados pelo construtor da classe *CPUcore*, conforme definidos pelo usuário projetista da arquitetura.