4
Estudos de Casos

Foram desenvolvidos três estudos de caso. Os dois primeiros tiveram como objetivo a avaliação e depuração do funcionamento da implementação proposta. Já o terceiro diz respeito ao objetivo principal desse trabalho, relacionado à emulação de arquiteturas baseadas em processadores de rede.

O primeiro estudo de caso realizado refere-se a uma arquitetura hipotética, idealizada com o objetivo único de auxiliar no desenvolvimento e testes da biblioteca. Esse desenvolvimento nasceu da necessidade de se buscar uma forma simples de se testar o progresso no desenvolvimento da ferramenta. Para essa finalidade, uma arquitetura complexa como a do IXP (ou mesmo a do MCS85), teria sido excessivamente complexa. A arquitetura escolhida se baseou em um exercício proposto em (Deitel & Deitel, 2001), no Capítulo 5. Trata-se de uma arquitetura baseada em uma CPU com apenas dois registradores (um de 16 e outro de 32 bits) e 49 instruções.

O segundo estudo de caso foi uma implementação do Intel MCS85, também conhecido como 8085, antecessor de 8 bits da família x86, à qual pertencem os modernos 386, 486, Pentium etc. Esse segundo estudo de caso serviu ao propósito de ilustrar o projeto de uma arquitetura genérica utilizando os componentes da biblioteca, bem como o de depurar a biblioteca.

Finalmente, o terceiro estudo de caso foi uma implementação do núcleo ARM (ARM, 2004) de um processador de rede Intel IXP. O núcleo ARM é a entidade que controla todos os demais componentes da arquitetura IXP. A versão da implementação ARM depende da versão do IXP em questão. Versões antigas do IXP(1200) usavam um núcleo StrongARM da Intel (STRONGARM, 2001), implementação da versão 4 da família ARM. Os IXP mais novos (2400 e 2800), empregam um núcleo XScale, implementação da versão 5TE do ARM, que é a especificação mais recente, sendo essa a utilizada na presente implementação.

O terceiro estudo foi o mais complexo dos três, pois o ARM é um processador moderno de 32 bits, com 31 registradores de uso geral, diversos modos de operação, e milhares de combinações para cada uma das 55 famílias de instruções, devido aos diversos tipos de modos de endereçamento e

combinações de registradores possíveis, criando várias possibilidades de execução condicional para cada instrução. Dada a imensa quantidade de instruções, delimitou-se um subconjunto de instruções da CPU para ser efetivamente implementado neste estudo. Em uma análise inicial, decidiu-se implementar uma única variante de cada família, o que totalizaria 55 instruções. Porém, ao longo do desenvolvimento, verificou-se que algumas instruções não tinham aplicação prática no contexto desse trabalho, motivando a decisão de não implementá-las. Com isto, foram efetivamente implementadas 48 instruções, o suficiente para executar o programa exemplo, que consistiu na validação do cabeçalho IP do emissor (Ipv4 Layer 3 Forwarding) na máquina virtual ARM. As instruções não implementadas nesse estudo de caso foram aquelas que tratavam da interação do ARM com co-processadores.

4.1. Arquitetura Hipotética para Testes

O primeiro estudo de caso, como já mencionado, foi motivado por um exercício proposto em (Deitel & Deitel, 2001), doravante denominado *CPU8*. Essa arquitetura evoluiu muito ao longo do trabalho, tendo começado como uma CPU de 8 bits (daí o seu nome original) com apenas um registrador que funcionava como acumulador. Com a evolução, adicionou-se um segundo registrador, de modo que se passou a ter o *AX* como acumulador, no qual era armazenado o resultado das instruções aritméticas, e o *BX* como registrador de uso geral em qualquer instrução.

Em sua versão final, a arquitetura *CPU8* foi modificada para conter um registrador de 16 bits como o acumulador *AX,* o *BX* foi aumentado para 32 bits e o conjunto de instruções foi expandido das 30 iniciais para um total de 49 instruções.

Nas versões iniciais, o Contador de Programa e o Ponteiro de Pilha eram registradores de 32 bits, para poder explorar o potencial da biblioteca, embora nunca tenham sido usados mais do que 1024 endereços nesta arquitetura. Já no fim do ciclo de desenvolvimento da biblioteca, ao se adicionar suporte a registradores de 64 bits, o Contador de Programa e o Ponteiro de Pilha foram modificados para tipos de 64 bits, apenas com o propósito de ilustrar a capacidade da biblioteca.

A Figura 4-1 mostra a declaração dos protótipos das funções que implementam a CPU emulada.

```
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 4-1 – Os Protótipos das Funções da CPU do Estudo de Caso CPU8

Essa arquitetura foi implementada com um registrador de 16 bits (ax) e outro de 32 bits (bx), implementados como instâncias da classe *Register <SIZE>,* já definida na Seção 3.1.1. Os registradores foram inicializados com zero, exceto o ponteiro de pilha que aponta para o topo da memória. As definições dos arquivos do lexer e do parser para esta implementação são ilustradas nas Figuras 4-2 e 4-3 respectivamente.

```
%{
#include <cstdio>
#include "CPU.cpp.h"
extern YYSTYPE CPUlval;
#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
   { \
   int c = getchar(); \
   result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
          ; \
   }
%}
%%
[ \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;}
          :
          :
          :
[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 4-2 – O arquivo CPU.l para o Estudo de Caso CPU8

No bloco de definições do arquivo do lexer (cpu.l) acima, são mantidas as declarações default e no bloco das regras são definidos os opcodes utilizados nas instruções (pelo projetista da arquitetura), ressaltando que nesta implementação as instruções possuem 8 bits de tamanho.

```
%%
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 4-3 – O arquivo CPU.y para o Estudo de Caso CPU8

No arquivo do parser, da mesma forma, não há modificações relevantes nas definições das regras, sendo incluída a definição dos tokens propriamente ditos. Nesta VM (CPU8) cada token é implementado como uma única instrução. Finalmente, no bloco das regras estão inseridas as implementações das instruções, conforme ilustrado na Figura 4-3.

A memória foi definida como uma instância da classe register<SIZE> como um vetor de tamanho MEM_SIZE de 1024 endereços com tamanho de palavra de 8 bits. A classe VirtualMachine encapsula as instâncias criadas de CPU e memória, conforme ilustrado na Figura 4-4.

