DASMx

Version 1.40, 18th October 2003

A microprocessor opcode disassembler

Copyright

Disclaimer

DASMx comes without any express or implied warranty. You use this software at your own risk. Conquest Consultants have no obligation to support or upgrade this software. Conquest Consultants cannot be held responsible for any act of copyright infringement or other violation of applicable law that results from use of this disassembler software.

Contents

Introduction

DASMx is a disassembler for a range of common microprocessors. The following main processor families are supported:

- Motorola 6800 family and single chip variants (including Hitachi 630*X* devices);
- Motorola 6809;
- MOS Technology 6502 and Rockwell 65C0*X*;
- Zilog Z80;
- Sharp LR35902 (single chip Z80 variant as used in the Nintendo GameBoy);
- Intel MCS-80/85™ family (i.e. 8080 and 8085);
- Intel MCS-48™ family (i.e. 8048 et al);
- Intel MCS-51™ family (i.e. 8051 et al);
- Microchip PIC16CXX family;
- RCA CDP1802 COSMAC and variants;
- Signetics 2650.

The disassembler takes as input a binary code/data image file (typically a ROM image) and generates either an assembler source file or a listing file. **DASMx** is a *multi-pass* disassembler with automatic symbol generation. **DASMx** can optionally use a symbol file containing user-defined symbols and specifications of data areas within the source image.

DASMx includes a powerful feature called *code threading*. Using known code entry points (e.g. reset and interrupt vectors) and by performing partial emulation of the processor, the disassembler is able to follow known code paths within a source binary image.

Use of code threading, together with the multi-pass operation and symbol table management permits readable assembly code output from source images that contain large amounts of data (which tend to confuse most disassemblers).

Historical note: Version 1.10 of **DASMx** superseded the Motorola 680x disassembler, **dasm6800** (last released as version 1.00 on 25th January 1997). The change of name reflected the wide range of processors then covered.

The key features of **DASMx** are:

- Disassembly of object code images for the following microprocessors:
  - Motorola 6800, 6802 and 6808;
  - Motorola 6801 and 6803;
  - Hitachi 6301 and 6303;
  - Motorola 6805;
  - Motorola 68HC05
  - Hitachi 6305;
  - Hitachi 63L05;
  - Motorola 6809;
  - MOS Technology/Rockwell 6502;
  - Rockwell 65C00/21 and 65C29;
  - Rockwell 65C02, 65C102 and 65C112;
  - Intel 8048;
  - Intel 8051;
  - Intel 8080 and 8085;
  - Microchip PIC16F83 and PIC16F84;
  - RCA CDP1802 COSMAC;
  - RCA CDP1805 and CDP1806;
  - Sharp LR35902 (i.e. GameBoy processor);
  - Signetics 2650;
  - Zilog Z80 and National Semiconductor NSC800.
- Multi-pass operation, with automatic symbol generation for jump, call and data target addresses;
- Code threading (used to automatically differentiate code from data);
- Control file containing user defined symbols, specifications of data areas and code entry points;
- Generation of full listing or assembler output file;
- Runs from the Windows command line.

Version history

Version	Date	Comments
0.90	28th July 1996	First public release (as **dasm6800**): with support for 6800/6802/6808 only.
1.00	25th January 1997	Second release (as **dasm6800**): 6801/6803 and 6809 support added; other improvements in performance and listing output.
1.10	16th July 1997	Third release (now renamed **DASMx**): 6502, Z80 and 8048 processor support added; minor improvements and bug fixes.
1.20	2nd April 1998	8080, 8085 and 2650 processor support added; improvements and bug fixes.
1.30	6th October 1999	6301, 6303, 65C00/21, 65C29, 65C02, 65C102, 65C112, 8051 and LR35902 processor support added; wide listing format showing execution cycles; checksum and CRC-32 calculation; number format improvements; new symbol file directives; other improvements and bug fixes.
1.40	18th October 2003	6805, 68HC05, 63L05, 6305, NSC800, CDP1802, CDP1805/1806, PIC16F83 and PIC16F84 processor support added; new checksum utility; bug fixes and improvements.

The changes from version 1.30 are:

- Disassembly of 6805, 68HC05, 63L05 and 6305 added;
- NSC800 CPU type added (identical instruction set to Z80);
- Disassembly of RCA CDP1802 COSMAC and CDP1805/1806 added;
- Disassembly of Microchip PIC16F83 and PIC16F84 added;
- New DWORD data type in symbol file;
- Symbol file `code` directive changed to allow length parameter;
- Checksum command line utility added to distribution;
- Origin now defaults to 0;
- RCA and Acorn ARM number formats added;
- Bug fixes and improvements to: 2650, 6502, 6809, 8051, & GameBoy;
- Bug fix: Signetics number type now allowed in symbol file;

- Improvements to code/data differentiation algorithm affecting disassembly for all processors.

