VS_DSP
Software Tools User's Manual

Revision 3.2

Aug 15, 2002

Revision history:

Rev. 3.2	Aug 15, 2002	vslink: allocation priority explanation
Rev. 3.1	May 10, 2002	VCC: packed strings, section 2.8.8
Rev. 3.0	January 28, 2002	Some reformatting
Rev. 2.9	June 8, 2001	VCC updated, VSSYM added
Rev. 2.8	March 9, 2001	VCC and coff2lod updated
Rev. 2.7	October 24, 2000	Explanations added to VCC errors/warnings
Rev. 2.6	October 13, 2000	Minor corrections
Rev. 2.5	August 22, 2000	VCC documentation updated
Rev. 2.4	June 29, 2000	Small changes to VCC documentation
Rev. 2.3	March 16, 2000	VCC and VSLINK documentation updated
Rev. 2.2	March 8, 2000	Copyright/disclaimer page added
Rev. 2.1	December 17, 1999	Updated text, added coff2lod
Rev. 2.0	November 17, 1999	Updated for software tools 2
Rev. 1.0	September 10, 1998	First release

Contents

Chapter 1

Introduction

This manual describes the VS_DSP software tools. The tools are:

- VCC - C Compiler
- VSA - Assembler
- VSAR - Archiver
- VSLINK - Linker
- VSSYM - COFF symbol lister
- VSOMD - Object module disassembler
- COFF2LOD - COFF to LOD file converter
- VSSIM - VS_DSP Simulator

VCC compiles a C source code into VS_DSP assembly language and into a COFF executable or object file suitable for further processing by for example the linker tool VSLINK.

VSA is a powerful and fast macro assembler for the VS_DSP processor. It contains a versatile preprocessor and supports symbolic expressions in place of constants.

VSAR is a archiver for common object file format (COFF) object files generated by VSA.

VSLINK is a linker for common object file format (COFF) object files generated by VSA and VCC. VSLINK can directly use the archive files created by VSAR as link libraries.

VSSYM is a symbol lister for common object file format (COFF) object files generated by VSA.

VSOMD is an object module disassembler for common object file format (COFF) object files generated by VSA.

COFF2LOD converts COFF object files to LOD files or uploads them to a VS_DSP board.

VSSIM simulates a defined VS_DSP architecture. Core parameters can be changed using a hardware configuration file. Cycle-based simulation can be controlled both interactively and using a command script file.

As additional material, chapters *Installing and First Steps Using VS_DSP Tools* and *VCC Programming Tips* are included.

Chapter 2

VCC - VS_DSP Optimizing C Compiler

2.1 Synopsis

`vcc [-S] [-h`*hw_desc*`] [-w`*num*`] [-f`*special*`] [-O`*level*`] [-I`*incpath*`] [-D`*define*`] [-o`*outfile*`] [`*infile*`]`

`vcc` is an ANSI-C compiler for VS_DSP core with a 16-bit and a 32-bit address space. `vcc` compiles ANSI-C source code into VS_DSP assembly language and automatically into a COFF executable or object file suitable for further processing by for example the linker tool `vslink`.

`vcc` also automatically runs the C preprocessor (currently an external program) on the input file. Currently an intermediate assembly file, which includes the original C code and moderately commented assembly code, is generated and the assembler is called automatically, unless the `-S` option is used. If needed, the core parameters can be changed using a hardware configuration file.

ANSI-C link libraries for the target system are provided. Both simulator and emulator versions of the libraries are available. In practice, the following are required to effectively use the compiler:

- `vsa` - the assembler, which is automatically called by `vcc`
- `cpp` - any C preprocessor, which is automatically called by `vcc`
- `vslink` - the linker to create executable programs
- `libc.o` and/or `libsim.o` standard libraries and the corresponding header files for the emulator and simulator (separate versions are available for 16-bit and 32-bit address spaces)
- `vssim` and/or `coff2lod` or `vsemu` to simulate and/or run the code in actual hardware

2.2 Options

`-o outfile`
Define output filename (object file)

`-S`
Generate assembly output only

`-h hw-desc`
Define a hardware configuration file where to read the hardware configuration from. If not defined, the standard hardware configuration file is used.

`-fno-cpp`
Do not run preprocessor

`-fsmall-code`
Use only 16-bit code address space

`-fhuge-code`
Code is 32-bit by default (implied __far)

`-fhuge-data`
Data is 32-bit by default (implied __far)

`-fseparate-data`
Section names according to file names

`-fusejmpi`
Use the instruction JMPI when jumping to __near code

`-I include_path`
Adds a directory to the include search path

`-D preprocessor`
Defines a preprocessor symbol

`-O level`
Define optimization level: 0 (off) .. 6 (max)

`-fprof proffile`
Enable usage of profile data in jump optimization

`-W num[,num]*` Enable warnings by the warning number

`-w num[,num]*` Disable warnings by the warning number

`-P num[,num]*` Promote warnings into errors by the warning number

`-p num[,num]*` Demote warnings into warnings by the warning number

The environmental variable $VSDSP_CPP defines the preprocessor command, default is "cpp". If you have gcc installed into your system, you can use "gcc -E".

The identifier __VSDSP__ will be defined in the preprocessor. This can be used in the C code to identify the target platform for conditional compilation.

If no output filename is given, the base of the input filename is taken and the appropriate extension is added (.a for assembly and .o for object file output). The option `-S` causes the compiler not to run the assembler.

If no input filename is given, the standard input is used.

If one or more profile files are defined, the code optimizer uses the information in jump optimization.

2.3 Main Features

2.3.1 Architectural Issues

The VS_DSP instruction set offers parallelism. Some instructions like constant loads, jumps, and loops can not have any parallel operations, but an ALU operation (a binary or unary arithmetic or logical operation, a multiply, or a multiply-accumulate) and upto two memory references (load or store) or a register-to-register move can occur in the same instruction. Instructions are rearranged to use this parallelism to reduce the code size and thus increase the execution speed.

The parallelism can be poorly exploited in program flow structures like `if-else`, but better in arithmetic operations. Special steps are taken in the code optimization to reduce the penalty of program flow changes on the overall parallelism.

The separation of registers into three register files (ALU, data address generation and control) requires that arithmetic and pointer types are allocated to appropriate registers. However, some pointer arithmetic and other manipulation must still be done in ALU. This requires register moves between the register files.

In a load/store architecture the operands are first fetched, the operation is performed, and then the result is stored. This isn't as such any slower than operations dealing directly with memory in other architectures, but it causes lower code density if a lot of loading and storing needs to be done. Because of this all data is kept in registers as long as possible.

Index register manipulation is done with post-modification only and there is no index register + constant offset addressing mode. However, accessing stack variables and function parameters requires pre-modification of the frame pointer. Code optimizer combines these modifications to the previous occurance of the index register: a separate pre-modification of a register is transformed into a post-modification.

