Step By Step

To

Creating An ASxxxx Assembler

June 2026

I N T R O D U C T I O N

The Step By Step document describes how to create a new ASxxxx assembler following a general outline of the process. The creation of two assemblers will be described.

The first is the OKI OLMS-40 series of 4-bit microprocessors. The ASOLMS40 assembler will support four variations of the MSM5840 microprocessor. The instruction set for the OLMS-40 series is very simple as almost all instructions are a single byte with implicit addressing modes. As a result no addressing mode parsing is required for this assembler.

The second assembler is for the Texas Insturments TMS370 microprocessor. This device has a more complex instruction set with instructions having none to three arguments. Each argument has as many as 12 different addressing modes and thus a substantial effort is required to evalute these modes.

The assemblers will make use of the base definitions file, xxx.h, the program structure file, xxxpst.c, the addressing file, xxadr.c, and the machine processing file, xxxmch.c. Each file contains a basic template of the required definitions and routines required by the new assembler and the ASxxxx core.

Table Of Contents

CHAPTER 1

ASOLMS40 STEP BY STEP

1.1 GETTING STARTED

This step by step process will show how to create a new assembler using the facilities of the ASxxxx assembler core functions. The creation of an assembler for the OKI OLMS-40 4-bit microcontroller family will illustrate the process.

The basic steps are:

1) Know your microprocessor by collecting the necessary information: mnemonics, binary instruction codes, addressing modes, and cycle counts (otional).

2) For simple processors create a binary table of instructions showing opcodes and cycle counts.

3) Create a list of addressing modes. Concentrate on modes that are common to multiple instructions.

1.2 MSM5840 INSTRUCTION SET

Studying the OKI OLMS-40 series processor documentation it is found that the MSM5840 core is the most complex of the series. The MSM5840 has 98 instructions, a 4K byte addressing range and a 128 byte RAM addressing range. The MSM5840 instruction set is shown in pages from the OKI Microcontroller Data Book, Third Edition, March 1988. Following the instruction pages is the binary table created from this information

INSTRUCTION SET

	Mnemonic	Description	Instruction Code 7	6	5	4	3	2	1	0	Byte	Cycle
Load, Store, Read, Clear	CLA	Clear Accumulator	0	0	0	1	0	0	0	0	1	1
	CLL	Clear DP_L	0	0	1	0	0	0	0	0	1	1
	CLH	Clear DP_H	0	1	1	0	0	0	0	0	1	1
	LAI	Load Accumulator with Immediate	0	0	0	1	I_3	I_2	I_1	I_0	1	1
	LLI	Load DP_L with Immediate	0	0	1	0	I_3	I_2	I_1	I_0	1	1
	LHI	Load DP_H with Immediate	0	1	1	0	0	I_2	I_1	I_0	1	1
	L	Load Accumulator with Memory	1	0	0	1	0	1	0	0	1	1
	LM	Load Accumulator with Memory then Modify DP_H	1	0	0	1	0	1	I_1	I_0	1	1
	LAL	Load Accumulator with DP_L	0	1	0	1	0	1	0	1	1	1
	LLA	Load DP_L with Accumulator	0	1	0	1	0	1	0	0	1	1
	LAW	Load Accumulator with W Register	1	0	0	0	0	1	0	0	1	1
	LAX	Load Accumulator with X Register	1	0	0	0	0	1	0	1	1	1
	LAY	Load Accumulator with Y Register	1	0	0	0	0	1	1	0	1	1
	LAZ	Load Accumulator with Z Register	1	0	0	0	0	1	1	1	1	1
	SI	Store Accumulator to Memory then Increment DP_L	1	0	0	1	0	0	0	0	1	1
	SMI	Store Accumulator to Memory then Modify DP_H and Increment DP_L	1	0	0	1	0	0	I_1	I_0	1	1
	LWA	Load W Register with Accumulator	1	0	0	0	0	0	0	0	1	1
	LXA	Load X Register with Accumulator	1	0	0	0	0	0	0	1	1	1
	LYA	Load Y Register with Accumulator	1	0	0	0	0	0	1	0	1	1
	LZA	Load Z Register with Accumulator	1	0	0	0	0	0	1	1	1	1
	LPA	Load Port Pointer with Accumulator	0	1	0	1	1	0	0	0	1	1
	LTI	Load Timer with Immediate	0 I_7	1 I_6	1 I_5	0 I_4	1 I_3	0 I_2	0 I_1	0 I_0	2	2
	RTH	Read Timer H	0	1	1	0	1	0	1	0	1	1
	RTL	Read timer L	0	1	1	0	1	0	1	1	1	1
Exchange	XA	Exchange Accumulator with Save Register A	0	1	0	0	1	0	0	1	1	1
	XL	Exchange DP_L with Save Register L	0	1	0	0	1	0	1	0	1	1
	XCH	Exchange DP_H and Carry with Save Register CH	0	1	0	0	1	0	0	0	1	1
	X	Exchange Accumulator with Memory	1	0	0	1	1	0	0	0	1	1
	XM	Exchange Accumulator with Memory then Modify DP_H	1	0	0	1	1	0	I_1	I_0	1	1
	XAX	Exchange Accumulator with Save Register AX	0	1	0	0	1	0	1	1	1	1
Increment/Decrement	INA	Increment Accumulator (Skip if Zero)	0	0	0	0	0	0	0	1	1	1
	INL	Increment DP_L (Skip if Zero)	0	1	0	1	0	1	1	1	1	1
	INM	Increment Memory (Skip if Zero)	0	1	0	1	1	1	0	1	1	1
	INW	Increment W Register (Skip if Zero)	1	0	0	0	1	0	0	0	1	1
	INX	Increment X Register (Skip if Zero)	1	0	0	0	1	0	0	1	1	1
	INY	Increment Y Register (Skip if Zero)	1	0	0	0	1	0	1	0	1	1
	INZ	Increment Z Register (Skip if Zero)	1	0	0	0	1	0	1	1	1	1

INSTRUCTION SET (CONT.)

	Mnemonic	Description	Instruction Code 7	6	5	4	3	2	1	0	Byte	Cycle
Increment/Decrement	DCA	Decrement Accumulator – Skip if Not All Ones	0	0	0	0	1	1	1	1	1	1
	DCL	Decrement DP_L	0	1	0	1	0	1	1	0	1	1
	DCM	Decrement Memory	0	1	0	1	1	1	0	0	1	1
	DCW	Decrement W Register (Skip if All Ones)	1	0	0	0	1	1	0	0	1	1
	DCX	Decrement X Register	1	0	0	0	1	1	0	1	1	1
	DCY	Decrement Y Register	1	0	0	0	1	1	1	0	1	1
	DCZ	Decrement Z Register	1	0	0	0	1	1	1	1	1	1
	DCH	Decrement DP_H – Skip if All Ones and C = Zero	0	1	0	1	1	1	1	1	1	1
Logical	CAO	Complement Accumulator of One	0	1	0	1	0	0	0	0	1	1
	AND	And Accumulator with Memory	0	1	0	0	0	1	0	0	1	1
	OR	Or Accumulator with Memory	0	1	0	0	0	1	0	1	1	1
	EOR	Exclusive or Accumulator with Memory	0	1	0	0	0	1	1	0	1	1
	RAL	Rotate Accumulator Left through Carry	0	1	0	0	0	1	1	1	1	1
Arithmetic	AC	Add Memory to Accumulator with Carry	0	1	0	0	1	1	0	0	1	1
	ACS	Add Memory to Accumulator with Carry, Skip if Carry	0	1	0	0	1	1	0	1	1	1
	AS	Add Memory to Accumulator, Skip if Carry	0	1	0	0	1	1	1	0	1	1
	AIS	Add Immediate to Accumulator, Skip if Carry	0	0	0	0	I_3	I_2	I_1	I_0	1	1
	DAS	Decimal adjust Accumulator in Subtraction	0	1	0	1	1	0	1	0	1	1
	CM	Compare Accumulator with Memory, Skip if Equal	0	1	0	1	1	1	1	0	1	1
	AWS	Add W Register to Accumulator, Skip if Carry	1	0	0	1	1	1	0	0	1	1
	AXS	Add X Register to Accumulator, Skip if Carry	1	0	0	1	1	1	0	1	1	1
	AYS	And Y Register to Accumulator, Skip if Carry	1	0	0	1	1	1	1	0	1	1
	AZS	Add Z Register to Accumulator, Skip if Carry	1	0	0	1	1	1	1	1	1	1
Bit Set/Reset/Test	SPB	Set Port Bit	1	0	1	1	0	0	I_1	I_0	1	1
	RPB	Reset Port Bit	1	0	1	1	0	1	I_1	I_0	1	1
	SMB	Set Memory Bit	1	0	1	1	1	0	I_1	I_0	1	1
	RMB	Reset Memory Bit	1	0	1	1	1	1	I_1	I_0	1	1
	TAB	Test Accumulator Bit (Skip if One)	1	0	1	0	0	0	I_1	I_0	1	1
	TMB	Test Memory Bit	1	0	1	0	0	1	I_1	I_0	1	1
	TKB	Test K Port Bit	1	0	1	0	1	0	I_1	I_0	1	1
	THB	Test H Port Bit	1	0	1	0	1	1	0	I_0	1	1
	TI	Test Interrupt flag	1	0	1	0	1	1	1	1	1	1
	TTM	Test Time flag	1	0	1	0	1	1	1	0	1	1
	TC	Test Carry flag	0	1	0	0	0	0	1	0	1	1
	SC	Set Carry flag	0	1	0	0	0	0	0	0	1	1
	RC	Reset Carry flag	0	1	0	0	0	0	0	1	1	1

INSTRUCTION SET (CONT.)

	Mnemonic	Description	Instruction Code 7	6	5	4	3	2	1	0	Byte	Cycle
Branch/Subroutine	J	Jump	0	0	1	1	0	I_{10}	I_9	I_8	2	2
			I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0		
	JC	Jump in Current Page	1	1	I_5	I_4	I_3	I_2	I_1	I_0	1	1
	JA	Jump with Accumulator	0	1	0	0	0	0	1	1	1	1
	CAL	Call Subroutine	0	0	1	1	1	I_{10}	I_9	I_8	2	2
			I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0		
	RT	Return from Subroutine	0	1	0	1	1	0	0	1	1	2
Input/Output	OBS	Output Byte String	0	1	1	1	0	0	0	0	1	2~17
	OTD	Output Table Data	0	1	1	1	0	0	0	1	1	2
	OA	Output Accumulator to Port A	0	1	1	1	0	0	1	0	1	1
	OB	Output Accumulator to Port B	0	1	1	1	0	0	1	1	1	1
	OP	Output Accumulator to Port P designated Port Pointer	0	1	1	1	0	1	0	0	1	1
	OAB	Output Memory and Accumulator to Ports A and B	0	1	1	1	0	1	0	1	1	1
	OPM	Output Memory to Port P designated Port Pointer	0	1	1	1	0	1	1	0	1	1
	IA	Input Port A in Accumulator	0	1	1	1	1	0	1	0	1	1
	IB	Input Port B in Accumulator	0	1	1	1	1	0	1	1	1	1
	IK	Input Port K in Accumulator	0	1	1	1	1	1	0	0	1	1
	IAB	Input Ports A and B in Memory and Accumulator	0	1	1	1	1	1	0	1	1	1
Control	EI	Enable Interrupt	0	1	0	1	0	0	1	1	1	1
	DI	Disable Interrupt	0	1	0	1	0	0	1	0	1	1
	ET	Enable Timer	0	1	1	0	1	1	1	1	1	1
	DT	Disable Timer	0	1	1	0	1	1	1	0	1	1
	ECT	Enable Counter	0	1	1	1	1	1	1	1	1	1
	DCT	Disable Counter	0	1	1	1	1	1	1	0	1	1
	HLT	Halt	0	1	1	0	1	1	0	1	1	1
	EXP	Exchange Program	0	1	1	0	1	0	0	1	1	1
	NOP	No Operation	0	0	0	0	0	0	0	0	1	1

MSM 5840
L1095

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP AIS	INA														DCA
1	CLA LAI	LAI	LAI	LAI	LAI	LAI										LAI
2	CLL LLI															LLI
3	J							J	CAL							CAL
4	SC	RC	TC	JA	AND	OR	XOR	RAL	XCH	XA	XL	XAX	AC	ACS	AS	X
5	CAD	X	DI	EI	LLA	LAL	DCL	INL	LPA	RT	DAS	X	DCM	INM	CM	X
6	CLH LHI							LHI	LTI	EXP	RTH	RTL	X	HLT	DT	ET
7	OBS	OTD	OA	OB	OP	OAB	OPM	X	X	X	IA	IB	IK	IAB	DCT	ECT
8	LWA	LXA	LYA	LZA	LAW	LAX	LAY	LAZ	INW	INX	INY	INZ	DCW	DCX	DCY	DCZ
9	SI SMI	SMI	SMI	SMI	LM	LM	LM	LM	XM	XM	XM	XM	AWS	AXS	AYS	AZS
A	TAB			TAB	TMB			TMB	TKB			THB	THB	THB	TTM	TI
B	SPB			SPB	RPB			RPB	SMB			SMB	RMB			RMB
C	JC															JC
D	JC															JC
E	JC															JC
F	JC															JC

Analyzing this table notice that various instructions can be grouped together with similar patterns:

1) AIS (0x00-0x1F), LAI (0x10-0x1F), LLI (0x20-0x2F

2) J (0x30-0x37), CAL (0x38-0x3F)

3) SMI (0x90-0x93), LM (0x94-0x97), XM (0x98-0x9B),
TAB (0xA0-0xA3), TMB (0xA4-0xA7), TKB (0xA8-0xAB),
SPB (0xB0-0xB3), RPB (0xB4-0xB7),
SMB (0xB8-0xBB), RMB (0xBC-0xBF)

and others with unique addressing:

1) LHI (0x60-0x67)

2) THB (0xA6-0xA7)

3) JC (0xC0-0xFF)

all other instructions are single byte with unique opcodes.

There are just 8 unused/illegal opcodes for the MSM5840 microprocessor. The other members of the OLMS-40 series have many more unused/illegal opcodes as will be seen later.

At this point it is useful to discuss how this information can be incorporated into appropriate source files of a new assembler. From the directory asxvp6xx\asxmisc copy the files xxx.h, xxxpst.c, xxxmch.c, and xxxadr.c to a directory created for the OKI OLMS-40 series assembler. Rename the files to indicate the processor type - ms40.h, ms40pst.c, ms40mch.c, and ms40adr.c. The file name suffix ms40 is arbitrary but was selected from the first processor type MS[M58]40 -> ms40.

The ms40.h file will contain some boiler plate text, definitions for instruction types, addressing and argument types, and explicit function definitions required by every ASxxxx assembler. The ms40pst.c file contains the core ASxxxx directives, processor specific mnemonic definitions, and the default merge mode addressing recipe. The ms40mch.c file will contain the basic starting point for the machine instruction processing and cycle counting. And ms40adr.c will contain the addressing mode processing.

1.3 INSTRUCTION TYPES

Edit the newly named ms40.h file to reflect the assembler being created. From the analysis of the instruction set and the binary table there are eight distinct instruction groups, address types, or argument types. Under instruction types define the following types -

```
/*
 * Instruction Types
 */
#define I_ADR11 50      /* J, CAL */
#define I_ADR6  51      /* JC */
#define I_IMM4  52      /* AIS, LAI, LLI */
#define I_IMM3  53      /* LHI */
                        /* SMI,  LM,  XM, TAB, TMB,*/
#define I_IMM2  54      /* TKB, SPB, RPB, SMB, RMB */
#define I_IMM1  55      /* THB */
#define I_LTI   56      /* LTI */

#define I_INH   57      /* All Remaining Are Inherent */
```

The type values MUST be 30 or greater so as not to conflict with the values assigned to the ASxxxx directives. Failure to adhere to this restriction will result in program execution errors.

Remove the 'Other' and 'Registers' as they will not be used.

1.4 EXTENDED ADDRESSING: MERGE MODES

The extended addressing modes need some discussion. The MSM5840 core has addressing and argument lengths which vary from 1 to 11 bits. For a purely absolute assembler this is not a problem as all addresses and arguments are known and simple masking can select the correct bits to combine with the microprocessor's opcodes. However, with a relocating assembler the arguments are not always known (labels and values may be externally defined in separately compiled modules). Thus a means of combining an external argument with an opcode is required. The ASxxxx assemblers use a method called 'merge mode'. This method passes the value or a reference to a value to the linker along with a recipe to combine this value with another absolute value, typically an opcode. Each ASxxxx assembler can define upto 16 unique 'merge mode' recipes for combining two values.

The recipe index is specified by the third 4-bit nibble of the relocation parameter word value (0x0000, 0x0100, ..., 0x0E00, 0x0F00). The MSM5840 requires 6 unique modes to handle the combining of addresses and arguments with the instruction opcodes. Create these entries under the Extended Addressing Modes:

```
/*
 * Extended Addressing Modes (MSM5840)
 */
#define M_ADR11 0x0100  /* 11-Bit Addressing Mode */
#define M_ADR6  0x0200  /* 6-Bit Addressing Mode */
#define M_IMM4  0x0300  /* 4-Bit Argument Mode */
#define M_IMM3  0x0400  /* 3-Bit Argument Mode */
#define M_IMM2  0x0500  /* 2-Bit Argument Mode */
#define M_IMM1  0x0600  /* 1-Bit Argument Mode */
```

The 'merge mode' recipes will be defined in the ms40pst.c file. The default merge mode, mode0[32], specifies a one-to-one bit replacement for 32 bits. The active bit entries are specified by setting bit 7 of the char to 1. Add the following six additional merge modes for the MSM5840 processor:

```
/*
 * Additional Relocation Mode Definitions
 *
 * Specification for the 11-bit addressing mode:
 */
char    mode1[32] = {   /* M_ADR11 */
        '\200', '\201', '\202', '\203', '\204', '\205', '\206', '\207',
        '\210', '\211', '\212', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};

/*
 * Specification for the 6-bit addressing mode:
 */
char    mode2[32] = {   /*M_ADR6 */
        '\200', '\201', '\202', '\203', '\204', '\205', '\006', '\007',
        '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};
```

```
/*
 * Specification for the 4-bit argument mode:
 */
char    mode3[32] = {   /* M_IMM4 */
        '\200', '\201', '\202', '\203', '\004', '\005', '\006', '\007',
        '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};

/*
 * Specification for the 3-bit argument mode:
 */
char    mode4[32] = {   /* M_IMM3 */
        '\200', '\201', '\202', '\003', '\004', '\005', '\006', '\007',
        '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};

/*
 * Specification for the 2-bit argument mode:
 */
char    mode5[32] = {   /* M_IMM2 */
        '\200', '\201', '\002', '\003', '\004', '\005', '\006', '\007',
        '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};

/*
 * Specification for the 1-bit argument mode:
 */
char    mode6[32] = {   /* M_IMM1 */
        '\200', '\001', '\002', '\003', '\004', '\005', '\006', '\007',
        '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};
```

An addtional structure is used by the assembler to assist in processing the merge mode:

```
/*
 *      *m_def is a pointer to the bit relocation definition.
 *      m_flag indicates that bit position swapping is required.
 *      m_dbits contains the active bit positions for the output.
 *      m_sbits contains the active bit positions for the input.
 *
 *      struct  mode
 *      {
 *              char *  m_def;          Bit Relocation Definition
 *              a_uint  m_flag;         Bit Swapping Flag
 *              a_uint  m_dbits;        Destination Bit Mask
 *              a_uint  m_sbits;        Source Bit Mask
 *      };
 */
```

Add the six new modes to the merge mode array:

```
struct  mode    mode[7] = {
    {   &mode0[0],      0,      0x0000FFFF,     0x0000FFFF      },
    {   &mode1[0],      0,      0x000007FF,     0x000007FF      },
    {   &mode2[0],      0,      0x0000003F,     0x0000003F      },
    {   &mode3[0],      0,      0x0000000F,     0x0000000F      },
    {   &mode4[0],      0,      0x00000007,     0x00000007      },
    {   &mode5[0],      0,      0x00000003,     0x00000003      },
    {   &mode6[0],      0,      0x00000001,     0x00000001      }
};
```

Note that in all the merge mode recipes the source bit is placed into the same bit position in the destination. This is indicated in the structure by setting the bit swapping flag, mflag, equal to 0. An example of a complex merge mode can be found in the AVR assembler where a merge mode recipe and the corresponding structure entry are shown here.

```
char mode7[32] = {      /* S_ILDST instructions */
        /* |--K-|KK--|----|-KKK| */
        '\200', '\201', '\202', '\212', '\213', '\215', '\006', '\007',
        '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};

    {   &mode7[0],      1,      0x00002C07,     0x0000003F      },
```

In this recipe bits 0, 1, and 2 are moved into bits 0, 1, and 2 of the destination while bits 3 and 4 are moved to bit positions 10 and 11 and bit 5 is moved to bit position 13 of the destination. (Note: the recipe numbers are in octal for historical reasons.) In this case the bit swapping flag is set to 1.