The changes between versions 1.20 and 1.30 were:

- Disassembly of Hitachi 6301 and 6303 added;
- Disassembly of Rockwell 65C00/21, 65C29, 65C02, 65C102 and 65C112 added;
- Disassembly of Intel 8051 added;
- Disassembly of Sharp LR35902 (GameBoy processor) added;
- Corrected documentation concerning Hitachi 6309 (which has, in fact, an identical instruction set to the 6809);
- Labelling and threading improvements for 8080, 8085 and Z80 disassembly (affects RST and indirect addressing instructions);
- Correction to instruction format for 2650 lodz/eorz/andz/…;
- New wide listing format showing execution cycles for each instruction;
- File size, checksum and CCITT CRC-32 calculated and shown in listing header;
- Auto number format determined by processor type (which can be overriden by a directive in the symbol file);
- User messages can now be specified and generated from the symbol file;
- Symbol file includes (which may be nested) now permitted.

The changes between versions 1.10 and 1.20 were:

- Disassembly of Intel 8080 and 8085 added (in addition to existing support for 8080 provided by Z80 disassembly);
- Disassembly of Signetics 2650 added;
- New symbol file command to skip areas of source image;
- Origin can now be specified in symbol file;
- New command line option to specify a single code entry point for threading;
- New command line option to list all processors supported;
- Fix to incorrect disassembly of 6801/6803 subd instruction (opcode 0x93);
- Bug fixes and other minor changes.

The changes between versions 1.00 and 1.10 were:

- All references to "dasm6800" replaced by "DASMx";
- Disassembly of 6502 added;
- Disassembly of Z80 added;
- Disassembly of 8048 added;
- Minor bug fix for code threading of 6801/6803 direct branch instructions;
- Minor changes to listing output;

- Bug fixes and other minor improvements.

The changes between versions 0.90 and 1.00 were:

- Disassembly of 6801/6803 added;
- Disassembly of 6809 added;
- Define byte pseudo-op now generates full listing;
- Two new commands supported in symbol file: `cpu` (to select processor type) and `addrtab` (to define a table of addresses, each of which points to data);
- New command line switch to select processor type;
- Performance improvement to pass 1;
- Minor changes to listing output;
- Bug fixes and other minor improvements.

Distribution

DASMx is copyright software. This version (1.40) may be distributed and used freely provided that all files are included in the distribution, no files (including the distribution zip file) are modified and no charge is made beyond that reasonable to cover copying (maximum $10 US). Conquest Consultants reserve the right to alter the free distribution and use terms for any future versions or derivatives of **DASMx** that may be produced.

DASMx version 1.40 is distributed as file **dasmx140.zip** in the *WinME / Win98 / Win95 / Programming Utilities* section of the Simtel.net archive. Provided that the above distribution terms are adhered to, this file may be freely copied to and mirrored at other ftp and web sites.

Operation

Before describing the operation of **DASMx** in detail, here is an overview of how the disassembler will be typically used in practice.

First, you must obtain a file containing a binary image of the code/data that you wish to disassemble. Typically, this will be from one or more ROMs or EPROMs that have been read using a PROM programmer. Some PROM programmers output data in a form of ASCII hexadecimal format (Intel and Motorola are two common formats). If that is the case, then you must use a conversion utility to generate a raw binary image. A good check that you have a correct binary image of a complete ROM is that the file length (shown by a DIR command) will be a power of two and will correspond to the length of the ROM. For example, the file size of a complete image of a 27256 EPROM will be 32,768 bytes.

Assuming at this stage that you do not know which areas of the binary image are code and which are data, it is sensible to use the code threading feature. For code threading to work, you must provide at least one code entry point. This requires `code`, `vector` or `vectab` entries in a symbol file. For example, if you are disassembling a ROM image from the uppermost region of the 6800 microprocessor address space, then four `vector` entries for the standard interrupt and reset vectors will be all that is initially required to provide the necessary entry points. You can also improve the readability of the disassembled output by defining symbols for all known hardware addresses (e.g. PIA registers and other ports).

Try modifying one of the supplied example symbol files to suit your application. It is important that the correct processor type is specified using a `cpu` directive in the symbol file (or by command line switch). The disassembler will not make much sense of Z80 code if it thinks that the processor is a 6502!