A single post-modification can be in the range of -7..7, otherwise the modification value must first be loaded into the index register's pair (modifier register). This of course needs an extra constant load instruction in addition to requiring another register. This is why it is better to generate two -7 modifications for an offset value of -14: the same number of instructions but fewer registers will be used. In addition to this one or both of the modifications can be either merged or parallelized, unlike the constant load instruction, which always takes up the whole instruction word.

The three-stage pipeline causes jump delay slots. The next instruction to a jump is always executed before the instruction in the jump target. Also, if the jump is conditional (uses ALU flags), the relevant flags may not be changed by the previous instruction. This causes one delay slot before any jump that uses flags in addition to the actual control change delay slot. It is important for performance to try to fill both these slots in instruction scheduling and code optimization.

2.3.2 Current data types for 16/32-bit architecture

- `char`, `short`, and `int` are single-word types
- `long` and `float` are double-word types
- `double` is a triple-word type
- pointer types are single-word (zero-page data) or double-word (far data)
- Q15 and Q31 fractional types supported
- enumerations, arrays, unions, structures and bit-fields supported

2.3.3 Current extensions

- fractional integral types: `__fract short`, `__fract int`, `__fract long`
- force variable allocation to X/Y memories: `__x`, `__y`
- force allocation to 16-bit or 32-bit address spaces: `__near`, `__far`
- allow parameter and return value passing in registers
  `__fract long mac(register __fract *a, register __y __fract *b);`
- register variables
  `register __d long sum;`

2.3.4 Global optimizer features

- Constant expression evaluation
- Constant merging (e.g. 2+a+4 → 6+a, 2*a*3 → 6*a, 4*a/2 → 2*a)
- Expression rearrangement
  - delay constant loads to preserve registers
- Logical expression optimization
  - preserves short-circuit evaluation
- Subexpression caching, which is used in
  - common subexpression elimination
  - variable load elimination

2.3.5 Code generation features

- Temporaries, register store, and automatic variables use software stack
  - recursion fully supported
- Integral arithmetic generates inline code
  - 16-bit and 32-bit (40-bit intermediates in V2) arithmetic supported
- Floating point uses a link library
  - some builtin functions implemented as inline code
  - double/float has 16-bit exponent, and 31/15-bit mantissa + sign bit
- Uses loop hardware for loops if possible. This requires:
  - loop is of format: `for( init ; check ; update )` where
    - `init` inializes a loop variable with a constant
    - `check` compares the same variable with constant or variable
    - `update` uses the loop variable with ++, - -, += or -= with a constant
  - loop body does not change the loop variable (nor comparison) value
  - loop body contains no labels, `goto` statements, or `return` statements

  It is more efficient if the loop variable is not used at all in the loop body.
  Examples:
  - `for (i=10; i > -1; i--) { *d++ = *s++; }`
  - `for (i=0; i < limit; i+=3) { *d++ = s[i]; }`
  - `for (i=0; i != 200; i++) { sum += *x++ * *y++; }`
- Parameter passing possible in registers
- `switch-case` is generated selectively as cascaded jumps or a jump table
  - the implementation which uses less memory is selected
  - jump table implementation is constant-time, but also uses data memory
  - cascaded jumps use only code memory, but some cases are faster than others
- Control structures are generated with the minimum number of jumps
  - Jump condition reversal to minimize the number of jumps
- The original C source code is included in the intermediate assembly file

Conditionals, i.e. all flow-control expressions, optionally with short-circuit evaluation, are generated with the minimum number of jumps: one jump for each condition plus one jump for the whole expression, if fall-through does not contain the other target branch. The individual conditions are reversed when needed. The backend code optimizer further optimizes jumps.

The `switch-case` construct is generated selectively either as a jump table or as an `if-else` cascade. The selection is based on memory consumption. Because data memory is 16-bit and code memory is 32-bit, the required case density for a jump table implementation is less than 0.5. This means that half of the possible case values between the lowest- and highest-numbered case must be defined. Other values cause a jump to the default branch (if one exists) or out of the `switch-case`.

Because the current architecture does not have a barrel shifter, 16-bit shifts are performed in the multiplier unit. Constant shifts take a maximum of three instructions, variable shifts take a maximum of six instructions (possible optimization effects not taken into account). 32-bit shifts are performed one bit at a time unless a more efficient way is possible. A constant shift of a 32-bit value by 16 bits is performed in two cycles, which is usually reduced to one cycle (or no cycles at all) in the code optimizer.

The code generation prefers to generate instruction sequences that can potentially be shortened in the code optimizer. For example two index register modifications are preferred over a constant load plus one index register modification.

Because the original C source code is included in the intermediate assembly language file (source.a), the programmer can keep an eye on the generated code quality if necessary. It is also easier to take a generated function and then further optimize it in the assembly level, when the original source is there for reference.

2.3.6 Code Optimizer

After code generation the code optimizer works on the result, trying to reduce the number of operations, parallelize them and minimize the number of no-operation instructions. Code transformations can be divided into several groups.

- Allows Better Optimization
  - merge successive labels, remove unused local labels
  - move LDC's backwards
  - move single MV's backwards

  The transformations in this group do not themselves reduce the size or increase the speed of the code, but they rearrange it so that other optimization possibilities arise.

Reducing the number of labels decreases the number of code blocks and increases the sizes of the remaining ones. Larger code blocks provide better selection of operations for other optimizations.

Relocation of constant load and register move instructions also group together operations that can occur in parallel in one instruction, thus indirectly reducing the code size.

- Redundancy Removals
  - eliminate identical LDC's
  - eliminate redundant operations - result and flags not used
  - eliminate redundant loads/stores (takes volatile data types into account)

  To simplify the code generator, not all special cases are handled separately. Redundant code can be generated, but these transformations remove these unneeded constructs.

  If the result of an instruction is overwritten before it is used, it can be safely removed. Few operations of this kind exist after the code generation, but other optimizations can generate them. These other optimizations depend on the redundancy removal transformations to eliminate them.

  Some of the ALU operation merging transformations reduce the register usage. If a register was saved into stack to be used as a temporary, this save and the corresponding restore are redundant after the removal of the usage. Some of these can be removed by the redundancy removal transformations. Mostly this doesn't decrease the code size, but it will reduce the power consumption because less functional blocks are activated and memory accesses are reduced (although stack is almost always in internal memory). Some actual redundant variable loads are also eliminated.

- Parallelizing Transforms
  - move MV's backwards and parallelize
  - remove loads to NULL, move loads/stores backwards, merge index arithmetic
  - move loads backwards over stores with alias detection
  - merge ALU operations
  - move arithmetic operations backwards, parallelize

  Instruction parallelism must be exploited to increase the code density. These transformations take the generated operations and either merge two simpler operations (such as a register move performed in ALU and a subsequent operation on that value) or combine two operations (for example an ALU operation and a load) into one instruction.

Normally memory loads can not be moved over a store, if they use the same memory bus (X or Y). However, if the optimizer can be certain that a load and a store do not reference the same memory location, their execution order can be safely reversed, unless either one of the memory locations is marked volatile.