And finally, the array of pointers to the mode structure elements must be updated to include the six additional merge modes:

```
/*
 * Array of Pointers to mode Structures
 */
struct  mode    *modep[16] = {
        &mode[0],       &mode[1],       &mode[2],       &mode[3],
        &mode[4],       &mode[5],       &mode[6],       NULL,
        NULL,           NULL,           NULL,           NULL,
        NULL,           NULL,           NULL,           NULL
};
```

1.5 MNEMONIC STRUCTURE

Having completed the initial process of defining addressing and argument modes it is time to update the ASxxxx directives and add the MSM5840 instruction opcodes to the mnemonic structure.

```
/*
 *      The mne structure is a linked list of the assembler
 *      mnemonics and directives.  The list of mnemonics and
 *      directives contained in the device dependent file
 *      xxxpst.c are hashed and linked into NHASH lists in
 *      module assym.c by syminit().  The structure contains
 *      the mnemonic/directive name, a subtype which directs
 *      the evaluation of this mnemonic/directive, a flag which
 *      is used to detect the end of the mnemonic/directive
 *      list in xxxpst.c, and a value which is normally
 *      associated with the assembler mnemonic base instruction
 *      value.
 */
struct  mne
{
        struct  mne *m_mp;      /* Hash link */
        char    *m_id;          /* Mnemonic (JLH) */
        char    m_type;         /* Mnemonic subtype */
        char    m_flag;         /* Mnemonic flags */
        a_uint  m_valu;         /* Value */
};
```

The MSM5840 processor uses a byte addressable program memory and addresses using upto 2 bytes (1 word). Thus any ASxxxx directive that uses more than 2 bytes can be removed by commenting out those directives:

```
/*
    {   NULL,   ".3byte",       S_DATA,         0,      O_3BYTE },
    {   NULL,   ".triple",      S_DATA,         0,      O_3BYTE },
    {   NULL,   ".dl",          S_DATA,         0,      O_4BYTE },
    {   NULL,   ".4byte",       S_DATA,         0,      O_4BYTE },
    {   NULL,   ".quad",        S_DATA,         0,      O_4BYTE },
    {   NULL,   ".long",        S_DATA,         0,      O_4BYTE },

    {   NULL,   ".blk3",        S_BLK,          0,      O_3BYTE },
    {   NULL,   ".blk4",        S_BLK,          0,      O_4BYTE },
    {   NULL,   ".blkl",        S_BLK,          0,      O_4BYTE },
*/
```

Directives only used under special conditions should also be commented out:

```
/*
    {   NULL,   ".msb",         S_MSB,          0,      0       },
    {   NULL,   ".lohi",        S_MSB,          0,      O_LOHI  },
    {   NULL,   ".hilo",        S_MSB,          0,      O_HILO  },
    {   NULL,   ".8bit",        S_BITS,         0,      O_1BYTE },
    {   NULL,   ".16bit",       S_BITS,         0,      O_2BYTE },
    {   NULL,   ".24bit",       S_BITS,         0,      O_3BYTE },
    {   NULL,   ".32bit",       S_BITS,         0,      O_4BYTE },
*/
```

Note the last entry in the mne structure has the symbol S_EOL in the fourth column. This is REQUIRED in the last entry, it signifies the end of the mnemonic structure during initialization.

Remove the entries from /* Special */ to the end of the structure. We will add entries for each of the instruction types defined in ms40.h. Each entry will have the instruction type, instruction mnemonic, and the opcode value. Those instructions having extended addressing and argument types will have instruction base opcode values in the structure. The last entry will have the symbol S_EOL in the fourth element.

1.6 INSTRUCTION MNEMONICS

```
/* MSM5840 */

    {   NULL,   "j",            I_ADR11,        0,      0x30    },
    {   NULL,   "cal",          I_ADR11,        0,      0x38    },

    {   NULL,   "jc",           I_ADR6,         0,      0xC0    },

    {   NULL,   "ais",          I_IMM4  ,       0,      0x00    },
    {   NULL,   "lai",          I_IMM4  ,       0,      0x10    },
    {   NULL,   "lli",          I_IMM4  ,       0,      0x20    },
```

```
{   NULL,   "lhi",          I_IMM3,         0,      0x60    },

{   NULL,   "smi",          I_IMM2,         0,      0x90    },
{   NULL,   "lm",           I_IMM2,         0,      0x94    },
{   NULL,   "xm",           I_IMM2,         0,      0x98    },
{   NULL,   "tab",          I_IMM2,         0,      0xA0    },
{   NULL,   "tmb",          I_IMM2,         0,      0xA4    },
{   NULL,   "tkb",          I_IMM2,         0,      0xA8    },
{   NULL,   "spb",          I_IMM2,         0,      0xB0    },
{   NULL,   "rpb",          I_IMM2,         0,      0xB4    },
{   NULL,   "smb",          I_IMM2,         0,      0xB8    },
{   NULL,   "rmb",          I_IMM2,         0,      0xBC    },


{   NULL,   "thb",          I_IMM1,         0,      0xAC    },

{   NULL,   "lti",          I_LTI,          0,      0x68    },

    /* alias for ais 0 */
{   NULL,   "nop",          I_INH,          0,      0x00    },
    /* alias for ais 1 */
{   NULL,   "ina",          I_INH,          0,      0x01    },

    /* ais 0x00 - 0x0F */
    /* alias for ais F */
{   NULL,   "dca",          I_INH,          0,      0x0F    },
    /* alias for lia 0 */
{   NULL,   "cla",          I_INH,          0,      0x10    },
    /* lai 0x10 - 0x1F */
    /* alias for lli 0 */
{   NULL,   "cll",          I_INH,          0,      0x20    },
    /* lli 0x20 - 0x2F */
    /* j   0x30 - 0x37 */
    /* cal 0x38 - 0x3F */
```

```
{   NULL,   "sc",           I_INH,          0,      0x40    },
{   NULL,   "rc",           I_INH,          0,      0x41    },
{   NULL,   "tc",           I_INH,          0,      0x42    },
{   NULL,   "ja",           I_INH,          0,      0x43    },
{   NULL,   "and",          I_INH,          0,      0x44    },
{   NULL,   "or",           I_INH,          0,      0x45    },
{   NULL,   "xor",          I_INH,          0,      0x46    },
{   NULL,   "ral",          I_INH,          0,      0x47    },
{   NULL,   "xch",          I_INH,          0,      0x48    },
{   NULL,   "xa",           I_INH,          0,      0x49    },
{   NULL,   "xl",           I_INH,          0,      0x4A    },
{   NULL,   "xax",          I_INH,          0,      0x4B    },
{   NULL,   "ac",           I_INH,          0,      0x4C    },
{   NULL,   "acs",          I_INH,          0,      0x4D    },
{   NULL,   "as",           I_INH,          0,      0x4E    },
    /* 0x4F - unused */

{   NULL,   "cao",          I_INH,          0,      0x50    },
    /* 0x51 - unused */
{   NULL,   "di",           I_INH,          0,      0x52    },
{   NULL,   "ei",           I_INH,          0,      0x53    },
{   NULL,   "lla",          I_INH,          0,      0x54    },
{   NULL,   "lal",          I_INH,          0,      0x55    },
{   NULL,   "dcl",          I_INH,          0,      0x56    },
{   NULL,   "inl",          I_INH,          0,      0x57    },
{   NULL,   "lpa",          I_INH,          0,      0x58    },
{   NULL,   "rt",           I_INH,          0,      0x59    },
{   NULL,   "das",          I_INH,          0,      0x5A    },
    /* 0x5B - unused */
{   NULL,   "dcm",          I_INH,          0,      0x5C    },
{   NULL,   "inm",          I_INH,          0,      0x5D    },
{   NULL,   "cm",           I_INH,          0,      0x5E    },
    /* 0x5F - unused */

    /* alias for lhi 0 */
{   NULL,   "clh",          I_INH,          0,      0x60    },
    /* lhi 0x60 - 0x67 */
    /* lti 0x68 */
{   NULL,   "exp",          I_INH,          0,      0x69    },
{   NULL,   "rth",          I_INH,          0,      0x6A    },
{   NULL,   "rtl",          I_INH,          0,      0x6B    },
{   NULL,   "hlt",          I_INH,          0,      0x6D    },
    /* 0x6C - unused */
{   NULL,   "dt",           I_INH,          0,      0x6E    },
{   NULL,   "et",           I_INH,          0,      0x6F    },
```

```
{   NULL,   "obs",          I_INH,          0,      0x70    },
{   NULL,   "otd",          I_INH,          0,      0x71    },
{   NULL,   "oa",           I_INH,          0,      0x72    },
{   NULL,   "ob",           I_INH,          0,      0x73    },
{   NULL,   "op",           I_INH,          0,      0x74    },
{   NULL,   "oab",          I_INH,          0,      0x75    },
{   NULL,   "opm",          I_INH,          0,      0x76    },
    /* 0x77 - 0x79 - unused */
{   NULL,   "ia",           I_INH,          0,      0x7A    },
{   NULL,   "ib",           I_INH,          0,      0x7B    },
{   NULL,   "ik",           I_INH,          0,      0x7C    },
{   NULL,   "iab",          I_INH,          0,      0x7D    },
{   NULL,   "dct",          I_INH,          0,      0x7E    },
{   NULL,   "ect",          I_INH,          0,      0x7F    },

{   NULL,   "lwa",          I_INH,          0,      0x80    },
{   NULL,   "lxa",          I_INH,          0,      0x81    },
{   NULL,   "lya",          I_INH,          0,      0x82    },
{   NULL,   "lza",          I_INH,          0,      0x83    },
{   NULL,   "law",          I_INH,          0,      0x84    },
{   NULL,   "lax",          I_INH,          0,      0x85    },
{   NULL,   "lay",          I_INH,          0,      0x86    },
{   NULL,   "laz",          I_INH,          0,      0x87    },
{   NULL,   "inw",          I_INH,          0,      0x88    },
{   NULL,   "inx",          I_INH,          0,      0x89    },
{   NULL,   "iny",          I_INH,          0,      0x8A    },
{   NULL,   "inz",          I_INH,          0,      0x8B    },
{   NULL,   "dcw",          I_INH,          0,      0x8C    },
{   NULL,   "dcx",          I_INH,          0,      0x8D    },
{   NULL,   "dcy",          I_INH,          0,      0x8E    },
{   NULL,   "dcz",          I_INH,          0,      0x8F    },

    /* alias for smi 0 */
{   NULL,   "si",           I_INH,          0,      0x90    },
    /* smi 0x90 - 0x93 */
    /* alias for lm 0 */
{   NULL,   "l",            I_INH,          0,      0x94    },
    /* lm  0x94 - 0x97 */
    /* alias for xm 0 */
{   NULL,   "x",            I_INH,          0,      0x98    },
    /* xm  0x98 - 0x9B */
{   NULL,   "aws",          I_INH,          0,      0x9C    },
{   NULL,   "axs",          I_INH,          0,      0x9D    },
{   NULL,   "ays",          I_INH,          0,      0x9E    },
{   NULL,   "azs",          I_INH,          0,      0x9F    },
```

```
    /* tab 0xA0 - 0xA3 */
    /* tmb 0xA4 - 0xA7 */
    /* tkb 0xA8 - 0xAB */
    /* thb 0xAC - 0xAD */
{   NULL,   "ttm",          I_INH,          0,      0xAE    },
{   NULL,   "ti",           I_INH,          S_EOL,  0xAF    }
    /* spb 0xB0 - 0xB3 */
    /* rpb 0xB4 - 0xB7 */
    /* smb 0xB8 - 0xBB */
    /* rmb 0xBC - 0xBF */
    /* jc  0xC0 - 0xFF */
```

The next step is the implementation of the instruction mnemonic to the binary code of the microprocessor. This process is usually implemented in two modules - ms40mch.c and ms40adr.c.

1.7 INSTRUCTION ADDRESSING MODES

The OKI OLMS-40 series has only two specific addressing modes - immediate (a number) and extended (an address). File ms40adr.c contains the starting point for the address/argument processing in an ASxxxx assembler. The OLMS-40 series does not have arguments that represent internal registers, thus the admode(), srch(), adsym(), and aindx elements may be deleted from ms40adr.c. What remains is the processing for an immediate argument, S_IMMED, and an address expression, S_EXT. S_IMMED and S_EXT are defined in ms40.h. Update the references to xxxaddr.c and xxx.h to ms40adr.c and ms40.h respectively.

1.8 MACHINE CODE GENERATION

The ms40mch.c file contains the basic skeleton to process machine mnemonics, initialize the assembler, and provide instruction cycle counts. The ASxxxx core always calls function minit() at the beginning of each assembly pass to (re)initialize the parameters for this assembler. The function machine() is called by the ASxxxx core to evaluate any machine mnemonic that was placed in the ms40pst.c mne structure. The evaluation of any and all arguments is determined by the processing in this function. The function mchpcr() is used by many assemblers to help in assembling program counter relative addressing. The MSM5840 series does not use pc relative addressing and thus mchprc() can be removed from ms40mch.c.

Edit the file ms40mch.c to replace any xxxmch.c and xxx.h references with ms40mch.c and ms40.h resectively. Also change the cpu string to "OKI OLMS-40 SERIES". The function machine() typically processes the machine mnemonics by using a switch() with case statements for each of the defined instruction types. Remove:

```
case I_XXX:
    break;
```

and add these case statements:

```
/* J, CAL */
case I_ADR11:
    break;

/* JC */
case I_ADR6:
    break;

/* AIS, LAI, LLI */
case I_IMM4:
    break;

/* LHI */
case I_IMM3:
    break;

/* SMI, LM, XM, TAB, TMB, TKB, SPB, RPB, SMB, RMB */
case I_IMM2:
    break;

/* THB */
case I_IMM1:
    break;

/* LTI */
case I_LTI:
    break;

case I_INH:
    break;
```

In the following discussion function addr() is defined in ms40adr.c. The other functions are part of the ASxxxx core and will be described in a later section.

Case I_ADR11 is the 11-bit addressing mode where the output code occupies two bytes. The output code is generated by merging the 11-Bit address with the opcode shifted to the Most Significant Byte. The MSM5840 has a maximum ROM address of 0x0FFF (0-4095). The output function, outrwm(), arguments are the address, the merge mode ored with control bits that inform the assembler or linker to verify the range as specified by the merge mode, and the shifted opcode value. Update the case statement to include a simple error message and the word merge mode code for this instruction type.

```
/* J, CAL */
case I_ADR11:
    if (addr(&e) != S_EXT) {
        xerr('w', "Argument should be an address - not #N");
    }
    outrwm(&e, M_ADR11 | R_MBRO, op << 8);
    break;
```

Case I_ADD6 for instruction JC is a special case. The jump address must be in the same page as the JC instruction itself. The page size is 64 bytes (ie 6-bits). The page base address is the address of the instruction anded with ~0x3F (. & ~0x3F). The ASxxxx assemble uses an addressing method called paging to define the current page and inform the linker about the page. The function outdp() performs this function. On most early microprocessors paging usually referred to a fixed address range of 0x00-0xFF or the upper 256 bytes of the addressing range. However the OLMS-40 series has a floating 64-byte page determined by the address of the JC instruction itself. This requires that the assembler notify the linker each time a JC instruction is assembled of the new page boundary. The 64-bit page length also requires that the assembler be initialized for a 64-byte page by adding the following code to the minit() function in ms40mch.c.

```
/*
 * Set Page Mask To 64 Bytes
 */
p_mask = 0x3F;
```

Update the Case I_ADR6 code with the following

```
/* JC */
case I_ADR6:
    /* Save Argument Pointer */
    jp = ip;
    /* Output The Current Page On The Fly */
    ip = ". & ~0x3F";
    expr(&e);
    outdp(dot.s_area, &e, 0);
    /* Restore The Argument Pointer */
    ip = jp;
    /* Process Argument */
    clrexpr(&e);
    if (addr(&e) != S_EXT) {
        xerr('w', "Argument should be an address - not #N");
    }
    outrbm(&e, M_ADR6 | R_PAGN, op);
    break;
```

and add a definition for the character pointer jp at the beginning of the function machine().

```
char *jp;
```

The character pointer, ip, is the pointer to the current position in the source code text line. This pointer, ip, is used by all lexical and expression processing by the ASxxxx core. To determine the current page the expression ". & ~0x3F" must be evaluated and exported to the linker. The process is to save the current position in jp, point ip at the string, evaluate the expression, output the page description, and then restore the text pointer. In preparation for the instruction argument processing the expression structure must be cleared (initialized).

Cases I_IMM4, I_IMM3, I_IMM2, and I_IMM1 each require immediate values prefixed with the character #. Each of these cases has a common form:

```
case I_XXXX:
    if (addr(&e) != S_IMMED) {
        xerr('w', "Argument should be #N - not an address");
    }
    outrbm(&e, M_XXXX | R_MBRO, op);
    break;
```

Replace this case with the following code:

```
/* AIS, LAI, LLI */
case I_IMM4:
    if (addr(&e) != S_IMMED) {
        xerr('w', "Argument should be #N - not an address");
    }
    outrbm(&e, M_IMM4 | R_MBRO, op);
    break;

/* LHI */
case I_IMM3:
    if (addr(&e) != S_IMMED) {
        xerr('w', "Argument should be #N - not an address");
    }
    outrbm(&e, M_IMM3 | R_MBRO, op);
    break;

/* SMI, LM, XM, TAB, TMB, TKB, SPB, RPB, SMB, RMB */
case I_IMM2:
    if (addr(&e) != S_IMMED) {
        xerr('w', "Argument should be #N - not an address");
    }
    outrbm(&e, M_IMM2 | R_MBRO, op);
    break;

/* THB */
case I_IMM1:
    if (addr(&e) != S_IMMED) {
        xerr('w', "Argument should be #N - not an address");
    }
    outrbm(&e, M_IMM1 | R_MBRO,op);
    break;
```

The LTI instruction has a byte opcode followed by a byte value. Update the Case I_LTI code with the following:

```
/* LTI */
case I_LTI:
    if (addr(&e) != S_IMMED) {
        xerr('w', "Argument should be #N - not an address");
    }
    outab(op);
    outrb(&e, R_OVRF);
    break;
```

Case I_INH instructions do not have arguments. Update Case I_INH to output just the single byte opcode.

```
case I_INH:
    outab(op);
    break;
```

We have reached the point where the code can be compiled with your favorite compiler. Create a project named asolms40 which must include all the following files:

ms40.h
ms40mch.c, ms40adr.c, ms40pst.c

asxxxx.h
asdata.c, asdbg.c, asexpr.c, aslex.c, aslist.c
asmain.c, asmcro.c, asout.c, assubr.c, assym.c

After fixing any syntax errors this results in a working ASxxxx assembler: asolms40

1.9 INSTRUCTION CYCLE COUNT

This assembler has NO CYCLE COUNT code and does not show cycle counts. The next step is to add the cycle count option to the assembler. The current implementation of asolms40 could implement cycle counts simply by adding "opcycles = 1;" to all instruction types except cases I_ADR11, I_LTI, and in I_INH when the opcode has a value of 0x59 (RT) or 0x71 (OTB) and the opcycles = 2 or for opcode 0x70 when opcycles = 17 (the maximum opcycles for this instruction). However an alternate method is to use a lookup table of 256 elements, the number of states in a byte sized opcode. The table is filled with the maximum cycle count for each opcode value. Unused opcode values will have a value of zero and the undefined bit set. ASxxxx assemblers normally limit cycle counts to 99 which is easily contained in a byte sized element. Using the information contained in the Binary Table described earlier a cycle count array was created:

```
/*
 * MSM5840 Cycle Count
 *
 *      opcycles = pg5840[op]
 */
static char pg5840[256] = {
/*--*--* 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F */
/*--*--* -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  - */
/*00*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*10*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*20*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*30*/   2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
/*40*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,UN,
/*50*/   1,UN, 1, 1, 1, 1, 1, 1, 1, 2, 1,UN, 1, 1, 1,UN,
/*60*/   1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,UN, 1, 1, 1,
/*70*/  17, 2, 1, 1, 1, 1, 1,UN,UN,UN, 1, 1, 1, 1, 1, 1,
/*80*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*90*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*A0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*B0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*C0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*D0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*E0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*F0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
};
```

This array replaces the cycle count array at the beginning of ms40mch.c.

During the instruction evaluation the output values are stored in a temporary array - cb[]. In all cases the first element of the cb[] array is the opcode from the instruction.

Replacing

```
if (opcycles == OPCY_NONE) {
        opcycles = pgxxx[cb[0] & 0xFF];
}
```

with

```
if (opcycles == OPCY_NONE) {
        opcycles = pg5840[cb[0] & 0xFF];
}
```

after the switch() statement and before the 'Translate To External Format' will add the cycle count option to the assembler.