Run the disassembler with code threading. This will identify all known areas of code. Data and unknown areas will be listed as byte data rather than disassembled into instruction mnemonics. Due to limitations of the code threading process (see below) not all code areas may be identified. Any additional code entry points or address vector tables can be added to the symbol file. Similarly, areas of byte, word or string data that can be identified from examination of the disassembly listing can also be recorded in the symbol file.

Using a repeated "disassemble, inspect listing, update symbol file" cycle a comprehensive disassembly of an image can be built up quite quickly.

Finally, if you are satisfied that you have identified all main data areas, try disassembling without code threading. This will help pick up areas of code that may have been missed by the code threading and subsequent manual investigation process.

Platform

DASMx is a Win32 console application. This means that it is a 32-bit application that requires Windows 95/98/Me or Windows NT/2000/XP to run. Typically, you will run the disassembler from a command line window.

Command line options

DASMx has the following command line options:

-a	Generate assembler output (default is to generate a full listing file).
-c*TYPE*	Set the CPU processor type – overrides any `cpu` statement in the symbol file, where *TYPE* is one of the types reported by the **-l** option (6800, 6809, 6502, Z80 etc.) (default is 6800).
-e*NNNN*	Specify a code entry point *NNNN* for threading.
-l	List all processors supported and exit.
-o*NNNN*	Set the origin, or start address to *NNNN* (default is top of address space less the length of the source image).
-t	Perform code threading (requires at least one code entry point to be specified).
-v	Display version information and exit.
-w	Wide listing format (shows instruction cycles and up to 8 data bytes per line).

When specifying addresses, the number *NNNN* should be specified using C language conventions (i.e. default is decimal, prefix with 0x for hex, prefix with 0 for octal).

Input files

The primary input file is a binary image of the code/data to be disassembled. This must be code for one of the supported microprocessors (or other manufacturer equivalent). **DASMx** will produce meaningless output for any other type of processor.

DASMx assumes a file extension of ".bin" unless otherwise specified for the binary image file.

DASMx looks for a symbol file of the same base name as the source binary file, but with a ".sym" file extension. If a symbol file is found, it will be used. Provision of a symbol file is optional, except where code threading is used (where a symbol file must be used to define at least one code entry point).

Symbol file syntax

The symbol file is a plain text file that may be created/modified with any text editor. The file contains lines that fall into one of three categories:

- Comment lines;
- Command lines;
- Blank lines.

Comment lines are denoted by ';' as the first non-whitespace character on the line. Command lines start with one of the specified keywords. Parameters follow the command keyword, separated by spaces or tabs. A comment may be added to the end of a command, preceded by the ';' character. Blank lines are ignored.

Number value parameters may be given in decimal (the default), octal or hex using standard C language conventions (e.g. 0x prefix for hex).

The symbol file command syntax contains an `include` directive which allows one symbol file to be included within another. Included files may be nested to any practical depth. A particular use of this feature is to have a symbol file containing a generic set of defintions for a processor or item of hardware. This can then be included within a symbol file with additional definitions for a specific software image that runs on that processor/hardware. The pair of example files, **gameboy.sym** and **tetris.sym**, shows this in action with generic GameBoy definitions in one file and specific defintions for a tetris game cartidge in the other.

Valid command keywords and their meaning are summarised in the table below.

Command	Function/syntax
cpu	Specify the processor type. *Syntax*: `cpu PIC16F83 | PIC16F84 | 1802 | 1805 | 2650 | 6502 | 65C00 | 65C02 | 65C59 | 65C102 | 65C112 | 6301 | 6303 | 6305 | 63L05 | 6800 | 6801 | 6802 | 6803 | 6805 | 68HC05 | 6808 | 6809 | 8048 | 8051 | 8080 | 8085 | Z80 | LR35902`
numformat	Specify number format (overriding default for processor) as ARM, Intel, Motorola, RCA, Signetics, C language hex (i.e. 0x prefix) or decimal. *Syntax*: `numformat A | I | M | R | S | C | D`
include	Include a file containing additional symbol commands. Include filess may be nested. *Syntax*: `include <filename>`
message	Generate a message to the console during disassembly. *Syntax*: `message "<message string>"` *or*: `message <word1> [<word2> <word3> ...]`
org	Define the start address for the first byte of the code/data image. Note that only one org statement should be present in a symbol file. *Syntax*: `org <address>`
symbol	Define a symbol corresponding to a value (usually an address). *Syntax*: `symbol <value> <name>`
vector	Define a location that contains a word pointing to a code entry (for example, the reset entry point). *Syntax*: `vector <address> [<vector name>] [<destination name>]`
vectab	Define a table of vectors (i.e. a jump table) of length <*count*>. Each vector will be used as a code entry point if threading is used. *Syntax*: `vectab <address> <name> [<count>]`
code	Define a code entry point (for code threading). Optionally, <*count*> may specify the length of the code region in instruction words. *Syntax*: `code <address> [<name>] [<count>]`
byte	Define a single data byte, or <*count*> length array of bytes. *Syntax*: `byte <address> <name> [<count>]`
word	Define a single data word, or <*count*> length array of words. *Syntax*: `word <address> <name> [<count>]`
addrtab	Define a table of addresses, which point to data, of length <*count*>. *Syntax*: `addrtab <address> <name> [<count>]`
string	Define a single data character, or <*count*> length string of chars. *Syntax*: `string <address> <name> [<count>]`
skip	Skip (i.e. omit from disassembly and listing) <*count*> length data bytes. *Syntax*: `skip <address> <count>`