- Back-End Jump Minimization
  - eliminate chained jumps
  - eliminate chained jumps in subroutine call returns
  - remove jumps to the next instruction
  - remove unnecessary jumps by condition reversal

  Although all control flow constructs are generated with great efficiency, when they are generated back-to-back, some unnecessary jumps may arise. These transformations combine two jumps together to shorten the code and speed up its execution.

- Delay Slot and Inter-Codeblock Optimization
  - move LDC's forward to fill delay slots/NOP's
  - fill unconditional jump delay slots
  - fill delay slots from fall-through and/or jump target code block
    - head merging - the same instruction in both branches
    - jump probability for each jump condition predicted per-function basis. If more than 45% of the jumps taken, prefer jump target when moving instructions, otherwise prefer fallthrough. This minimizes the average execution time.
    - transform instructions in the jump delay slots to make them movable
    - if ALU-op in the delay slot, find parallel load
    - if load in the delay slot, find parallel ALU-op

  Most of the previous transformations have been intrablock optimizations. Delay slot optimization uses and preserves interblock dependencies. Moving instructions from one block to another causes more opportunities for operation merging and parallelization.

  The profiler output from the simulator can be used (`-fprof`) to assist the compiler in selecting the most often executed branch of the code. When the more probable branch is shortened, the average runtime is decreased.

- Code Merging
  - tail merging - move instructions from jump source

`If-else` constructs and some other code structures cause very similar operations to be generated at the end of their code blocks. Moving identical operations forward, combining two or more of them, reduces the code size and at the same time makes the target code block larger. Larger code blocks may contain more opportunities for merging and parallelization of operations.

- Hardware Loop Optimization
  - fill LOOP delay slots
  - move LDC's out of the hardware loop
  - move index register modifications out of the hardware loop

  Zero-overhead hardware loops are generated from a subset of `for` statements. For digital signal processing (DSP) applications multiply-accumulate loops are usually the most time-consuming parts of the code, so loops are handled as a special case in the code optimizer. Code that can be safely moved from inside of the loop to the outside is identified and moved.

  These optimizations partly compensate for the lack of high-level strength reduction transformations in cases where the loop contains only a small amount of code.

2.3.7 Extensions and differencies to ANSI-C features

- typed function declaration enforced:
  K&R-style declarations, i.e. typeless arguments are not allowed
- `float` is not automatically promoted to `double` in function calls, use `"%hf"` in `printf` for float types (Note: float is not fully tested).
- `static` variable storage is not automatically initialized to zero if there is no initializer specified. Specify a full or partial initializer if the initial state of the storage matters.
- packed string support, see section 2.8.8
- binary constants supported: `0b10100101`.

2.4 Link Libraries

Target-system (partial) libraries are provided. See section 2.5.14 for a list of the implemented modules. There are two versions for different code memory models:

1. `libc16` for 16-bit code memory (`-fsmall-code`)
2. `libc32` for 32-bit code memory

The 16-bit version only allows 64k words of code, but reduces the function call overhead and is thus better for digital signal processing tasks. Also, the 16-bit link library routines do not handle far data. You can still use far data in your programs, because 16-bit data and 32-bit data addressing can be mixed, but you can't directly manipulate it with the library routines.

Separate libraries are used for simulator (`libsim.a`) and emulator (`libc.a`).

2.5 Implementation-defined Behavior

This part documents implementation-defined behavior as the ANSI-C standard requires.

2.5.1 Translation

Diagnostic messages are printed to the standard output stream, unless the compiler output is directed there, in which case the messages appear in the standard error stream.

Diagnostic messages are in the following format:

- `file:line: warning num:` *specific message*
- `file:line: ERROR num:` *specific message*

A result file is produced even if any number of warning messages are produced, but not if any error message is produced. See warning and error message list for specific error messages.

2.5.2 Environment

The environment does not provide any arguments to the main function. If any formal parameters are defined in the main function prototype or declaration, their usage is undefined.

All streams are considered interactive, but the output may be buffered on the host side (with the emulator board).

2.5.3 Identifiers

All characters are significant in identifiers with and without external linkage. Case distinctions are significant in all identifiers.

2.5.4 Characters

The source and execution character set is ISO Latin-1. Multibyte characters are not supported.

The execution character set supports 16 bits. Source character set is directly mapped as a subset to the execution character set.

Plain `char` is considered `signed`.

2.5.5 Integers

Integers are represented as two's-complement numbers. `char`, `short` and `int` are 16-bit (one word), `long` is 32-bit (two words).

Converting integer to a shorter signed integer causes the value to get truncated. Converting an unsigned integer to a signed integer of equal length causes values that can not be represented to get truncated (they become negative).

Bitwise operations on signed integers behave exactly the same as the corresponding operations on unsigned integers, i.e. signedness does not affect the result.

Remainder on integer division is non-negative for non-negative division results and non-positive for negative division results.

Right-shifting a negative-valued signed integral type copies the sign, i.e. it is an arithmetic shift.

Pointer types are considered unsigned.

2.5.6 Floating Point

The `float` type has a 16-bit exponent and 16-bit mantissa, including the sign bit. Arithmetic performed between `float` types is done in this format for speed instead of conversion to `double` first. Because of the limited accuracy, this floating point type is not very usable. Future plans include changing the `float` format to 8-bit exponent and 24-bit mantissa.

The `double` type has a 16-bit exponent and 32-bit mantissa, including the sign bit. More testing has been performed for `double` than for `float`. Some trigonometric functions for `double` are included in the link libraries.

2.5.7 Arrays and Pointers

`size_t` is `unsigned int` for 16-bit address space (with `-fsmall-code`, i.e. libc16) and `unsigned long` for 32-bit address space (without `-fsmall-code`, i.e. libc32).

With 16-bit address space casting a pointer to an integer preserves the pointer so it can be without loss converted back to a pointer of the same or other type. With 32-bit address space a long is required. Casting a `short` or `int` integer type to a pointer preserves the value so it can be without loss converted back to an integer with the original value.

`ptrdiff_t` is `signed int` for 16-bit address space and `signed long` for 32-bit address space.

Array initializers can be fully bracketed, incompletely bracketed, or a mixture of both.

2.5.8 Registers, X and Y Memory

The `register` keyword with a register specification of the type `__a0` causes parameters to be allocated in the corresponding register. The register size and the type size must match.

Automatic variables can be assigned into registers in the same way. With the `register` keyword but without a register specification the compiler tries to allocate pointer types into index registers and integral types into arithmetic registers. This works with both variables and parameters.

Storage can be allocated from either X memory or Y memory by directly specifying `__x` and `__y`, respectively. The default is to allocate from X memory. Specifiers `__far` and `__near` can be used to specify/force 32-bit and 16-bit linkage, respectively. A pointer to a storage element specified as `__far` will be 32-bit, while a pointer to a storage element specified as `__near` will be 16-bit. Notice that the Y memory space is restricted to 64 kilowords, and is thus always `__near`.