```
#define MEM_SIZE 1024
// 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 4-4 – A Criação da CPU e Memória para o Estudo de Caso CPU8

```
#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 */
        CPU8.CPU.writeRegCycles = 10;
        CPU8.CPU.writeMemCycles = 10;

        cout << "\t\t\t\tEMULADOR DE CPU\n\n";
        for(int j = 0;j < MEM_SIZE; j++)
                CPU8.RAM[j]=0;

        // CÓDIGO E DADOS DO PROGRAMA
        // armazena A
        CPU8.RAM[0x00000384] = 0x0D;
        // armazena B
        CPU8.RAM[0x00000385] = 0x15;

        // halt
        CPU8.RAM[0x00000202] = 0x58;

        // Imprime o código do programa
        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 4-5 – A Instância da VM CPU8 e os Dados Carregados nos Registradores

O programa de testes executados por esta VM emulada tem por finalidade apenas mostrar o correto funcionamento da arquitetura a partir da execução de todas as operações implementadas.

A VM é criada a partir da instância de CPU8 como subclasse da classe VirtualMachine, conforme ilustrado na Figura 4-5. Assim, os valores são carregados nos registradores e a partir do uso desses registradores, todas as operações são testadas e os resultados mostrados na execução do programa. Nessa VM hipotética foram implementadas operações sobre os registradores e entre registradores e a memória, para funções de leitura, escrita, adição, subtração e operações lógicas, entre outras.

A Figura 4-5 ilustra também fragmentos do código que implementa as operações de soma sobre o registrador AX. Finalmente, a emulação dessa arquitetura implementada é ilustrada na Figura 4-6.

Figura 4-6 – A Execução da Máquina Virtual de CPU8

4.2. Arquitetura baseada no Processador MCS85

O segundo estudo de caso é uma implementação do processador Intel MCS85, também conhecido como 8085. Esse processador tem as seguintes características:

- Um total de 7 registradores de uso geral, de 8 bits: A, B, C, D, E, H, L;
- O registrador A é usado como acumulador nas operações aritméticas e lógicas;
- Os demais podem ser operados aos pares, como se fossem um único registrador de 16 bits: BC, DE e HL;
- Contador de programa (PC) e ponteiro de pilha (SP) de 16 bits. Com isso, a arquitetura permite endereçar até 2^{16} = 65536 endereços;
- Registrador de instrução (IR) de 8 bits, o que permite até 256 códigos de operação (opcodes). No entanto, 10 não são implementados, resultando num total de 246 opcodes.
- Flags condicionais: Zero, Sinal, Paridade, Carry e Carry Auxiliar.

```
class CPUcore
{
  public:
        CPUcore();                      // Construtor da Classe
        long   execute(long);           // Controla a execução da CPU
        void   reset_cpu(void);         // Reseta a CPU
        uint16 opcode;
        uint16 program_counter;         // Contador de programa
        uint16 stack_pointer;           // Ponteiro da pilha
        struct CELL
                {
                uint8 sign     :1;
                uint8 zero     :1;
                uint8 b5       :1;
                uint8 aux_carry :1;
                uint8 b3       :1;
                uint8 parity   :1;
                uint8 b1       :1;
                uint8 carry    :1;
                } flags;                // Registrador de flags
        long   cycle_count;             // Conta o número de ciclos executados
        Register<8> A, B, C, D, E, H, L;// Cria os registradores de 8 bits da CPU
        Register<16> BC, DE, HL;        // Cria os registradores de 16 bits da CPU
        CPUParser parser;               // Cria o Parser Yacc

        // Variáveis para armazenar o número de ciclos de cada instrução
                uint16 MOVRegRegCycles;
                uint16 MOVRegMemCycles;
            :
            :
            :
                uint16 MOVMemRegCycles;
};
#endif // CPUCORE_H
```

Figura 4-7 –Os Protótipos das Funções da CPU do Estudo de Caso MCS85

Na Figura 4-7 mostramos a definição da classe *CPUcore*, que especifica o núcleo da CPU emulada, com os respectivos protótipos das funções membros da classe da arquitetura do MCS85.

```
#define MEM_SIZE 65536
// 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 4-8 – A Criação da CPU e Memória para o Estudo de Caso MCS85

Os registradores foram implementados como instâncias da classe *Register<SIZE>* e a memória foi definida como um vetor de tamanho MEM_SIZE de 65536 endereços, com tamanho de palavra de 8 bits.

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

VirtualMachine MCS85;

void
main(int argc, char** argv)
{
        long num_ciclos;
        Register<16> r16;
        Register<8> a, b;
        Register<9> c;

        cout << "\t\t\t\t\tEMULADOR DE MCS85\n\n";
        for(int j = 0;j < SIZE; j++)
                MCS85.RAM[j]=0;

        // CÓDIGO DO PROGRAMA
        // MVI_A A
        MCS85.RAM[0x0000] = 0x3E;
        MCS85.RAM[0x0001] = 0x1F;
        // MVI_B B
        MCS85.RAM[0x0002] = 0x06;
        MCS85.RAM[0x0003] = 0x34;
    :
    :
    :
        // halt
        MCS85.RAM[0x0031] = 0x76;
        num_ciclos = MCS85.CPU.execute(1000);
}
```

Figura 4-9 – A Instância da VM MCS85 e os Dados Carregados nos Registradores

De maneira análoga à implementação da arquitetura hipotética, a classe VirtualMachine encapsula as instâncias criadas de CPU e memória, conforme ilustrado na Figura 4-8.

A VM é criada a partir da instância de MCS85 como uma subclasse de VirtualMachine, conforme ilustrado na Figura 4-9.

As definições dos arquivos do lexer e do parser para essa implementação são ilustradas nas Figuras 4-10 e 4-11 respectivamente.

```
%{
#include <cstdio>
#include "CPU.cpp.h"
#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
   { \
   int c = getc(stdin); \
   result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
          ; \
   }
%}
%%
[ \n] ;
[\r] ;
00 {return NOP;}
%{ // Opcodes Inexistentes %}
08 ;                                                    CB ;
10 ;                                                    D9 ;
18 ;                                                    DD ;
28 ;                                                    ED ;
38 ;                                                    FD
 ;
%{ // OPERAÇÕES DE TRANSFERÊNCIA DE DADOS %}
%{ // LXIs - Load Register Pair Imediate  %}
01 {return LXIRegData;}
11 {return LXIRegData;}
21 {return LXIRegData;}
31 {return LXIRegData;}
%{ // STAX  - Store Accumulator Indirect  %}
02 {return STAX;}
12 {return STAX;}


%{ // ADC %}
88 {return ADCReg;}
89 {return ADCReg;}

%{ // Diversos %}
C3 {return JUMP;}
76 {return HALT;}
%%
```

Figura 4-10 – O arquivo CPU.l para o Estudo de Caso MCS85

O bloco de definições do arquivo do lexer (cpu.l) é mantido sem nenhuma modificação e no bloco das regras foram definidos os opcodes utilizados nas instruções, ressaltando que, nessa implementação, as instruções possuem 8 bits de tamanho.