Output files

By default, **DASMx** generates a disassembly listing file. This is similar to the full listing file generated by most assemblers. Optionally, **DASMx** can be made to produce an assembly file instead. This could then be used as a source file to an assembler of your choice (with certain provisos concerning pseudo-ops and number formats noted later).

As an aid to readability, **DASMx** inserts a comment line after all breaks in a sequence of instructions (e.g. after an unconditional branch or jump, or a return from subroutine). Comment lines are also inserted between code and data areas. This use of comment lines breaks the output listing into identifiable sections and aids manual inspection of the resultant disassembly listing.

Note that output files tend to be large. For example, a 32 Kbyte ROM image will generate a listing file of around half a megabyte in length.

The output file is named based upon the name of the source image file, but with a file extension of ".lst" for the list file or ".asm" for the assembly output file.

Listing file

The list file format is largely self-explanatory. Program counter and code/data byte values are given in hex. Code/data is also shown as ASCII characters (where printable) as an aid to identifying strings within the binary image. If the wide listing format is selected then instruction cycle counts are also given for every instruction.

Instruction cycles are shown within [square braces]. If an instruction takes a variable number of cycles to execute (e.g. a conditional branch on many processors) then two values are shown: the minimum and the maximum.

Code threading

Code threading is a very powerful feature that will automatically identify known areas of code. It can prove particularly useful in the early stages of disassembly of an image that contains large areas of data. Such data areas would otherwise be disassembled incorrectly as code and would add many erroneous symbols to the symbol table.

Code threading works by performing a partial emulation of the processor; executing instructions starting from one or more known entry points. Code threading follows calls to subroutines and conditional and unconditional branches. In certain cases, the code threading may fail to follow certain code paths (i.e. leaving valid code still defined as data). The following are examples of where the code threader will fail to follow a correct execution path:

- Pushing an address onto the stack and then, later, performing a return from subroutine instruction (i.e. as a method of performing a jump);
- Performing an indexed branch instruction (e.g. using addresses taken from a vector table);
- Use of undocumented instruction opcodes – since threads are abandoned when an invalid opcode is detected;
- Self-modifying code.

Indexed branch instructions are highlighted in the output listing by automatically generated comments. These are an indication that you need to manually identify what the contents of the index register will be prior to the branch (often obvious – look for a preceding load index register instruction.) Then, you can add a `code` or a `vectab` entry to the symbol file and repeat the disassembly.

In rare cases, code threading may incorrectly identify data as code:

- A call to a subroutine that never returns (e.g. the subroutine discards the return address); the other side of the call containing data rather than code.
- A conditional branch that is always, or never, executed (and the other side of the branch contains data rather than code).

Normally this latter scenario is pretty unlikely and requires a particularly perverse programmer of the original code. However, it is a technique that may be encountered on those processors that have a "better" (i.e. fewer cycles and/or fewer bytes) conditional jump than unconditional jump. So, in general, code threading will identify guaranteed known areas of code that may be a subset of the overall actual code. Most of the above problem areas can be dealt with by manual inspection of the disassembly listing and subsequent additions to the symbol file.

A thread of execution will be abandoned for one of two reasons. If a branch or subroutine call is made outside the address range corresponding to the source image then that thread is not followed. Also, if an invalid instruction is detected then the thread terminates immediately. This will produce a command line error message identifying the address where the problem occurred. Normally this represents an error condition that can be corrected by the person operating the disassembler:

- The processor type is incorrectly specified;
- The source binary image is not real code;
- An incorrect code entry point has been supplied;
- So called "undocumented" instructions have been used.

In rare cases, the original programmer may have done something that causes the code threader to incorrectly identify data as code. These cases may also result in invalid instruction messages.

Microprocessor specifics

The following sub-sections detail items of note relating to disassembly for the specific microprocessors (and their variants) supported by **DASMx**.

