6
Referências

(Almquist, P., 1992) Almquist, P., **Network Working Request for Comments**: **1349 - Type of Service in the Internet Protocol Suite.** Consultant, USA, July, 1992.Disponível em:
http://www.faqs.org/rfcs/rfc1349.html

(Ayala-Rincón et al., 2003) Ayala-Rincón, M., Hartenstein, R., Neto, R.N., Jacobi, R.P., and Llanos, C. **Architectural Specification, Exploration and Simulation Through Rewriting-Logic.,** In Colombian Journal of Computation, Vol 3(2):20-34, 2003. Disponível em: http://www.unab.edu.co/editorialunab/revistas/rcc/pdfs/r32_art2_r.pdf

(Braden et al., 1998) Braden, R.[1], Borman, D.[2], and Partridge, C.[3], **Network Working Request for Comments: 1071 - Computing the Internet Checksum.** [1]ISI, [2]Cray Research, [3]BBN Laboratories, USA, September, 1988. Disponível em: http://www.faqs.org/rfcs/rfc1071.html

(Charitakis et al., 2003) Charitakis, I.[1], Pnevmatikatos, D.[1], Markatos, E.[1], and Anagnostakis, K.[2], **Code Generation for Packet Header Intrusion.** [1] Institute of Computer Science (ICS) Foundation of Research and Technology - Hellas (FORTH), Crete, Greece, [2]Distributed Systems Laboratory, CIS Department, University of Pennsylvania, Philadelfia, USA. 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2003), Vienna, Austria, September 2003.

(Deitel & Deitel, 2001) Deitel, H.M. & Deitel, P.J, **C++ Como Programar.** 3ª Edição, Editora Bookman, Brasil.

(Freitas & Martins, 2002) Freitas, H.C. e Martins, C.A.P.S. **NPSIM: Simulador de Processador de Rede.** Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, Brasil, 2002.

(John, 2002) John. **Compute 16-bit One's Complement Sum** The Math Forum, Drexel University, USA, 2002. Disponível em: http://mathforum.org/library/drmath/view/54379.html

(Kleitz, 1997) **Microprocessor and Microcontroller Fundamentals: The 8085 and 8051 Hardware and Software.** Pearson Education, 1st edition (August 11, 1997)

(Kounavis et al., 2003) Kounavis, M.E., Campbell, A.T., Chou, S.T. and Vicente, J.B., **A Programming Environment for Network Processors.** Network Processor Conference, West, San Jose, CA, October 2003.

(Levine et al., 1992) Levine, J.R., Mason, T. and Brown, D. **Lex & Yacc.** O'Reilly & Associates Inc, 2ª Edição, Estados Unidos.

(Reynolds et al., 1994) Reynolds, J. and Postel, J., **Network Working Request for Comments: 1700 - Assigned Numbers.** [1]ISI, [2]Cray Research, [3]BBN Laboratories, USA, October, 1994. Disponível em: http://www.faqs.org/rfcs/rfc1700.html

(Seshadri & Lipasti, 2002) Madhusudanan, S and Lipasti, M. **A Case for Vector Network Processors**, University of Wisconsin, Electrical and Computer Engineering, Madison, USA, 1992. Disponível em: http://www.ece.wisc.edu/~pharm/papers/npcwest2002.pdf

(Spalink et al., 2000) Spalink, T., Karlin, S. and Peterson, L., **Evaluating Network Processors in IP Forwarding**, Technical Report, Computer Science Dep, Princeton University, Nov 15 2000.

(ARCHC, 2001) **The ArchC Architecture Description Language.** Disponível em: http://www.archc.org/

(ARM, 2000) **Arm Limited Copyright© (2000) ARM Architecture Reference Manual**. Disponível em: http://www.deetc.isel.ipl.pt/microprocessadores/recursos/arm/arm/DDI0100E_ARM_ARM.pdf

(ARM, 2004) **Arm Corporate Backgrounder (2004).** Disponível em: http://www.arm.com/miscPDFs/3822.pdf

(INTEL IXP1200XX, 2001) **Intel IXP12XX Product Line of Network Processors**. Disponível em: http://www.intel.com/design/network/products/npfamily/ixp1200.htm

(STRONGARM, 2001) Intel Corporation, (Julho, 2001) “Intel® StrongARM SA-1111 Microprocessor Companion Chip Developer's Manual”. Disponível em: http://www.intel.com/design/strong/manuals/278242.htm

(SYSTEMC, 1999**)** SystemC™ Network**.** Disponível em: http://www.systemc.org

7
Apêndice A Código-Fonte da Biblioteca Implementada

A seguir, neste Apêndice são listados todo os códigos-fontes de toda a Biblioteca desenvolvida nesta Dissertação de Mestrado, incluindo os 03 (três) estudos de casos apresentados. Primeiramente são listados os códigos-fontes comuns às 03 implementações e a seguir aqueles que são dependentes da implementação e são alterados em cada Estudo de Caso.

7.1.
Arquivos de Cabeçalho (Código-Fonte) Comuns aos 03 Estudos de Casos

7.1.1.
Arquivo CPUParser.h

```
#ifndef __CPU_PARSER__
#define __CPU_PARSER__
#include "CPU.cpp.h"
int CPUparse();
class CPUParser
{
public:
  CPUParser();
  CPUParser(char* fname);
  int Parse();
};
#endif
```

7.1.2.
Arquivo Memória.h

```
//Memoria.h
//Declaração da classe Memory
#ifndef MEMORIA_H
#define MEMORIA_H
/* As funções de leitura e escrita na memória definem a variável accessResult como 1 (um) se o acesso
for bem sucedido, e como 0 (zero) se o acesso falhar. O cliente da classe deve explicitamente testar o
valor de accessResult para saber se a operação foi ou não bem sucedida! */
#include "Register.h"
```

```
#include "Tipos.h"      // Inclui meus Typedefs
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
};
#endif // MEMORIA_H
```