Notice that a pointer to a far element is different than a far pointer to an element.

- `__far short * __near ptrToFar;`
- `__near short * __far ptrToNear;`

The first declaration specifies a 32-bit pointer which itself is placed inside the 16-bit address space, while the second example specifies a 16-bit pointer which is placed inside the 32-bit address space. The same applies to `__x` and `__y`.

- `__y short * __x ptrToY;`
- `__x short * __y ptrToX;`

The first declaration specifies a Y-memory pointer which itself is placed in X memory, while the second one declares a X-memory pointer which is put in Y memory.

Let us finally consider function pointers and register variables.

- `register __i2 void (*funcPtr)(void);`
- `void (* register __i2 funcPtr)(void);`

The meaning of these declarations may not be self-evident. The first tries to allocate a pointer to a function which returns a `void` value in address register `I2`, but as the sizes do not match (nothing vs. a 16-bit register), an error is generated. The latter declares a pointer to a function returning `void` and allocates storage from address register `I2`.

2.5.9 Structures, Unions, Enumerations, and Bitfields

If a member of a union object is accessed using a member of a different type, the behavior is undefined, unless the types are of the same size, in which case the bit pattern is copied.

Plain `int` bitfield is treated as a `signed int` type. `char`, `short`, and `long` are also allowed as bitfield integral types, but they have no direct relevance to the underlaying storage allocation. Specifying `unsigned` for bitfields is recommended, because it generates faster code.

The order of allocation of bitfields proceeds from least-significant bits to most significant bits.

A bitfield can not straddle storage-unit boundaries. If this was about to happen, the current allocation unit is completed and next one is started. If the bitfield specifies a field larger than can be represented in `int`, the current allocation unit is completed and storage for a full `long` type is allocated.

Enumeration types are represented as `int`.

Note that returning structures or unions from functions can lead to inefficient code.

2.5.10 Qualifiers

Any reference to an object whose type is qualified by the keyword `volatile` is considered an access to that object. A read or write will not be optimized and the order of accesses will not be altered.

2.5.11 Declarators

Any number of declarators can modify an arithmetic, structure or union type. The number of successive pointer modifiers in declarations is limited to 32, but you can have more by using typedefs.

2.5.12 Statements

Any number of `case` values are allowed in a `switch` statement.

2.5.13 Preprocessing Directives

An external preprocessor is currently used.

2.5.14 Library Functions

Target-system (partial) libraries are provided.

libc16 *libc16* provides routines that use the 16-bit-code-only calling convention for functions. This mode is used for simple applications where all of the code can be fitted into the 16-bit code address space. 32-bit addressing can still be used for data, but the library functions only supports 16-bit addressing.

libc32 *libc32* is used for applications where the code and/or data space needs to be bigger than 64 kilowords. The calling convention uses full 32-bit call and return addresses, and the library functions use 32-bit data addresses.

- errno.h
  - errno not supported as most errors are handled on the host side
- float.h
- limits.h
- math.h
- stddef.h
- assert.h
- ctype.h
  - ISO Latin-1 supported
- stdarg.h
- stdio.h
  - formatted input/output (*printf, *scanf) partially supported

  - available functions perform the operations in the host environment

- stdlib.h
  - memory management and similar functions are left to the application or OS
  - random() / srandom() added for better pseudo-random number generation

- string.h

2.5.15 Locale-specific Behavior

No locales are implemented.

2.6 Pragma Statements

VCC currently understands one `#pragma` statement:
`#pragma msg` *num* `[on|off]`

This pragma can be used to turn warnings on and off in the code, for example disable a specific warning for a portion of the code.

```
#pragma msg 137 off
    long __fract af = 0x5555eeee, bf = 0x11112222;
#pragma msg 137 on
```

Only messages that are warnings can be disabled.

2.7 Warning and Error messages

Individual warnings can be disabled with the `-w` option. A comma-separated list of warning numbers can be specified, and also several `-w` options can be used. Errors can not be disabled. Warnings can be enabled with the `-W` option: warnings 132, 150, 153, and 334 are disabled by default, because they can be mainly used for debugging to detect possible differences between systems.

It is also possible to promote warnings into errors by using the `-P` option. A promoted warning can be demoted back into a warning with the `-p` option. These options have no effect on genuine errors.

The following list contains the identification codes and explanations of the error and warning messages of VCC.

2.7.1 Syntax and Sematic Errors

`0 internal error`
Some internal assumptions failed in the code generation.

`0 Invalid expression`
Invalid expression encountered when evaluating an expression type.

`0 unknown operator`
Unknown operator encountered when evaluating an expression type.

`0 assembler stage failed`
Code was generated with no errors, but the assembler run failed for some reason.

`0 syntax error`
A syntax error was encountered in the source code. Sometimes the reported line number is not exactly where the syntax error is.

`0 Can't combine derived types -- a missing semicolon?`
If you leave out the semicolon after a structure type definition, you may get this error. Check that the semicolon is in the right place.

`1 invalid pragma line: '`*line*`'`
The pragma line has the wrong number of parameters or contains an invalid command.

`8 invalid type conversion`
The specified type conversion is not possible. For example an object type can't be converted into another object.

`9 undefined identifier '`*id*`'`
An identifier was used in an expression, but it has not been previously declared.

`10 need an integral value for array index`

`10 need a pointer for array base`
Array indexing needs the base to be of a pointer (or an array) type, and the index to be of an integral type (or vice versa, because of the definition of the indexing operation). This error is generated if this kind of pointer-integral pair is not used.

`11 Too many successive pointer references in declaration`
Only 32 consequtive pointer modifiers can be used in declarations. If you need more, use typedefs.

`12 '`*type*`' has an incomplete type`

`12 incomplete structure/union`

`12 invalid cast into an incomplete type`
A type is used, but it has not been completely defined before the usage. The usual reason is that an array type does not have the size defined, a structure or union does not have the field list defined, or a function type does not have the parameter list defined.

`12 invalid structure/union reference`
A structure or union indirection or member reference was specified for an incomplete or invalid type, or a type which was neither a structure nor a union.

`13 struct/union member name missing`
A structure or union indirection or member reference doesn't specify a name.

`14 undefined struct/union member '`*name*`'`
A field with the specified name was not found from the structure or union. Either the structure or union does not have that field, or the definition is incomplete.

`15 invalid function call`
A function call doesn't contain an address specification or its type is not a function or a function pointer.

`16 too few parameters for call`
A function call expression specifies less parameters than the function specification (prototype).

`17 too many parameters for call`
A function call expression specifies more parameters than the function specification (prototype).

`20 invalid constant expression for case`
Case selectors must have constant values and an integral type.