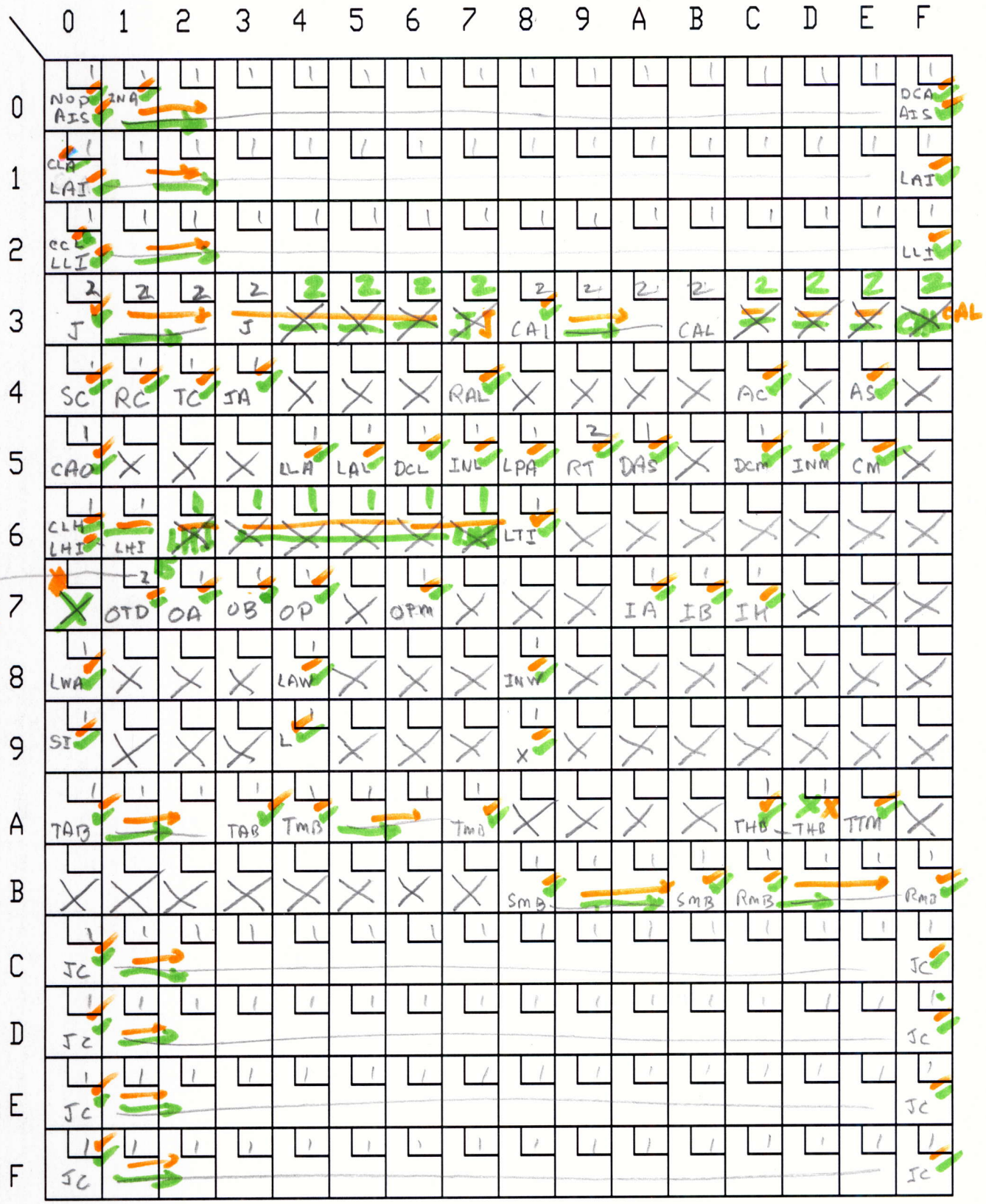
The OKI OLMS-40 series has three additional members that can be added to this assembler: MSM5842, MSM58421, and MSM58422. The MSM5842 and the pair, MSM58421 and MSM58422, each supports a subset of the MSM5840 instruction set. A second binary table was created to analyze the differences between the processor cores, it is shown here:

msm 5842

768

1 1536

2 1536



2

2+15

1024

512

1536

Comparing this table with that of the MSM5840 shows a large number of unimplemented instructions. From this binary table two cycle count tables are created - one for the MSM5842 and a second for the MSM58421 and MSM58422. The MSM58421 and MSM58422 variants have the same core with different peripherals.

```
/*
 * MSM5842 Cycle Count
 *
 *      opcycles = pg5842[op]
 */
static char pg5842[256] = {
/*--*--* 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F */
/*--*--* -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  - */
/*00*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*10*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*20*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*30*/   2, 2, 2, 2,UN,UN,UN,UN, 2, 2, 2, 2,UN,UN,UN,UN,
/*40*/   1, 1, 1, 1,UN,UN,UN, 1,UN,UN,UN,UN, 1,UN, 1,UN,
/*50*/   1,UN,UN,UN, 1, 1, 1, 1, 1, 2, 1,UN, 1, 1, 1,UN,
/*60*/   1, 1,UN,UN,UN,UN,UN,UN, 1,UN,UN,UN,UN,UN,UN,UN,
/*70*/  UN, 2, 1, 1, 1,UN, 1,UN,UN,UN, 1, 1, 1,UN,UN,UN,
/*80*/   1,UN,UN,UN, 1,UN,UN,UN, 1,UN,UN,UN,UN,UN,UN,UN,
/*90*/   1,UN,UN,UN, 1,UN,UN,UN, 1,UN,UN,UN,UN,UN,UN,UN,
/*A0*/   1, 1, 1, 1, 1, 1, 1, 1,UN,UN,UN,UN, 1,UN, 1,UN,
/*B0*/  UN,UN,UN,UN,UN,UN,UN,UN, 1, 1, 1, 1, 1, 1, 1, 1,
/*C0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*D0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*E0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*F0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
};
```

```
/*
 * MSM58421 And MSM58422 Cycle Count
 *
 *      opcycles = pg5842x[op]
 */
static char pg5842x[256] = {
/*--*--* 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F */
/*--*--* -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  - */
/*00*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*10*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*20*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*30*/   2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
/*40*/   1, 1, 1, 1,UN,UN,UN, 1,UN,UN,UN,UN, 1,UN, 1,UN,
/*50*/   1,UN,UN,UN, 1, 1, 1, 1, 1, 2, 1,UN, 1, 1, 1,UN,
/*60*/   1, 1, 1, 1, 1, 1, 1, 1, 1,UN,UN,UN,UN,UN,UN,UN,
/*70*/  UN, 2, 1, 1, 1,UN, 1,UN,UN,UN, 1, 1, 1,UN,UN,UN,
/*80*/   1,UN,UN,UN, 1,UN,UN,UN, 1,UN,UN,UN,UN,UN,UN,UN,
/*90*/   1,UN,UN,UN, 1,UN,UN,UN, 1,UN,UN,UN,UN,UN,UN,UN,
/*A0*/   1, 1, 1, 1, 1, 1, 1, 1,UN,UN,UN,UN, 1,UN, 1,UN,
/*B0*/  UN,UN,UN,UN,UN,UN,UN,UN, 1, 1, 1, 1, 1, 1, 1, 1,
/*C0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*D0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*E0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/*F0*/   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
};
```

Add these tables to ms40mch.c after the existing cycle count table.

1.10 PROCESSOR SELECTION

To select a specific processor add the following to ms40.h:

```
/*
 * Processor Selection
 */
#define S_CPU   30

#define MSM5840         0
#define MSM5842         1
#define MSM58421        2
#define MSM58422        3
```

and these entries to the mnemonic structure in ms40pst.c just before the /* MSM5480 */ comment:

```
/* CPU Select */

{   NULL,   ".msm5840",     S_CPU,          0,      MSM5840 },
{   NULL,   ".msm5842",     S_CPU,          0,      MSM5842 },
{   NULL,   ".msm58421",    S_CPU,          0,      MSM58421},
{   NULL,   ".msm58422",    S_CPU,          0,      MSM58422},
```

The specific core is selected by including one of these directives in the assembly source file. Add the following as the first case statement in ms40mch.c.

```
case S_CPU:
    opcycles = OPCY_CPU;
    cputyp = op;
    sym[2].s_addr = op;
    lmode = SLIST;
    break;
```

and add this variable definition before the function machine() in ms40mch.c.

```
int cputyp;
```

The value is also saved in sym[2].s_addr which is the symbol .__.CPU. shown in the symbol table after assembly. lmode is the list mode variable specifying source list only. This symbol and others will be detailed in a later section.

Initialize the cputyp during each pass of the assembler by adding the following to the function minit() in ms40mch.c.

```
/*
 * Select MSM5840 As Default CPU
 */
cputyp = MSM5840;
sym[2].s_addr = MSM5840;
```

What remains is to select the proper cycle count table and report unimplemented instructions for each of the cores.

Create a four element array with each element pointing to the proper cycle count table. The cputyp will be the index, 0-3. Indexes 2 and 3 will point to the same table, pg5842x.

```
static char *cpupag[4] = {
    pg5840, pg5842, pg5842x, pg5842x
};
```

Place this just after the three cycle count tables in ms40mch.c.

To select the proper cycle count table replace

```
if (opcycles == OPCY_NONE) {
        opcycles = pg5840[cb[0] & 0xFF];
}
```

with

```
if (opcycles == OPCY_NONE) {
        opcycles = cpupag[cputyp][cb[0] & 0xFF];
}
```

1.11 UNSUPPORTED INSTRUCTIONS

The decision to use a cycle count table to specify the cycle counts also makes it easy to indicate a specific instruction as unimplemented. When the cycle count lookup returns the value UN the instruction is unimplemented. Replace the just added cycle count code with

```
if (opcycles == OPCY_NONE) {
  opcycles = cpupag[cputyp][cb[0] & 0xFF];

  if (opcycles == UN) {
    switch(cputyp) {
    case MSM5840:
        xerr('x', "Invalid Or Unsupported MSM5840 Instruction");
        break;
    case MSM5842:
        xerr('x', "Invalid Or Unsupported MSM5842 Instruction");
        break;
    case MSM58421:
        xerr('x', "Invalid Or Unsupported MSM58421 Instruction");
        break;
    case MSM58422:
        xerr('x', "Invalid Or Unsupported MSM58422 Instruction");
        break;
    default:
        break;
    }
  }
}
```

1.12 ASSEMBLER VERIFICATION

To verify the correct operation of the assembler a comprehensive test file should be created which tests every insruction (or an instruction type) and all addressing modes with each instruction type. The method that seems to work the best is to hand code each instruction from the documentation and compare it with the assembler output. Once this process has been completed with arguments that are NOT global references then modify your arguments to include global references and verify again. You can verify for the same result if the global reference has a value of zero.

This completes the creation of the asolms40 assembler. For details in writing assembler code for these processors refer to the OKI OLMS-40 documentation.

CHAPTER 2

AS370 STEP BY STEP

2.1 GETTING STARTED

This step by step process will show how to create a new assembler using the facilities of the ASxxxx assembler core functions. The creation of an assembler for the Texas Instruments TMS370 8-bit microcontroller family will illustrate the process.

The basic steps are:

1) Know your microprocessor by collecting the necessary information: mnemonics, binary instruction codes, addressing modes, and cycle counts (otional).

2) For simple processors create a binary table of instructions showing opcodes and cycle counts.

3) Create a list of addressing modes. Concentrate on modes that are common to multiple instructions.

2.2 TMS370 INSTRUCTION SET

Studying the TMS370 processor documentation it can be seen that the TMS370 core is considerably more complex than the OKI MSM5840 processor descibed in the previous Step By Step. The TMS370 has over 45 instructions, 14 specific addressing modes, a 64K byte addressing range, a 256 byte register range, and 256 byte peripheral register range. The TMS370 instruction set and opcode maps are shown in pages from the Texas Instruments TMS Family Data Book, 1988.

Following the TMS 370 opcode map is an opcode map organized to reflect the organization of the opcycle table.

12.3 Instruction Set Overview

The following tables provide a listing of the instruction set symbols, a listing of the instruction set itself including pertinent characteristics, and an opcode/instruction map.

Table 12-2. TMS370 Symbol Definitions

SYMBOL	DEFINITION	SYMBOL	DEFINITION
A	Register A or R0 in Register File	B	Register B or R1 in Register File
Rn	Register n of Register File	Pn	Register n of Peripheral File ($0 \leq n \leq 255$)
s	Source operand	d/D	Destination operand (8-bit/16-bit)
Rs	Source register in Register File	Ps	Source register in Peripheral File ($0 \leq s \leq 255$)
Rd	Destination register in Register File	Pd	Destination register in Peripheral File ($0 \leq d \leq 255$)
Rps	Source register pair	Rpd	Destination register pair
iop8	8-bit Immediate operand	iop16	16-bit Immediate operand
off8	8-bit Signed Offset	off16	16-bit Signed Offset
Rp	Register pair	label	16-bit label
ST	Status Register	SP	Stack Pointer
PC	Program Counter	PCN	16-bit address of next instruction
#	Immediate operand	@	Indirect addressing operand
MSB	Most significant byte	LSB	Least significant byte
MSb	Most significant bit	LSb	Least significant bit
cnd	Condition	( )	Contents of
→	Is assigned to	←	Becomes equal to
[ ]	Indicates an optional entry. The brackets themselves are not entered.	< >	Indicates an entry that must be typed in. For example, <label> indicates that a label must be entered. The brackets themselves are not entered.
C	Carry flag	N	Sign flag
V	Overflow/borrow flag	Z	Zero flag
XADDR	16-bit address	name	symbol defined for a bit
Rname	symbol defined register bit	Pname	symbol defined peripheral bit

Table 12-3 lists all instruction formats, opcodes, byte lengths, cycles/instruction, operand types, status bits affected, and an operational description.

Table 12-3. TMS370 Family Instruction Overview

MNEMONIC		OPCODE	BYTES	CYCLES t_C	STATUS C	N	Z	V	OPERATION DESCRIPTION
ADC	B,A	69	1	8	x	x	x	x	(s) + (d) + (C) → (d)
	Rs,A	19	2	7					Add the source, destination, and carry bit
	Rs,B	39	2	7					together. Store at the destination address.
	Rs,Rd	49	3	9					
	#iop8,A	29	2	6					
	#iop8,B	59	2	6					
	#iop8,Rd	79	3	8					
ADD	B,A	68	1	8	x	x	x	x	(s) + (d) → (d)
	Rs,A	18	2	7					Add the source and destination operands at
	Rs,B	38	2	7					the destination address.
	Rs,Rd	48	3	9					
	#iop8,A	28	2	6					
	#iop8,B	58	2	6					
	#iop8,Rd	78	3	8					
AND	A,Pd	83	2	9	0	x	x	0	(s) AND (d) → (d)
	B,A	63	1	8					AND the source and destination operands
	B,Pd	93	2	9					together and store at the destination
	Rs,A	13	2	7					address.
	Rs,B	33	2	7					
	Rs,Rd	43	3	9					
	#iop8,A	23	2	6					
	#iop8,B	53	2	6					
	#iop8,Rd	73	3	8					
	#iop8,Pd	A3	3	10					
BR	label	8C	3	9	-	-	-	-	XADDR → (PC)
	@Rp	9C	2	8					Branch to the destination address.
	label(B)	AC	3	11					
	off8(Rp)	F4 EC	4	16					
(1)					0	x	x	0	If (s) AND (d) ≠ 0,
BTJO	A,Pd,off8	86	3	10					then PCN + offset → (PC)
	B,A,off8	66	2	10					If the AND of the source and destination
	B,Pd,off8	96	3	10					operands ≠ 0 (corresponding 1 bits)
	Rs,A,off8	16	3	9					The PC will add the offset, and the
	Rs,B,off8	36	3	9					jump will be taken.
	Rs,Rd,off8	46	4	11					
	#iop8,A,off8	26	3	8					
	#iop8,B,off8	56	3	8					
	#iop8,Rd,off8	76	4	10					
	#iop8,Pd,off8	A6	4	11					
(1)					0	x	x	0	If (s) AND (not d) ≠ 0
BTJZ	A,Pd,off8	87	3	10					then (PCN) + offset → (PC)
	B,A,off8	67	2	10					destination operands ≠ 0 (jump if
	B,Pd,off8	97	3	10					corresponding 1 and 0 bits). The PC
	Rs,A,off8	17	3	9					will add the offset and the jump
	Rs,B,off8	37	3	9					will be taken.
	Rs,Rd,off8	47	4	11					
	#iop8,A,off8	27	3	8					
	#iop8,B,off8	57	3	8					
	#iop8,Rd,off8	77	4	10					
	#iop8,Pd,off8	A7	4	11					

Note: 1.Add two to cycle count if jump is taken.

Legend:
0 Status Bit always cleared.
1 Status Bit always set.
x Status Bit cleared or set on results.
- Status Bit not affected.

Table 12-3. TMS370 Family Instruction Overview (Continued)

MNEMONIC		OPCODE	BYTES	CYCLES t_C	STATUS C N Z V	OPERATION DESCRIPTION
CALL	label	8E	3	13	- - - -	Push PC MSB, PC LSB,
	@Rp	9E	2	12		XADDR → (SP)
	label(B)	AE	3	15		
	off8(Rp)	F4 EE	4	20		
CALLR	label	8F	3	15	- - - -	Call Relative
	@Rp	9F	2	14		Push PC MSB, PC LSB,
	label(B)	AF	3	17		PCN + (XADDR) → (PC)
	off8(Rp)	F4 EF	4	22		
CLR	A	B5	1	8	0 0 1 0	0 → (d)
	B	C5	1	8		Clear the destination operand.
	Rd	D5	2	6		
CLRC		B0	1	9	0 x x 0	0 → (C)
						Clears the carry bit.
CMP	label,A	8D	3	11	x x x x	Compare; (d) - (s) computed.
	@Rp,A	9D	2	10		Set flags on the result of the source
	label(B),A	AD	3	13		operand subtracted from the destination
	off8(Rp),A	F4 ED	4	18		operand. Operands are not affected by
	off8(SP),A	F3	2	8		operation.
	B,A	6D	1	8		
	Rs,A	1D	2	7		
	Rs,B	3D	2	7		
	Rs,Rd	4D	3	9		
	#iop8,A	2D	2	6		
	#iop8,B	5D	2	6		
	#iop,Rd	7D	3	8		
CMPBIT	Rname	75	3	8	0 x x 0	Complement Bit; invert the bit
	Pname	A5	3	10		
COMPL	A	BB	1	8	x x x 0	Two's complement;
	B	CB	1	8		00h - (s) → (d)
	Rd	DB	2	10		
DAC	B,A	6E	1	10	x x x x	(s) + (d) + (C) → (d) (BCD)
	Rs,A	1E	2	9		The source, destination, and the carry bit
	Rs,B	3E	2	9		are added, and the BCD sum is stored at the
	Rs,Rd	4E	3	11		destination address.
	#iop8,A	2E	2	8		
	#iop8,B	5E	2	8		
	#iop8,Rd	7E	3	10		
DEC	A	B2	1	8	x x x x	(d) - 1 → (d)
	B	C2	1	8		Decrement destination operand by 1.
	Rd	D2	2	6		
DINT		F0 00	2	6	0 0 0 0	0 → (ST)(global interrupt enable bits)
						0 → IE1, 0 → IE2.

Note: Add two to cycle count if jump is taken.

Legend:
0 Status Bit always cleared.
1 Status Bit always set.
x Status Bit cleared or set on results.
- Status Bit not affected.

Table 12-3. TMS370 Family Instruction Overview (Continued)

MNEMONIC	OPCODE	BYTES	CYCLES t_C	STATUS C N Z V	OPERATION DESCRIPTION
DIV Rs,A	F4 F8	3	47-63	0 x x 0	A:B/Rs → A(= quo),B(= REM) Integer divide, 16 by 8 bit.
			14	1 1 1 1	Detected overflow
(1) DJNZ A,off8	BA	2	10	- - - -	(d) - 1 → (d); If (d) ≠ 0, then PCN + offset → (PC) Decrement and jump if not 0.
B,off8	CA	2	10		
Rd,off8	DA	3	8		
DSB B,A	6F	1	10	x x x x	(d) - (s) - 1 + (C) → (d) (BCD) The source operand is subtracted from the destination; this sum is then reduced by 1 and the carry bit is then added to it. The result is stored as a BCD number.
Rs,A	1F	2	9		
Rs,B	3F	2	9		
Rs,Rd	4F	3	11		
#iop8,A	2F	2	8		
#iop8,B	5F	2	8		
#iop8,Rd	7F	3	10		
EINT	F0 0C	2	6	0 0 0 0	0Ch → (ST)(global interrupt enable bit) 1 → IE1, 1 → IE2.
EINTH	F0 04	2	6	0 0 0 0	04h → (ST)(high priority global interrupt enable bit). 1 → IE1, 0 → IE2
EINTL	F0 08	2	6	0 0 0 0	08h → (ST)(low priority global interrupt enable bit) 0 → IE1, 1 → IE2
IDLE	F6	1	6	- - - -	(PC) → (PC) until interrupt (PC) + 1 → (PC) after return from interrupt Stops µC execution until an interrupt.
INC A	B3	1	8	x x x x	(d) + 1 → (d) Increase the destination operand by 1.
B	C3	1	8		
Rd	D3	2	6		
INCW #off8,Rp	70	3	11	x x x x	(Rp) + offset → (Rp) Add 8-bit signed offset to register pair.
INV A	B4	1	8	0 x x 0	NOT(d) → (d) 1's complement the destination operand.
B	C4	1	8		
Rd	D4	2	6		
(1) JBITO Rd,off8	77	4	10	0 x x 0	Jump If Bit = 0
Pd,off8	A7	4	11		
(1) JBIT1 Rd,off8	76	4	10	0 x x 0	Jump If Bit = 1
Pd,off8	A6	4	11		
JMP off8	00	2	7	- - - -	PCN + off8 → (PC) Jump unconditionally using an 8-bit offset.

Note: 1. Add two to cycle count if jump is taken.

Legend:
0 Status Bit always cleared.
1 Status Bit always set.
x Status Bit cleared or set on results.
- Status Bit not affected.

Table 12-3. TMS370 Family Instruction Overview (Continued)

MNEMONIC	OPCODE	BYTES	CYCLES t_C	STATUS C N Z V	OPERATION DESCRIPTION
JMPL label	89	3	9	- - - -	PCN + D → (PC)
@Rp	99	2	8		Jump unconditionally using
label(B)	A9	3	11		a 16-bit offset
off8(Rp)	F4 E9	4	16		
(1) Jcnd				- - -	- Conditional jump
JC	03	2	5		Carry
JEQ	02	2	5		Jump Equal
JG	0E	2	5		Greater Than, signed
JGE	0D	2	5		Greater Than or Equal, signed
JHS	0B	2	5		Higher or Same, unsigned
JL	09	2	5		Less Than, signed
JLE	0A	2	5		Less Than or Equal, signed
JLO	0F	2	5		Lower Value, unsigned
JN	01	2	5		Negative, signed
JNC	07	2	5		No Carry
JNE	06	2	5		Jump Not Equal
JNV	0C	2	5		No Overflow, signed
JNZ	06	2	5		Not Zero
JP	04	2	5		Positive, signed
JPZ	05	2	5		Positive or Zero, signed
JV	08	2	5		Overflow, signed
JZ	02	2	5		Zero
LDSP	FD	1	7	- - - -	(B) → (SP) Load stack pointer with contents of register B.
LDST #iop8	F0	2	6	x x x x	(d) → (ST) Load ST Register

Note: 1.Add two to cycle count if jump is taken.