7.1.3. Arquivo Register.h

```
//Register.h
//Declaração da classe Register
#ifndef REGISTER_H
#define REGISTER_H
#include "Tipos.h"  // Inclui meus Typedefs
// Classe Base, contém a estrutura de armazenagem de dados
class BasicRegister
{
  public:
            uint64 Bits;    // Armazena o conteúdo do registrador
            uint16 RegSize;  // Armazena o tamanho do registrador
};
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);
            void   Mul(BasicRegister, BasicRegister);
            void   Mul(BasicRegister);
            void   Div(BasicRegister, BasicRegister);
            void   Div(BasicRegister);
            void   And(BasicRegister, BasicRegister);
            void   And(BasicRegister);
            void   Xor(BasicRegister, BasicRegister);
            void   Xor(BasicRegister);
            void   Or(BasicRegister, BasicRegister);
            void   Or(BasicRegister);
```

```
        void   Not(BasicRegister);
        void   Not(void);
        void   Neg(BasicRegister);
        void   Neg(void);
        void   ShiftR(BasicRegister, uint16);
        void   ShiftR(uint16);
        void   ShiftL(BasicRegister, uint16);
        void   ShiftL(uint16);
        void   RotateR(BasicRegister, uint16);
        void   RotateR(uint16);
        void   RotateL(BasicRegister, uint16);
        void   RotateL(uint16);
        BasicRegister& operator=(const uint64);
        BasicRegister& operator=(const BasicRegister);
        uint64 operator+(const BasicRegister);
        uint64 operator-(const BasicRegister);
        uint64 operator*(const BasicRegister);
        uint64 operator/(const BasicRegister);
        uint64 operator&(const BasicRegister);
        uint64 operator^(const BasicRegister);
        uint64 operator|(const BasicRegister);
        uint64 operator>>(const uint16);
        uint64 operator<<(const uint16);
        uint64 operator>>(const BasicRegister);
        uint64 operator<<(const BasicRegister);
        BasicRegister& operator~(void);
        BasicRegister& operator+=(const BasicRegister);
        BasicRegister& operator-=(const BasicRegister);
        BasicRegister& operator*=(const BasicRegister);
        BasicRegister& operator/=(const BasicRegister);
        BasicRegister& operator&=(const BasicRegister);
        BasicRegister& operator^=(const BasicRegister);
        BasicRegister& operator|=(const BasicRegister);
        BasicRegister& operator>>=(const uint16);
        BasicRegister& operator<<=(const uint16);
        BasicRegister& operator>>=(const BasicRegister);
        BasicRegister& operator<<=(const BasicRegister);
};
#endif // REGISTER_H
```

7.1.4. Arquivo Tipos.h

```
//Tipos.h
//Declaração dos meus typedefs
#ifndef TIPOS_H
#define TIPOS_H
/* ATENÇÃO: O tipo "__int64" é específico do MS Visual C. Para compilar em outras arquiteturas,
deve-se antes substituí-lo pelo inteiro de 64 bits adequado para o compilador em questão. */
typedef unsigned __int64 uint64;
```

```
typedef unsigned long int uint32;
typedef unsigned short int uint16;
typedef unsigned char uint8;
#endif // TIPOS_H
```

7.2. Arquivos de Cabeçalho (Código-Fonte) do Estudo de Caso da Arquitetura Hipotética

7.2.1. Arquivo CPU.cpp.h

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define  WRITE_AX          258
#define  WRITE_BX          259
#define  WRITE_MEM         260
#define  LOAD_AX           261
#define  LOAD_BX           262
#define  STORE_AX          263
#define  STORE_BX          264
#define  ADD_AX_MEM        265
#define  ADD_BX_MEM        266
#define  ADD_AX_BX         267
#define  ADD_MEM_MEM       268
#define  SUB_AX_MEM        269
#define  SUB_BX_MEM        270
#define  SUB_AX_BX         271
#define  SUB_MEM_MEM       272
#define  AND_AX_MEM        273
#define  AND_BX_MEM        274
#define  AND_AX_BX         275
#define  AND_MEM_MEM       276
#define  XOR_AX_MEM        277
#define  XOR_BX_MEM        278
#define  XOR_AX_BX         279
#define  XOR_MEM_MEM       280
#define  OR_AX_MEM         281
#define  OR_BX_MEM         282
#define  OR_AX_BX          283
#define  OR_MEM_MEM        284
#define  NOT_AX            285
#define  NOT_BX            286
#define  NEG_AX            287
#define  NEG_BX            288
#define  SHIFTR_AX         289
#define  SHIFTL_AX         290
#define  ROTR_AX           291
```

```
#define   ROTL_AX          292
#define   SHIFTR_BX        293
#define   SHIFTL_BX        294
#define   ROTR_BX          295
#define   ROTL_BX          296
#define   ICALL            297
#define   IRET             298
#define   CALL             299
#define   RET              300
#define   PUSH             301
#define   POP              302
#define   PRINT            303
#define   JUMP             304
#define   RESET            305
#define   HALT             306
extern YYSTYPE CPUlval;
```

7.2.2. Arquivo CPUcore.h


```
//CPUcore.h
//Declaração da classe CPUcore
#ifndef CPUCORE_H
#define CPUCORE_H
#include "Register.cpp"    // Inclui a classe Register
#include "CPUParser.h"          // Inclui a classe do Parser Yacc
#include "Tipos.h"              // Inclui meus Typedefs

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
                    uint16 writeRegCycles;
                    uint16 writeMemCycles;
                    uint16 loadCycles;
                    uint16 storeCycles;
                    uint16 addRegMemCycles;
```

```
        uint16 addMemMemCycles;
        uint16 addRegRegCycles;
        uint16 subRegMemCycles;
        uint16 subMemMemCycles;
        uint16 subRegRegCycles;
        uint16 andRegMemCycles;
        uint16 andMemMemCycles;
        uint16 andRegRegCycles;
        uint16 xorRegMemCycles;
        uint16 xorMemMemCycles;
        uint16 xorRegRegCycles;
        uint16 orRegMemCycles;
        uint16 orMemMemCycles;
        uint16 orRegRegCycles;
        uint16 shiftCycles;
        uint16 rotateCycles;
        uint16 notCycles;
        uint16 negCycles;
        uint16 callCycles;
        uint16 retCycles;
        uint16 icallCycles;
        uint16 iretCycles;
        uint16 pushCycles;
        uint16 popCycles;
        uint16 jumpCycles;
};
#endif // CPUCORE_H
```

7.2.3. Arquivo VM.h