```
%{
#include <iostream.h>
#include <cstdio>
#include <cstdlib>
#include "VM.h"          // Inclui a classe CPUcore
#include "Tipos.h"               // Inclui meus Typedefs
Register<16> address, destino16, fonte16;
Register<8> destino8, fonte8;
Register<9> AUX9;
Register<4> AUX4;
Register<3> ACC3, AUX3;
uint8 aux, aux1;
extern VirtualMachine MCS85;
extern int CPUerror(const char* s);
extern int CPUlex();
extern YYSTYPE CPUlval;
%}
%token MOVRegReg, MOVRegMem, MOVMemReg, MVIRegData, MVIMemData
%token LXIRegData, LDA, STA, LHLD, SHLD, LDAX, STAX, XCHG
%token ADDReg, ADDMem, ADI, ADCReg, ADCMem, ACI
%token SUBReg, SUBMem, SUI, SBBReg, SBBMem, SBI
%token INRReg, INRMem, DCRReg, DCRMem, INX, DCX, DAD, DAA
%token ANAReg, ANAMem, ANI, XRAReg, XRAMem, XRI, ORAReg, ORAMem
%token CMPReg, CMPMem, CPI, RLC, RRC, RAL, RAR, CMA, CMC, STC
%token JUMP, JUMPCond, CALL, CALLCond, RET, RETCond, RST, PCHL
%token PUSH, PUSH_PSW, POP, POP_PSW, XTHL, SPHL, INport, OUTport
%token EI, DI, HLT, NOP, RIM, SIM
%token RESET HALT
%%
instruction:
        | POP_PSW   {
                    MCS85.CPU.cycle_count = MCS85.CPU.cycle_count -
                            MCS85.CPU.POP_PSWCycles;
                    YYACCEPT;}
        | XTHL      {
                    MCS85.CPU.cycle_count = MCS85.CPU.cycle_count -
                            MCS85.CPU.XTHLCycles;
                    YYACCEPT;}
        | HALT      {printf("\n O programa terminou e o MCS85 parou!\n\n");
                    exit(0);
                    YYACCEPT;}
%%
extern "C"
{
int
```

Figura 4-11 – O arquivo CPU.y para o Estudo de Caso MCS85

Na construção do arquivo do parser são declaradas algumas variáveis auxiliares no bloco de definições, juntamente com a definição dos tokens propriamente ditos.

Nas definições das regras estão inseridas as implementações das instruções. Nessa arquitetura, de maior complexidade do que a anterior, o código implementado foi desenvolvido de tal forma que, após o lexer correlacionar o número do opcode lido com uma das regras existentes, ele identifica o token correspondente àquele número, retornando o token ao parser.

Na implementação dessa arquitetura, ao contrário do que acontece na CPU8, cada token não corresponde a uma única instrução implementada e assim cada token retornado ao parser pode identificar uma família de opcodes, ao invés de um único. Isso é efetivamente implementado no bloco das regras com o uso de uma estrutura de *switch* ... *case* nas referidas implementações, conforme mostrado na Figura 4-11.

Assim o parser necessita não só do token como também do valor numérico do opcode, para poder decidir qual instrução a ser executada entre as disponíveis na família de instruções vinculada àquele token.


```
// CODIGO DO PROGRAMA
        // MVI_A A
        MCS85.RAM[0x0000] = 0x3E;
        MCS85.RAM[0x0001] = 0x1F;
        // MVI_B B
        MCS85.RAM[0x0002] = 0x06;
        MCS85.RAM[0x0003] = 0x34;
        // MVI_C C
        MCS85.RAM[0x0004] = 0x0E;
        MCS85.RAM[0x0005] = 0x56;
        // MVI_D D
        MCS85.RAM[0x0006] = 0x16;
        MCS85.RAM[0x0007] = 0x78;
        // MVI_E E
        MCS85.RAM[0x0008] = 0x1E;
        MCS85.RAM[0x0009] = 0x9A;
        // MVI_H H
        MCS85.RAM[0x000A] = 0x26;
        MCS85.RAM[0x000B] = 0xBC;
        // MVI_L L
        MCS85.RAM[0x000C] = 0x2E;
        MCS85.RAM[0x000D] = 0xDE;
        // ADDs Regs
        MCS85.RAM[0x000E] = 0x80;
        MCS85.RAM[0x000F] = 0x81;
        MCS85.RAM[0x0010] = 0x82;
        MCS85.RAM[0x0011] = 0x83;
        MCS85.RAM[0x0012] = 0x84;
        MCS85.RAM[0x0013] = 0x85;
        MCS85.RAM[0x0014] = 0x87;