Legend:
0 Status Bit always cleared.
1 Status Bit always set.
x Status Bit cleared or set on results.
- Status Bit not affected.

12

Table 12-3. TMS370 Family Instruction Overview (Continued)

MNEMONIC		OPCODE	BYTES	CYCLES t_C	STATUS C	N	Z	V	OPERATION DESCRIPTION
MOV	A,B	C0	1	9	0	x	x	0	(s) → (d) Replace the destination operand with the source operand.
	A,Rd	D0	2	7					
	A,Pd	21	2	8					
	A,label	8B	3	10					
	A,@Rp	9B	2	9					
	A,label(B)	AB	3	12					
	A,off8(Rp)	F4 EB	4	16					
	A,off8(SP)	F2	2	7					
	Rs,A	12	2	7					
	Rs,B	32	2	7					
	label,A	8A	3	10					
	@Rp,A	9A	2	9					
	label(B),A	AA	3	12					
	off8(Rp),A	F4 EA	4	17					
	off8(SP),A	F1	2	7					
	B,A	62	1	8					
	B,Rd	D1	2	7					
	B,Pd	51	2	8					
	Rs,Rd	42	3	9					
	Rs,Pd	71	3	10					
	Ps,A	80	2	8					
	Ps,B	91	2	8					
	Ps,Rd	A2	3	10					
	#iop8,A	22	2	6					
	#iop8,B	52	2	6					
	#iop8,Rd	72	3	8					
	#iop8,Pd	F7	3	10					
MOVW	Rps,Rpd	98	3	12	0	x	x	0	(s) → (Rpd) Copy the source register word to the destination register pair.
	#iop16,Rpd	88	4	13					
	#iop16(B),Rpd	A8	4	15					
	off8(Rs),Rpd	F4 E8	5	20					
MPY	B,A	6C	1	47	0	x	x	0	(s) × (d) → (A:B) Multiply the source and destination operands, store the result in Registers A (MSB) and B (LSB).
	Rs,A	1C	2	46					
	Rs,B	3C	2	46					
	Rs,Rd	4C	3	48					
	#iop8,A	2C	2	45					
	#iop8,B	5C	2	45					
	#iop8,Rn	7C	3	47					
NOP		FF	1	7	-	-	-	-	No operation
OR	A,Pd	84	2	9	0	x	x	0	(s) OR (d) → (d) Logically OR the source and destination operands, and store the results at the destination address.
	B,A	64	1	8					
	B,Pd	94	2	9					
	Rs,A	14	2	7					
	Rs,B	34	2	7					
	Rs,Rd	44	3	9					
	#iop8,A	24	2	6					
	#iop8,B	54	2	6					
	#iop8,Rd	74	3	8					
	#iop8,Pd	A4	3	10					

Legend:
0 Status Bit always cleared.
1 Status Bit always set.
x Status Bit cleared or set on result.
- Status Bit not affected.

Table 12-3. TMS370 Family Instruction Overview (Continued)

MNEMONIC	OPCODE	BYTES	CYCLES t_C	STATUS C N Z V	OPERATION DESCRIPTION
POP A	B9	1	9	0 x x 0	((SP)) → (d)
B	C9	1	9		(SP) -1 → (SP)
Rd	D9	2	7		
ST	FC	1	8	x x x x	
PUSH A	B8	1	9	0 x x 0	(SP) + 1 → (SP)
B	C8	1	9		(s) → ((SP))
Rd	D8	2	7		Copy the operand onto the stack.
ST	FB	1	8	- - - -	Copy the Status Register onto the Stack
RL A	BE	1	8	x x x 0	Bit(n) → Bit(n + 1)
B	CE	1	8		Bit(7) → Bit(0) and Carry
Rd	DE	2	6		
RLC A	BF	1	8	x x x 0	Bit(n) → Bit(n + 1)
B	CF	1	8		Carry → Bit(0)
Rd	DF	2	6		Bit(7) → Carry
RR A	BC	1	8	x x x 0	Bit(n + 1) → Bit(n)
B	CC	1	8		Bit(0) → Bit(7) and Carry
Rd	DC	2	6		
RRC A	BD	1	8	x x x 0	Bit(n + 1) → Bit(n)
B	CD	1	8		Carry → Bit(7)
Rd	DD	2	6		Bit(0) → Carry
RTI	FA	1	12	x x x x	Pop PCL, PCH, POP ST Return From Interrupt
RTS	F9	1	9	- - - -	Pop PCL, PCH
SBB B,A	6B	1	8	x x x x	(d) - (s) - 1 + (C) → (d)
Rs,A	1B	2	7		Subtract with borrow.
Rs,B	3B	2	7		Destination minus source minus 1 plus
Rs,Rd	4B	3	9		carry; stored at the destination address.
#iop8,A	2B	2	6		
#iop8,B	5B	2	6		
#iop8,Rd	7B	3	8		
SBIT0 Rd	73	3	8	0 x x 0	Set Bit to 0
Pd	A3	3	10		
SBIT1 Rd	74	3	8	0 x x 0	Set Bit to 1
Pd	A4	3	10		
SETC	F8	1	7	1 0 1 0	Axh → (ST) Set the carry bit. IE1 and IE2 unchanged.

Note: 1.Add two to cycle count if jump is taken.

Legend:
0 Status Bit always cleared.
1 Status Bit always set.
x Status Bit cleared or set on results.
- Status Bit not affected.

Table 12-3. TMS370 Family Instruction Overview (Concluded)

MNEMONIC		OPCODE	BYTES	CYCLES t_C	STATUS C N Z V	OPERATION DESCRIPTION
STSP		FE	1	8	- - - -	(SP) → (B) Copy the SP into Register B.
SUB	B,A	6A	1	8	x x x x	(d) - (s) → (d) Store the destination operand minus the source operand into the destination.
	Rs,A	1A	2	7		
	Rs,B	3A	2	7		
	Rs,Rd	4A	3	9		
	#iop8,A	2A	2	6		
	#iop8,B	5A	2	6		
	#iop8,Rd	7A	3	8		
SWAP	A	B7	1	11	0 x x 0	s(7-4,3-0) → d(3-0,7-4) Swap the operand's hi and lo nibbles.
	B	C7	1	11		
	Rd	D7	2	9		
TRAP	n	EF-E0	1	14	- - - -	Vector n → (PC), n = 0 → 15 Trap to Subroutine; Push PCN Trap 0 = EF
TST	A	B0	1	9	0 x x 0	Test; Set flags from register.
	B	C6	1	10		
XCHB	A	B6	1	10	0 x x 0	(B) ←→ (Rn) Swap the contents of Register B with (Rn).
	B	C6	1	10		
	Rd	D6	2	8		
XOR	A,Pd	85	2	9	0 x x 0	(s) XOR (d) → (d) Logically exclusive OR the source and destination operands, store at the destination address.
	B,A	65	1	8		
	B,Pd	95	2	9		
	Rs,A	15	2	7		
	Rs,B	35	2	7		
	Rs,Rd	45	3	9		
	#iop8,A	25	2	6		
	#iop8,B	55	2	6		
	#iop8,Rd	75	3	8		
	#iop8,Pd	A5	3	10		

Note: 1.Add two to cycle count if jump is taken.

Legend:
0 Status Bit always cleared.
1 Status Bit always set.
x Status Bit cleared or set on results.
- Status Bit not affected.

Table 12-4. TMS370 Family Opcode/Instruction Map

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	JMP ra 2/7							INCW #n,Rd 3/11	MOV Ps,A 2/8			CLRC TST A 1/9	MOV A,B 1/9	MOV A,Rd 2/7	TRAP 15 1/14	LDST n 2/6
1	JN ra 2/5		MOV A,Pd 2/8			MOV B,Pd 2/8		MOV Rs,Pd 3/10		MOV Ps,B 2/7				MOV B,Rd 2/7	TRAP 14 1/14	MOV n(SP),A 2/7
2	JZ ra 2/5	MOV Rs,A 2/7	MOV #n,A 2/6	MOV Rs,B 2/7	MOV Rs,Rd 3/9	MOV #n,B 2/6	MOV B,A 1/8	MOV #n,Rd 3/8			MOV Ps,Rd 3/10	DEC A 1/8	DEC B 1/8	DEC Rn 2/6	TRAP 13 1/14	MOV A,n(SP) 2/7
3	JC ra 2/5	AND Rs,A 2/7	AND #n,A 2/6	AND Rs,B 2/7	AND Rs,Rd 3/9	AND #n,B 2/6	AND B,A 1/8	AND #n,Rd 3/8	AND A,Pd 2/9	AND B,Pd 2/9	AND #n,Pd 3/10	INC A 1/8	INC B 1/8	INC Rn 2/6	TRAP 12 1/14	CMP n(SP),A 2/8
4	JP ra 2/5	OR Rs,A 2/7	OR #n,A 2/6	OR Rs,B 2/7	OR Rs,Rd 3/9	OR #n,B 2/6	OR B,A 1/8	OR #n,Rd 3/8	OR A,Pd 2/9	OR B,Pd 2/9	OR #n,Pd 3/10	INV A 1/8	INV B 1/8	INV Rn 2/6	TRAP 11 1/14	extend inst,2 opcodes
5	JPZ ra 2/5	XOR Rs,A 2/7	XOR #n,A 2/6	XOR Rs,B 2/7	XOR Rs,Rd 3/9	XOR #n,B 2/6	XOR B,A 1/8	XOR #n,Rd 3/8	XOR A,Pd 2/9	XOR B,Pd 2/9	XOR #n,Pd 3/10	CLR A 1/8	CLR B 1/8	CLR Rn 2/6	TRAP 10 1/14	
6	JNZ ra 2/5	BTJO Rs,A 3/9	BTJO #n,A 3/8	BTJO B,Rd 3/9	BTJO Rs,Rd 4/11	BTJO #n,B 3/8	BTJO B,A 2/10	BTJO #n,Rd 4/10	BTJO A,Pd 3/11	BTJO B,Pd 3/10	BTJO #n,Pd 4/11	XCHB A 1/10	XCHB TESTB 1/10	XCHB Rn 2/8	TRAP 9 1/14	IDLE 1/6
7	JNC ra 2/5	BTJZ Rs,A 3/9	BTJZ #n,A 3/8	BTJZ Rs,B 3/9	BTJZ Rs,Rd 4/11	BTJZ #n,B 3/8	BTJZ B,A 2/10	BTJZ #n,Rd 4/10	BTJZ A,Pd 3/10	BTJZ B,Pd 3/10	BTJZ #n,Pd 4/11	SWAP A 1/11	SWAP B 1/11	SWAP Rn 2/9	TRAP 8 1/14	MOV #n,Pd 3/10
8	JV ra 2/5	ADD Rs,A 2/7	ADD #n,A 2/6	ADD Rs,B 2/7	ADD Rs,Rd 3/9	ADD #n,B 2/6	ADD B,A 1/8	ADD #n,Rd 3/8	MOVW #16,Rd 4/13	MOVW Rs,Rd 3/12	MOVW #16(B),Rd 4/15	PUSH A 1/9	PUSH B 1/9	PUSH Rs 2/7	TRAP 7 1/14	SETC 1/7
9	JL ra 2/5	ADC Rs,A 2/7	ADC #n,A 2/6	ADC Rs,B 2/7	ADC Rs,Rd 3/9	ADC #n,B 2/6	ADC B,A 1/8	ADC #n,Rd 3/8	JMPL lab 3/9	JMPL @Rd 2/8	JMPL lab(B) 3/10	POP A 1/9	POP B 1/9	POP Rd 2/7	TRAP 6 1/14	RTS 1/9
A	JLE ra 2/5	SUB Rs,A 2/7	SUB #n,A 2/6	SUB Rs,B 2/7	SUB Rs,Rd 3/9	SUB #n,B 2/6	SUB B,A 1/8	SUB #n,Rd 3/8	MOV lab,A 3/10	MOV @Rs,A 2/9	MOV lab(B),A 3/12	DJNZ A,ra 2/10	DJNZ B,ra 2/10	DJNZ Rn,ra 3/8	TRAP 5 1/14	RTI 1/12
B	JHS ra 2/5	SBB Rs,A 2/7	SBB #n,A 2/6	SBB Rs,B 2/7	SBB Rs,Rd 3/9	SBB #n,B 2/6	SBB B,A 1/8	SBB #n,Rd 3/8	MOV A,lab 3/10	MOV A,@Rd 2/9	MOV A,lab(B) 3/12	COMPL A 1/8	COMPL B 1/8	COMPL Rn 2/10	TRAP 12 1/14	PUSH ST 1/8
C	JNV ra 2/5	MPY Rs,A 2/46	MPY #n,A 2/45	MPY Rs,B 2/46	MPY Rs,Rd 3/48	MPY #n,B 2/45	MPY B,A 1/47	MPY #n,Rs 3/47	BR lab 3/9	BR @Rd 2/8	BR lab(B) 3/11	RR A 1/8	RR B 1/8	RR Rn 2/6	TRAP 3 1/14	POP ST 1/8
D	JGE ra 2/5	CMP Rs,A 2/7	CMP #n,A 2/6	CMP Rs,B 2/7	CMP Rs,Rd 3/9	CMP #n,B 2/6	CMP B,A 1/8	CMP #n,Rd 3/8	CMP lab,A 3/11	CMP @Rs,A 2/10	CMP lab(B),A 3/13	RRC A 1/8	RRC B 1/8	RRC Rn 2/6	TRAP 2 1/14	LDSP 1/7
E	JG ra 2/5	DAC Rs,A 2/9	DAC #n,A 2/8	DAC Rs,B 2/9	DAC Rs,Rd 3/11	DAC #n,B 2/8	DAC B,A 1/10	DAC #n,Rd 3/10	CALL lab 3/13	CALL @Rd 2/12	CALL lab(B) 3/15	RL A 1/8	RL B 1/8	RL Rn 2/6	TRAP 1 1/14	STSP 1/8
F	JLO ra 2/5	DSB Rs,A 2/9	DSB #n,A 2/8	DSB Rs,B 2/9	DSB Rs,Rd 3/11	DSB #n,B 2/8	DSB B,A 1/10	DSB #n,Rd 3/10	CALLR lab 3/15	CALLR @Rd 2/14	CALLR lab(B) 3/17	RLC A 1/8	RLC B 1/8	RLC Rn 2/6	TRAP 0 1/14	NOP 1/7
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Note:

All conditional jumps (opcodes 01-0F), BTJO, and BTJZ instructions use two additional cycles if the branch is taken. The BTJO and BTJZ instructions have a relative address as the last operand.

Second byte of two-byte instructions (F4xx):

	E	F
8	MOVW n(Rn) 4/15	DIV Rn,A 3/14-63
9	JMPL n(Rn) 4/16	
A	MOV n(Rn),A 4/17	
B	MOV A,n(Rn) 4/16	
C	BR n(Rn) 4/16	
D	CMP n(Rn) 4/18	
E	CALL n(Rn) 4/20	
F	CALLER n(Rn) 4/22	
	E	F

ra - relative address
Rn - Register
Rs - Register containing source byte
Rd - Register containing destination byte
Ps - Peripheral register containing source byte
Pd - Peripheral register containing destination byte
Pn - Peripheral register
n - Immediate 8-bit number
#16 - Immediate 16-bit number
lab - 16-bit label

Tms 370

	Ø0	Ø1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	9 Jmp	7 JN	7 JEQ JZ	7 JC	7 JP	7 JPZ	7 JNE JNZ	7 JNC	7 JV	7 JL	7 JLE	7 JHS	7 JNV	7 JGE	7 JG	7 JLO
0x10	X	X	7 MOV Rs,A	7 AND	7 OR	7 XOR	11 BTJO	11 BTJZ	7 ADD	7 ADC	7 SUB	7 SBB	46 MPY	7 CMP	9 DAC	9 DSB Rs,A
2	X	8 MOV A,Pd	6 MOV #N,A	6 AND	6 OR	6 XOR	10 BTJO	10 BTJZ	6 ADD	6 ADC	6 SUB	6 SBB	45 MPY	6 CMP	8 DAC	8 DSB #N,A
3	X	X	7 MOV Rs,B	7 AND	7 OR	7 XOR	11 BTJO	11 BTJZ	7 ADD	7 ADC	7 SUB	7 SBB	46 MPY	7 CMP	9 DAC	9 DSB Rs,B
4	X	X	9 MOV Rs,Rd	9 AND	9 OR	9 XOR	13 BTJO	13 BTJZ	9 ADD	9 ADC	9 SUB	9 SBB	48 MPY	9 CMP	11 DAC	11 DSB Rs,Rd
5	X	8 MOV B,Pd	6 MOV #N,B	6 AND	6 OR	6 XOR	10 BTJO	10 BTJZ	6 ADD	6 ADC	6 SUB	6 SBB	45 MPY	6 CMP	8 DAC	8 DSB #N,B
6	X	X	8 MOV B,A	8 AND	8 OR	8 XOR	12 BTJO	12 BTJZ	8 ADD	8 ADC	8 SUB	8 SBB	47 MPY	8 CMP	10 DAC	10 DSB B,A
7	11 INCW #N,Rd	10 MOV Rs,Pd	8 MOV #N,Rd	8 AND	8 OR	8 XOR	12 BTJO	12 BTJZ	8 ADD	8 ADC	8 SUB	8 SBB	47 MPY	8 CMP	10 DAC	10 DSB #N,Rd
8	8 MOV Ps,A	X	X	9 AND A,Pd	9 OR	9 XOR	13 BTJO	12 BTJZ A,Pd	13 MOVW #16,Rd	9 JMPL Lab	10 MOV lab,A	10 MOV A,lab	11 BR lab	11 CMP Lab,A	13 CALL lab	15 CALR lab
9	X	7 MOV Ps,B	X	9 AND B,Pd	9 OR	9 XOR	12 BTJO	12 BTJZ B,Pd	12 MOVW Ps,Rd	8 JMPL @Rd	9 MOV @Ps,A	9 MOV A,@Rd	10 BR @Rd	10 CMP @Ps,A	12 Call @Rd	14 CALR @Rd
A	X	X	10 MOV Ps,Rd	10 AND #N,Pd	10 OR	10 XOR	13 BTJO	13 BTJZ #N,Pd	15 MOVW #16(B),Rd	10 JMPL lab(B)	12 MOV lab(B),A	12 MOV A,lab(B)	13 BR lab(B)	13 CMP lab(B),A	15 CALL lab(B)	17 CALR lab(B)
B	9 CLRC ST A	X	8 DEC A	8 INC A	8 INV A	8 CLR A	10 XCHB A	11 SWAP A	9 PUSH A	9 POP A	12 DJNZ A,ra	8 COMPL A	8 RR A	8 RRC A	8 RL A	8 RLC A
C	9 MOV A,B	X	8 DEC B	8 INC B	8 INV B	8 CLR B	10 XCHB B	11 SWAP B	9 PUSH B	9 POP B	12 DJNZ B,ra	8 COMPL B	8 RR B	8 RRC B	8 RL B	8 RLC B
D	7 MOV A,Rd	7 MOV B,Rd	6 DEC Rn	6 INC Rn	6 INV Rn	6 CLR Rn	8 XCHB Rn	9 SWAP Rn	7 PUSH Rs	7 POP Rd	10 DJNZ Rn,ra	10 COMPL Rn	6 RR Rn	6 RRC Rn	6 RL Rn	6 RLC Rn
E	14 TRAP 15	14 TRAP 14													14 TRAP 1	14 TRAP 0
F	6 LDST IN	7 MOV N(SP),A	7 MOV A,N(SP)	8 CMP N(SP),A	P1	X	6 IDLE	10 MOV #N,Pd	7 SETC	9 RTS	12 RTI	8 PUSH ST	8 POP ST	7 LDSP	8 STSP	7 NOP

2.3 INSTRUCTION TYPES

Analyzing this table notice that various instructions can be grouped together with similar patterns:

1) JMP and the conditional jumps have the same program relative addressing and comprise 16 opcodes, 0x00 - 0x0F.

2) The TRAP instruction has 16 opcodes, 0xE0 - 0xEF.

3) AND, OR, XOR, BTJO, and BTJ1 each share the same 10 addressing modes and comprise a total of 50 opcodes. BTJ0 and BTJI have 1 additional argument for the branch destination.

4) ADD, ADC, SUB, SBB, MPY, CMP, DAC, and DSB share the same 7 addressing modes for a total of 56 opcodes. CMP has 4 additional opcodes in the table and 1 that is an extension.

5) DEC, INC, INV, CLR, XCHB, SWAP, PUSH, POP, DJNZ, COMPL, RR, RRC, RL, and RLC share 3 addressing modes for 42 opcodes. DJNZ has 1 additional argument for the branch destination. PUSH and POP use 1 additional addressing mode, ST, for 2 more opcodes. The INCW, the 16 bit version of INC, uses 1 opcode.

6) JMPL, BR, CALL, and CALLR share 3 addressing modes for 12 opcodes. JMPL and CALLR use 16 bit program relative addressing, BR and CALL do not.

7) MOV is an instruction unto itself using 24 opcodes spread throughout the opcode table. MOVW, the 16 bit version of MOV, uses 3 opcodes.

8) LDST has one addressing mode and uses just 1 opcode.

9) JBIT0, JBIT1, SBIT0, and SBIT0 each have 2 addressing modes. These extension instructions all have 0xF4 as a prefix byte.

10) The remaining instructions are 1 byte opcodes with no arguments: CLRC, TSTA, TSTB, IDLE, SETC, RTS, RTI, LDSP, STSP, and NOP.

For an assembler of this complexity it is useful to develop the assembler one instruction at a time. From the directory

asxvp6xx\asxmisc copy the files xxx.h, xxxpst.c, xxxmch.c, and xxxadr.c to a directory created for the TMS370 assembler, as370. Rename the files to indicate the processor type - m370.h, m370pst.c, m370mch.c, and m370adr.c. The file name suffix m370 is arbitrary but was selected from TMS370 processor type.