```
//VM.h
//Declaração da classe VirtualMachine
#ifndef VM_H
#define VM_H

#include "Memoria.cpp"     // Inclui a classe Memory
#include "CPUcore.h"    // Inclui a classe CPUcore
#include "Tipos.h"                  // Inclui meus Typedefs
#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
```

7.3.
Arquivos de Cabeçalho (Código-Fonte) do Estudo de Caso do MCS85

7.3.1.
Arquivo CPU.cpp.h

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define	MOVRegReg	258
#define	MOVRegMem	259
#define	MOVMemReg	260
#define	VMIRegData	261
#define	VMIMemData	262
#define	LXIRegData	263
#define	LDA	264
#define	STA	265
#define	LHLD	266
#define	SHLD	267
#define	LDAX	268
#define	STAX	269
#define	XCHG	270
#define	ADDReg	271
#define	ADDMem	272
#define	ADI	273
#define	ADCReg	274
#define	ADCMem	275
#define	ACI	276
#define	SUBReg	277
#define	SUBMem	278
#define	SUI	279
#define	SBBReg	280
#define	SBBMem	281
#define	SBI	282
#define	INRReg	283
#define	INRMem	284
#define	DCRReg	285
#define	DCRMem	286
#define	INX	287
#define	DCX	288
#define	DAD	289
#define	DAA	290
#define	ANAReg	291
#define	ANAMem	292
#define	ANI	293
#define	XRAReg	294
#define	XRAMem	295
#define	XRI	296
#define	ORAReg	297
```

```
#define   ORAMem      298
#define   ORI         299
#define   CMPReg      300
#define   CMPMem            301
#define   CPI         302
#define   RLC         303
#define   RRC         304
#define   RAL         305
#define   RAR         306
#define   CMA         307
#define   CMC         308
#define   STC         309
#define   JUMP        310
#define   JUMPCond    311
#define   CALL        312
#define   CALLCond    313
#define   RET         314
#define   RETCond     315
#define   RST         316
#define   PCHL        317
#define   PUSH        318
#define   PUSH_PSW    319
#define   POP         320
#define   POP_PSW     321
#define   XTHL        322
#define   SPHL        323
#define   INport      324
#define   OUTport     325
#define   EI          326
#define   DI          327
#define   HLT         328
#define   NOP         329
#define   RIM         330
#define   SIM         331
#define   RESET       332
#define   HALT        333
extern    YYSTYPE     CPUlval;
```


7.3.2. Arquivo CPUcore.h

```
//CPUcore.h
//Declaração da classe CPUcore
#ifndef CPUCORE_H
#define CPUCORE_H
#include "Memoria.cpp"          // Inclui a classe Memory
#include "Register.cpp"         // Inclui a classe Register
#include "CPUParser.h"          // Inclui a classe do Parser Yacc
```

```
#include "Tipos.h"                  // Inclui meus Typedefs
#define SIZE 65536
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;
            uint16 VMIRegDataCycles;
            uint16 VMIMemDataCycles;
            uint16 LXIRegDataCycles;
            uint16 LDACycles;
            uint16 STACycles;
            uint16 LHLDCycles;
            uint16 SHLDCycles;
            uint16 LDAXCycles;
            uint16 STAXCycles;
            uint16 XCHGCycles;
            uint16 ADDRegCycles;
            uint16 ADDMemCycles;
            uint16 ADICycles;
            uint16 ADCRegCycles;
            uint16 ADCMemCycles;
            uint16 ACICycles;
            uint16 SUBRegCycles;
            uint16 SUBMemCycles;
            uint16 SUICycles;
            uint16 SBBRegCycles;
            uint16 SBBMemCycles;
```

```
uint16 SBICycles;
uint16 INRRegCycles;
uint16 INRMemCycles;
uint16 DCRRegCycles;
uint16 DCRMemCycles;
uint16 INXCycles;
uint16 DCXCycles;
uint16 DADCycles;
uint16 DAACycles;
uint16 ANARegCycles;
uint16 ANAMemCycles;
uint16 ANICycles;
uint16 XRARegCycles;
uint16 XRAMemCycles;
uint16 XRICycles;
uint16 ORARegCycles;
uint16 ORAMemCycles;
uint16 ORICycles;
uint16 CMPRegCycles;
uint16 CMPMemCycles;
uint16 CPICycles;
uint16 RLCCycles;
uint16 RRCCycles;
uint16 RALCycles;
uint16 RARCycles;
uint16 CMACycles;
uint16 CMCCycles;
uint16 STCCycles;
uint16 JUMPCycles;
uint16 JUMPCondTRUECycles;
uint16 JUMPCondFALSECycles;
uint16 CALLCycles;
uint16 CALLCondTRUECycles;
uint16 CALLCondFALSECycles;
uint16 RETCycles;
uint16 RETCondTRUECycles;
uint16 RETCondFALSECycles;
uint16 RSTCycles;
uint16 PCHLCycles;
uint16 PUSHCycles;
uint16 PUSH_PSWCycles;
uint16 POPCycles;
uint16 POP_PSWCycles;
uint16 XTHLCycles;
uint16 SPHLCycles;
uint16 INportCycles;
uint16 OUTportCycles;
uint16 EICycles;
uint16 DICycles;
uint16 HLTCycles;
uint16 NOPCycles;
```

```
        uint16 RIMCycles;
        uint16SIMCycles;
};
#endif      //PUCORE_H
```

7.3.3. Arquivo VM.h

```
//VM.h
//Declaração da classe VirtualMachine
#ifndef VM_H
#define VM_H
#include "Memoria.cpp"     // Inclui a classe Memory
#include "CPUcore.h"   // Inclui a classe CPUcore
#include "Tipos.h"                  // Inclui meus Typedefs
#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
```

7.4. Arquivos de Cabeçalho (Código-Fonte) do Estudo de Caso do IXP

7.4.1. Arquivo CPU.cpp.h

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define    ADC             258
#define    AND             259
#define    ADD             260
#define    BRANCH          261
#define    BRANCH_LINK     262
#define    BIC             263
#define    BKPT            264
#define    BLX1            265
#define    BLX2            266
#define    BX              267
#define    CDP_MCR_MRC     268
#define    CLZ             269
#define    CMN             270
```