`30 parameter` *n* `is a pointer to a different page`

`30 assignment argument is a pointer to a different page`

`30 assignment argument is a pointer to a different object`
Because storage can be allocated from either X or Y memory, pointer types pointing to storage in different pages must not be mixed. In parameter passing and assignment the storage of the pointer itself doesn't matter.

`31 pointer to a constant object used as a pointer to a non-constant`

`31 pointer to a constant object assigned to a pointer to a non-constant`
Storage for constants may be allocated from ROM areas and trying to change constants does not work. Mixing pointers to constant and non-constant storage may sometimes cause subtle errors.

`33 initializer already specified for '`*var*`'`
One variable can only have one initializer, although it can be declared multiple times.

`34 braces missing from structure initializer for '`*var*`'`

`34 braces missing from union initializer for '`*var*`'`

`34 extra braces in initializer for scalar variable '`*var*`'`
Structure and union initializers require braces to delimit the initializer. Partial bracing is only allowed for arrays. Bracing is not allowed for scalar variables.

`34 malformed initializer`
This error is generated if dimensions in an array initializer do not match with the array type dimensions.

`35 initializer for an incomplete type (variable '`*var*`')`
Variables with invalid and incomplete types can not have initializers.

`36 control cannot reach this statement`
This warning is generated whenever the compiler knows that a certain statement can't be reached. The reason may be a premature `return` statement, infinite loop without `break` statements, or a constant flow control statement.

`37 duplicate statement label '`*label*`' (see `*file*`:`*line*`)`
Statement labels must be unique inside a function. If there are two labels with the same name, the location of the previous one is also displayed.

`38 control reaches end of non-void function`
A value must be returned by a function but the end of the function (fall-through) can be reached. A `return` statement should be added before the end of the function.

`40 break not inside loop or switch`
A `break` statement was reached but it isn't contained in a loop or a `switch` statement.

`41 case not inside switch`
A `case` or `default` statement was reached but it isn't contained in a `switch` statement.

`42 case expression not integral`
Case selectors must have constant values and an integral type.

`43 duplicate case value (see `*file*`:`*line*`)`
Case values in a `switch` must be unique.

`44 continue not inside loop`
A `continue` statement was reached but it isn't contained in a loop.

`46 duplicate default (see` *file*`:`*line*`)`
Only one `default` case is allowed for a `switch`.

`52 missing if expression`

`52 invalid if expression`
The flow-control expression in an `if` statement is missing or invalid.

`54 switch expression not integral`

`55 invalid switch expression`
The flow-control expression in a `switch` statement does not have an integral type.

`56 no case values for switch`

`57 empty body for switch`
A `switch` statement does not have any case values or does not have a body (e.g. `switch(1);`)

`59 invalid storage class`
Some storage class specifiers are mutually exclusive.

`60 invalid type specifier combination`
Some types are impossible, for example `float double` and `short long`.

`61 invalid array size` *size*
`(__near array can only be` *size* `words)`
Arrays in near storage can only be of certain size.

`62 ANSI-C prohibits the use of 0-size arrays`
Zero-sized arrays are not possible in ANSI-C.

`63 integral constant value needed for enum`
Enumeration values must be integral constant expressions.

`67 illegal object`
Variables with the type `void` are not possible.

`68 illegal object for structure/union`
Structure and union members can not be function types.

`69 structure/union includes an instance of itself`
Recursive structures and unions are not allowed. A missing asterisk (*) may be the reason.

`72 type clash in redeclaration of '`*type*`' (previous in` *file*`:`*line*`)`
A previous declaration of a type exists and the declarations are inconsistent.

`78 undefined statement label '`*label*`'`
A `goto` statement uses a label that is not declared in the function.

`79 enum type` *type* `has duplicate values` *enum1* `and` *enum2* `=` *value*
This warning is given if two enumerations in an enumeration type have the same value. If you do not want to have this warning, disable it using the `-w79` compiler option or the pragma statement `#pragma msg 79 off`.

`83 indexing (`*n*`) outside the array bounds (0..`*N*`)`
A constant index is outside the array.

`85 return type mismatch for function '`*func*`'`
The expression type in the `return` statement does not match with the return type of the function.

`92 statement without effect`
An expression statement without any side effects is encountered.

`93 possibly unused variable '`*var*`'`
A variable is not used, as far as the compiler can determine.

`94 variable '`*var*`' may be used uninitialized`
A variable could be used before it has been initialized, at least as far as the compiler can determine.

`99 constant object can not be changed`
An assignment tries to modify a constant object.

2.7.2 Expression Errors

`100 Register type (`*reg*`) and type size (`*size*`) do not match`
Variable or parameter specification defined a register, but the size does not match the type size.

`101 formal parameter` *n* `specifies a void type`
Only the first and only formal parameter should have the void type.

`102 parameter` *n* `has an incompatible type`
The actual parameter can not be automatically type converted into the formal parameter type. This happens for example when an object is passed instead of a pointer to the object.

`103 mismatching parameter types for selection statement`
The parameters in the selection statement (`?:`) are not compatible.

`104 invalid selector for selection statement`
The selector expression must be an integral or pointer type.

`105 invalid parameters for logical or/and`
Logical operations require integral or pointer types.

`106 invalid parameter for unary ~`
Unary not requires an integral type.

`107 invalid parameter for unary !`
Unary logical not requires an integral or pointer type.

`108 invalid parameters for binary |`

`108 invalid parameters for binary ^`

`109 invalid parameters for binary &`
Bitwise operations require integral types.

`110 unary & requires a non-register parameter`

`111 unary & requires a non-bitfield object`

`112 the parameter for unary & must be a lvalue`
The address-of operator required an object located in memory that is not a bitfield object. The parameter must also be a lvalue expression.

`113 invalid parameters for equality comparison`
Only arithmetic and pointer types have equality and non-equality comparisons. Comparison of pointers to the integral value zero is also allowed.

`114 invalid parameters for comparison`
Only arithmetic and pointer types have smaller-than and greater-than operations defined.

`115 invalid parameters for shift expression`
Bit shifting is only possible on integral types and by integral number of bits.

`116 Invalid parameters for binary +`
Only arithmetic and pointer-integral additions are possible.

`117 incompatible pointer types for binary -`
Pointer difference requires that both pointers have the same type.

`118 invalid parameters for binary -`
Only arithmetic, pointer-integral, and pointer-difference subtractions are possible.

`119 invalid parameters for binary *`
Multiplications are only defined for arithmetic types.

`120 invalid parameter for unary *`
The indirection operation requires a pointer type.

`121 invalid parameters for binary /`
Division is only defined for arithmetic types.

`122 invalid parameters for binary %`
Modulo is only defined for integral types.

`123 pre/post in/decrement expression must be a lvalue`
The post/pre increment/decrement expressions require a lvalue expression.

`124 pre/post in/decrement only valid for arithmetic or pointer type`
Increment/decrement expressions are only valid for arithmetic and pointer types that point to an object (other than void).