Edit the newly named m370.h file to reflect the assembler being created. Remove the 'Other' definition, comment out the the 'Registers' and 'Extended Addressing Modes' definitions.

Beginning with the TRAP instruction replace

```
#define I_XXX   50              /* xxxxxx */
```

with

```
#define I_TRAP  60              /* TRAP */
```

in the m370.h file.

The type values MUST be 30 or greater so as not to conflict with the values assigned to the ASxxxx directives. Failure to adhere to this restriction will result in program execution errors.

2.4 MNEMONIC STRUCTURE

```
/*
 *      The mne structure is a linked list of the assembler
 *      mnemonics and directives.  The list of mnemonics and
 *      directives contained in the device dependent file
 *      xxxpst.c are hashed and linked into NHASH lists in
 *      module assym.c by syminit().  The structure contains
 *      the mnemonic/directive name, a subtype which directs
 *      the evaluation of this mnemonic/directive, a flag which
 *      is used to detect the end of the mnemonic/directive
 *      list in xxxpst.c, and a value which is normally
 *      associated with the assembler mnemonic base instruction
 *      value.
 */
struct  mne
{
        struct  mne *m_mp;      /* Hash link */
        char    *m_id;          /* Mnemonic (JLH) */
        char    m_type;         /* Mnemonic subtype */
        char    m_flag;         /* Mnemonic flags */
        a_uint  m_valu;         /* Value */
};
```

The TMS370 processor uses a byte addressable program memory with addresses using upto 2 bytes (1 word). Thus any ASxxxx directive that uses more than 2 bytes can be removed by commenting out those directives. Open the file m370pst.c and comment out the following directives as shown:

```
/*
    {   NULL,   ".3byte",       S_DATA,         0,      O_3BYTE },
    {   NULL,   ".triple",      S_DATA,         0,      O_3BYTE },
    {   NULL,   ".dl",          S_DATA,         0,      O_4BYTE },
    {   NULL,   ".4byte",       S_DATA,         0,      O_4BYTE },
    {   NULL,   ".quad",        S_DATA,         0,      O_4BYTE },
    {   NULL,   ".long",        S_DATA,         0,      O_4BYTE },

    {   NULL,   ".blk3",        S_BLK,          0,      O_3BYTE },
    {   NULL,   ".blk4",        S_BLK,          0,      O_4BYTE },
    {   NULL,   ".blkl",        S_BLK,          0,      O_4BYTE },
*/
```

Directives only used under special conditions should also be commented out:

```
/*
    {   NULL,   ".msb",         S_MSB,          0,      0       },
    {   NULL,   ".lohi",        S_MSB,          0,      O_LOHI  },
    {   NULL,   ".hilo",        S_MSB,          0,      O_HILO  },
    {   NULL,   ".8bit",        S_BITS,         0,      O_1BYTE },
    {   NULL,   ".16bit",       S_BITS,         0,      O_2BYTE },
    {   NULL,   ".24bit",       S_BITS,         0,      O_3BYTE },
    {   NULL,   ".32bit",       S_BITS,         0,      O_4BYTE },
*/
```

Note the last entry in the mne structure has the symbol S_EOL in the fourth column. This is REQUIRED in the last entry, it signifies the end of the mnemonic structure during initialization.

2.5 THE TRAP INSTRUCTION

Remove the entries from /* Special */ to the end of the structure. Entries will be added for each of the instruction types defined in m370.h. Each entry will have the instruction type, instruction mnemonic, and the opcode value. Those instructions having extended addressing and argument types will have instruction base opcode values in the structure. The last entry will have the symbol S_EOL in the fourth element.

```
/* machine */

{   NULL,   "trap",         I_TRAP,         S_EOL,  0xE0    }
```

Open the file m370mch.c and change all the xxx references to reflect the new as370 assembler. Replace

```
case I_XXX:
```

with

```
/* TRAP */
case I_TRAP:
```

The argument of the TRAP instruction must be in the range 0 to 15 and translate to a number in the range 15 to 0. If the argument is a constant then the value could be calculated as 15 - N, where 0 <= N <= 15 and the argument could be processed by the expr() function with the expression value left in

e1.e_addr: (see the section on ASxxxx functions for a description of the evaluation structure which holds the result of the expr() function and the functions used)
Thus the assembly statement

```
trap    3       ; trap to interrupt level 3
```

would invoke the following case statement in m370mch.c:

```
case I_TRAP:
    expr(&e1);
    v = 15 - e1.e_addr;
    if ((v < 0) || (v > 15)) {
      xerr('a', "The valid argument range is 0 - 15");
    }
    outab(op | (v & 0x0F));
    break;
```

ASxxxx assemblers are relocatable and the arguments could be global, ie not defined in the current source file. If the trap is invoked with a global argument:

```
trap    T
```

then the evaluation of argument T results in an undefined value for e1.e_addr (globals always evaluate to 0) as the assembler has no idea what the value of T is. Thus an alternate method of processing the argument is required. In this case, where a computation is required, a special function is needed to evaluate the expression. This function in conjunction with the 'merge mode' of the ASxxxx core can provide the evaluation and the error reporting for a global argument.

The function is exprps() which creates an alternate argument to be evaluted by the general expr() function. Basically the argument is prefixed and suffixed to produce a new argument. The new argument required for evaluation is 15 - T. The case statement becomes:

```
/* TRAP */
case I_TRAP:
    /* Make Argument Globally Compatible */
    exprps("15 - (", &e1, ")");
    /* Output Result */
    outrbm(&e1, M_TRAP | R_MBRO, op);
    break;
```

Enclosing the argument in ( ) allows the argument to be an expression, not just a single symbol. The M_TRAP merge mode combines the opcode and the 4 bit value. If the argument is a constant and out of range then the assembler will report an out of bounds error. If the argument is a global then the linker will report any out of bounds argument.

2.6 THE EXPRPS() FUNCTION

The exprps() function must be added to the m370mch.c file:

```
/*
 * Expression with Prefix and Suffix
 */
void
exprps(char *pre, struct expr *esp, char *post)
{
        int c;
        char pbufr[NCPS];
        char *p;

        /* Copy Prefix To String Buffer */
        strcpy(pbufr, pre);

        /* Append Input Argument To String */
        p = pbufr + strlen(pbufr);
        unget(getnb());
        while (((c = get()) != 0) && (c != ';') && (c != ',')) *p++ = c;
        *p = 0;
        unget(c);
        /* Trim Argument String */
        while ((*(--p) == ' ') || (*p == '\t')) *p = 0;

        /* Append Suffix To String */
        strcat(p, post);

        /* Process Created Argument String */
        p = ip;
        ip = pbufr;
        expr(esp);
        ip = p;
}
```

2.7 EXTENDED ADDRESSING: MERGE MODES

The ASxxxx assemblers use a method called 'merge mode' to combine a value with a constant. This method passes the value or a reference to a value to the linker along with a recipe to combine this value with another absolute value, typically an op-code. Each ASxxxx assembler can define upto 16 unique 'merge mode' recipes for combining two values. The recipe index is specified by the third 4-bit nibble of the relocation parameter word value (0x0000, 0x0100, ..., 0x0E00, 0x0F00). The TMS370 requires 1 unique mode to handle the combining of addresses and arguments with the instruction opcodes. Create this entry under the Extended Addressing Modes in m370.h:

```
/*
 * Extended Addressing Modes
 */
#define M_TRAP  0x0100  /* 4-Bit argument Mode */
```

The 'merge mode' recipe will be defined in the m370pst.c file. The default merge mode, mode0[32], specifies a one-to-one bit replacement for 32 bits. The active bit entries are specified by setting bit 7 of the char to 1. Add the following additional merge mode for the TMS370 processor:

```
/*
 * Additional Relocation Mode Definitions
 *
 * Specification for the 4-bit argument mode:
 */
char    mode1[32] = {   /* M_TRAP */
        '\200', '\201', '\202', '\203', '\004', '\005', '\006', '\007',
        '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};
```

An addtional structure is used by the assembler to assist in processing the merge mode:

```
/*
 *     *m_def is a pointer to the bit relocation definition.
 *      m_flag indicates that bit position swapping is required.
 *      m_dbits contains the active bit positions for the output.
 *      m_sbits contains the active bit positions for the input.
 *
 *      struct  mode
 *      {
 *              char *  m_def;          Bit Relocation Definition
 *              a_uint  m_flag;         Bit Swapping Flag
 *              a_uint  m_dbits;        Destination Bit Mask
 *              a_uint  m_sbits;        Source Bit Mask
 *      };
 */
```

Add the new mode to the merge mode array:

```
struct  mode    mode[2] = {
    {   &mode0[0],      0,      0x0000FFFF,     0x0000FFFF      },
    {   &mode1[0],      0,      0x0000000F,     0x0000000F      }
};
```

Note that in all the merge mode recipes the source bit is placed into the same bit position in the destination. This is indicated in the structure by setting the bit swapping flag, mflag, equal to 0. An example of a complex merge mode can be found in the AVR assembler where a merge mode recipe and the corresponding structure entry are shown here.

```
char mode7[32] = {      /* S_ILDST instructions */
        /* |--K-|KK--|----|-KKK| */
        '\200', '\201', '\202', '\212', '\213', '\215', '\006', '\007',
        '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
        '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
        '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037'
};

    {   &mode7[0],      1,      0x00002C07,     0x0000003F      },
```

In this recipe bits 0, 1, and 2 are moved into bits 0, 1, and 2 of the destination while bits 3 and 4 are moved to bit positions 10 and 11 and bit 5 is moved to bit position 13 of the destination. (Note: the recipe numbers are in octal for

historical reasons.) In this case the bit swapping flag is set to 1.

And finally, the array of pointers to the mode structure elements must be updated to include the additional merge mode:

```
/*
 * Array of Pointers to mode Structures
 */
struct  mode    *modep[16] = {
        &mode[0],       &mode[1],       NULL,           NULL,
        NULL,           NULL,           NULL,           NULL,
        NULL,           NULL,           NULL,           NULL,
        NULL,           NULL,           NULL,           NULL
};
```

We have reached the point where the code can be compiled with your favorite compiler. Create a project named as370 which must include all the following files:

m370.h
m370mch.c, m370adr.c, m370pst.c

asxxxx.h
asdata.c, asdbg.c, asexpr.c, aslex.c, aslist.c
asmain.c, asmcro.c, asout.c, assubr.c, assym.c

Compiling the assembler at this point should produce an assembler, as370, that understands only 1 instruction: TRAP.

2.8 THE JMP INSTRUCTION

The next instruction to add is the JMP and all its 15 conditional variants. These instructions have a single argument. The argument is normally an assembly label. The assembler is required to compute the relative distance from the instruction to the label and use this offset value to create the instructions binary code. When the label is local and in the same progamming area an absolute offset can be calculated. However if the label is in a different program area or is global the assembler uses the program counter relative mode, R_PCR, and the linker performs the offset calculation for the instruction. The function mchpcr() found in the m370mch.c file is a common routine to many ASxxxx assemblers and examines the label type to determine if the absolute or PCR mode should be used. The PCR

mode assumes the relative offset is computed from the address following the last byte of the offset. In general when using the PCR mode all arguments are evaluated and output before the offset is calculated and output.

```
int     mchpcr(struct expr *esp, int *v, int n);
```

The function requires the expression result, an integer n - the number bytes in the offset for an absolute argument, and v - a pointer to an integer to return the value of an absolute offset. The function returns a 1 for an absolute offset and 0 for a relocatable offset.

Add the following case to m370mch.c:

```
/* JMP, JN, JEQ, JZ, JC, JP, JPZ, JNE, JNZ */
/* JNC, JV, JL, JLE, JHS, JNV, JGE, JG, JLO */
case I_JMP:     /* off8 */
    expr(&e1);
    outab(op);
    if (mchpcr(&e1, &ofst, 1)) {
        if ((ofst < -128) || (ofst > 127))
            xerr('a', "Short Relative Address Is Out Of Range");
        outab(ofst);
    } else {
        outrb(&e1, R_PCR);
    }
    if (e1.e_mode != S_USER)
        rerr();
    break;
```

The absolute offset is verified to be in the proper range and that the expression is valid.

Add I_JMP to the instruction types in m370.h before the I_TRAP definition.

```
#define I_JMP   57      /* JMP, JN, JEQ, JZ, JC, JP, JPZ, JNE, JNZ */
                        /* JNC, JV, JL, JLE, JHS, JNV, JGE, JG, JLO */
```

And add the JMP mnemonics to the structure in m370pst.c.

```
{   NULL,   "jmp",          I_JMP,          0,      0x00    },
{   NULL,   "jn",           I_JMP,          0,      0x01    },
{   NULL,   "jz",           I_JMP,          0,      0x02    },
{   NULL,   "jc",           I_JMP,          0,      0x03    },
{   NULL,   "jp",           I_JMP,          0,      0x04    },
{   NULL,   "jpz",          I_JMP,          0,      0x05    },
{   NULL,   "jnz",          I_JMP,          0,      0x06    },
{   NULL,   "jnc",          I_JMP,          0,      0x07    },
{   NULL,   "jv",           I_JMP,          0,      0x08    },
{   NULL,   "jl",           I_JMP,          0,      0x09    },
{   NULL,   "jle",          I_JMP,          0,      0x0A    },
{   NULL,   "jhs",          I_JMP,          0,      0x0B    },
{   NULL,   "jnv",          I_JMP,          0,      0x0C    },
{   NULL,   "jge",          I_JMP,          0,      0x0D    },
{   NULL,   "jg",           I_JMP,          0,      0x0E    },
{   NULL,   "jlo",          I_JMP,          0,      0x0F    },

{   NULL,   "jeq",          I_JMP,          0,      0x02    },
{   NULL,   "jne",          I_JMP,          0,      0x06    },
```

Note that JEQ and JZ are equivalent as are JNE and JNZ.

2.9 INHERENT INSTRUCTION

The inherent instructions do not have arguments, they have only an opcode. The TMS370 assembly language also specifies several LDST based instructions which have implicit arguments - DINT, EINT, EINTH, and EINTL.
Add these instruction types to m327.h after the I_JMP definition:

```
#define I_INH   63      /* CLRC, TSTA, IDLE, SETC, RTS, RTI */
                        /* LDSP, STSP, NOP */
#define I_INH2  64      /* DINT, EINT, EINTH, EINTL */
```

Add these mnemonics at the end of the mnemonic structure in m370pst.c:

```
{   NULL,   "clrc",         I_INH,          0,      0xB0    },
{   NULL,   "tsta",         I_INH,          0,      0xB0    },
{   NULL,   "tstb",         I_INH,          0,      0xC6    },
{   NULL,   "idle",         I_INH,          0,      0xF6    },
{   NULL,   "setc",         I_INH,          0,      0xF8    },
{   NULL,   "rts",          I_INH,          0,      0xF9    },
{   NULL,   "rti",          I_INH,          0,      0xFA    },
{   NULL,   "ldsp",         I_INH,          0,      0xFD    },
{   NULL,   "stsp",         I_INH,          0,      0xFE    },
{   NULL,   "nop",          I_INH,          0,      0xFF    },

{   NULL,   "dint",         I_INH2,         0,      0xF000  },
{   NULL,   "eint",         I_INH2,         0,      0xF00C  },
{   NULL,   "einth",        I_INH2,         0,      0xF004  },
{   NULL,   "eintl",        I_INH2,         S_EOL,  0xF008  }
```

Replace the S_EOL in the TRAP mnemonic with a 0 and add the comma after the }.

Update the m370mch.c code by adding these two case statements:

```
/* CLRC, TSTA, IDLE, SETC, RTS, RTI */
/* LDSP, STSP, NOP */
case I_INH:
        outab(op);
        break;

/* DINT, EINT, EINTH, EINTL */
case I_INH2:
        outaw(op);
        break;
```

2.10 INSTRUCTION ADDRESSING

To continue adding instruction mnemonics the TMS370 addressing mode processing must be writen. The processing will be added to the function addr() in the file m370adr.c.

The TMS370 has the following argument types:

```
A       - Register A    Accumulator
B       - Register B    Scratch
SP      - Register SP   Stack Pointer
ST      - Register ST   Status
Rn      - Working Registers 0-255
Pn      - Port Registers 0-255

N(B)    - B  with offset N (or #N)
N(SP)   - SP with offset N (or #N)
N(Rn)   - Rn with offset N (or #N)

@N      - Indirect N
#N      - Immediate Number

Label or Number
```

this will require the assembler to evaluate each of these argument types. Update the addressing mode definitions in m370.h to this:

```
#define S_A     35              /* A */
#define S_B     36              /* B */
#define S_SP    37              /* SP */
#define S_ST    38              /* ST */
#define S_NA    39              /* N(A) - Illegal */
#define S_NB    40              /* N(B) */
#define S_NSP   41              /* N(SP) */
#define S_NST   42              /* N(ST) - Illegal */
#define S_R     43              /* Rn, n = 0-255 */
#define S_P     44              /* Pn, n = 0-255 */
#define S_IR    45              /* @Rn */
#define S_NR    46              /* N(Rn) */
#define S_IMM   47              /* #__ */
#define S_EXT   48              /* Label or Number */
```

Note that modes N(A) and N(ST) are illegal and are never used by the assembler. These modes are simply a byproduct of evaluating the other modes.

The registers A, B, SP, and ST use the admode() function which is used in most ASxxxx assemblers to parse registers. Replace the register entries in the struct adsym reg[] with the following:

```
{   "a",    S_A     },
{   "b",    S_B     },
{   "sp",   S_SP    },
{   "st",   S_ST    },
{   "",     0000    }
```

admode(reg) returns 1 if the argument is a register in the reg list of entries else a zero is returned. The register mode is returned in aindx.

Typically the processor registers are placed in multiple adsym structures. However the TMS370 has two sets of registers each having 256 elements. Constructing tables for these registers takes a lot of space and time doing a search of the adsym structures. Therefor functions are created to parse Rn and Pn registers. Add the following function, rgmode(), for Rn in m370adr.c:

2.11 Rn REGISTERS

```
/*
 *      Register Searching - Rn
 */
int
rgmode(void)
{
        int c, d, n;
        char *p;

        p = ip;
        c = ccase[getnb() & 0x7F];
        d = digit(*ip++, 10);
        if ((c == 'r') && (d >= 0)) {
            n = d;
            while ((d = digit(*ip, 10)) >= 0) {
                n = 10*n + d;
                ip++;
            }
            d = ctype[*ip & 0x7F];
            if ((*ip == 0) || (d == 0) || (d == BINOP)) {
                 aindx = n;
                 if (n > 255) xerr('a', "Only Rn, n=0-255 Allowed");
                 return(1);
            }
        }
        ip = p;
        return(0);
}
```

The function scans the source text line skipping white space and requiring the first found character to be 'r' and the next character to be a digit between 0 and 9. If these two conditions are not met then the parse terminates with the pointer into the source text line restored to its initial value and 0 is return. If the parse continues more digits are scanned to generate the register number. The number conversion terminates on a non digit character. The terminating character must 0, a character type equal to zero (SPACE, TAB, or unary operator) or a binary operator. If these conditions are not met then the parse terminates with the pointer into the source text line restored to its initial value and a 0 is returned. Else the converted number is placed in aindx and a 1 is return. If the register number is greater then 255 then an error message is issued.

Note that the two arrays, ccase[] and ctype[], are ASxxxx core arrays. ccase[] is a 128 character array performing the the translation of upper case to lower case letters. ctype[] is a 128 byte character array that translates a character to its character type - SPACE, ILLEGAL, LETTER, DIGIT, BINOP, etc.

The function ptmode() performs the same function as rgmode() but for Pn registers. The Pn, port, registers are typically allocated 8 or 16 registers for each peripheral in the specific TMS370 chip variation. Addressing these ports is more convenient using hexadecimal notation, thus the hexadecimal mode is enabled if the first digit following the p is 0 with the succeeding digits treated as hexadecimal digits.

Add the following function ptmode() to m370adr.c after the rgmode() function:

2.12 Pn REGISTERS

```
/*
 *      Port Searching - Pn
 *
 *         if first digit is 0 evaluate as HEX
 *         else evaluate as DECIMAL
 */
int
ptmode(void)
{
        int c, d, n, r;
        char *p;

        p = ip;
        c = ccase[getnb() & 0x7F];
        d = digit(*ip++, 10);
        if ((c == 'p') && (d >= 0)) {
            n = d;
            r = (d == 0) ? 16 : 10;
            while ((d = digit(*ip, r)) >= 0) {
                 n = r*n + d;
                 ip++;
            }
            d = ctype[*ip & 0x7F];
            if ((*ip == 0) || (d == 0) || (d == BINOP)) {
                 aindx = n;
                 if (n > 255) xerr('a', "Only Pn, n=0-255 Allowed");
                 return(1);
            }
        }
        ip = p;
        return(0);
}
```

2.13 REGISTER CODING

The three functions, admode(), rgmode(), and ptmode(), are the functions used to parse any argument contaning a TMS370 register.

Add the following code to parse the registers A, B, SP, ST, Rn (n = 0-255), and Pn (n = 0-255 or 0x00-0xFF). Insert the code just after the aindx = 0; statement.

```
if (admode(reg)) {      /* A, B, SP, ST */
    esp->e_mode = aindx;
    return(esp->e_mode);
}
if (rgmode()) {         /* Rn, n = 0-255 */
    esp->e_mode = S_R;
    return(esp->e_mode);
}
if (ptmode()) {         /* Pn, n = 0-255 */
    esp->e_mode = S_P;
    return(esp->e_mode);
}
```

The returned modes will be S_A, S_B, S_SP, S_ST, S_R, or S_P with aindx containing the register number for S_R or S_P.