```
#define   CMP     271
#define   EOR     272
#define   LDC     273
#define   LDM1    274
#define   LDM2    275
#define   LDM3    276
#define   LDR     277
#define   LDRB    278
#define   LDRBT   279
#define   LDRH    280
#define   LDRSB   281
#define   LDRSH   282
#define   LDRT    283
#define   MLA     284
#define   MOV     285
#define   MRS     286
#define   MSR     287
#define   MUL     288
#define   VMN     289
#define   ORR     290
#define   RSB     291
#define   RSC     292
#define   SBC     293
#define   SMLAL   294
#define   SMULL   295
#define   STC     296
#define   STM1    297
#define   STM2    298
#define   STR     299
#define   STRB    300
#define   STRBT   301
#define   STRH    302
#define   STRT    303
#define   SUB     304
#define   SWI     305
#define   SWP     306
#define   SWPB    307
#define   TEQ     308
#define   TST     309
#define   UMLAL   310
#define   UMULL   311
#define   MISTOS  312
#define   HALT    313
extern YYSTYPE CPUlval;
```

7.4.2. Arquivo CPUcore.h

```
//CPUcore.h
```

```
//Declaração da classe CPUcore
#ifndef CPUCORE_H
#define CPUCORE_H
#include "Memoria.cpp"     // Inclui a classe Memory
#include "Register.cpp"    // Inclui a classe Register
#include "CPUParser.h"             // Inclui a classe do Parser Yacc
#include "Tipos.h"                 // Inclui meus Typedefs
class CPUcore
{
   public:
            CPUcore();                  // Construtor da Classe
            long   execute(long);       // Controla a execução da CPU
            void   reset_cpu(void);     // Reseta a CPU
            long   cycle_count;         // Conta o número de ciclos executados
            uint32 opcode;              // Armazena o valor do opcode sendo executado
            Register<32> R00, R01, R02, R03, R04;       // Registradores gerais de 32 bits da CPU
            Register<32> R05, R06, R07, R08, R09;       // Registradores gerais de 32 bits da CPU
            Register<32> R10, R11, R12, R13, R14;       // Registradores gerais de 32 bits da 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
            struct {
                        uint8 N: 1;
                        uint8 Z: 1;
                        uint8 C: 1;
                        uint8 V: 1;
                        uint8 Q: 1;
                        uint8 I: 1;
                        uint8 F: 1;
                        uint8 T: 1;
                        uint8 M: 5;
            } CPSR;                     // Current Program Status Register
            struct {                    uint8 N: 1;
                        uint8 Z: 1;
                        uint8 C: 1;
                        uint8 V: 1;
                        uint8 Q: 1;
                        uint8 I: 1;
                        uint8 F: 1;
                        uint8 T: 1;
                        uint8 M: 5;
            } SPSR_svc, SPSR_abt, SPSR_und, SPSR_irq, 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 ADDCycles;
uint16 ANDCycles;
uint16 BRANCHCycles;
uint16 BRANCH_LINKCycles;
uint16 BICCycles;
uint16 BKPTCycles;
uint16 BLX1Cycles;
uint16 BLX2Cycles;
uint16 BXCycles;
uint16 CDPCycles;
uint16 CLZCycles;
uint16 CMNCycles;
uint16 CMPCycles;
uint16 EORCycles;
uint16 LDCCycles;
uint16 LDM1Cycles;
uint16 LDM2Cycles;
uint16 LDM3Cycles;
uint16 LDRCycles;
uint16 LDRBCycles;
uint16 LDRBTCycles;
uint16 LDRHCycles;
uint16 LDRSBCycles;
uint16 LDRSHCycles;
uint16 LDRTCycles;
uint16 MCRCycles;
uint16 MLACycles;
uint16 MOVCycles;
uint16 MRCCycles;
uint16 MRSCycles;
uint16 MSRCycles;
uint16 DAACycles;
uint16 MULCycles;
uint16 VMNCycles;
uint16 ORRCycles;
uint16 RSBCycles;
uint16 RSCCycles;
uint16 SBCCycles;
uint16 SMLALCycles;
uint16 SMULLCycles;
uint16 STCCycles;
uint16 STM1Cycles;
uint16 STM2Cycles;
uint16 STRCycles;
uint16 STRBCycles;
uint16 STRBTCycles;
uint16 STRHCycles;
uint16 STRTCycles;
uint16 SUBCycles;
uint16 SWICycles;
uint16 SWPCycles;
```

```
        uint16 SWPBCycles;
        uint16 TEQCycles;
        uint16 TSTCycles;
        uint16 UMLALCycles;
        uint16 UMULLCycles;
};
#endif // CPUCORE_H
```

7.4.3. Arquivo VM.h