Motorola 6800, 6802 and 6808

The Motorola 6800, 6802 and 6808 share an identical instruction set.

Assembler mnemonics follow the Motorola standard definitions (see reference [1]). Note that there are two common styles for instructions that involve the A and B registers:

- The A or B register name is separated by whitespace from the base instruction (e.g. **lda b value**);
- The A or B register name is used as a suffix to the instruction mnemonic (e.g. **ldab value**).

DASMx uses the latter style. This point also applies to the 6801/6803 and 6809 mnemonics generated by the disassembler.

Motorola 6801 and 6803

The Motorola 6801 and 6803 share an identical instruction set that is an object code compatible superset of that of the base 6800. These processors contain on-chip timer and I/O plus an expanded interrupt vector area over that of the 6800. Definitions for these in a symbol file will be useful for disassembly of any 6801/6803 code. See the supplied 6803 symbol file, **ebcgame.sym**, for an example that could be used as a template for other 6801/6803 disassembly.

Hitach 6301 and 6303

The Hitachi 6301 and 6303 are enhanced versions of the Motorola 6801/6803 with an enhanced object code compatible instruction set. Differences include a few additional instructions and pipelining that improves some instruction times.

Motorola 6805

The 6805 is another single chip microprocessor from Motorola. But, unlike the 6801/6803 it has an instruction set that no longer object code compatible with the original 6800. Mostly found in embedded applications, the 6805 formed the starting point for a series of microprocessors from Motorola and Hitachi.

Hitach 63L05

The Hitachi 63L05 has an identical instruction set to the Motorola 6805, but with different cycle counts for some instructions.

Motorola 68HC05 and 68HC705

The Motorola 68HC05 and 68HC705 have an instruction set that is a superset of the Motorola 6805. It has three extra instructions: **stop**, **wait** and **mul**. There are also cycle count differences from the base 6805.

Hitach 6305

The Hitachi 6305 has an instruction set that is a superset of the Motorola 6805. It has three extra instructions: **stop**, **wait** and **daa**. There are also cycle count differences from the base 6805.

Motorola 6809

The Motorola 6809 has an instruction set that is compatible with that of the 6800 *at the assembler level* (i.e. it is *not* binary compatible, but every 6800 instruction mnemonic is present in the 6809 instruction set). The 6809 also has many additional instructions that are not present in the 6800.

*Note:the Hitachi 6309 was incorrectly included in earlier versions of **DASMx** as having an identical instruction set to the 6809. This mistake was due to incorrect information in a Hitachi data book. It is now understood that the 6309 has a greatly expanded set of instructions over the 6809. Full support for the 6309 may be added in a future version of **DASMx**.*

MOS Technology/Rockwell 6502

The MOS Technology/Rockwell 6502 has a similar instruction set to that of the 6800 (but totally opcode incompatible).

A number of 6502 variants, with expanded instruction sets and addressing capabilities have appeared over the years. **DASMx** copes with some, but not all, of these variants (see next sections). If you know that a processor is based on the 6502 architecture, but are unsure of the variant then try disassembling with the CPU type set to 6502, 65C02 and 65C00. Inspect the results and select whichever gives the most intelligent disassembly. [Tip: try this with code threading and select the processor that gives least threading errors.]

Rockwell 65C00/21 and 65C29

The Rockwell 65C00/21 and 65C29 each contain two enhanced CMOS 6502 CPU cores plus on-chip masked ROM, RAM, two timers and general purpose I/O. Instruction set differences over the basic NMOS 6502 include new instructions for unsigned multiply, memory bit set and reset, branch on bit set/reset, unconditional branch and push/pop for the index registers. With the exception of the multiply instruction, these new instructions are a subset of the additional instructions in the 65C02.

Note that the CPU type for the 65C00/21 should be specified as `65C00` (i.e. without the trailing "/21").

Rockwell 65C02, 65C102 and 65C112

The Rockwell 65C02 is an improved version of, and object code compatible with, the original NMOS 6502 with twelve new basic instructions (giving 59 new opcodes with variants). The 65C02 is pin compatible with the original 6502. The 65C102 is similar, but with minor pinout differences to provide for multi-processor bus operation. The 65C112 has no internal clock oscillator and is designed as a slave processor to the 65C102. The extra instructions include all of the additions found in the 65C00/21 and 65C29 dual processors – with the exception of the multiply instruction found in those devices.

Zilog Z80

The Zilog Z80 (also made by Mostek, Sharp, NEC and other second sources) has an instruction set that is binary compatible with that of the Intel 8080, but with many additional instructions. Although each 8080 instruction has an identical Z80 instruction, Zilog chose to use a different mnemonic style for almost every instruction. Consequently, Z80 assembler (even if restricted to the 8080 subset) appears quite different even though the resulting binary image is identical.