        // MVI_H H
        MCS85.RAM[0x0015] = 0x26;
        MCS85.RAM[0x0016] = 0x00;
        // MVI_L L
        MCS85.RAM[0x0017] = 0x2E;
        MCS85.RAM[0x0018] = 0x00;
        // ADD Mem
        MCS85.RAM[0x0019] = 0x86;
```

Figura 4-12 – As Operações de MOV e ADD em MCS85

A VM emulada para o MCS85 permitiu a implementação de todas as instruções previstas para esta arquitetura. De forma semelhante à

implementação da arquitetura hipotética, inicialmente são carregados os valores nos registradores, a partir dos quais as operações são exercitadas, mostrados na Figura 4-12.

Nessa VM do MCS85 foram implementadas operações sobre os registradores e entre registradores e a memória, para funções de leitura, escrita, adição, subtração e operações lógicas, entre outras.

A Figura 4-13 ilustra a execução da VM que implementa o MCS85, exibindo o nome da instrução executada, os valores contidos nos registradores bem como os valores das flags da CPU.

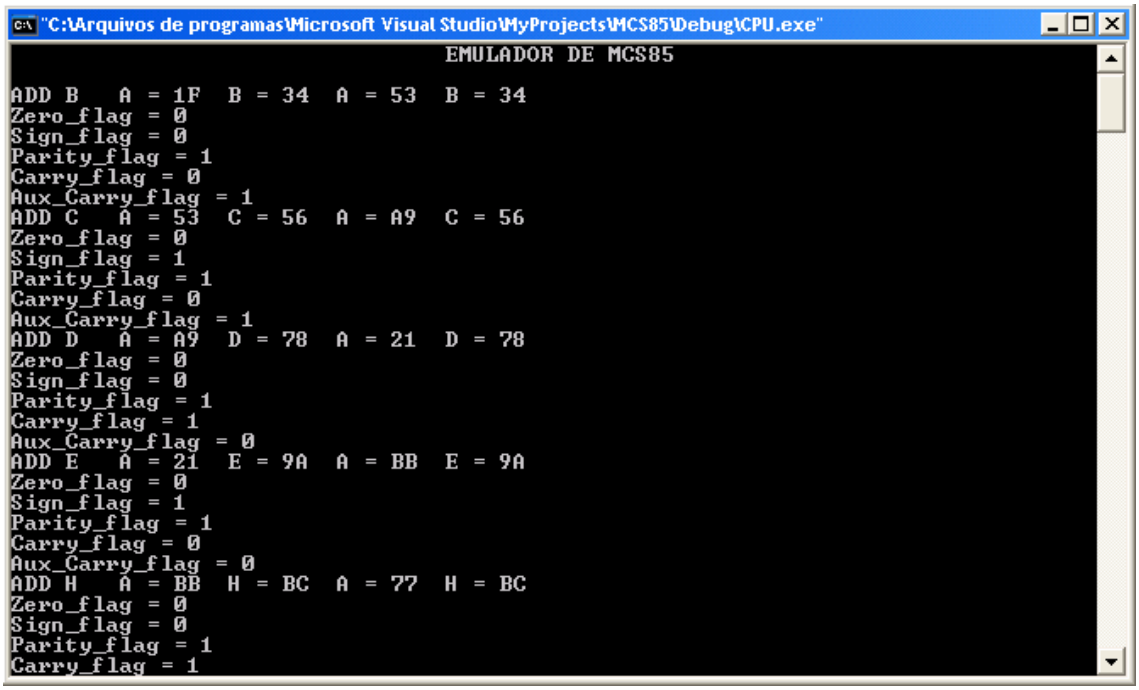

Figura 4-13 – A Execução da Máquina Virtual do MCS85

4.3. Arquitetura baseada no Núcleo ARM do Processador IXP

Nesse estudo de caso, implementou-se um subconjunto das instruções da família v5TE do ARM, que é a implementação mais recente da Intel. A Figura 4-14 ilustra essa arquitetura. O processador emulado apresenta as seguintes características, conforme descritas em (Charitakis et al.,2003).

- 31 Registradores de uso geral, de 32 bits, normalmente utilizando apenas 16 registradores (geralmente apenas 16 são visíveis);

- Contador de programa (PC), no caso sendo um dos registradores de uso geral (R15). Vale ressaltar que o programador pode usar este registrador de forma genérica, como qualquer outro, porém isso pode ter resultados imprevisíveis;
- O ponteiro de pilha (SP) nesse caso não é pré-definido. Normalmente, os programas empregam R13 para esse propósito, mas isso não é amarrado pela especificação da ARM;
- Registradores de Estado: são 6 (seis) registradores de estado (1 Current Program Status Register – CPSR e 5 Saved Program Status Registers – SPSR);
- Registrador de instrução (IR) de 32 bits.

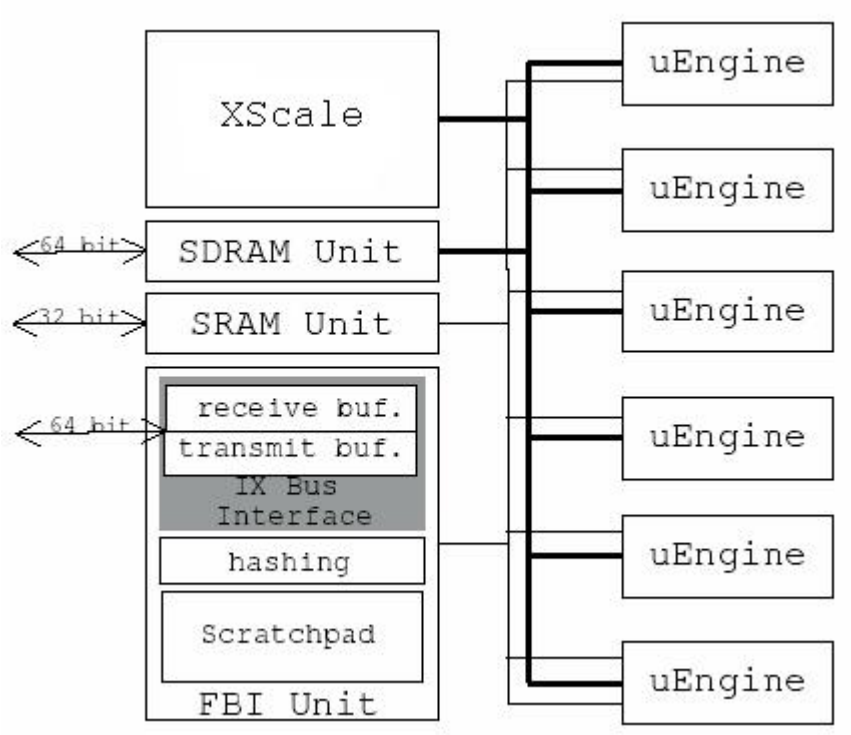

Figura 4-14 – A Arquitetura do Processador IXP

A definição da classe *CPUcore*, que especifica a CPU emulada para o ARM no IXP com os respectivos protótipos das funções membros da classe, é ilustrada na Figura 4-15.