```
//VM.h
//Declaração da classe VirtualMachine
#ifndef VM_H
#define VM_H
#include "Memoria.cpp"     // Inclui a classe Memory
#include "CPUcore.h"    // Inclui a classe CPUcore
#include "Tipos.h"                  // Inclui meus Typedefs
#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
```

8
Apêndice B Descrição dos Opcodes Implementados no Núcleo ARM do IXP

Os *opcodes* implementados no estudo de caso do Núcleo ARM do IXP foram desenvolvidos a partir do Manual de Referência da Arquitetura ARM (ARM, 2000), que contém a especificação detalhada do conjunto de instruções da Arquitetura ARM. A seguir são detalhados os *opcodes* utilizados nesta implementação.

1. ADC (ADD with Carry) – Implementado na categoria MISTOS

OPCODES: (1) **E0A1200X** [X = 0 a F]

(2) **E0BAA00X** [X = 0 a F]

(1) Soma R01 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) + a flag de carry (IXP.CPSR.C) e armazena o resultado em R02.

R02 = R01 + CPSR.C + Rxx (xx de 00 a 14 ou PC15)

(2) Soma R10 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) + a flag de carry (IXP.CPSR.C) e armazena o resultado em R10. Atualiza a Flag de Carry. (Nota: esta instrução deveria atualizar todas as flags, mas por simplicidade, somente a flag de carry é atualizada.)

R10 = R10 + CPSR.C + Rxx (xx de 00 a 14 ou PC15)

---

2. ADD – Implementado na categoria MISTOS

OPCODES: (1) **E081200X** [X = 0 a F]

(2) **E081100X** [X = 0 a F]

(3) **E092A00X** [X = 0 a F]

(4) **E09AA00X** [X = 0 a F]

(1) Soma R01 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R02.

R02 = R01 + Rxx (xx de 00 a 14 ou PC15)

(2) Soma R01 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado no próprio R01.

R01 = R01 + Rxx (xx de 00 a 14 ou PC15)

(3) Soma R02 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R10. Atualiza a Flag de Carry. (Esta instrução deveria atualizar todas as flags, mas por limitação da implementação, somente a flag de carry é atualizada).

R10 = R02 + Rxx (xx de 00 a 14 ou PC15)

(4) Soma R10 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R10. Atualiza a Flag de Carry. (Esta instrução deveria atualizar todas as flags, mas por limitação da implementação, somente a flag de carry é atualizada).

R10 = R10 + Rxx (xx de 00 a 14 ou PC15)

---

3. AND

OPCODES: (1) **E001200X** [X = 0 a F]
(2) **E002700X** [X = 0 a F]
(3) **E002800X** [X = 0 a F]
(4) **E002900X** [X = 0 a F]
(5) **E004A00X** [X = 0 a F]
(6) **E00A000X** [X = 0 a F]
(7) **E00AA00X** [X = 0 a F]
(8) **E000700X** [X = 0 a F]

(1) Efetua o AND bit-a-bit de R01 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R02.

R02 = R01 & Rxx (xx de 00 a 14 ou PC15)

(2) Efetua o AND bit-a-bit de R02 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R07.

R07 = R02 & Rxx (xx de 00 a 14 ou PC15)

(3) Efetua o AND bit-a-bit de R02 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R08.

R08 = R02 & Rxx (xx de 00 a 14 ou PC15)

(4) Efetua o AND bit-a-bit de R02 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R09.

R09 = R02 & Rxx (xx de 00 a 14 ou PC15)

(5) Efetua o AND bit-a-bit de R04 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R10.

R10 = R04 & Rxx (xx de 00 a 14 ou PC15)

(6) Efetua o AND bit-a-bit de R10 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R00.

R00 = R10 & Rxx (xx de 00 a 14 ou PC15)

(7) Efetua o AND bit-a-bit de R10 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R10.

R10 = R10 & Rxx (xx de 00 a 14 ou PC15)

(8) Efetua o AND bit-a-bit de R00 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R07.

R07 = R00 & Rxx (xx de 00 a 14 ou PC15)

---

4. BRANCH

OPCODE: **EAXXXXXX** [Cada X = 0 a F]

(1) Estende o valor de 24 bits representado pelos X´s acima para 32 bits, preservando o sinal. P ex, 0x012345 se torna 0x000123456, mas 0x812345 se torna 0xFF812345.

(2) Efetua um SHIFT LEFT 2 do resultado acima (isto é, garante que será múltiplo de 4).

(3) Soma o resultado final ao Contador de Programa (PC15), que contém o endereço da instrução atual (Branch) + 8 (Isso é padronizado). Qualquer leitura do PC no ARM deve retornar sempre o endereço da instrução atual + 8.

---

5. BRANCH & LINK

OPCODE: **EBXXXXXX** [Cada X = 0 a F]

Funcionamento idêntico ao BRANCH, porém armazena o endereço de retorno no Link Register (R14). Útil para chamadas de sub-rotina, pois será necessário o endereço de retorno pra voltar da mesma. O endereço de retorno é o endereço da próxima instrução após o BRANCH&LINK!

---

6. BIC – Implementado na categoria MISTOS

OPCODE: **E1C1200X** [X = 0 a F]

Efetua o AND bit-a-bit de R01 com o COMPLEMENTO do registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R02.

R02 = R01 & $\overline{Rxx}$ (xx de 00 a 14 ou PC15)

---

7. BKPT

OPCODE: **E12XXX7X** [X = 0 a F]

Causa a ocorrência de um breakpoint via software. Esse breakpoint pode ser manipulado por um manipulador de exceção instalado no vetor “prefetch abort”. Em algumas implementações que incluem hardware de depuração, este pode opcionalmente sobrepujar o comportamento padrão da instrução.