The Z80 has a great many (so called) undocumented instructions that (sometimes) perform useful functions. **DASMx** does not currently support these additional instructions.

Like the 6502, the Z80 has spawned many variants with opcode compatible instruction supersets. **DASMx** can be used on code for these devices with the standard caveat that any of the new instructions will not be disassembled as valid code (and therefore code threading is not advised.)

National Semiconductor NSC800

The National Semiconductor NSC800 has an identical instruction set to the Zilog Z80. The differences between a Z80 and the NSC800 were all electrical. The NSC800 was fabricated in a CMOS process called P^2CMOS. It also had a bus architecture that was compatible with the Intel 8080 (i.e. multiplexed address and data bus) and was therefore not pin compatible with a standard Z80.

Sharp LR35902 (GameBoy processor)

The Sharp LR35902 is the processor used in the hugely popular Nintendo GameBoy. This processor is a single chip variant of the Zilog Z80. The instruction set is based on a subset of that of the Z80 but with some additional instructions. Of those instructions that are shared with the Z80, most are opcode compatible but there are a few differences.

As a single chip microcontroller, the LR35902 contains various on-chip I/O and timer functions. These are accessed through a 256 byte memory page starting at address 0xFF00. The supplied file, **gameboy.sym**, contains a set of known symbol

definitions for these memory mapped registers. This generic GameBoy processor symbol file may be included in the main symbol file for the disassembly of a specific binary image. The supplied **tetris.sym** file shows an example of this.

> **WARNING**: unlike all the other processors supported by **DASMx**, it has not been possible to obtain *official* manufacturer's data on the Sharp LR35902. The information used is derived from a number of different public domain documents – some of which conflict over certain details. Consequently, the LR35902 disassembly should be considered provisional and potentially subject to error.
>
> If anyone has access to genuine Sharp (or other official) data on this device please contact the author: pclare@bigfoot.com.

Intel MCS-80/85™ (8080 and 8085)

The Intel 8080 and 8085 share an almost identical instruction set. The Intel 8085 is an enhanced version of the 8080, with two additional instructions (`rim` and `sim`) used to control new serial in and out pins and interrupt inputs.

When disassembling 8080 (and, with provisos, 8085) code the user has the option of generating either Intel or Zilog mnemonics. To generate Intel mnemonics, simply specify the CPU type to be `8080` or `8085` as required.

Generating Zilog Z80 style mnemonics from Intel 8080 code is possible because the 8080 has an instruction set that is a compatible binary subset of those of the Z80. Simply specify the CPU type is as `Z80` and **DASMx** will correctly disassemble 8080 code into Zilog mnemonics. This will not suit Intel assembler die-hards, but may be preferred by those more familiar with the Z80.

WARNING: if **DASMx** is used as a Z80 disassembler on 8085 code and either of the two 8085 specific instructions are used (`rim` and `sim`) then problems will result. In such cases Zilog disassembly is probably best avoided. If you really must have Zilog mnemonics then read the following description of how these instructions are handled and be prepared for code threading to work incorrectly.

`rim` is a one byte instruction, but **DASMx** will attempt to disassemble this as the two byte `jr nz` Z80 instruction. This will both generate a false label and ignore the next byte in the 8085 opcode stream. Since that could be the first byte in a multi-byte opcode it could take a number of erroneously disassembled instructions before synchronisation is achieved.

`sim` is a one byte instruction that will be disassembled as the first byte of the three byte `ld hl` immediate instruction. The results will be similar to those for `rim`.

Intel MCS-48™ family (8048 etc.)

DASMx will disassemble opcodes for the following Intel MCS-48™ family devices (and equivalents from second source manufacturers): 8021, 8022, 8035, 8039, 8041, 8741, 8048, 8049 and 8748. The CPU type should be set to `8048` and the term "8048" is used throughout this documentation to refer to this family of devices.

The 8021 instruction set is a much reduced subset of the full 8048 set of instructions.

The 8022 has a very similar instruction set to the 8021, but with slightly more of the 8048 instructions and a few new instructions to handle the on-chip analogue to digital converter.

The 8041/8741 has almost the same instruction set as the 8048, but with just a few instructions missing.

DASMx can disassemble code for the 8021, 8022, 8041 and 8741 variants with the caveat that data areas may be disassembled as 8048 instructions that are in fact illegal on the variant.