```
class CPUcore
{
  public:
                CPUcore();              // Construtor da Classe
                long   execute(long);   // Controla a execução da CPU
                void   reset_cpu(void); // Reseta a CPU
                uint32 opcode;
                long   cycle_count;     // Conta o número de ciclos executados
                Register<32> R00, R01, R02, R03, R04;// Registradores gerais de 32 bits CPU
                Register<32> R05, R06, R07, R08, R09;// Registradores gerais de 32 bits CPU
                Register<32> R10, R11, R12, R13, R14;// Registradores gerais de 32 bits CPU
                Register<32> PC15;                   // Contador de Programa (R15)
                Register<32> R13_svc, R14_svc;       // Registradores do Modo Supervisor
                Register<32> R13_abt, R14_abt;       // Registradores do Modo Abort
                Register<32> R13_und, R14_und;       // Registradores do Modo Undefined
                Register<32> R13_irq, R14_irq;       // Registradores do Modo Interrupt
                Register<32> R08_fiq, R09_fiq, R10_fiq; // Registradores do Modo Fast Interrupt
                Register<32> R11_fiq, R12_fiq, R13_fiq; // Registradores do Modo Fast Interrupt
                Register<32> R14_fiq;                // Registradores do Modo Fast Interrupt
                Register<32> CPSR;                   // Current Program Status Register
                Register<32> SPSR_svc, SPSR_abt;     // Saved Program Status Registers
                Register<32> SPSR_und, SPSR_irq;     // Saved Program Status Registers
                Register<32> SPSR_fiq;               // Saved Program Status Registers
                CPUParser parser;                    // Cria o Parser Yacc

                // Variáveis para armazenar o número de ciclos de cada instrução
              : uint16 ADCCycles;
              :
              : uint16 UMULLCycles;
};
#endif // CPUCORE_H
```

Figura 4-15 –Os Protótipos das Funções da CPU do Estudo de Caso IXP

Todos os registradores foram implementados da mesma forma que na arquitetura do MCS85, como instâncias da classe *Register* <*SIZE*> já definida na Seção 3-1. A memória foi definida como um vetor de tamanho MEM_SIZE de 65536 endereços com tamanho de palavra de 8 bits, a partir de uma instância da classe Memory.

A implementação do IXP, bem como as citadas anteriormente, traz a classe VirtualMachine encapsulando as instâncias criadas de CPU e memória, conforme ilustrado na Figura 4-16.

```
#define MEM_SIZE 65536
// 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 4-16 – A Criação da CPU e Memória para o Estudo de Caso IXP

A VM é criada a partir da instância de IXP como uma subclasse de VirtualMachine, conforme ilustrado na Figura 4-17.

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

VirtualMachine IXP;

void
main(int argc, char** argv)
{
        long num_ciclos;
        Register<32> r32;
        cout << "\t\t\t\tEMULADOR DE IXP\n\n";
        for(int j = 0;j < MEM_SIZE; j++)
                IXP.RAM[j]=0;

        // Inicializa os Registradores
        // Conteúdo de R00
        r32 = 0x00000004;
        IXP.RAM.Write(0x00000A00, r32);
        // Conteúdo de R01
        r32 = 0x000000A0;
        IXP.RAM.Write(0x00000A04, r32);
        // Ao inicializar, o LDM usa R01 como índice.
        IXP.CPU.R01 = 0x000009FC;

        // Carrega na memória o pacote IP
        IXP.RAM[0x000000A0] = 0x54; // Versão e IHL
        IXP.RAM[0x000000A1] = 0xE0  // Tipo de Serviço (Network Control Packet)
        IXP.RAM[0x000000A2] = 0x06; // Comprimento Total do pacote
        IXP.RAM[0x000000A3] = 0x00; // Comprimento Total do pacote ( 6 palavras de 32 bits)
        IXP.RAM[0x000000A4] = 0x23; // Identificação
        IXP.RAM[0x000000A5] = 0x06; // Identificação
        IXP.RAM[0x000000A6] = 0x15; // Fragment Offset
        IXP.RAM[0x000000A7] = 0x00; // Fragment Offset
        IXP.RAM[0x000000A8] = 0xFF; // Time To Live (TTL)
        IXP.RAM[0x000000A9] = 0x06; // Protocolo (no caso, TCP)
        IXP.RAM[0x000000AA] = 0x00; // Header Checksum
        IXP.RAM[0x000000AB] = 0x00; // Header Checksum
        // Endereço da fonte (192.168.12.225)
        IXP.RAM[0x000000AC] = 0xE1;         // Source Address
        IXP.RAM[0x000000AD] = 0x0C;         // Source Address
        IXP.RAM[0x000000AE] = 0xA8;         // Source Address
        IXP.RAM[0x000000AF] = 0xC0;         // Source Address
        // Endereço do destino (192.168.207.202)
        IXP.RAM[0x000000B0] = 0xCA;         // Destination Address
        IXP.RAM[0x000000B1] = 0xCF;         // Destination Address
        IXP.RAM[0x000000B2] = 0xA8;         // Destination Address
        IXP.RAM[0x000000B3] = 0xC0;         // Destination Address
        // Dados do Pacote
        IXP.RAM[0x000000B4] = 0x21;
        IXP.RAM[0x000000B5] = 0x43;
        IXP.RAM[0x000000B6] = 0x65;
        IXP.RAM[0x000000B7] = 0x87;
}
```

Figura 4-17 – A Instância da VM do IXP e os Dados Carregados nos Registradores

Os arquivos do lexer e do parser para esta implementação com suas respectivas definições encontram-se, respectivamente, nas Figuras 4-18 e 4-19.

```
%%
[ \n] ;
[\r] ;
F7F000F0 {return HALT;}
%{ // ADC %}
([0-9][A-F])(2A)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return ADC;}
([0-9][A-F])(2B)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return ADC;}

%{ // ADD %}
([0-9][A-F])(28)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return ADD;}
([0-9][A-F])(29)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return ADD;}
%{ // LDC %}
([0-9][A-F])(C1)([0-9][A-F])([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return LDC;}
([0-9][A-F])(C3)([0-9][A-F])([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return LDC;}
([0-9][A-F])(C5)([0-9][A-F])([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return LDC;}
([0-9][A-F])(C7)([0-9][A-F])([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return LDC;}
([0-9][A-F])(C9)([0-9][A-F])([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return LDC;}
      :
      :
      :
      :
      :
%{ // ADD & STRH %}
([0-9][A-F])(08)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return MISTOS;}
%{ // ADC & STRH %}
([0-9][A-F])(0A)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return MISTOS;}
%{ // SMULL & SBC & STRH %}
([0-9][A-F])(0C)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return MISTOS;}
%{ // SMLAL & RSC & STRH %}
([0-9][A-F])(0E)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return MISTOS;}
%{ // ORR & STRH %}
([0-9][A-F])(18)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return MISTOS;}
%{ // MOV & STRH %}
([0-9][A-F])(1A)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return MISTOS;}
%{ // BIC & STRH %}
([0-9][A-F])(1C)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return MISTOS;}

%{ // AND %}
([0-9][A-F])(00)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return AND;}
([0-9][A-F])(20)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return AND;}
([0-9][A-F])(21)([0-9][A-F])([0-9][A-F])((([0-9]+)|([A-F]+))+) {return AND;}
%%
```

Figura 4-18 – O arquivo CPU.l para o Estudo de Caso IXP

Nessa implementação as instruções possuem 32 bits de tamanho. No bloco de definições do arquivo do lexer (cpu.l) não foram necessárias modificações. Já no bloco das regras, foram definidos os opcodes utilizados nas instruções. Devido à grande complexidade da arquitetura emulada e da vasta possibilidade de opcodes utilizados em função da existência de 55 famílias de opcodes para esta especificação do ARM, o código de especificação do analisador léxico gerou um número expressivo de regras. A Figura 4-18 exemplifica parte deste código.

No bloco de definições das regras estão inseridas as implementações das instruções. Nessa arquitetura, a exemplo da anterior (MCS85), o código implementado faz com que o lexer, após correlacionar o número do opcode lido com uma das regras existentes, identifique o token correspondente àquele número e retorne esse token ao parser.