```
R14_abt =PC15;
SPSR_abt.N = CPSR.N;
SPSR_abt.Z = CPSR.Z;
SPSR_abt.C = CPSR.C;
SPSR_abt.V = CPSR.V;
SPSR_abt.Q = CPSR.Q;
SPSR_abt.I = CPSR.I;
SPSR_abt.F = CPSR.F;
SPSR_abt.T = CPSR.T;
SPSR_abt.M = CPSR.M;
CPSR.M = 0x17;
CPSR.T = 0;
CPSR.I = 0;
PC15 = 0x0000000C;
```

---

8. BLX1

OPCODE: **FAXXXXXX** [Cada X = 0 a F]

Instrução semelhante ao BRANCH & LINK! Armazena o endereço de retorno em R14.

1) Estende o valor de 24 bits representado pelos X´s acima para 32 bits, preservando o sinal. Por exemplo 0x012345 se torna 0x000123456, mas 0x812345 se torna 0xFF812345.

2) Efetua um SHIFT LEFT 2 do resultado acima (isto é, garante que será múltiplo de 4).

3) Seta o Bit 1 do resultado acima para o valor do bit H do *opcode* (bit 24 da instrução – no caso implementado é zero).

4) Soma o resultado final ao Contador de Programa (PC15), que contém o endereço da instrução atual (Branch) + 8; Isso é padronizado. Qualquer leitura do PC no ARM deve retornar sempre o endereço da instrução atual + 8.

---

9. BLX2

OPCODE: **E12FFF3X** [X = 0 a F]

Armazena o endereço de retorno em R14. Salta a execução para o endereço armazenado no registrado especificado como X no opcode, i e, X = 0, o registrador é R00, X = 1, R01 e assim por diante.

PC15 = Rxx & 0xFFFFFFFE

T flag = bit 0 de Rxx (A flag T especifica se a instrução após o salto é ARM ou THUMB – o conjunto Thumb é um subconjunto das instruções ARM e não foi implementado).

Link Register (R14) = endereço da instrução após o BLX2.

---

10. BX

OPCODE: **E12FFF21X** [X = 0 a F]

Análogo a BLX2, porém não armazena o endereço de retorno.

PC15 = Rxx & 0xFFFFFFFE

T flag = bit 0 de Rxx

11. CDP – Coprocessor Data Processing

OPCODE: **Instrução Não implementada**

---

12. CLZ

OPCODE: **E16F2F1X** [X = 0 a F]

Conta o número de zeros binários à esquerda de um número antes do primeiro 1 binário. Opera sobre um dos 16 registradores de R00 a R15 (PC15), especificados pelo X no *opcode*.

---

13. CMN

OPCODE: **E171000X** [X = 0 a F]

Compara o conteúdo de um registrador com o negativo de outro valor aritmético. Atualiza as flags com base no resultado da soma deste com o conteúdo do registrador.

Alu_out = R01 + Rxx;

CPSR.N = Alu_out[31];

CPSR.Z = 1 se Alu_out = 0 ou 0 caso contrário;

CPSR.C = CarryFrom(R01 + Rxx);

CPSR.V = OverflowFrom(R01 + Rxx);

---

14. CMP

OPCODE: **E151000X** [X = 0 a F]

Compara o conteúdo de um registrador com outro valor aritmético. Atualiza as flags com base no resultado da subtração deste com o conteúdo do registrador.

Alu_out = R01 - Rxx;

CPSR.N = Alu_out[31];

CPSR.Z = 1 se Alu_out = 0 ou 0 caso contrário;

CPSR.C = NOT BorrowFrom(R01 - Rxx);

CPSR.V = OverflowFrom(R01 - Rxx);

---

15. EOR

OPCODE: **E021200X** [X = 0 a F]

Efetua um XOR de R01 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R02.

R02 = R01 XOR Rxx (xx de 00 a 14 ou PC15)

16. LDC – Load Coprocessor

OPCODE: **Instrução Não implementada**

17. LDM1

OPCODE: **E991XXXX** [X = 0 a F]

Carrega múltiplos registradores, especificados pelos X´s com valores obtidos de posições seqüenciais da memória RAM. O endereço base é o valor armazenado em R01 (nesta versão implementada) e os endereços seguintes são obtidos somando-se 4 ao anterior.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	1	0	1	0	1	0	0	1	0

A Figura 8-1 mostra os 16 bits menos significativos da instrução. Os bits zerados significam que esses registradores não serão modificados. Só são alterados aqueles que estão com os bits iguais a 1 (um). No exemplo da Figura, seria feito um LOAD seqüencial a partir de um endereço de memória armazenado em R01 e os registradores a serem atualizados seriam R01, R04, R06, R08, R09, R13, R14 e R15 (PC15).

18. LDM2

OPCODE: **E9D1XXXX** [X = 0 a F]

Semelhante a LDM1, mas o bit 15 do *opcode* é sempre ZERO, ou seja, esta instrução não altera o Contador de Programa.

19. LDM3

OPCODE: **E9D1XXXX** [X = 0 a F]

Semelhante a LDM1. Neste implementação, entretanto, ela é exatamente igual.

---

20. LDR

OPCODE: (1) **E691200X** [X = 0 a F]
(2) **E691300X** [X = 0 a F]
(3) **E691400X** [X = 0 a F]
(4) **E691500X** [X = 0 a F]
(5) **E691600X** [X = 0 a F]
(6) **E691100X** [X = 0 a F]

(1) Carrega um dado de 32 bits de um endereço de memória, cujo valor está armazenado em R01, para o registrador R02. O valor contido no registrador especificado pelo X no opcode acima é somado a R01 após a decodificação do endereço.

(2) Carrega um dado de 32 bits de um endereço de memória, cujo valor está armazenado em R01, para o registrador R03. O valor contido no registrador especificado pelo X no opcode acima é somado a R01 após a decodificação do endereço.

(3) Carrega um dado de 32 bits de um endereço de memória, cujo valor está armazenado em R01, para o registrador R04. O valor contido no registrador especificado pelo X no opcode acima é somado a R01 após a decodificação do endereço.

(4) Carrega um dado de 32 bits de um endereço de memória, cujo valor está armazenado em R01, para o registrador R05. O valor contido no registrador especificado pelo X no opcode acima é somado a R01 após a decodificação do endereço.