`125 invalid sizeof() expression`

`126 sizeof() of an incomplete type`

`126 incomplete/invalid type for sizeof()`
The size of the type or object could not be determined. Either the type or object does not exist, has an incomplete or invalid type, or is a function type.

`127 need a modifiable lvalue for assignment`
Assignment needs an expression which evaluates into an object address.

`128 invalid type for assignment`
For example function types can not be assigned. Function pointers can, however.

`129 invalid parameters for assignment`
Invalid type combination for assignment or operation-assignment.

`130 function '`*func*`' called without a prototype`

`133 parameters specified for a prototypeless function`
Take these warnings very seriously, if you are mixing near and far code or are using register parameters. If you are calling a function without a prototype, no parameter checking can be performed. Even when you are using stack parameters, the responsibility of passing correctly sized values falls on you.

`132 parameter` *num* `will be promoted`
If an arithmetic parameter needs a type conversion, this warning is generated. This warning is disabled by default, use `-W132` to enable.

`134 obfuscated C not recommended`
Although ANSI-C allows the indexing operator to have the base (pointer type) and index (integral type) in either order, using `index[base]` obfuscates the meaning of the code and should not be used.

`136 __fract converted to integral type (bitwise copy)`

`137 integral type converted to __fract (bitwise copy)`
Fractional types have values smaller than one and greater or equal to minus one and converting them into non-fractional integral types and vice versa is meaningless as such. Because of this, conversion between fractional and non-fractional integral types is handled as a bitwise copy of the bit-pattern. Note that this warning is not generated from explicit type conversions.

`138 float type used as a parameter in logical or/and`
Logical expressions can use pointer and integral types as truth values. If a floating point type is used, this warning is generated and the expression is converted into integral type.

`139 integral value converted to pointer`
This warning is generated when an integral type is implicitly converted to pointer type. Note that this warning is not generated from explicit type conversions.

`140 pointer value converted to integral`
This warning is generated when a pointer type is implicitly converted to integral type. Note that this warning is not generated from explicit type conversions.

`141 overflow in conversion from floating to integral type`
Type conversion in the constant expression evaluation caused an overflow.

`142 value` *float* `too large for fractional type`
Converting a floating type constant could not be performed because it was too large. Only values from [-1..1) can be converted into signed fractional types, [0..2) can be converted into unsigned fractional types. Note that support for unsigned fractional types is not fully implemented yet and may change in the future.

`143 unable to evaluate constant cast expression`
Evaluation of type conversion for a constant failed.

`144 bitwise not ('~') of a logical value is always true`
If a bitwise not is performed on a logical value, it evaluates to either -1 or -2. This warning is generated if this expression is used as a truth value.

`145 negative shifts undefined`
Negative shifts have undefined behavior. In the case of constant shifts, negative shifts can be detected, a warning generated, and the shifting corrected: the order of the shift is reversed. However, trying to use non-constant negative shifts will fail utterly.

`146 constant divide by zero`
Constant expression evaluation detected a division by zero.

`147 possible division by zero ignored`
Constant expression evaluation removed a division in the form of `0/a`. This removal is not valid if `a` is zero.

`148 unsigned constant value exceeds range`

`149 signed constant value exceeds range`
Constant value no longer fits into the type in the constant expression evaluation.

`150 shift amount is handled as a short quantity`
Only 16 bits are used as a shift amount. This warning is disabled by default, use `-W150` to enable.

`151 a pointer type cast into a shorter type`

`152 a function type cast into a shorter type`
Because mixing 16/32-bit pointers can cause hidden bugs, converting pointers into smaller types is reported. If you want to extract part of the pointer value, cast it first into `long` and then into shorter type: `(short)(long)pointer`.

`153 a type cast into a shorter type`
This warning is disabled by default, use `-W154` to enable.

`154 pointer to` *page* `cast into pointer to` *page*
If a pointer is implicitly converted to point to another page, this warning is generated. Explicit conversions between pages are allowed without warnings.

2.7.3 Code Generation Stage Errors

`200 symbol '`*sym*`' has an incomplete type`
Storage could not be allocated for an object because the size of the object is not known.

`201 invalid initializer`
Automatic (stack) variables can only be initialized if they are scalars, pointers, arrays, or they have constant initializers.

`202 constant strings need a char array type`
String initializer can only be used in conjunction with a pointer or array variable.

`203 constant strings require char array type`

`204 constant strings normally initialize char arrays`
String initializer can not initialize non-integral types, such as floating point types. The compiler allows you to initialize `char`, `short`, and `int` arrays with string initializers because they are the same size, but warning 204 is generated. Other types cause error 203 to be generated.

`205 bitfields can not hold pointers`
Bitfields can not hold pointers.

`206 invalid initializer for '`*var*`'`
The initializer has an invalid type, is not a constant expression when one is required, or the constant value can not be converted into a floating point initializer, because it contains a relocatable memory address.

`208 bitfield initializer can not include symbols`
The initializer can not be converted to bitfield value, because it contains a relocatable memory address.

`209 invalid type in initializer of '`*var*`'`
The initializer contained an invalid type.

`210 can't have initializers with this type of variable ('var')`
Variables with very strange types may not have initializers.

`211 invalid case type`
Case value must be 16- or 32-bit (non-fractional) integral type.

`212 return in a non-function`
`return` statement was encountered although the object was not a function. Actually, this is one of the errors that should be impossible to get.

`213 return without expression in a function returning a value`
`return` statement without expression was encountered, but the function should return a value.

`214 invalid for initialization expression`
`for` statement's initialization expression was not empty, but had an invalid type.

`215 invalid for update expression`
`for` statement's update expression was not empty, but had an invalid type.

`216 invalid for loop expression`
`for` statement's loop expression was not empty, but had an invalid type. Note: an empty expression means an infinite loop.

`217 invalid do expression`
`do` statement's loop expression had an invalid type.

`218 invalid while expression`
`while` statement's loop expression had an invalid type.

`220 invalid asm expression`
Note: inline assembly is not yet supported.

`222 automatic conversion from float to integral type`
The float value in an initializer was converted to an integral value.

`224 automatic conversion from integral to float type`
The integral value in an initializer was converted to a float value.

`225 auto variables changed to static in` *func*
Stack variables were changed to static variables for a function containing no recursion as requested with `-fauto-to-static` option.

`227 constant switch() expression`
`switch` selection expression is a constant.

`228 invalid operator for floating type`
Modulo operator is not defined for floating type.

229 invalid types (or not implemented)
Some type combinations are not implemented for division and modulo operators.

231 invalid assignment
Assignment has incompatible, invalid, or incomplete types.

235 register variable value not currently available
Too many registers were used and a register variable value has become temporarily unavailable. Rephrase the expression by changing the evaluation order or perform it in smaller pieces.

236 invalid symbol
A type name or a function was used as a variable.

240 too many register parameters for a function

241 register '*reg*' specified twice for a function
A register can be used only once in a formal parameter list. The same register can be used as a parameter and as a return value register.