The 8048 jump and call instructions operate on an 11-bit address (i.e. within a 2 Kbyte memory bank). A memory bank select bit (controlled by the `sel mb0` and `sel mb1` instructions) is combined with the 11-bit jump/call address to give full 12-bit addressing within the 4 Kbyte address space of the 8048. This presents a problem for the code threading and automatic label generation functions of **DASMx** since a destination address can only be fully calculated if the last memory bank select operation is known. Tracking the state of the memory bank select bit is currently beyond the capabilities of **DASMx**. For this reason, it is advised that code threading be not used if the size of the 8048 source image exceeds 2 Kbytes. If images greater than this are disassembled, even with threading disabled, some errors in automatically generated labels may be expected.

Intel MCS-51™ family (8051 etc.)

Intel introduced the 8051 to provide an upgrade path from the 8048. It would do all that the 8048 would do and more. The heritage of the 8048 is obvious in the architecture and instruction set of the 8051.

Like the 8048, the 8051 was initially available in a number of variants (e.g. 8031 and 8751). Subsequently, many further variants of the 8051 have been produced by Intel and by other manufacturers. Some of these added to the instruction set.

DASMx will only correctly disassemble code for the original 8051 devices that shared the MCS-51™ instruction set.

Signetics 2650

The Signetics 2650 is a rather oddball processor when compared to most other 8-bit processors handled by **DASMx**. It operates on 8-bit data and can address 32,768

bytes of memory organised in four pages of 8,192 bytes each. It has a large range of addressing modes, made possible by the use of bits encoded in the second byte of two and three byte instructions. It has a 3-bit stack pointer, which means that subroutines can be nested to, at most, eight deep.

RCA/Intersil CDP1802 COSMAC

The CDP1802 is a single chip implementation of the earlier CDP1801/CDP18101 two-chip pairing. Its main novelty at the time of launch was fabrication using CMOS technology (at a time when most microprocessors were being made using NMOS). Its internal register architecture is also a little bit different from most contemporary processors. It is well endowed with registers – sixteen 16-bit to be precise – any one of which can be designated the Program Counter and another the Stack Pointer. Unusually for processors of the era it also had rudimentary DMA capabilities.

With just one exception, every possible 8-bit instruction opcode is valid. This means that attempting to disassemble data will usually produce sequences of "code".

RCA/Intersil CDP1805 and CDP1806

Using the one invalid opcode in the CDP1802 instruction set as a prefix instruction allowed the instruction set of the CDP1805 and CDP1806 to be expanded over that of the similar CDP1802 COSMAC.

Microchip PIC16F83 and PIC16F84

The Microchip PIC16F83 and PIC16F84 are both members of the Microchip PIC16CXX family of 8-bit microcontrollers. These devices include on-board flash memory for program storage. Other members of the family have ROM instead of flash memory. These are known as the PIC16CR83 and PIC16CR84. The processors are classified as "8-bit" due to the basic size of data transfers. However, program memory is organised in 14-bit words with each instruction occupying a single 14-bit word. **DASMx** assumes that the code image contains these 14-bit words, each aligned to a 16-bit boundary – each 16 bits of the code image containing the 14 actual bits with the top two bits set to zero. These 16 bit words are assumed to be in little endian format.

The PIC16F83 *et al* represent just a common example of processors in the PIC16CXX family. They have an expanded instruction set over the PIC16C5X, for example. Consequently, **DASMx** may be used to disassemble code intended for other PIC processors with some success. Future versions of **DASMx** may add explicit support for all these variants.

Assembler pseudo operations

Assembler pseudo operations (e.g. that to define a data word) are *not* in a standard style that matches the chosen processor. The pseudo-ops are common across all processor disassembly output. In general, the pseudo-ops follow Intel conventions:

- The ';' character to denote a comment;
- The ':' character following a label;
- **db**, to define a data byte, character or string;
- **dw**, to define a data word;
- **org**, to specify a starting address.

If these do not suit your preferred assembler, then use of search and replace in a text editor can probably effect the required changes.

Number format

Microprocessor manufacturers have chosen a variety of different formats[1] for representing hexadecimal numbers.

DASMx supports seven different hex number format styles. These are summarised in the table below, with an example in each case for the hex number F12C.

Number format	numformat parameter	Example
ARM	A	&F12C
Intel	I	0F12CH
Motorola	M	$F12C
RCA	R	#F12C
Signetics	S	H'F12C'
C language	C	0xF12C
Decimal	D	61740

DASMx chooses a default number format according to the CPU type setting. A numformat statement in the symbol file can override the default choice. The number format defaults for the processors supported by **DASMx** are given in the following table.

[1] Some sort of formatting is essential; otherwise a hex number starting with an alpha character could be confused with a label or symbol name.