(5) Carrega um dado de 32 bits de um endereço de memória, cujo valor está armazenado em R01, para o registrador R06. O valor contido no registrador especificado pelo X no opcode acima é somado a R01 após a decodificação do endereço.

(6) Carrega um dado de 32 bits de um endereço de memória, cujo valor está armazenado em R01, para o registrador R01. O valor contido no registrador especificado pelo X no opcode acima é somado a R01 após a decodificação do endereço.

---

21. LDRB

OPCODE: **E6D1200X** [X = 0 a F]

Idem ao anterior, porém carrega apenas um byte no registrador R02.

---

22. LDRBT

OPCODE: **E6F1200X** [X = 0 a F]

Com as limitações impostas nessa implementação, essa instrução é exatamente igual à anterior. Obviamente, numa implementação completa isso não seria o caso.

---

23. LDRH – Implementado na categoria MISTOS

OPCODE: **E09120BX** [X = 0 a F]

Análogo ao LDRB, porém carrega uma palavra de 16 bits em R02.

---

24. LDRSB – Implementado na categoria MISTOS

OPCODE: **E09120DX** [X = 0 a F]

Análogo ao LDRB, porém lida com dados com sinal.

---

25. LDRSH – Implementado na categoria MISTOS

OPCODE: **E09120FX** [X = 0 a F]

Análogo ao LDRH, porém lida com dados com sinal.

---

26. LDRT

OPCODE: **E6B1200X** [X = 0 a F]

Análogo ao LDR, porém não pode alterar o Program_Counter.

---

27. MCR – Move to Coprocessor from ARM Register

OPCODE: **Instrução Não implementada**

---

28. MLA – Implementado na categoria MISTOS

OPCODE: **E022109X** [X = 0 a F]

Multiplica o conteúdo de R0 com o registrador indicado por X, soma o resultado com R01 e armazena o resultado final em R02.

R02 = (X * R0) + R01

---

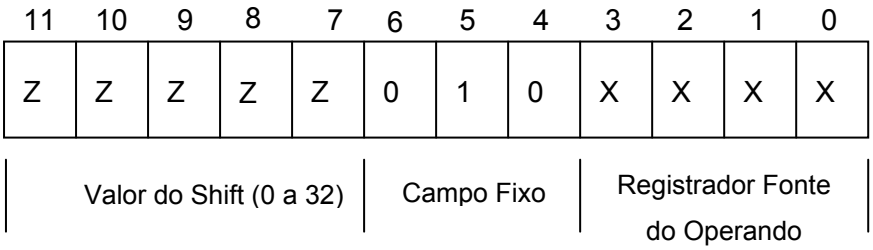
29. MOV – Implementado na categoria MISTOS

OPCODE: **E1A0200X** [X = 0 a F]

Move o conteúdo do registrador dado por X para R02.

OPCODE: (1) **E1A00ZZX** [X = 0 a F] (MOV com Shift Right Lógico)
(2) **E1A06ZZX** [X = 0 a F] (MOV com Shift Right Lógico)
(3) **E1A07ZZX** [X = 0 a F] (MOV com Shift Right Lógico)

(1) Move o conteúdo do registrador X para R00 deslocado pelo valor especificado por ZZ, da seguinte forma:



O campo ZZ do OPCODE acima tem uma parte fixa, os três últimos bits são fixos em 0b010. Só os cinco bits mais significativos codificam o valor do shift. O operando é o registrador dado por X, seu conteúdo é shift pelo valor dado acima e o resultado, armazenado em R00.

(2) Idem ao anterior, porém o registrador de destino é R06.

(3) Idem ao anterior, porém o registrador de destino é R07.

---

30. MRC – Move to ARM Register from Coprocessor

OPCODE: **Instrução Não implementada**

---

31. MRS – Move PSR to General Purpose Register

OPCODE: **Instrução Não implementada**

---

32. MSR – Move to Status Register from ARM Register

OPCODE: **Instrução Não implementada**

---

33. MUL

OPCODE: **E002009X** [X = 0 a F]

Multiplica o conteúdo de R0 com o registrador indicado por X, e armazena o resultado final em R02.

R02 = (X * R0)

---

34. VMN

OPCODE: **E1E0200X** [X = 0 a F]

Move o complemento dois do conteúdo do registrador dado por X para R02.

---

35. ORR - Implementado na categoria MISTOS

OPCODE: **E181200X** [X = 0 a F]

Efetua o OR bit-a-bit de R01 com o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R02.

R02 = R01 OR Rxx (xx de 00 a 14 ou PC15)

---

36. RSB - Implementado na categoria MISTOS

OPCODE: **E061200X** [X = 0 a F]

Subtrai R01 do registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R02.

R02 = Rxx - R01 (xx de 00 a 14 ou PC15)

---

37. RSC - Implementado na categoria MISTOS

OPCODE: **E0E1200X** [X = 0 a F]

Subtrai R01 e o inverso da Flag Carry do registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) e armazena o resultado em R02.

R02 = Rxx - R01 – NOT(Carry Flag) (xx de 00 a 14 ou PC15)

---

38. SBC - Implementado na categoria MISTOS

OPCODE: **E0C1200X** [X = 0 a F]

Subtrai o inverso da Flag Carry e o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) de R01 e armazena o resultado em R02.

R02 = R01 – Rxx - NOT(Carry Flag) (xx de 00 a 14 ou PC15)

---

39. SMLAL - Implementado na categoria MISTOS

OPCODE: **E0E2109X** [X = 0 a F]

Multiplicação de 64 bits. Armazena os 32 bits menos significativos do resultado em R01 e os 32 bits mais significativos em R02.

R01 = (X * R00)[31:0] + R01

R02 = (X * R00)[63:32] + R02 + CarryFrom((X * R00)[31:0] + R01)

---

40. SMULL - Implementado na categoria MISTOS

OPCODE: **E0C2109X** [X = 0 a F]

Multiplicação de 64 bits. Armazena os 32 bits menos significativos do resultado em R01 e os 32 bits mais significativos em R02.

R01 = (X * R00)[31:0]

R02 = (X * R00)[63:32]

---

41. STC - Store Coprocessor

OPCODE: **Instrução Não implementada**

---

42. STM1

OPCODE: **E981XXXX** [X = 0 a F]

Análogo ao LDM1, porém carrega o conteúdo dos registradores na memória.

---

43. STM2

OPCODE: **E9C1XXXX** [X = 0 a F]

Análogo ao STM1, porém user-mode (numa implementação completa). No caso desta implementação é igual ao opcode STM1.

---

44. STR

OPCODE: **E681200X** [X = 0 a F]

Análogo ao LDR, porém carrega o conteúdo dos registradores na memória.

---

45. STRB

OPCODE: **E6C1200X** [X = 0 a F]

Análogo ao LDRB, porém carrega o conteúdo dos registradores na memória.