243 currently only either __x or __y allowed (symbol '*sym*)'
Only either __x or __y is allowed, not both.

244 too large bit field *d* (*D* max)
Bit fields upto the size of the long type are possible. Larger field sizes are not allowed, after all, the full range couldn't be manipulated if the size were larger than the largest type size.

247 typedef not allowed in function prototype!
Creating new types can't be performed in function prototypes.

248 redefinition of symbol '*sym*' (previous in *file*:*line*)
A symbol in the same scope was declared twice.

249 typedef not allowed in function parameters!
Creating new types can't be performed inside a parameter list.

250 symbol '*sym*' hides previous declaration (*file*:*line*)
A symbol hides a previous declaration.

252 extra initializers for array '*sym*' (dim *d*)
More initializers were specified than there is room in an array.

253 initializer for an incomplete type (variable '*var*')
Variables with incomplete types can be declared, but these can't contain initializers.

254 '*type*': can't have initializers with this variable type '*type*'
If you manage to create some obscure types, it may be that the compiler is not able to create initializations for them.

255 integral value needed for dimension (symbol '*sym*')

256 constant value needed for dimension (symbol '*sym*')
Array dimension has to be a constant integral value.

257 typedef not allowed in structure field!
Creating new types can't be performed in structure definitions.

258 invalid base type for bit field
(integral type required)
Only integral types are allowed for bit field base types. Floating point and pointer types are not allowed, and it would be quite futile to specify structure or union types or arrays.

259 too large bit field for this type of variable
The base type is too small to include all the bit field bits.

260 integral expression needed for bit-field size

261 non-constant bit-field size

262 invalid bit-field size specification
Bit field specification needs to be a constant integral expression.

264 redefinition of type '*type*' (previous in *file*:*line*)
A type was redefined but the definition was not compatible.

265 redefinition of structure '*struct*'(previous in *file*:*line*):
member '*memb*' name differs ('*name*')
A structure type was redefined, but there were differences in the structure members. A member name was different between the definitions.

267 redefinition of structure '*struct*'(previous in *file*:*line*):
member '*memb*' type differs ('*name*')
A structure type was redefined, but there were differences in the structure members. A member type was different between the definitions.

268 redefinition of structure '*struct*'(previous in *file*:*line*):
member '*memb*' field size differs ('*name*')
A structure type was redefined, but there were differences in the structure members. A bit field size was different between the definitions.

265 redefinition of union '*union*' (previous in *file*:*line*):
member '*memb*' name differs ('*name*')
A union type was redefined, but there were differences in the union members. A member name was different between the definitions.

267 redefinition of union '*union*' (previous in *file*:*line*):
member '*memb*' type differs ('*name*')
A union type was redefined, but there were differences in the union members. A member type was different between the definitions.

`268 redefinition of union '`*union*`' (previous in `*file*`:`*line*`): member '`*memb*`' field size differs ('`*name*`')`
A union type was redefined, but there were differences in the union members. A bit field size was different between the definitions.

`269 redefinition of structure '`*struct*`'(previous in `*file*`:`*line*`): too few members`
A structure type was redefined, but there were less members in the new definition.

`269 redefinition of union '`*union*`' (previous in `*file*`:`*line*`): too few members`
A union type was redefined, but there were less members in the new definition.

`270 redefinition of structure '`*struct*`'(previous in `*file*`:`*line*`): extra members`
A structure type was redefined, but there were more members in the new definition.

`269 redefinition of union '`*union*`' (previous in `*file*`:`*line*`): extra members`
A union type was redefined, but there were more members in the new definition.

`271 redefinition of enumeration type '`*type*`' (previous in `*file*`:`*line*`)`
An enumeration type was redefined.

`272 specifying unsigned and signed creates an unsigned type`
If both `signed` and `unsigned` is specified, the result type will be `unsigned`.

`273 too few structure initializers for '`*struct*`'`
A structure initializer did not initialize all members.

`274 too many structure initializers for '`*struct*`'`
Extra initializers were specified for a structure.

`275 trailing NUL not included in initializer of '`*var*`'`
An array can be initialized with a string, which always includes a trailing NUL character. In this case, the NUL character is not included in the array.

`276 too large bit field, a full variable allocated`
If a bit field size is larger than the basic storage element size, i.e. 16 bits, a 32-bit storage location is allocated.

`277 uninitialized constant '`*const*`'`
A variable with a constant type can not be written to. Because of this, constants should always have initializers specified or their values remain unspecified.

`278 enum type '`*type*`' value '`*name*`' = '`*val*`' out of range`
The specified value for an enumeration does not fit into a `signed short` type.

`279 redundant keywords in type declaration`
A type declarator has the same keyword specified more than once. This can happen easily if typedef types are used.

`280 external identifier '`*id*`' can not be initialized`
Combining an external variable with an initializer is not allowed.

`281 enum type '`*type*`' contains no members`
An empty enumeration type is kind of useless. Note that you can have incomplete enumeration types; an empty enumeration list is different than missing enumeration list.

`282 value is not part of the enumeration type`
A constant value was cast into an enumeration type, but it is not part of the enumeration. Another case where you get this warning is if a case statement used a constant that is not part of the enumeration type used in the switch selector.

`283 hex constant too large for char, high bits may be lost`
Character constants can be inserted into strings using the hex escape "\x1234", but large values do not fit into the 16-bit character type.

`284 invalid number`
A floating point or integral constant was malformed.

`285 operation invalid for pointer to void`
Pointer arithmetic is not possible for void pointers.

`286 switch did not specify cases for all enum '`*enum*`' values`
You get this warning if not all enumeration values are handled in a switch and there is no `default` case.

`287 symbol '`*sym*`': no prototype declared for function pointer`
A function pointer has an empty formal parameter list.

`288 no statement after label`
The C syntax originally requires that there is a statement after each label.

`299 non-ANSI use of ellipsis punctuator`
The ellipsis (...) can only be used as the last formal parameter and it can not be the only parameter for a function.

`300 initializer specified for an external object`
If an object is declared `extern`, it should not have an initializer.

`334 short * short -> short`
If a program assumes that int is a 32-bit type, multiplying short types may present a surprise, because the result type, although int, is also a 16-bit type. This warning is disabled by default, use `-W334` to enable.

2.8 Specific Information about Implementation

2.8.1 Function Calls

Functions preserve all registers except

1. the return value register, if any (`A0`, `A`, or `A/B0` unless specified explicitly)
2. parameter registers, if values are passed in registers
3. the multiplier pipeline register `P`
4. `I7`
5. Guard bits - may be set according to base register value

Stack parameters are located so that the stack pointer register (`I6`) points to the last word of the first actual parameter when control is transferred to the function. All parameters are located in X-memory. `long` values are placed into memory lowest part first (i.e. in lower address). The caller removes the parameters from stack.

If a function is declared with the `auto` storage specifier, the caller advances the stack pointer before the call. In this case, the stack pointer points directly above the first parameter. In effect, the first `LDX(I6)+1,NULL` in the function is eliminated. Be extra careful with prototypes, if you use this feature.