If the argument is not one of these modes then the parse continues on to the next mode. The next simple mode is @Rn, the indirect mode S_IR. This requires the first character to be '@' and a valid rgmode() parse. Add the following code after the code added above:

```
if (((c = getnb()) == '@') && rgmode()) {       /* @Rn */
    esp->e_mode = S_IR;
    return(esp->e_mode);
}
```

2.14 N(REGISTER) ADDRESSING

The remaining argument modes are N(B), N(SP), and N(R). These modes require source text line manipulation to allow for multiple arguments and argument seperation. Given the following cases:

```
case 1  ---     N(B)
           ^ parse begins here (only one argument)
case 2  ---     N(SP),arg2
           ^ parse begins here (first argument)

case 3  ---     arg1,N(SP)
                     ^ parse begins here (second argument)

case 4  ---     arg1,N(R),arg3
                     ^ parse begins here (second argument)
```

Cases 1 and 3 are equivalent as each is parsing the last argument on the line. Cases 2 and 4 are equivalent as parsing must be aware of the second or third argument and requires unique processing.

1) The first step is to scan forward for a ','. If a comma is found then separate the arguments by terminating the source text by replacing the comma with a 0 and saving the position of the replaced comma.

2) Now perform a reverse search for a '(' indicating this might be of the searched for form. If the '(' is not found then the argument cannot of the form N(B), N(SP), or N(R); restore the ',' if required and proceed to the next addressing mode type check.

3) With the '(' found scan for a register with admode(reg) and verify the next character is ')'. If both conditions are true then this is one of the searched for forms. If not then go to step 5.

4) Save the parsing position (the ip following the verification of ')') and replace the '(' with a 0 to terminate the string. Now evaluate the expression from the beginning of the text to the termination. Select the addressing mode by register, restore the '(', restore the ',' if required, and restore the parsing position to ip. Return the expression evaluation and addressing mode.

5) With the '(' found scan for a register with rgmode() and verify the next character is ')'. If both conditions are true then this is one of the searched for forms. If not then restore the ',' if required and proceed to the next addressing mode type check.

6) Save the parsing position (the ip following the verification of ')') and replace the '(' with a 0 to terminate the string. Now evaluate the expression from the beginning of the text to the termination. Set the addressing mode to S_NR, restore the '(', restore the ',' if required, and restore the parsing position to ip. Return the expression evaluation, register number in aindx, and the addressing mode.

Add the following code to m370addr.c just after the preceeding addressing code.

```
/* N(B), N(SP), N(R) */
ip = p;
q = strchr(ip, ',');
if (q != NULL) *q = 0;
r = strrchr(ip, '(');
if (r != NULL) {
    ip = r + 1;
    if (admode(reg) && (getnb() == ')')) {
        s = ip;
        *r = 0;
        ip = p;
        expr(esp);
        switch(aindx) {
        case S_A:       /* N(A) - Illegal */
            esp->e_mode = S_NA;         break;
        case S_B:       /* N(B) */
            esp->e_mode = S_NB;         break;
        case S_SP:      /* N(SP) */
            esp->e_mode = S_NSP;        break;
        case S_ST:      /* N(ST) - Illegal */
            esp->e_mode = S_NST;        break;
        default:        break;
        }
        if (q != NULL) *q = ',';
        *r = '(';
        ip = s;
        return(esp->e_mode);
    }
    ip = r + 1;
    if (rgmode() && (getnb() == ')')) {
        s = ip;
        *r = 0;
        ip = p;
        expr(esp);
        esp->e_mode = S_NR;     /* N(Rn) */
        if (q != NULL) *q = ',';
        *r = '(';
        ip = s;
        return(esp->e_mode);
    }
}
if (q != NULL) *q = ',';
```

2.15 IMMEDIATE AND LABEL ADDRESSING

The remaining addressing modes, S_IMM and S_EXT, were included in the original xxxadr.c file. These modes are an immediate number, #N, or a label (which could be just a numerical value).

This completes the addressing mode code used by the as370 assembler. What remains is to define all of the remaining instruction types, add the code for each of the instruction types, and add the instruction cycles table to the file m370mch.c.

2.16 TYP1 INSTRUCTIONS

For each of the instruction types a section will be added to the mnemonic structure in m370pst.c and code added to the machine() switch statement.

Add the following to m370pst.c for the first case statement:

```
{   NULL,   "and",          I_TYP1,         0,      0x13    },
{   NULL,   "or",           I_TYP1,         0,      0x14    },
{   NULL,   "xor",          I_TYP1,         0,      0x15    },
{   NULL,   "btjo",         I_TYP1,         0,      0x16    },
{   NULL,   "btjz",         I_TYP1,         0,      0x17    },
```

and the following to m370mch.c as the first case:

```
/* AND, OR, XOR, BTJO, BTJZ */
case I_TYP1:
```

Each instruction has two regular arguments. BTJO and BTJZ have a third argument, the jump address offset. The two or three arguments are processed before any binary output. Add the following code:

```
    t1 = addr(&e1);
    v1 = aindx;
    comma(1);
    t2 = addr(&e2);
    v2 = aindx;
    if ((op == 0x16) || (op == 0x17)) { /* BTJO, BTJZ */
        comma(1);
        expr(&e3);
    }
```

The first two arguments are processed for their addressing modes while the third argument is just evaluated as an expression which may be a label or a number. The value of the opcode from the mnemonic table selects the BTJO and BTJZ instructions and that a third argument is required.

Add the following code which explicitly selects the valid argument pairs supported by these instructions. The if statements are intentionally encased in a switch statement to catch errors and allow the third argument for a BTJO or BTJZ to be processed. Note that the base opcode is updated for each addresing mode.

```
    switch(0) {
    case 0:
        if ((t1 == S_R) && (t2 == S_A)) {       /* Rn,A */
            outab(op + 0x00);
            outab(v1);
            break;
        }
        if ((t1 == S_IMM) && (t2 == S_A)) {     /* #N,A */
            outab(op + 0x10);
            outrb(&e1, R_OVRF);
            break;
        }
        if ((t1 == S_R) && (t2 == S_B)) {       /* Rn,B */
            outab(op + 0x20);
            outab(v1);
            break;
        }
        if ((t1 == S_R) && (t2 == S_R)) {       /* Rn,Rn */
            outab(op + 0x30);
            outab(v1);
            outab(v2);
            break;
        }
        if ((t1 == S_IMM) && (t2 == S_B)) {     /* #N,A */
            outab(op + 0x40);
            outrb(&e1, R_OVRF);
            break;
        }
```

```
        if ((t1 == S_B) && (t2 == S_A)) {       /* B,A */
            outab(op + 0x50);
            break;
        }
        if ((t1 == S_IMM) && (t2 == S_R)) {     /* #N,Rn */
            outab(op + 0x60);
            outrb(&e1, R_OVRF);
            outab(v2);
            break;
        }
        if ((t1 == S_A) && (t2 == S_P)) {       /* A,Pn */
            outab(op + 0x70);
            outab(v2);
            break;
        }
        if ((t1 == S_B) && (t2 == S_P)) {       /* B,Pn */
            outab(op + 0x80);
            outab(v2);
            break;
        }
        if ((t1 == S_IMM) && (t2 == S_P)) {     /* #N,Pn */
            outab(op + 0x90);
            outrb(&e1, R_OVRF);
            outab(v2);
            break;
        }
        aerr();
        break;

    default:    aerr();         break;
    }
```

The BTJO and BTJZ have a program counter relative offset output from the third argument. This code is equivalent to that used for the JMP instructions. The code for TYP1 instructions is complete with this addition.

```
    if ((op == 0x16) || (op == 0x17)) { /* BTJO, BTJZ */
      if (mchpcr(&e3, &ofst, 1)) {
        if ((ofst < -128) || (ofst > 127))
          xerr('a', "Short Relative Address Is Out Of Range");
        outab(ofst);
      } else {
        outrb(&e3, R_PCR);
      }
      if (e3.e_mode != S_USER)
        rerr();
    }
    break;
```

2.17 TYP2 INSTRUCTIONS

Add the following to m370pst.c for the second case statement:

```
{   NULL,   "add",          I_TYP2,         0,      0x18    },
{   NULL,   "adc",          I_TYP2,         0,      0x19    },
{   NULL,   "sub",          I_TYP2,         0,      0x1A    },
{   NULL,   "sbb",          I_TYP2,         0,      0x1B    },
{   NULL,   "mpy",          I_TYP2,         0,      0x1C    },
{   NULL,   "cmp",          I_TYP2,         0,      0x1D    },
{   NULL,   "dac",          I_TYP2,         0,      0x1E    },
{   NULL,   "dsb",          I_TYP2,         0,      0x1F    },
```

and the following to m370mch.c as the second case:

```
/* ADD, ADC, DUB, SBB, MPY, CMP, DAC, DSB */
case I_TYP2:
    t1 = addr(&e1);
    v1 = aindx;
    comma(1);
    t2 = addr(&e2);
    v2 = aindx;
```

Each instruction has 7 valid argument pairs. However the CMP instruction has 6 additional valid argument pairs. The common argument pairs are processed by the following if statements:

```
    if ((t1 == S_R) && (t2 == S_A)) {   /* Rn,A */
        outab(op + 0x00);
        outab(v1);
        break;
    }
    if ((t1 == S_IMM) && (t2 == S_A)) { /* #N,A */
        outab(op + 0x10);
        outrb(&e1, R_OVRF);
        break;
    }
    if ((t1 == S_R) && (t2 == S_B)) {   /* Rn,B */
        outab(op + 0x20);
        outab(v1);
        break;
    }
    if ((t1 == S_R) && (t2 == S_R)) {   /* Rn,Rn */
        outab(op + 0x30);
        outab(v1);
        outab(v2);
        break;
    }
    if ((t1 == S_IMM) && (t2 == S_B)) { /* #N,B */
        outab(op + 0x40);
        outrb(&e1, R_OVRF);
        break;
    }
    if ((t1 == S_B) && (t2 == S_A)) {   /* B,A */
        outab(op + 0x50);
        break;
    }
    if ((t1 == S_IMM) && (t2 == S_R)) { /* #N,Rn */
        outab(op + 0x60);
        outrb(&e1, R_OVRF);
        outab(v2);
        break;
    }
```

The unique CMP argument pairs are processed by this additional code:

```
    if (op == 0x1D) {   /* CMP ---,--- */
        if ((t1 == S_EXT) && (t2 == S_A)) {     /* N,A */
            outab(op + 0x70);
            outrw(&e1, 0);
            break;
        }
        if ((t1 == S_IR) && (t2 == S_A)) {      /* (Rn),A */
            outab(op + 0x80);
            outab(v1);
            break;
        }
        if ((t1 == S_NB) && (t2 == S_A)) {      /* N(B),A */
            outab(op + 0x90);
            outrw(&e1,0);
            break;
        }
        if ((t1 == S_NSP) && (t2 == S_A)) {     /* N(SP),A */
            outab(0xF3);
            outrb(&e1, R_OVRF);
            break;
        }
        if ((t1 == S_IR) && (t2 == S_A)) {      /* @Rn,A */
            outaw(0xF4ED);      opcycles = 18;
            outab(v1);  break;
        }
        if ((t1 == S_NR) && (t2 == S_A)) {      /* N(Rn),A */
            outaw(0xF4ED);      opcycles = 18;
            outrb(&e1, R_OVRF);
            outab(v1);  break;
        }
    }
    aerr();
    break;
```

All other argument pairs will report an addressing error. All instructions using the 0xF4 prefix also have their cycle count explicitly set as the count cannot be included in a standard 256 byte cycle count table.

2.18 TYP3 INSTRUCTIONS

The 14 TYP3 instructions each have 3 argument types. The PUSH and POP instructions have one additional mode. The DJNZ instruction has a second argument, the jmp address.
Add the following to m370pst.c for the TYP3 case:

```
{   NULL,   "dec",          I_TYP3,         0,      0xB2    },
{   NULL,   "inc",          I_TYP3,         0,      0xB3    },
{   NULL,   "inv",          I_TYP3,         0,      0xB4    },
{   NULL,   "clr",          I_TYP3,         0,      0xB5    },
{   NULL,   "xchb",         I_TYP3,         0,      0xB6    },
{   NULL,   "swap",         I_TYP3,         0,      0xB7    },
{   NULL,   "push",         I_TYP3,         0,      0xB8    },
{   NULL,   "pop",          I_TYP3,         0,      0xB9    },
{   NULL,   "djnz",         I_TYP3,         0,      0xBA    },
{   NULL,   "compl",        I_TYP3,         0,      0xBB    },
{   NULL,   "rr",           I_TYP3,         0,      0xBC    },
{   NULL,   "rrc",          I_TYP3,         0,      0xBD    },
{   NULL,   "rl",           I_TYP3,         0,      0xBE    },
{   NULL,   "rlc",          I_TYP3,         0,      0xBF    },
```

and the following to m370mch.c as the TYP3 case:

```
/* DEC, INC, INV, CLR, XCHB, SWAP, PUSH */
/* POP, DJNZ, COMPL, RR, RRC, RL, RLC */
case I_TYP3:
    t1 = addr(&e1);
    v1 = aindx;
    if (op == 0xBA) {   /* DJNZ */
        comma(1);
        expr(&e2);
    }
    switch(t1) {
    case S_A:   outab(op + 0x00);       break;
    case S_B:   outab(op + 0x10);       break;
    case S_ST:
        if (op == 0xB8) {       /* PUSH ST */
                op = 0xFB;
                outab(op);      break;
        }
        if (op == 0xB9) {       /* POP ST */
                op = 0xFC;
                outab(op);      break;
        }
        xerr('a', "Only PUSH ST and POP ST are Legal");
        break;
    case S_R:   outab(op + 0x20);
                outab(v1);      break;
    default:    aerr();         break;
    }
    if (op == 0xBA) {   /* DJNZ */
      if (mchpcr(&e2, &ofst, 1)) {
        if ((ofst < -128) || (ofst > 127))
          xerr('a', "Short Relative Address Is Out Of Range");
        outab(ofst);
      } else {
        outrb(&e2, R_PCR);
      }
      if (e2.e_mode != S_USER)
        rerr();
      break;
    }
    break;
```

2.19 THE MOV INSTRUCTION

The MOV instruction has 27 variations of unique argument pairs. To simplify the processing the instruction was organized into 6 groupings: A the first argument, A the second argument, B the first argument, B the second argument, Rn the second argument, and Pn the second argument. No argument pairs are duplicated. The instruction opcodes are explicitly output for each argument pair.

Add the following to m370pst.c for the MOV instruction:

```
{   NULL,   "mov",          I_MOV,          0,      0x12    },
```

and the following to m370mch.c as the MOV case:

```
/* MOV */
case I_MOV:
    t1 = addr(&e1);
    v1 = aindx;
    comma(1);
    t2 = addr(&e2);
    v2 = aindx;
    if (t1 == S_A) {
        if (t2 == S_B) {        /* A,B */
            outab(0xC0);        break;
        }
        if (t2 == S_R) {        /* A,Rn */
            outab(0xD0);
            outab(v2);  break;
        }
        if (t2 == S_P) {        /* A,Pn */
            outab(0x21);
            outab(v2);  break;
        }
        if (t2 == S_IR) {       /* A,@Rn */
            outab(0x9B);
            outab(v2);  break;
        }
        if (t2 == S_EXT) {      /* A,label */
            outab(0x8B);
            outrw(&e2, 0);      break;
        }
        if (t2 == S_NB) {       /* A,label(B) */
            outab(0xAB);
            outrw(&e2, 0);      break;
        }
        if (t2 == S_NSP) {      /* A,N(SP) */
            outab(0xF2);
            outrb(&e2, R_SGND); break;
        }
        if (t2 == S_NR) {       /* A,N(Rn) */
            outaw(0xF4EB);      opcycles = 16;
            outrb(&e2, R_SGND);
            outab(v2);  break;
        }
        aerr();
        break;
    }
```

```
    if (t2 == S_A) {
        if (t1 == S_B) {          /* B,A */
            outab(0x62);          break;
        }
        if (t1 == S_R) {          /* Rn,A */
            outab(0x12);
            outab(v1);  break;
        }
        if (t1 == S_IMM) {        /* #N,A */
            outab(0x22);
            outrb(&e1, R_SGND); break;
        }
        if (t1 == S_P) {          /* Pn,A */
            outab(0x80);
            outab(v1);  break;
        }
        if (t1 == S_IR) {         /* @Rn,A */
            outab(0x9A);
            outab(v1);  break;
        }
        if (t1 == S_EXT) {        /* label,A */
            outab(0x8A);
            outrw(&e1, 0);        break;
        }
        if (t1 == S_NB) {         /* label(B),A */
            outab(0xAA);
            outrw(&e1, 0);        break;
        }
        if (t1 == S_NSP) {        /* off8(SP),A */
            outab(0xF1);
            outrb(&e1, R_SGND); break;
        }
        if (t1 == S_NR) {         /* off8(Rn),A */
            outaw(0xF4EA);        opcycles = 17;
            outrb(&e1, R_OVRF);
            outab(v1);  break;
        }
        aerr();
        break;
    }
```

```
    if (t1 == S_B) {
        if (t2 == S_R) {        /* B,Rn */
            outab(0xD1);
            outab(v2);  break;
        }
        if (t2 == S_P) {        /* B,Pn */
            outab(0x51);
            outab(v2);  break;
        }
        aerr();
        break;
    }
    if (t2 == S_B) {
        if (t1 == S_R) {        /* Rn,B */
            outab(0x32);
            outab(v1);  break;
        }
        if (t1 == S_IMM) {      /* #N,B */
            outab(0x52);
            outrb(&e1, R_OVRF); break;
        }
        if (t1 == S_P) {        /* Pn,B */
            outab(0x91);
            outab(v1);  break;
        }
        aerr();
        break;
    }
    if (t2 == S_R) {
        if (t1 == S_R) {        /* Rn,Rn */
            outab(0x42);
            outab(v1);  outab(v2);      break;
        }
        if (t1 == S_IMM) {      /* #N,Rn */
            outab(0x72);
            outrb(&e1, R_OVRF);
            outab(v2);  break;
        }
        if (t1 == S_P) {        /* Pn,Rn */
            outab(0xA2);        /* Reverse Arguments */
            outab(v2);  outab(v1);      break;
        }
        aerr();
        break;
    }
```

```
    if (t2 == S_P) {
        if (t1 == S_R) {        /* Rn,Pn */
            outab(0x71);        /* Reverse Arguments */
            outab(v2);  outab(v1);      break;
        }
        if (t1 == S_IMM) {      /* #N,Pn */
            outab(0xF7);
            outrb(&e1, R_OVRF);
            outab(v2);  break;
        }
        aerr();
        break;
    }
    aerr();
    break;
```

All other argument pairs will result in an addressing error.

2.20 THE MOVW INSTRUCTION

The MOVW instruction operates on 16 bit arguments. Add the following code for the case IMOVW:
Add the following to m370pst.c for the MOVW instruction:

```
{   NULL,   "movw",         I_MOVW,         0,      0x88    },
```

and the following to m370mch.c as the MOVW case:

```
/* MOVW */
case I_MOVW:
    t1 = addr(&e1);
    v1 = aindx;
    comma(1);
    t2 = addr(&e2);
    v2 = aindx;
    if (t2 == S_R) {
        switch(t1) {
        case S_IMM:     /* MOVW #N,Rn */
            outab(op + 0x00);
            outrw(&e1, 0);
            outab(v2);
            break;
        case S_R:       /* MOVW Rn,Rn */
            outab(op + 0x10);
            outab(v1);
            outab(v2);
            break;
        case S_NB:      /* MOVW #N(B),Rn */
            outab(op + 0x20);
            outrw(&e1, 0);
            outab(v2);
            break;
        case S_NR:      /* MOVW #N(Rn),Rn */
            outaw(0xF4E8);      opcycles = 20;
            outrb(&e1, R_OVRF); outab(v1);
            outab(v2);
            break;
        default:
            aerr();
            break;
        }
        break;
    }
    aerr();
    break;
```

The MOVW instruction only supports 4 argument pairs, all other pairs will result in an addressing error.

2.21 THE DIV AND INCW INSTRUCTIONS

The DIV and INCW instructions only support a single argument pair. Add the following to m370pst.c for the DIV and INCW instructions:

```
{   NULL,   "div",          I_DIV,          0,      0xF4F8  },

{   NULL,   "incw",         I_INCW,         0,      0x70    },
```

and the following to m370mch.c as the DIV and INCW cases:

```
/* DIV */
case I_DIV:     /* Rn,A */
    t1 = addr(&e1);
    v1 = aindx;
    comma(1);
    t2 = addr(&e2);
    v2 = aindx;
    if ((t1 == S_R) && (t2 == S_A)) {
        outaw(op);      opcycles = 63;
        outab(v1);
        break;
    }
    aerr();
    break;

/* INCW */
case I_INCW:    /* #N,Rn */
    t1 = addr(&e1);
    v1 = aindx;
    comma(1);
    t2 = addr(&e2);
    v2 = aindx;
    if ((t1 == S_IMM) && (t2 == S_R)) {
        outab(op);
        outrb(&e1, R_OVRF);
        outab(v2);
        break;
    }
    aerr();
    break;
```

2.22 CALL AND BR INSTRUCTIONS

The CALL and BR instructions access the full addressing range of the TMS370 processor. CALL places a return address on the stack before making a jump whereas BR simply jumps to the destination address.