```
%{
#include <iostream.h>
#include <cstdio>
#include <cstdlib>
#include "VM.h"                    // Inclui a classe CPUcore
#include "Tipos.h"                 // Inclui meus Typedefs
Register<8> aux8;
Register<16> register_list, aux16;
Register<32> address, aux32;
Register<33> aux33;
uint32 aux, signal;
extern VirtualMachine IXP;
extern int CPUerror(const char* s);
extern int CPUlex();
extern YYSTYPE CPUlval;
%}
%token ADC, AND, ADD, BRANCH, BRANCH_LINK, BIC, BKPT, BLX1, BLX2, BX,
CDP_MCR_MRC
%token CLZ, CMN, CMP, EOR, LDC, LDM1, LDM2, LDM3, LDR, LDRB, LDRBT, LDRH
%token LDRSB, LDRSH, LDRT, MLA, MOV, MRS, MSR, MUL, MVN, ORR
%token RSB, RSC, SBC, SMLAL, SMULL, STC, STM1, STM2, STR, STRB, STRBT, STRH
%token STRT, SUB, SWI, SWP, SWPB, TEQ, TST, UMLAL, UMULL
%token MISTOS, HALT

%%
instruction:
        | ADC             {printf("\nADC\n");
                           switch((IXP.CPU.opcode)&0xFFFFF000)
                                   {case 0x0000:
                                    break;
                                   }
                           IXP.CPU.cycle_count = IXP.CPU.cycle_count - IXP.CPU.ADCCycles;
                           YYACCEPT;}

        | HALT            {printf("\n O programa terminou e o IXP parou!\n\n");
                           exit(0);
                           YYACCEPT;}
 %%
 extern "C"
 {
 int
 CPUwrap()
 {
   return 1;
 }
 }
 int
 CPUerror(const char *s)
 {
cerr << "Syntax error: " << s << endl;
       return 0;
}
```

Figura 4-19 – O arquivo CPU.y para o Estudo de Caso IXP

Isso é necessário porque, nesse caso, cada token pode corresponder não apenas a uma única instrução e sim identificar uma família de opcodes.

Assim, como no caso do MCS85, o parser, de posse do token juntamente com seu valor numérico de *opcode* associado, tem subsídios para decidir qual instrução a ser executada entre as disponíveis na família de instruções vinculada àquele token. Essa implementação do analisador sintático é exemplificada na Figura 4-19.

4.3.1. A Implementação de Um Programa Executado no IXP

O programa a ser executado na Máquina Virtual (VM) do IXP, implementado para testar esta arquitetura, valida o cabeçalho de um pacote IP versão 4. O checksum do cabeçalho é calculado pelo somatório de todos os conjuntos de 16 bits do cabeçalho.

O código apresentado nesta implementação foi baseado em (Seshadri & Lipasti, 2002). O pacote está armazenado na memória da VM a partir do endereço 0xA0.

```
// Ao inicializar, o LDM usa R01 como índice.
IXP.R01 = 0x000009FC;

// Carrega na memória o pacote IP
IXP.RAM.Array[0x000000A0] = 0x54;   // Versão e IHL
IXP.RAM.Array[0x000000A1] = 0xE0;   // Tipo de Serviço (Network Control Packet)
IXP.RAM[0x000000A2] = 0x06;         // Comprimento Total do pacote
IXP.RAM[0x000000A3] = 0x00;         // Comprimento Total do pacote (6 palavras de 32 bits)
IXP.RAM[0x000000A4] = 0x23;         // Identificação
IXP.RAM[0x000000A5] = 0x06;         // Identificação
IXP.RAM[0x000000A6] = 0x15;         // Fragment Offset
IXP.RAM[0x000000A7] = 0x00;         // Fragment Offset
IXP.RAM[0x000000A8] = 0xFF;         // Time To Live (TTL)
IXP.RAM[0x000000A9] = 0x06;         // Protocolo (no caso, TCP)
IXP.RAM[0x000000AA] = 0x00;         // Header Checksum
IXP.RAM[0x000000AB] = 0x00;         // Header Checksum
// Endereço da fonte (192.168.12.225)
IXP.RAM[0x000000AC] = 0xE1;         // Source Address
IXP.RAM[0x000000AD] = 0x0C;         // Source Address
IXP.RAM[0x000000AE] = 0xA8;         // Source Address
IXP.RAM[0x000000AF] = 0xC0;         // Source Address
// Endereço do destino (192.168.207.202)
IXP.RAM[0x000000B0] = 0xCA;         // Destination Address
IXP.RAM[0x000000B1] = 0xCF;         // Destination Address
IXP.RAM[0x000000B2] = 0xA8;         // Destination Address
IXP.RAM[0x000000B3] = 0xC0;         // Destination Address
// Dados do Pacote
IXP.RAM[0x000000B4] = 0x21;
IXP.RAM[0x000000B5] = 0x43;
IXP.RAM[0x000000B6] = 0x65;
IXP.RAM[0x000000B7] = 0x87;
```

Figura 4-20 – Código IXP carregando o Programa

O código que carrega o pacote na memória virtual, em preparação para a execução do programa é ilustrado na Figura 4-20 e os valores dos campos do cabeçalho, são os seguintes:

- *Version = 04*
- *IHL = 05 (cabeçalho sem o campo de Options)*
- *Comprimento Total do Pacote = 0x06 = 06 palavras de 32 bits*
- *Identificação = 0x0623*
- *Fragment Offset = 0x0015*
- *Time To Live (TTL) = 0xFF*
- *Protocolo = 0x06 (TCP)*
- *Header Checksum = 0x0000 (conforme livro do Tanenbaum)*
- *Source Address = 192.168.12.225 (C0.A8.0C.E1)*
- *Destination Address = 192.168.207.202 (C0.A8.CF.CA)*