---

46. STRBT

OPCODE: **E6E1200X** [X = 0 a F]

Análogo ao LDRBT, porém carrega o conteúdo dos registradores na memória.

---

47. STRH- Implementado na categoria MISTOS

OPCODE: **E08120BX** [X = 0 a F]

Análogo ao LDRH, porém carrega o conteúdo dos registradores na memória.

---

48. STRT

OPCODE: **E6A1200X** [X = 0 a F]

Análogo ao LDRT, porém carrega o conteúdo dos registradores na memória.

---

49. SUB

OPCODE: **E041200X** [X = 0 a F]

Subtrai o registrador especificado por X (0 sendo R00; 1, R01, ...; F, PC15 (R15)) de R01 e armazena o resultado em R02.

R02 = R01 - Rxx (xx de 00 a 14 ou PC15)

---

50. SWI – Software Interrupt

OPCODE: **Instrução Não implementada**

---

51. SWP

OPCODE: **E101209X** [X = 0 a F]

Permuta uma palavra de 32 bits entre registradores e memória. No caso desta implementação, salva o conteúdo do registrador X no endereço de memória cujo valor está contido em R01 e o conteúdo antigo deste endereço é colocado em R02.

---

52. SWPB

OPCODE: **E141209X** [X = 0 a F]

Permuta um byte entre registradores e memória. No caso desta implementação, salva o conteúdo do registrador X no endereço de memória cujo valor está contido em R01 e o conteúdo antigo deste endereço é colocado em R02.

---

53. TEQ

OPCODE: **E131000X** [X = 0 a F]

Compara um registrador com outro valor aritmético e atualiza algumas flags.

Aux = R01 XOR Rxx (xx de 00 a 14 ou PC15)

---

54. TST

OPCODE: **E111000X** [X = 0 a F]

Semelhante ao TEQ.

Aux = R01 AND Rxx (xx de 00 a 14 ou PC15)

---

55. UMLAL

OPCODE: **E0A2109X** [X = 0 a F]

Multiplicação de 64 bits. Armazena os 32 bits menos significativos do resultado em R01 e os 32 bits mais significativos em R02. Nessa implementação, é igual ao SMLAL.

R01 = (X * R00)[31:0] + R01

R02 = (X * R00)[63:32] + R02 + CarryFrom((X * R00)[31:0] + R01)

---

56. UMULL

OPCODE: **E082109X** [X = 0 a F]

Multiplicação de 64 bits. Armazena os 32 bits menos significativos do resultado em R01 e os 32 bits mais significativos em R02. Nessa implementação, é igual ao SMULL.

R01 = (X * R00)[31:0]

R02 = (X * R00)[63:32]

9
Apêndice C Formato do Cabeçalho do Datagrama IPv4

A seguir são detalhados todos os campos do Datagrama IPv4, que serão utilizados no programa de teste executado no IXP. A Figura 9-1 traz o formato do cabeçalho do Datagrama IPv4, a fim de entender os campos do pacote.

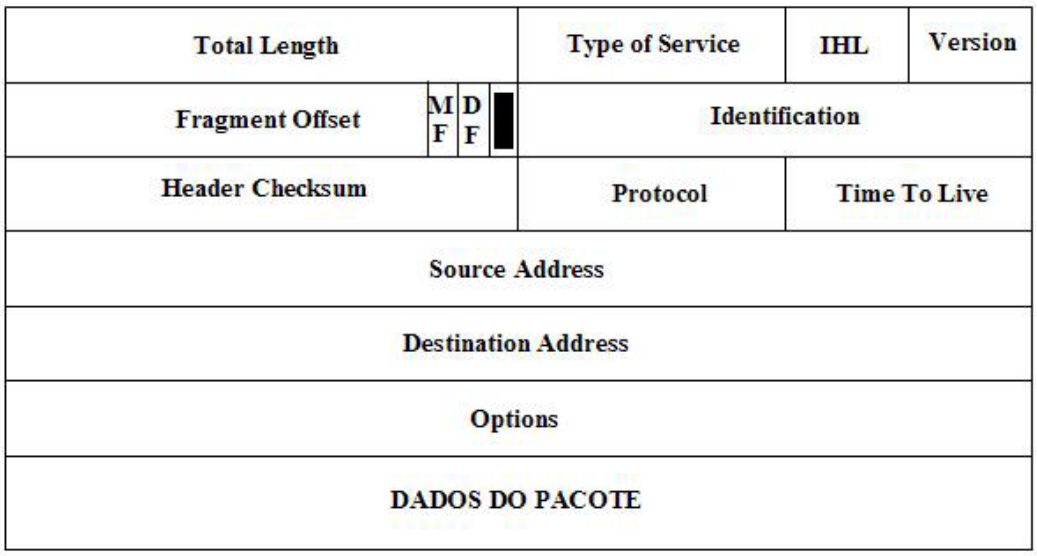


Figura 9-1 – Formato do Cabeçalho do Datagrama IPv4

- *Version* – 04 (Ipv4)
- *IHL (Internet Header Length)* – Comprimento do cabeçalho em palavras de 32 bits. O valor mínimo é 5 (cinco), caso o campo Options seja omitido.
- *Type of Service* – tem seus campos explicados abaixo, conforme Figura 9-2.

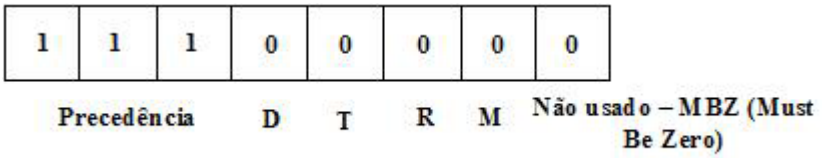


Figura 9-2 – Detalhamento do Campo Type of Service do Datagrama IPv4

O campo ***precedência*** varia de 0 (normal) a 7 (Network Control Packet); os campos ***D*** (Minimize Delay), ***T*** (Maximize Throughput), ***R*** (Maximize Reliability) e ***M*** (Minimize Monetary Cost) indicam respectivamente qual a prioridade da rede, podendo ser o atraso, vazão, confiabilidade ou custo. Na prática, o hardware atual ignora esse bits e se for tudo zero, é considerado ***Serviço Normal***, conforme especificado em (Almquist, P., 1992). A Figura 9-3 detalha os campos mencionados acima.

Precedence. 3 bits.

Value	Description
0	Routine.
1	Priority.
2	Immediate.
3	Flash.
4	Flash override.
5	CRITIC/ECP.
6	Internetwork control.
7	Network control.

D. 1 bit.
Minimize delay.

Value	Description
0	Normal delay.
1	Low delay.

T. 1 bit.
Maximize throughput.

Value	Description
0	Normal throughput.
1	High throughput.

Figura 9-3 – Detalhamento dos Campos Precedência, D, T, R e M

- *Total Length* – Comprimento total do pacote, soma dos tamanhos do cabeçalho e dados.
- *Identification* – Serve para identificar a qual datagrama pertence um fragmento, pois todos os fragmentos de um certo datagrama possuem o mesmo valor neste campo.
- *DF (Don't Fragment)* – Sinaliza ao emissor para não fragmentar o datagrama, pois o receptor pode ser incapaz de remontar o pacote fragmentado.
- *MF (More Fragmets)* – Indica se o datagrama contém fragmentos adicionais. A Figura 9-4 ilustra os detalhes de valores assumidos no campos DF e MF.

DF, Don't fragment. 1 bit.

Value	Description
0	Fragment if necessary.
1	Do not fragment.

MF, More fragments. 1 bit.

Value	Description
0	This is the last fragment.
1	More fragments follow this fragment.

Figura 9-4 – Detalhamento dos Campos DF e MF

- *Fragment Offset* – Posição do fragmento atual em seu datagrama.
- *Time To Live (Tempo de Vida)* – Valor em segundos, campo de 8 bits, de modo que seu valor máximo é 255 segundos.
- *Protocol* – Número do protocolo da camada de transporte, conforme definido em (Reynolds et al., 1994). No caso do TCP, usado no nosso exemplo, o número é 06.
- *Header Checksum* – Contém o checksum do pacote. Os detalhes deste cálculo estão disponíveis em (John, 2002).
- *Source Address* – Define o endereço IPv4 de origem do host.
- *Destination Address* – Define o endereço IPv4 de destino do host.