Add the following to m370pst.c for the CALL and BR instructions:

```
{   NULL,   "call",         I_CALL,         0,      0x8E    },
{   NULL,   "br",           I_CALL,         0,      0x8C    },
```

and the following to m370mch.c as the I_CALL case:

```
/* CALL, BR */
case I_CALL:
    t1 = addr(&e1);
    v1 = aindx;
    switch(t1) {
    case S_EXT: /* Label */
        outab(op + 0x00);
        outrw(&e1, 0);
        break;
    case S_IR:  /* @Rn */
        outab(op + 0x10);
        outab(v1);
        break;
    case S_NB:  /* Label(B) */
        outab(op + 0x20);
        outrw(&e1, 0);
        break;
    case S_NR:  /* off8(Rn) */
        if (op == 0x8C) {       /* BR */
            outaw(0xF4EC);      opcycles = 16;
        }
        if (op == 0x8E) {       /* CALL */
            outaw(0xF4EE);      opcycles = 20;
        }
        outrb(&e1, R_SGND);
        outab(v1);
        break;
    default:
        aerr();
        break;
    }
    break;
```

2.23 CALLR AND JMPL INSTRUCTIONS

The CALLR and JMPL instructions access the full addressing range of the TMS370 processor. CALLR places a return address on the stack before making a jump whereas JMPL simply jumps to the destination address. These instructions use 16 bit program counter relative addressing. The assembler or linker will calculate the offset from the address of the first byte following the instruction.

Add the following to m370pst.c for the CALLR and JMPL instructions:

```
{   NULL,   "callr",        I_CALLR,        0,      0x8F    },
{   NULL,   "jmpl",         I_CALLR,        0,      0x89    },
```

and the following to m370mch.c as the I_CALLR case:

```
/* CALLR, JMPL */
case I_CALLR:
    t1 = addr(&e1);
    v1 = aindx;
    switch(t1) {
    case S_EXT: /* Label */
    case S_NB:  /* Label(B) */
        if (t1 == S_EXT) outab(op + 0x00);
        if (t1 == S_NB)  outab(op + 0x20);
        if (mchpcr(&e1, &ofst, 2)) {
            outaw(ofst);
        } else {
            outrw(&e1, R_PCR);
        }
        break;
    case S_IR:  /* @Rn */
        outab(op + 0x10);
        outab(v1);
        break;
    case S_NR:  /* off8(Rn) */
        switch(op) {
        case 0x89:      outaw(0xF4E9);  /* JMPL */
                        opcycles = 16;   break;
        case 0x8F:      outaw(0xF4EF);  /* CALLR */
                        opcycles = 22;  break;
        default:        break;
        }
        outrb(&e1, R_SGND);
        outab(v1);
        break;
    default:
        aerr();
        break;
    }
    break;
```

2.24 THE LDST INSTRUCTION

The LDST instruction has a single addressing mode argument, an 8 bit constant. Add the following to m370pst.c for the LDST instruction after the I_TRAP code:

```
{   NULL,   "ldst",         I_LDST,         0,      0xF0    },
```

and the following to m370mch.c as the I_LDST case:

```
/* LDST */
case I_LDST:
    expr(&e1);
    outab(op);
    outrb(&e1, R_USGN);
    break;
```

2.25 THE BITS INSTRUCTIONS

The final type is I_BITS. These are assembler constructed instructions operating on individual bits within an 8 bit argument. These instructions actually use the existing instructions XOR, AND, OR, BTJZ, and BTJO:

```
CMPBIT  BITN,Rn          ->      XOR     #iop8,Rn
CMPBIT  BITN,Pn          ->      XOR     #iop8,Pn
                where #iop8 =   1 << BITN,   BITN = 0-7

SBIT0   BITN,Rn          ->      AND     #iop8,Rn
SBIT0   BITN,Pn          ->      AND     #iop8,Pn
                where #iop8 = ~(1 << BITN),  BITN = 0-7

SBIT1   BITN,Rn          ->      OR      #iop8,Rn
SBIT1   BITN,Pn          ->      OR      #iop8,Pn
                where #iop8 =   1 << BITN,   BITN = 0-7

JBIT0   BITN,Rn,label    ->      BTJZ    #iop8,Rn,label
JBIT0   BITN,Pn,label    ->      BTJZ    #iop8,Pn,label
                where #iop8 =   1 << BITN,   BITN = 0-7

JBIT1   BITN,Rn,label    ->      BTJO    #iop8,Rn,label
JBIT1   BITN,Pn,label    ->      BTJO    #iop8,Pn,label
                where #iop8 =   1 << BITN,   BITN = 0-7
```

The conversion from a bit number to an 8 bit value required by the instructions is performed by the expression analysis or by the linker. The conversion should be compatible with the use of a global defined bit value or expression.

Add the following to m370pst.c for the I_BITS instructions:

```
{   NULL,   "sbit0",        I_BITS,         0,      0x03    },
{   NULL,   "sbit1",        I_BITS,         0,      0x04    },
{   NULL,   "cmpbit",       I_BITS,         0,      0x05    },
{   NULL,   "jbit1",        I_BITS,         0,      0x06    },
{   NULL,   "jbit0",        I_BITS,         0,      0x07    },
```

and the following to m370pst.c for the I_BITS case:

```
/* SBIT0, SBIT1, CMPBIT  #iop8,Rn / #iop8,Pn */
/* JBIT1, JBIT0  #iop8,Rn,label / #iop8,Pn,label */
case I_BITS:
    /* Make Argument Globally Compatible */
    if (op == 0x03) {   /* SBIT0 */
        exprps("~(1 << ((", &e1, ") & 7))");
    } else {            /* SBIT1, CMPBIT, JBIT0, JBIT1 */
        exprps("1 << ((", &e1, ") & 7)");
    }
    comma(1);
    t2 = addr(&e2);
    v2 = aindx;
    /* JBIT0, JBIT1 */
    if ((op == 0x06) || (op == 0x07)) {
        comma(1);
        expr(&e3);
    }
    switch(t2) {
    case S_R:   outab(op | 0x70);       break;
    case S_P:   outab(op | 0xA0);       break;
    default:    aerr();                 break;
    }
    outrb(&e1, 0);
    outab(v2);
    /* JBIT0, JBIT1 */
    if ((op == 0x06) || (op == 0x07)) {
        if (mchpcr(&e3, &ofst, 1)) {
            outab(ofst);
        } else {
            outrb(&e3, R_PCR);
        }
    }
    break;
```

2.26 CYCLE COUNTS

To complete the as370 assembler the cycle count array must be built from the opcode/instruction map. For each opcode element the number of cycles for that opcode is placed in the table. Unused opcodes are filled with the symbol UN. Note that the opcode at 0xF4 is also filled with UN. Those instructions with the 0xF4 opcode prefix explicitly set the value of opcycles for the instruction. Replace xxxpg[] references with tms370[] and replace the xxxpg[] array with the following array:

```
/*
 * tms370 Cycle Count
 *
 *      opcycles = tms370[opcode]
 */
static char tms370[256] = {
/*--*--* 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F */
/*--*--* -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  - */
/*00*/   9, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
/*10*/  UN,UN, 7, 7, 7, 7,11,11, 7, 7, 7, 7,46, 7, 9, 9,
/*20*/  UN, 8, 6, 6, 6, 6,10,10, 6, 6, 6, 6,45, 6, 8, 8,
/*30*/  UN,UN, 7, 7, 7, 7,11,11, 7, 7, 7, 7,46, 7, 9, 9,
/*40*/  UN,UN, 9, 9, 9, 9,13,13, 9, 9, 9, 9,48, 9,11,11,
/*50*/  UN, 8, 6, 6, 6, 6,10,10, 6, 6, 6, 6,45, 6, 8, 8,
/*60*/  UN,UN, 8, 8, 8, 8,12,12, 8, 8, 8, 8,47, 8,10,10,
/*70*/  11,10, 8, 8, 8, 8,12,12, 8, 8, 8, 8,47, 8,10,10,
/*80*/   8,UN,UN, 9, 9, 9,13,12,13, 9,10,10,11,11,13,15,
/*90*/  UN, 7,UN, 9, 9, 9,12,12,12, 8, 9, 9,10,10,12,14,
/*A0*/  UN,UN,10,10,10,10,13,13,15,10,12,12,13,13,15,17,
/*B0*/   9,UN, 8, 8, 8, 8,10,11, 9, 9,12, 8, 8, 8, 8, 8,
/*C0*/   9,UN, 8, 8, 8, 8,10,11, 9, 9,12, 8, 8, 8, 8, 8,
/*D0*/   7, 7, 6, 6, 6, 6, 8, 9, 7, 7,10,10, 6, 6, 6, 6,
/*E0*/  14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,
/*F0*/   6, 7, 7, 8,UN,UN, 6,10, 7, 9,12, 8, 8, 7, 8, 7
};
```

2.27 ASSEMBLER VERIFICATION

To verify the correct operation of the assembler a comprehensive test file should be created which tests every insruction (or an instruction type) and all addressing modes with each instruction type. The method that seems to work the best is to hand code each instruction from the documentation and compare with with the assembler output. Once this process has been completed with arguments that are NOT global references then modify your arguments to include global references and verify again. You can verify for the same result if the global reference has a value of zero.

This completes the creation of the as370 assembler. For details in writing assembler code for this processor refer to the Texas Instruments TMS370 documentation.

CHAPTER 3

ASXXXX STEP BY STEP EVALUATION

3.1 GETTING STARTED

The 'Step By Step' process includes references to many of the ASxxxx core subroutines that analyze the text of the source lines during the assembly process. This section will describe many of these functions so that they may be used when creating an ASxxxx assembler.

The functions are loosely organized into four sections: strings, symbols, errors, and expressions.

3.2 STRING FUNCTIONS

The 'string' functions perform various functions for scanning the source code text input to the assemblers. All functions use the global pointer, char *ip, as the pointer into the assembler-source text line in ib[].

Note that references to constant NCPS, currently defined in asxxxx.h as having a value of 256, is a default length.

3.2.1 comma()

```
/*)Function     int     comma(flag)
 *
 *      int     flag    when flag is non zero a 'q' error is
 *                      generated if a COMMA is not found.
 *
 *      The function comma() skips SPACEs and TABs and returns
 *      a '1' if the next character is a COMMA else a '0' is
 *      returned.  If a COMMA is not found and flag is non zero
 *      then a 'q' error is reported.
 */
```

3.2.2 skipcomma()

```
/*)Function     int     skpcomma(void)
 *
 *      The function skpcomma() checks for and then skips an
 *      immediate COMMA.  The function returns '1' if a COMMA
 *      was found else a '0' is returned.
 */
```

3.2.3 endline()

```
/*)Function     char    endline(void)
 *
 *      The function endline() scans the assembler-source text
 *      line skipping white space (SPACES and TABS) and returns
 *      the next character or a (0) if the end of the line or
 *      a comment delimiter (;) is found.
 */
```

3.2.4 get()

```
/*)Function     int     get(void)
 *
 *      The function get() returns the next character in the
 *      assembler-source text line, at the end of the line a
 *      NULL character is returned.
 */
```

3.2.5 getnb()

```
/*)Function     int     getnb(void)
 *
 *      The function getnb() scans the current assembler
 *      source text line returning the first character not
 *      a SPACE or TAB.
 */
```

3.2.6 unget()

```
/*)Function     void    unget(c)
 *
 *              int     c       value of last character read
 *                              from assembler-source text line
 *
 *      If (c) is not a NULL character then the global pointer
 *      ip is updated to point to the preceeding character in
 *      the assembler-source text line.
 *
 *      NOTE:   This function does not push the character (c)
 *              back into the assembler-source text line, only
 *              the pointer ip is changed.
 */
```

3.2.7 getmap()

```
/*)Function     int     getmap(d)
 *
 *              int     d       value to compare with the
 *                              assembler-source text
 *                              line character
 *
 *      The function getmap() converts the 'C' style characters
 *      \b, \f, \n, \r, and \t to their equivalent ascii values
 *      and also converts 'C' style octal constants '\123'
 *      to their equivalent numeric values.  If the first
 *      character is equivalent to (d) then a (-1) is returned,
 *      if the end of the line is detected then a 'q' error
 *      terminates the parse for this line, or if the first
 *      character is not a \ then the character value
 *      is returned.
 */
```

3.2.8 more()

```
/*)Function     int     more(void)
 *
 *      The function more() scans the assembler-source text
 *      line skipping white space (SPACES and TABS) and returns
 *      a (0) if the end of the line or a comment delimeter (;)
 *      is found, or a (1) if their are additional characters
 *      in the line.
 */
```

3.2.9 getid()

```
/*)Function     void    getid(id, c)
 *
 *              char *  id      a pointer to a string of
 *                              maximum length NCPS-1
 *              int     c       mode flag
 *                              >=0  this is first character to
 *                                   copy to the string buffer
 *                              <0   skip white space, first
 *                                   character must be a LETTER
 *
 *      The function getid() scans the current assembler-source
 *      text line from the current position copying the next
 *      LETTER | DIGIT string into the external string buffer
 *      id[].  The string ends when a non LETTER or DIGIT
 *      character is found. The maximum number of characters
 *      copied is NCPS-1.  If the input string is larger than
 *      NCPS-1 characters then the string is truncated.  The
 *      string is always NULL terminated.  If the mode argument
 *      (c) is >=0 then (c) is the first character copied to
 *      the string buffer, if (c) is <0 then intervening white
 *      space (SPACES and TABS) are skipped and the first
 *      character found must be a LETTER else a 'q' error
 *      terminates the parse of this assembler-source text line.
 */
```

3.2.10 getst()

```
/*)Function     void    getst(id, c)
 *
 *              char *  id      a pointer to a string of
 *                              maximum length NCPS-1
 *              int     c       mode flag
 *                              >=0  this is first character to
 *                                   copy to the string buffer
 *                              <0   skip white space, first
 *                                   character must be a LETTER
 *
 *      The function getst() scans the current assembler-source
 *      text line from the current position copying the next
 *      character string into the external string buffer (id).
 *      The string ends when a SPACE or ILL character is found.
 *      The maximum number of characters copied is NCPS-1.
 *      If the input string is larger than NCPS-1 characters
 *      then the string is truncated.  The string is always
 *      NULL terminated.  If the mode argument (c) is >=0 then
 *      (c) is the first character copied to the string buffer,
 *      if (c) is <0 then intervening white space (SPACES and
 *      TABS) are skipped and the first character found must be
 *      a LETTER else a 'q' error terminates the parse of this
 *      assembler-source text line.
 */
```

3.3 SYMBOL FUNCTIONS

The functions used to lookup and create named symbols are described in this section. Also a function to allocate memory and save a string is described.

3.3.1 slookup()

```
/*)Function     sym *   slookup(id)
 *
 *              char *  id      symbol name string
 *
 *      The function slookup() searches the symbol hash tables
 *      for a symbol name match returning a pointer to the sym
 *      structure else it returns a NULL.
 */
```

3.3.2 lookup()

```
/*)Function     sym *   lookup(id)
 *
 *              char *  id      symbol name string
 *
 *      The function lookup() searches the symbol hash tables
 *      for a symbol name match returning a pointer to the sym
 *      structure.  If the symbol is not found then a
 *      sym structure is created, initialized, and linked to
 *      the appropriate hash table.  A pointer to this new
 *      sym structure is returned.
 */
```

3.3.3 symeq()

```
/*)Function     int     symeq(p1, p2, flag)
 *
 *              int     flag    case sensitive flag
 *              char *  p1      name string
 *              char *  p2      name string
 *
 *      The function symeq() compares the two name strings
 *      for a match.  The return value is 1 for a match and
 *      0 for no match.
 *
 *              flag == 0       case sensitive compare
 *              flag != 0       case insensitive compare
 */
```

3.3.4 strsto()

```
/*)Function     char *  strsto(str)
 *
 *              char *  str     pointer to string to save
 *
 *      Allocate space for "str", copy str into new space.
 *      Return a pointer to the allocated string.
 */
```

3.4 ERROR FUNCTIONS

3.4.1 err()

```
/*)Function     void    err(c)
 *
 *              int     c       error type character
 *
 *      The legacy function err() reports errors using
 *      the default error descriptions by calling xerr()
 *      with a NULL string.
 */
```

3.4.2 xerr()

```
/*)Function     void    xerr(c, str)
 *
 *              int     c       error type character
 *              char *  str     the error message string
 *
 *      The function xerr() logs the error code character
 *      suppressing duplicate errors.  If the error code
 *      is 'q' then the parse of the current assembler-source
 *      text line is terminated.
 */
```

3.4.3 aerr(), qerr(), and rerr()

```
/*)Functions:   void    aerr(void)
 *              void    qerr(void)
 *              void    rerr(void)
 *
 *      The functions aerr(), qerr(), and rerr() report their
 *      respective error type.  These are included only for
 *      convenience.
 */
```

3.5 EXPRESSION FUNCTIONS

The expression functions are the main entities used when creating a new assembler. These functions process arithmetic and logical expressions, check expression types, change expression types, and convert numerical strings to internal binary values.

3.5.1 digit()

```
/*)Function     int     digit(c, r)
 *
 *              int     c       digit character
 *              int     r       current radix
 *
 *      The function digit() returns the value of c
 *      in the current radix r.  If the c value is not
 *      a number of the current radix then a -1 is returned.
 */
```

3.5.2 absexpr()

```
/*)Function     a_uint  absexpr()
 *
 *      The function absexpr() evaluates an expression,
 *      verifies it is absolute (i.e. not position dependent
 *      or relocatable), and returns its value.
 */
```

3.5.3 expr()

```
/*)Function     void    expr(esp)
 *
 *              expr *  esp     pointer to an expr structure
 *
 *      The function expr() initializes variables, calls
 *      the evaluation function exprx(), and may call the
 *      complex argument exprscan() function.  The evaluation
 *      stores its value and relocation information into
 *      the expr structure supplied by the user.
 */
```

3.5.4 exprx()

```
/*)Function     void    exprx(esp, n)
 *
 *              expr *  esp     pointer to an expr structure
 *              int     n       a firewall priority
 *
 *      The function exprx() evaluates an expression and
 *      stores its value and relocation information into
 *      the expr structure supplied by the user. This
 *      function should only be called after initialization
 *      from expr() or recursively during exprx() processing.
 */
```

3.5.5 clrexpr()

```
/*)Function     void    clrexpr(esp)
 *
 *              expr *  esp     pointer to expression structure
 *
 *      The function clrexpr() clears the expression structure.
 */
```

3.5.6 is_abs()

```
/*)Function     int     is_abs(esp)
 *
 *              expr *  esp     pointer to an expr structure
 *
 *      The function is_abs() tests the evaluation of an
 *      expression to verify it is absolute.  If the evaluation
 *      is absolute then 1 is returned, else 0 is returned.
 *
 *      Note:   The area type (i.e. ABS) is not checked because
 *              the linker can be told to explicitly relocate
 *              an absolute area.
 */
```

3.5.7 abscheck()

```
/*)Function     void    abscheck(esp)
 *
 *              expr *  esp     pointer to an expr structure
 *
 *      The function abscheck() tests the evaluation of an
 *      expression to verify it is absolute.  If the evaluation
 *      is relocatable then an 'r' error is noted and the
 *      expression made absolute.
 */
```

3.5.8 rngchk()

```
/*)Function     a_uint  rngchk(n)
 *
 *              a_uint  n       a signed /unsigned value
 *
 *      The function rngchk() verifies that the
 *      value of n is a signed or unsigned value
 *      within the range of the current exprmasks()
 *      settings and returns the value masked to
 *      the current exprmasks() settings.
 */
```

3.5.9 exprmasks()

```
/*)Function     void    exprmasks(n)
 *
 *              int     n       T Line Bytes in Address
 *
 *      The function exprmasks() configures the assembler
 *      for 16, 24, or 32-Bit Data/Addresses.
 */
```

CHAPTER 4

ASXXXX STEP BY STEP OUTPUT

4.1 GETTING STARTED

The 'Step By Step' process includes references to many of the ASxxxx core subroutines that output binary data from the assembly process. This section will describe all of these functions so that they may be used when creating an ASxxxx assembler.

4.2 ABSOLUTE OUTPUT

The absolute functions are used to output values that are constants. Subroutines for multiples of a byte are provided to match various sizes of outputs from 1 to 4 bytes.

Output an absolute value:

```
void    outab(v)                /* output bits <7:0> */
void    outaw(v)                /* output bits <15:0> */
void    outa3b(v)               /* output bits <23:0> */
void    outa4b(v)               /* output bits <31:0> */

argument:
        a_uint v                unsigned integer
```

4.3 RELOCATABLE OUTPUT

The relocatable functions are used to output values that may not be constants. These functions automatically check for an absolute value before outputting relocation information to the linker. Subroutines for multiples of a byte are provided to match various sizes of outputs from 1 to 4 bytes. These functions are in general used when addresses or values are global, ie not in the current assembly file or are not in the same current assembly area.

Output a relocatable value:

```
void    outrb(esp, r)           /* output bits <7:0> */
void    outrw(esp, r)           /* output bits <15:0> */
void    outr3b(esp, r)          /* output bits <23:0> */
void    outr4b(esp, r)          /* output bits <31:0> */

arguments:
        int r                   relocation mode
        struct expr *esp        expression evaluation
```

The relocation modes most invoked by the user part of the ASxxxx assemblers are R_SGND, R_USGN, R_OVRF, R_PAG0, or R_PAGN. These and others are detailed in the chapter describing the ASxxxx internals.

4.4 MERGE MODE OUTPUT

The merge mode functions are used to output values that require merging with a constant. These functions automatically check for an absolute value before outputing relocation and merging information to the linker. Values that are constants will be merged by the assembler. Subroutines for multiples of a byte are provided to match various sizes of outputs from 1 to 4 bytes. These functions are in general used when addresses or values are part of an opcode and their values are global, ie not in the current assembly file or are not in the same current assembly area.

Output a relocatable value merged with a constant:

```
void    outrbm(esp, r, v)       /* output bits <7:0> */
void    outrwm(esp, r, v)       /* output bits <15:0> */
void    outr3bm(esp, r, v)      /* output bits <23:0> */
void    outr4bm(esp, r, v)      /* output bits <31:0> */

arguments:
        a_uint  v               unsigned int destination
        int     r               relocation mode
        struct expr *esp        expression evaluation
```

The functions require a merge mode value that references a mode defined in the xxxpst.c file. Other relocation modes such as R_MBRS, R_MBRU, or RMBRO may be Or'd with the merge mode to test conditions of the resulting mode value.