```
// CODIGO DO PROGRAMA

// LDM1 (Carrega todos os registradores)
r32 = 0xE9917FFF;
IXP.RAM.Write(0x00, r32);
// LDR R2
r32 = 0xE6912000;
IXP.RAM.Write(0x04, r32);
// LDR R3
r32 = 0xE6913000;
IXP.RAM.Write(0x08, r32);
// LDR R4
r32 = 0xE6914000;
IXP.RAM.Write(0x0C, r32);
// LDR R5
r32 = 0xE6915000;
IXP.RAM.Write(0x10, r32);
// LDR R6
r32 = 0xE6916000;
IXP.RAM.Write(0x14, r32);
// AND R7 (Extrai o IHL - Header Length)
r32 = 0xE002700B;
IXP.RAM.Write(0x18, r32);
// MOV com shift right 4
r32 = 0xE1A07227;
IXP.RAM.Write(0x1C, r32);

// Começa o cálculo do checksum
// ADD (R10 = R02 + R03)
r32 = 0xE092A003;
IXP.RAM.Write(0x20, r32);
// ADC (R10 = R10 + R04)
r32 = 0xE0BAA004;     // Cálculo do checksum
IXP.RAM.Write(0x24, r32);
// ADC (R10 = R10 + R05)
r32 = 0xE0BAA005;     // Cálculo do checksum
IXP.RAM.Write(0x28, r32);
// ADC (R10 = R10 + R06)
r32 = 0xE0BAA006;     // Cálculo do checksum
IXP.RAM.Write(0x2C, r32);
// AND (R00 = R10 & R12) R12 = 0xFFFF0000
r32 = 0xE00A000C;     // Cálculo do checksum
IXP.RAM.Write(0x30, r32);
// MOV R0, R0 com shift right 16
r32 = 0xE1A00820;
IXP.RAM.Write(0x34, r32);
// AND (R10 = R10 & R14) R14 = 0x0000FFFF
r32 = 0xE00AA00E;     // Cálculo do checksum
IXP.RAM.Write(0x38, r32);
// ADD (R10 = R10 + R06)
r32 = 0xE09AA000;     // Cálculo do checksum
IXP.RAM.Write(0x3C, r32);
// AND (R00 = R10 & R13) R14 = 0x00010000
r32 = 0xE00A000D;     // Cálculo do checksum
IXP.RAM.Write(0x40, r32);
// MOV R0, R0 com shift right 16
r32 = 0xE1A00820;
IXP.RAM.Write(0x44, r32);
// AND (R10 = R10 & R14)
r32 = 0xE00AA00E;     // Cálculo do checksum
IXP.RAM.Write(0x48, r32);
// ADD (R10 = R10 + R00)
r32 = 0xE09AA000;     // Cálculo do checksum
IXP.RAM.Write(0x4C, r32);
// halt
r32 = 0xF7F000F0;
IXP.RAM.Write(0x50, r32);

//          EXECUTA A SIMULAÇÃO!
printf("\n*** Carregando os valores dos registradores a partir da memoria! ***\n");
num_ciclos = IXP.execute(2);
printf("\n*** Carregando o cabeçalho do pacote IP nos registradores! ***\n");
num_ciclos = IXP.execute(10);
printf("\n*** Extrai o Internet Header Length (IHL)! ***\n");
num_ciclos = IXP.execute(4);
printf("\n*** Calcula o Checksum do pacote! ***\n");
num_ciclos = IXP.execute(26);
```

Figura 4-21 – Código IXP testando a Validação do Cabeçalho IPv4

A seguir é feita uma análise do programa propriamente dito. O registrador *R01* é usado como ponteiro para o pacote IP na memória, dessa forma, *R01* deve conter o endereço inicial do pacote que é 0xA0. O código do programa é mostrado na Figura 4-21.

A instrução LDM1 carrega todos os dados necessários para a execução do programa, exceto o cabeçalho do pacote IP, o qual será carregado pelos LDR a seguir. Os registradores são inicializados conforme mostrado na Figura 4-22.

```
// Inicializa os Registradores
// Conteúdo de R00
r32 = 0x00000004;    IXP.RAM.Write(0x00000A00, r32);
// Conteúdo de R01
r32 = 0x000000A0;    IXP.RAM.Write(0x00000A04, r32);
// Conteúdo de R02
r32 = 0x00000000;    IXP.RAM.Write(0x00000A08, r32);
// Conteúdo de R03
r32 = 0x00000000;    IXP.RAM.Write(0x00000A0C, r32);
// Conteúdo de R04
r32 = 0x00000000;    IXP.RAM.Write(0x00000A10, r32);
// Conteúdo de R05
r32 = 0x00000000;    IXP.RAM.Write(0x00000A14, r32);
// Conteúdo de R06
r32 = 0x00000000;    IXP.RAM.Write(0x00000A18, r32);
// Conteúdo de R07
r32 = 0x00000000;    IXP.RAM.Write(0x00000A1C, r32);
// Conteúdo de R08
r32 = 0x00000000;    IXP.RAM.Write(0x00000A20, r32);
// Conteúdo de R09
r32 = 0x00000000;    IXP.RAM.Write(0x00000A24, r32);
// Conteúdo de R10
r32 = 0x00000000;    IXP.RAM.Write(0x00000A28, r32);
// Conteúdo de R11
r32 = 0x000000F0;    IXP.RAM.Write(0x00000A2C, r32);
// Conteúdo de R12
r32 = 0xFFFF0000;    IXP.RAM.Write(0x00000A30, r32);
// Conteúdo de R13
r32 = 0x00010000;    IXP.RAM.Write(0x00000A34, r32);
// Conteúdo de R14
r32 = 0x0000FFFF;    IXP.RAM.Write(0x00000A38, r32);