Manufacturer	cpu parameter	Format
Microchip	PIC16F83	C language
Microchip	PIC16F84	C language
RCA	1802	RCA
RCA	1805	RCA
Signetics	2650	Signetics
MOS Technology	6502	Motorola
Rockwell	65C00	Motorola
Rockwell	65C02	Motorola
Rockwell	65C29	Motorola
Rockwell	65C102	Motorola
Rockwell	65C112	Motorola
Hitachi	6301	Motorola
Hitachi	6303	Motorola
Hitachi	6305	Motorola
Hitachi	63L05	Motorola
Motorola	6800	Motorola
Motorola	6801	Motorola
Motorola	6802	Motorola
Motorola	6803	Motorola
Motorola	6805	Motorola
Motorola	68HC05	Motorola
Motorola	6808	Motorola
Motorola	6809	Motorola
Intel	8048	Intel
Intel	8051	Intel
Intel	8080	Intel
Intel	8085	Intel
Zilog	Z80	Intel
Sharp	LR35902	Intel

The number formatting applies to all operands in disassembled instructions with the exception of small positive or negative offsets in 6809 index instructions. These are given as a signed decimal number.

Future enhancements

Whilst there is no guarantee that future versions of this disassembler software will be released, some or all of the following areas are likely to receive attention in any future version:

- Fixing any errors discovered in the instruction mnemonics or disassembly of an opcode to its instruction;
- Rationalisation of the pseudo-ops such that the assembler output can be fed directly into at least one common assembler without further text editing;
- Improved code threading (through use of a more complete emulation of the processor);
- Improved symbol table output in listing file;
- Specifying comments in the symbol file for inclusion in the output files;
- Additional memory map output in listing file;
- Better support for 8048 code greater than 2 Kbytes and for 8048 variants;
- Support for additional microprocessors;
- Support for further variants of the currently supported processors;
- Disassembly of commonly known "undocumented" instructions.

Fixing actual disassembly errors (if any are discovered) will be treated with priority.

Note that it is not currently intended to support platforms other than Windows 95/98/Me or Windows NT/2000/XP. In particular, there will be no 16-bit versions for DOS or any other 16-bit operating systems. If the demand exists, a Linux version may be produced.

Contacting the author

Feedback to Conquest Consultants may be made via pclare@bigfoot.com.

References

The following publications were referred to in the course of the development of **DASMx**. This may also be considered to be a useful reference list for anyone programming these processors at assembler level and/or inspecting the output of **DASMx**.

[1] *M6800 Microprocessor Applications Manual*, Motorola Semiconductor Products Inc., First Edition, 1975.

[2] *Hitachi Microcomputer Databook 8-bit HD6800 & 16-bit HD68000*, Hitachi Ltd., March 1983.

[3] *Programming the 6502*, Rodnay Zaks, Sybex, ISBN 0-89588-046-6, Third Edition, 1980.

[4] *6502 Assembly Language Programming*, Lance A.Leventhal, Osborne/McGraw-Hill, ISBN 0-931988-27-6, 1979.

[5] *6502 Assembly Language Programming*, Second Edition, Lance A.Leventhal, Osborne/McGraw-Hill, ISBN 0-07-881216-X, 1986.

[6] *R650X and R651X Microprocessors (CPU)*, Rockwell, 29000D39, Data Sheet D39, Revision 6, February 1984.

[7] *MCS6500 Microcomputer Family Programming Manual*, MOS Technology Inc., Second Edition, Publication Number 6500-50A, January 1976.

[8] *1984 Data Book*, Semiconductor Products Division, Rockwell International, March 1984.

[9] *TLCS-Z80 System Manual*, Toshiba, 4419 '84-05(CK), June 1984.

[10] *Microcomputer Components Databook*, Mostek, MK79778, July 1979.

[11] *Z80-Assembly Language Programming Manual*, Zilog, 03-0002-01, Rev B, April 1980.

[12] *The MCS-80/85 Family User's Manual*, Intel, ISBN 1-55512-009-1, 1986.

[13] *MCS-48™ User's Manual*, Intel, 9800270D, July 1978.

[14] *48-Series Microprocessors Handbook*, National Semiconductor, 1980.

[15] *Component Data Catalog*, Intel, 1980.

[16] *An Introduction to Microcomputers: Volume 1, Basic Concepts*, Second Edition, Adam Osborne, Osborne/McGraw-Hill, ISBN 0-931988-34-9, 1980.

[17] *Osborne 4 & 8-Bit Microprocessor Handbook*, Adam Osborne & Gerry Kane, Osborne/McGraw-Hill, ISBN 0-931988-42-X, 1980.

[18] *2650A/2650A-1 Data Sheet*, Signetics.