4.5 r OPTIONS

These are the most used "R_" defined values used in the relocatable output functions to test various conditions or specify a special output mode.

```
#define R_NORM  0x0000  /* Default, Or just use 0 */

#define R_SGND  0x0004  /* Signed */
#define R_USGN  0x0008  /* Unsigned */
#define R_OVRF  0x0008  /* Overflow */

#define R_MBRS  0x0004  /* Merge Bit Range Signed */
                        /* An alias for Signed */
#define R_MBRU  0x0008  /* Merge Bit Range Unsigned */
                        /* An alias for Unsigned */
#define R_MBRO  0x0008  /* Merge Bit Range Overflow */
                        /* An alias for Overflow */

#define R_LSB   0x0000  /* LSB */
#define R_MSB   0x000C  /* MSB */

#define R_PAG0  0x0010  /* Page '0'    .setdp */
#define R_PAGN  0x0020  /* Page 'nnn'  .setdp */

#define R_PCR   0x0040  /* PC adjust (default)    */
```

4.6 DIRECT PAGE DESCRIPTOR

The direct page descriptor informs the assembler and linker of the address and size of the region to designate as a page. The default page length is 256 and requires the base address to be a multiple of the page length. The page length can be changed by setting the ASxxxx internal page mask, p_mask, to the page lenth - 1. The page length must be a power of 2. The p_mask value must be set in the minit() function of the xxxmch.c file. An example of this usage was described in the Step By Step creation of the asolms40 assembler.

The machine specific directive .setdp, Set Direct Page, is the traditional ASxxxx method of specifying a direct page. See the coding in the as6800 assembler for a single fixed address method of setting a direct page or the coding in as6809 to modified the .setdp directive for setting any 256 byte boundary as the direct page.

Output a relocatable page description:

```
void    outdp(carea, esp, r)
```

arguments:

int	r	paging type
struct expr	*esp	expression evaluation
struct area	*carea	current area

The paging type, r, is always set to 0. The expression contains the base address of the page. The current area, carea, is a pointer to the area structure where the the direct page is located.

CHAPTER 5

ASXXXX STEP BY STEP INTERNALS

5.1 GETTING STARTED

The 'Step By Step' process includes references to many of the ASxxxx core elements that include various structures, variables, pointers, and definitions. This section will describe all of these so that they may be used when creating an ASxxxx assembler.

5.2 VARIABLES

This first section will decscibe various variables that may prove useful while writing an assembler.

5.2.1 a_uint

```
typedef unsigned INT32 a_uint;
```

The defined type 'a_uint' is used for all address and unsigned variable value calculations. Its size is required to be at least 32-bits to allow upto 32-bit addressing or 32-bit value manipulation.

5.2.2 ib[], ic[], and ip

```
/* assembler-source text line for processing */
char    ib[NINPUT*2];

/* assembler-source text line for listing */
char    ic[NINPUT*2];

/* The pointer into the assembler-source text line in ib[] */
char    *ip;
```

As the assembler reads each line from a source file, include file, or macro the line is copied into two line buffers, ib[] and ic[]. The line buffer ib[] is scanned and any key words are replaced by their definitions. The character pointer ip is set to the beginning of the line buffer ib[] in preparation for parsing the source line.

5.2.3 dot

During processing of the assembly source code the program address is represented by the '.' symbol. Internally the address is maintained in the structure sym[0]. As a more descriptive name 'dot' is defined to be structure sym[0].

```
#define dot     sym[0]          /* Dot, current loc */
```

5.2.4 passlmt

```
int     passlmt;
```

An assembler needing more than the default number of passes required to resolve all forward references must set passlmt to a number greater than 0 to enable the multi-pass processing. Set the pass limit value in the function minit() in the xxxmch.c file.

The assembler option -n# can also be used to change the number of passes during this stage of the assembly.

5.2.5 p_mask

```
a_uint  p_mask;
```

The page length mask informs the assembler and linker of the size of the region to designate as a page. The default page length is 256 and requires the base address to be a multiple of the page length. The page length can be changed by setting the internal page mask, p_mask, to the page lenth - 1. The page length must be a power of 2. The p_mask value must be set in the minit() function of the xxxmch.c file.

5.2.6 cycldgts

```
int     cycldgts;
```

The number of cycle count digits to display. The valid range is bounded to be >= 2 and <= 4. An assembler requiring other than the default of 2 digits should set this value in minit() in the xxxmch.c file.

5.2.7 opcycles

```
int     opcycles;
```

The number of execution states or clock cycles is loaded into the variable opcycles for each instruction. For most assemblers two digits is sufficient to display the cycle count and use an opcode table to specify the cycles for each opcode. A 256 byte opcode table with cycldgts=2 allows cycle counts from 1 to 99.

The ASxxxx assembler allows counts upto 4 digits or 1 to 9999. The translate code modifies the OPCY codes used by a specific assembler to the corresponding CYCL codes of the ASxxxx core.

```
/*
 * Translate To External Format
 */
if (opcycles == OPCY_NONE) { opcycles  =  CYCL_NONE; } else
if (opcycles  & OPCY_NONE) { opcycles |= (CYCL_NONE | 0x3F00); }
```

5.2.8 lmode

```
int     lmode;
```

Under normal circumstances the listing of any evaluated mnemonic or machine dependent text line is controlled by the .list or .nlist assembler directives. However this default operation can be modified by setting the lmode value within the case statement of the particular instruction mnemonic or machine directive in the function machine(). When the noted .list parameters are enabled the following options will print those options and inhibit all other listing parameters.

NLIST	No listing	
SLIST	Source and Line Number	(lin,src)
ALIST	Address	(loc)
BLIST	Address With Allocation	(loc,eqt)
CLIST	Code	(bin)
ELIST	Equate or IF Evaluation	(eqt)

A typical example is taken from the asolms40 assembler for the cpu selection directive where only the source and line number are listed:

```
case S_CPU:
        opcycles = OPCY_CPU;
        cputyp = op;
        sym[2].s_addr = op;
        lmode = SLIST;
        break;
```

5.2.9 cb[]

At the completion of parsing an instruction mnemonic and its arguments the cb[] byte array holds the sequential stream of bytes generated by the evaluation of the current input source text line. In general for instruction parsing the first byte output is the opcode which is then followed by argument values. This fact allows one to use the first cb[] value, cb[0], as an index into an array of cycle counts indexed by the opcode value.

```
char    cb[NCODE];      /* array of assembler output values */
```

5.2.10 mchterm_ptr

```
int     (*mchterm_ptr)(struct expr *esp);
```

The element mchterm_ptr is a pointer to a function which provides auxiliary functionality to the term() function in asexpr.c. Its only argument is a pointer to an expression structure. If an expression is evaluated then the function returns a non zero value with the expression results in the expression structure else 0 is returned. The function may perform parsing using the variable ip during the evaluation. The value of ip must have its initial value if a 0 is returned.

To enable the additional functionality for all expression analysis set mchterm_ptr to the address of a function having the a struct expr *esp argument. This must be placed in minit() in the xxxmch.c file. A typical example follows:

```
/* asexpr.c term() Extension */
mchterm_ptr = mchterm;
```

and add the function mchterm() to the code of xxxmch.c. The following is the code from aspdp11 showing the added functionality. Note the use of exprx() in this function. This function is REQUIRED if an expression evaluation is to be performed. DO NOT use expr().

```
/*
 * Machine specific expression terms.
 */
int
mchterm(struct expr *esp)
{
        char *p;

        p = ip;
        if (getnb() == '^') {
                /* Option Must Immediately Follow ^ */
                switch(ccase[*ip++ & 0x007F]) {
                case 'c':       /* Complement Argument */
                        *(--ip) = '~';
                        *(--ip) = ' ';
                        exprx(esp, 100);
                        break;
                case 'f':       /* Single Word Floating Point */
                        atowrd();
                        esp->e_addr = rslt[3];
                        break;
                /*
                case 'r':       **
                                 * RAD50 Is A Special Radix Used
                                 * To Build File Name Strings.
                                 * ^r Is Processed in pdpadr.c
                                 * And .rad50 Processes Words.
                                 **
                 */
                /*
                 * ^B, ^0, ^Q, ^D, ^H, And ^X
                 * Are Processed In asexpr.c
                 */
                default:        /* No Valid ^Option Found */
                        ip = p;
                        return(0);
                }
                return 1;
        }
        ip = p;
        return 0;
}
```

For the case where the functionality is desired only during a specific expression analysis load mchterm_ptr with the address of the desired function, evaluate the expression using expr(), and then load mchterm_ptr with a NULL to inhibit any further use of the function.

5.2.11 mchopt_ptr

```
int     (*mchoptn_ptr)(char *id, int v);
```

The element mchopt_ptr is a pointer to a function which provides auxiliary functionality to the .enabl and .dsabl directives of the assembler. Its arguments are a character string pointer and an integer. The character string is a string argument of the .enabl or .dsabl directive. If the integer value is 0 then the parameter should be disabled else the parameter should be enabled. If the string does not match any parameter the function should return 0 else a non zero value should be returned signifying the parameter was valid.

To enable the additional functionality for the .enabl and .dsabl directives load mchopt_ptr with the address of a function having a character string pointer and an interger value. This must be placed in minit() in the xxxmch.c file. A typical example follows:

```
/* .enabl/.dsabl Extension */
mchoptn_ptr = mchoptn;
```

and add the function mchopt() to the code of xxxmch.c. The following is the code from aspdp11 showing the added .enabl and .dsabl items.

```
/*
 * Machine specific .enabl/.dsabl terms.
 */
int
mchoptn(char *id, int v)
{
        /* Floating Point Truncation */
        if (symeq(id, "fpt", 1)) { fpt = v; } else
        /* Address Memory Absolute */
        if (symeq(id, "ama", 1)) { ama = v; } else
        /* Self Modifying Instruction */
        if (symeq(id, "smi", 1)) { smi = v; } else
        /* Automatic .WORD Generation, An Option In ASMAIN.C */
        if (symeq(id, "awg", 1)) { awg = v ? 2 : 0; } else
        /* Explicit Register Types */
        if (symeq(id, "rtyp", 1)) { rtyp = v; } else {
                return(0);
        }
        return(1);
}
```

5.3 STRUCTURES

5.3.1 struct mne

The mne structure is a linked list of the assembler mnemonics and directives. The list of mnemonics and directives contained in the device dependent file xxxpst.c are hashed and linked into NHASH lists in module assym.c by syminit(). The structure contains the mnemonic/directive name, a subtype which directs the evaluation of this mnemonic/directive, a flag which is used to detect the end of the mnemonic/directive list in xxxpst.c, and a value which is normally associated with the assembler mnemonic base instruction value. The m_flag element can include any additional instruction flags as long as they donot conflict with the S_EOL bit.

```
struct  mne
{
        struct  mne *m_mp;      /* Hash link */
        char    *m_id;          /* Mnemonic (JLH) */
        char    m_type;         /* Mnemonic subtype */
        char    m_flag;         /* Mnemonic flags */
        a_uint  m_valu;         /* Value */
};
```

5.3.2 struct expr

The expr structure is used to return the evaluation of an expression. The structure supports three valid cases:

```
(1)     The expression evaluates to a constant,
        mode = S_USER, flag = 0, addr contains the
        constant, and base = NULL.
(2)     The expression evaluates to a defined symbol
        plus or minus a constant, mode = S_USER,
        flag = 0, addr contains the constant, and
        base = pointer to area symbol.
(3)     The expression evaluates to an external
        global symbol plus or minus a constant,
        mode = S_NEW, flag = 1, addr contains the
        constant, and base = pointer to symbol.
```

```
struct  expr
{
        char    e_mode;         /* Address mode */
        char    e_flag;         /* Symbol flag */
        a_uint  e_addr;         /* Address */
        union   {
                struct area *e_ap;
                struct sym  *e_sp;
        } e_base;               /* Rel. base */
        char    e_rlcf;         /* Rel. flags */
};
```

5.3.3 struct sym

The sym structure is a linked list of symbols defined in the assembler source files. The first symbol is "." defined in asdata.c. The entry 'struct tsym *stsym' links any temporary symbols following this symbol and preceeding the next normal symbol. The structure also contains the symbol's name, type (NEW, USER or LOCAL), flag (global, assigned, and multiply defined), a pointer to the area structure defining where the symbol is located, a reference number assigned by outgsd() in asout.c, the symbols address relative to the base address of the area where the symbol is located, the symbols base address from the previous assembler pass, and a pointer to an expression that the linker should evaluate (if required).

```
struct  sym
{
        struct  sym  *s_sp;     Hash link
        struct  tsym *s_tsym;   Temporary symbol link
        char    *s_id;          Symbol (JLH)
        char    s_type;         Symbol subtype
        char    s_flag;         Symbol flags
        struct  area *s_area;   Area line, 0 if absolute
        int     s_ref;          Ref. number
        a_uint  s_addr;         Address
        a_uint  p_addr;         Previous Pass Address
        char    *s_expr;        Expression to evaluate
};

/*
 * Internal Definitions
 */
#define dot     sym[0]          /* Dot, current loc */
```

5.4 EXTENDED ADDRESSING: MERGE MODES

The ASxxxx assemblers use a method called 'merge mode' to combine a value with a constant. This method passes the value or a reference to a value to the linker along with a recipe to combine this value with another absolute value, typically an op-code. Each ASxxxx assembler can define upto 16 unique 'merge mode' recipes for combining two values. The recipe index is specified by the third 4-bit nibble of the relocation parameter word value (0x0000, 0x0100, ..., 0x0E00, 0x0F00).

5.4.1 Mode R_NORM

The basic 'R_NORM' one-to-one merge mode recipe is shown here. The character array is in octal for historical reasons:

```
/*
 * Basic Relocation Mode Definition
 *
 *      #define         R_NORM  0000            No Bit Positioning
 */
char    mode0[32] = {   /* R_NORM */
        '\200', '\201', '\202', '\203', '\204', '\205', '\206', '\207',
        '\210', '\211', '\212', '\213', '\214', '\215', '\216', '\217',
        '\220', '\221', '\222', '\223', '\224', '\225', '\226', '\227',
        '\230', '\231', '\232', '\233', '\234', '\235', '\236', '\237'
};
```

or can be represented in hex form:

```
char    mode0[32] = {   /* R_NORM */
        '0x80', '0x81', '0x82', '0x83', '0x84', '0x85', '0x86', '0x87',
        '0x88', '0x89', '0x8A', '0x8B', '0x8C', '0x8D', '0x8E', '0x8F',
        '0x90', '0x91', '0x92', '0x93', '0x94', '0x95', '0x96', '0x97',
        '0x98', '0x99', '0x9A', '0x9B', '0x9C', '0x9D', '0x9E', '0x9F'
};
```

Character 'mode0[0]' designates bit 0 of the source. The bit is active if the most significant bit is set. Bits <4:0> specify where this bit is to be placed in the destination.

5.4.2 Mode Structure

The mode structure is used by the assembler to more quickly process the relocation mode of local merge modes. Each additional mode must be added to the mode[] array.

```
*m_def is a pointer to the bit relocation definition.
m_flag indicates that bit position swapping is required.
m_dbits contains the active bit positions for the output.
m_sbits contains the active bit positions for the input.

struct  mode
{
        char *  m_def;          Bit Relocation Definition
        int     m_flag;         Bit Swapping Flag
        int     m_mask;         Bit Mask
        int     m_mbro;         Bit Range Overflow Mask
};

struct  mode    mode[1] = {
    {   &mode0[0],      0,      0x0000FFFF,     0x0000FFFF
        }
};
```

Any specific additional modes required for an assembler are defined in the xxx.h file for mode1[32], ..., through mode15[32] and could have more descriptive names:

```
#define R_0100  0x0100          /* Extended mode  1 */
...
#define R_0F00  0x0F00          /* Extended mode 15 */
```

5.4.3 Array of Mode Structure Pointers

The array of pointers to mode structures is indexed by the mode number to select the appropriate mode structure.

```
struct  mode    *modep[16] = {
        &mode[0],       NULL,           NULL,           NULL,
        NULL,           NULL,           NULL,           NULL,
        NULL,           NULL,           NULL,           NULL,
        NULL,           NULL,           NULL,           NULL
};
```

Add a pointer for each added mode structure into this array.

5.5 CHARACTER TABLES

5.5.1 ASCII Reference Table

```
/*
 * ASCII Table
 */
/*----  \000    \001    \002    \003    \004    \005    \006    \007    */
/*-------------------------------------------------------------------------*/
/*\000  NUL     SOH     STX     ETX     EOT     ENQ     ACK     BELL    */
/*\010  BS      TAB     LF      VT      FF      CR      SO      SI      */
/*\020  DLE     DC1     DC2     DC3     DC4     NAK     SYN     ETB     */
/*\030  CAN     EM      SUB     ESC     FS      GS      RS      US      */
/*\040  SPACE   !       "       #       $       %       &       `       */
/*\050  (       )       *       +       `       -       .       /       */
/*\060  0       1       2       3       4       5       5       7       */
/*\070  8       9       :       ;       <       =       >       ?       */
/*\100  @       A       B       C       D       E       F       G       */
/*\110  H       I       J       K       L       M       N       O       */
/*\120  P       Q       R       S       T       U       V       W       */
/*\130  X       Y       Z       [       \       ]       ^       _       */
/*\140  '       a       b       c       d       e       f       g       */
/*\150  h       i       j       k       l       m       n       o       */
/*\160  p       q       r       s       t       u       v       w       */
/*\170  x       y       z       {       |       }       ~       DEL     */
```

5.5.2 Character Types

```
/*
 * Definitions for Character Types
 */
#define SPACE   '\000'
#define ETC     '\000'
#define LETTER  '\001'
#define DIGIT   '\002'
#define BINOP   '\004'
#define RAD2    '\010'
#define RAD8    '\020'
#define RAD10   '\040'
#define RAD16   '\100'
#define ILL     '\200'

#define DGT2    (DIGIT|RAD16|RAD10|RAD8|RAD2)
#define DGT8    (DIGIT|RAD16|RAD10|RAD8)
#define DGT10   (DIGIT|RAD16|RAD10)
#define LTR16   (LETTER|RAD16)
```

5.5.3 Character Type Array

```
/*
 *      an array of character types,
 *      one per ASCII character
 */
char    ctype[128] = {
/*NUL*/ ILL,    ILL,    ILL,    ILL,    ILL,    ILL,    ILL,    ILL,
/*BS*/  ILL,    SPACE,  ILL,    ILL,    SPACE,  ILL,    ILL,    ILL,
/*DLE*/ ILL,    ILL,    ILL,    ILL,    ILL,    ILL,    ILL,    ILL,
/*CAN*/ ILL,    ILL,    ILL,    ILL,    ILL,    ILL,    ILL,    ILL,
/*SPC*/ SPACE,  ETC,    ETC,    ETC,    LETTER, BINOP,  BINOP,  ETC,
/*(*/   ETC,    ETC,    BINOP,  BINOP,  ETC,    BINOP,  LETTER, BINOP,
/*0*/   DGT2,   DGT2,   DGT8,   DGT8,   DGT8,   DGT8,   DGT8,   DGT8,
/*8*/   DGT10,  DGT10,  ETC,    ETC,    BINOP,  ETC,    BINOP,  ETC,
/*@*/   ETC,    LTR16,  LTR16,  LTR16,  LTR16,  LTR16,  LTR16,  LETTER,
/*H*/   LETTER, LETTER, LETTER, LETTER, LETTER, LETTER, LETTER, LETTER,
/*P*/   LETTER, LETTER, LETTER, LETTER, LETTER, LETTER, LETTER, LETTER,
/*X*/   LETTER, LETTER, LETTER, ETC,    ETC,    ETC,    BINOP,  LETTER,
/*`*/   ETC,    LTR16,  LTR16,  LTR16,  LTR16,  LTR16,  LTR16,  LETTER,
/*h*/   LETTER, LETTER, LETTER, LETTER, LETTER, LETTER, LETTER, LETTER,
/*p*/   LETTER, LETTER, LETTER, LETTER, LETTER, LETTER, LETTER, LETTER,
/*x*/   LETTER, LETTER, LETTER, ETC,    BINOP,  ETC,    ETC,    ETC
};
```

5.5.4 Upper To Lower Array

```
/*
 *      an array of characters which
 *      perform the case translation function
 */
char    ccase[128] = {
/*NUL*/ '\000', '\001', '\002', '\003', '\004', '\005', '\006', '\007',
/*BS*/  '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
/*DLE*/ '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
/*CAN*/ '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037',
/*SPC*/ '\040', '\041', '\042', '\043', '\044', '\045', '\046', '\047',
/*(*/   '\050', '\051', '\052', '\053', '\054', '\055', '\056', '\057',
/*0*/   '\060', '\061', '\062', '\063', '\064', '\065', '\066', '\067',
/*8*/   '\070', '\071', '\072', '\073', '\074', '\075', '\076', '\077',
/*@*/   '\100', '\141', '\142', '\143', '\144', '\145', '\146', '\147',
/*H*/   '\150', '\151', '\152', '\153', '\154', '\155', '\156', '\157',
/*P*/   '\160', '\161', '\162', '\163', '\164', '\165', '\166', '\167',
/*X*/   '\170', '\171', '\172', '\133', '\134', '\135', '\136', '\137',
/*`*/   '\140', '\141', '\142', '\143', '\144', '\145', '\146', '\147',
/*h*/   '\150', '\151', '\152', '\153', '\154', '\155', '\156', '\157',
/*p*/   '\160', '\161', '\162', '\163', '\164', '\165', '\166', '\167',
/*x*/   '\170', '\171', '\172', '\173', '\174', '\175', '\176', '\177'
};
```