// Ao inicializar, o LDM usa R01 como índice.
IXP.R01 = 0x000009FC;
```

Figura 4-22 – Código IXP inicializando os Registradores

As instruções *LDR* carregam o cabeçalho do pacote IP nos registradores de R02 a R06.

Finalmente, depois de carregar o cabeçalho nos registradores, iniciam-se os cálculos sobre o mesmo. O primeiro passo é extrair o valor do IHL com o *AND R07*, que efetua o *AND* do conteúdo de *R02* (o IHL é um dos campos da palavra de 32 bits armazenada neste registrador) com a máscara 0x000000F0 armazenada no registrador *R11* e armazena o resultado em *R07*.

Após extrair o *IHL*, o programa inicia o cálculo do *Checksum* do pacote, que é calculado efetuando a soma módulo um das palavras de 16 bits do pacote. Nesse sistema de cálculo, é efetuada uma adição binária convencional, mas é necessário somar o *carry_out* da operação de volta no bit menos significativo do resultado, conforme mostrado em (Braden et al., 1998).

O conteúdo do cabeçalho é ilustrado na tabela abaixo, sendo omitido o conteúdo do campo de dados, pois este não participa do cálculo do checksum.

00 06 E0 54 = 000 0000 0000 0110 1110 0000 0101 0100	(R2)
00 15 06 23 = 0000 0000 0001 0101 0000 0110 0010 0011	(R3)
(00 00) 06 FF = (0000 0000 0000 0000) 0000 0110 1111 1111	(R4)
C0 A8 0C E1 = 1100 0000 1010 1000 0000 1100 1110 0001	(R5)
C0 A8 CF CA = 1100 0000 1010 1000 1100 1111 1100 1010	(R6)

Tabela 1: O Conteúdo do Cabeçalho do Pacote IP utilizado no Cálculo de Checksum

A seqüência do cálculo de checksum propriamente dito a ser realizado é ilustrada na tabela abaixo:

aux = R2 + R3
carry_out = CarryFrom(aux)
aux = aux + carry_out
aux = aux + R4
carry_out = CarryFrom(aux)
aux = aux + carry_out
aux = aux + R5
carry_out = CarryFrom(aux)
aux = aux + carry_out
aux = aux + R6
carry_out = CarryFrom(aux)
aux = aux + carry_out
checksum = aux

Tabela 2: O Cálculo de Checksum

A partir desse cálculo o valor de checksum é dividido ao meio, em duas palavras de 16 bits, onde após efetuar todos os cálculos acima mencionados, teríamos o seguinte: checksum = 81 6C CA 22. Esse valor dividido ao meio resulta nos valores: 81 6C e CA 22. Então é feita a soma desses valores, chegando-se ao resultado de 00 01 4B 8E. Como pode ser visto, essa soma de 16 bits produziu um carry_out no bit 17, de modo que o somamos de novo no bit menos significativo, obtendo o resultado final: 4B 8F (RESULTADO FINAL)

Na execução do programa, este resultado estará armazenado em R10 na linha do último ADD a ser executado.

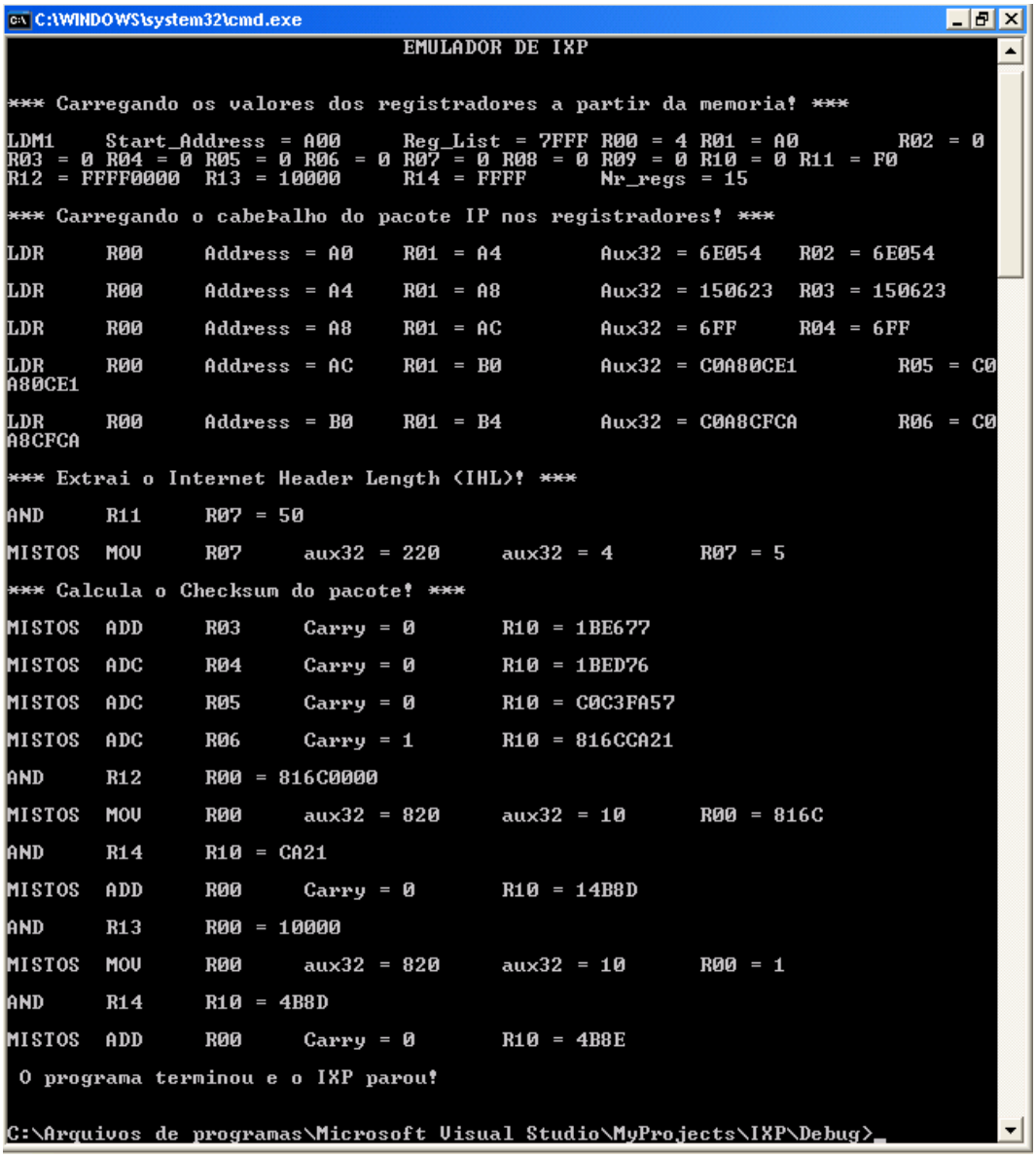

Figura 4-23 – Exemplo do Código IXP implementado

Finalmente é implementada a última instrução, o HALT, que não existe no IXP, tendo sido criada apenas por comodidade para interromper a execução do emulador. Na Figura 4-23 apresentamos o programa do núcleo ARM do IXP emulado.