2.8.2 Stack Frame

	I4 Offset	X	Y
new SP	..	rest of the parameters	unused
	..+M+1	start of call parameters	unused
	..+N	register save	register save
	..+2	3rd var	2nd var HI
	+1	1st var	2nd var LO
new FP	+0	new FP	old FP
	-1	LR0	I5
	-2	MR0	I7
old SP	-3	1st parameter	unused
	..-4	rest of the parameters	unused

2.8.3 Bitfield Allocation

Bitfield allocation proceeds from least-significant bits to most significant bits. The allocation is always performed from 16-bit words. A bitfield is always allocated fully inside

one word, it can't continue from one word to another. If a field larger than 16 bits is specified, storage for a `long` type is allocated.

```
struct NOSTRADDLE {
    unsigned var1:8;       /* bits 0..7 of the 1st word */
    unsigned var2:4;       /* bits 8..11 of the 1st word */
    unsigned long var3:24; /* 32 bits, 2nd and 3rd word */
    unsigned var4:4;       /* bits 0..3 of the 4th word */
};
```

Because bits are allocated from lowest to highest and multiword variables are also stored in memory from least-significant word to most-significant word, the size of the actual bitfield storage unit size does not usually matter.

2.8.4 Multiplications

Because `int` is a 16-bit type, and C type promotions convert `short` values to `int` for most binary operations, i.e. operations on `short` values have 16-bit results (int), and not 32-bit like in most other systems. Specifically this means that you should be very careful with code constructs containing multiplications, because their result usually do not fit in 16-bit `int`. To do things safely, one of the multiplication operands must be cast into the `long` type to assure that the result will be 32 bits long.

- Wrong: `(sVal1*sVal2 + sVal3) >> 14`
- **Right:** `((long)sVal1*sVal2 + sVal3) >> 14`

Note: as an implementation side-effect, specific down-shifts of multiplication results will shift the 32-bit result, although their result is 16 bits.

- Works, but wrong: `sVal1*sVal2 >> 14`
- **Right:** `(long)sVal1*sVal2 >> 14`

Multiplications are performed in fractional mode (shifting up the result by one bit) whenever either one of the values is fractional. Full precision is retained in the result, i.e. for 32x32-bit and 16x32-bit fractional multiplication, full 32 bits of result are calculated.

Fractional values can also be multiplied by integral values, in which case the fractional shift is not performed and the lower part of the multiplication gives the result, i.e. multiplying fractional value by integral value 2 will double the fractional value. For fractional multiplications, use either a cast to fractional type or a floating point constant. Because fractional types are higher in precedence than floating point types in promotion type selection, the floating point constant will be automatically converted into fractional number by the constant expression evaluator. The following statements are therefore equivalent:

- `f32_w = (__fract long)13408963 * f32_x;`
- `f32_w = 0.00624403497204185 * f32_x;`

Because of the as-if-rule of the C standard, the multiplication is performed with as short operands as possible, i.e. casts which do not affect the value are skipped when evaluating the operands. Depending on the values to multiply, 16x16-bit multiplication with either 16- or 32-bit result, 16x32-bit multiplication with 32-bit result, and 32x32-bit multiplication with 32-bit result are generated. Also, unsigned multiplications are supported, so that a cast from signed short to unsigned long does not cause a longer multiplication to be generated.

2.8.5 Fractional Division

32-bit and 16-bit fractional divisions (Q15/Q15 → Q15, Q31/Q31 → Q31) are supported. The result is also fractional. The absolute value of the dividend must be less than the absolute value of the divider for correct results. Currently you can't divide fractional values by integral values.

Fractional modulo operator is not implemented.

2.8.6 Generation of Constants

VS_DSP has two special registers called NULL and ONES. These registers are used when possible to generate constants in ALU (allowing other operations in parallel) instead of using a load constant operation, which occupies a full instruction word. This is the reason constants `0`, `1`, `-1` (`0xffffU`), and `32767` (`0x7fff`) are better than others.

Note that constant `0xffff` is by default signed, thus a long value. Append U to make it an unsigned short value (`0xffffU`).

2.8.7 Grouping of Values

If a program uses a lot of global state information, a lot of address loading will be generated. Each variable reference generates a load constant instruction. If multiple variables are accessed right after eachother, you can get rid of some of the address loads by combining the variables into a structure. Then only the first access needs to load the address, subsequent accesses can get the address by modifying the existing one in parallel with other operations.

2.8.8 Using Packed Characters

New in 1.12 - A special extension is provided to allow packing two 8-byte characters into one 16-bit word. If you declare a string constant with a leading escape $\backslash p$, the string will be packed and the type is unsigned short pointer instead of signed char.

```
unsigned short *packedStr = "\pThis string packs two bytes "
"into each word\n";
```

The first byte is packed into high bits of the first word, the second byte to the low eight bits of the first word and so on. If the string length is odd, the second to last word will have the lower eight bits set to zero. The string always ends in a `NUL` character, so that you can copy packed strings with the normal string functions.

Packed strings can be also used with initializers.

```
unsigned short packedStr[] = "\pThis string packs two bytes "
"into each word\n";
```

Notice that `sizeof()` and for example `strlen()` return the length of the packed string, not the number of original characters. The original size can be calculated with the following code.

```
int packedstrlen(__far const unsigned short *s) {
    int size = 2*strlen(s);
    if (size && (s[size/2-1] & 0xff) == 0)
        size--;
    return size;
}
```

Packed strings are mainly intended to be used in applications where a lot of strings have to be stored and manipulating them is not time-critical. Strings can be stored packed in EEPROM and then depacked into RAM for processing.

```
void unpackstr(__far char *d, __far const unsigned short *s) {
    register __d __far const unsigned short *sr = s;
    register __c __far unsigned char *dr = d;
    while (1) {
        register __b0 unsigned short sd = *sr++;
        if ((*dr++ = (sd>>8)) == 0)
            break;
        if ((*dr++ = (sd & 0xff)) == 0)
            break;
    }
}
```

Chapter 3

VSA - VS_DSP Symbolic Assembler

3.1 Synopsis

`vsa [-o` *objFile* `| -ol` *objFile* `| -mf` *memFile* `| -m] [-v] [-V] [-l` *listFile*`] [-L] [-c` *confFile*`] [-w] [-W] [-I` *preIncDir* `...] [-i` *postIncDir* `...] [-D` *ppSymbol* `...]` *sourceFile*

Command line options may appear in any order.

3.2 Description

VSA is a powerful and fast macro assembler for the VS_DSP processor. This documentation will cover all the details of the compiler not decribed in the VS_DSP User's Manual.

Before using VSA, it is a good idea to set the operating system environmental variable `VSDSP_DIR` to point to the directory containing the hardware configuration file `hw_desc` and memory configuration file `mem_desc`. In UNIX, this can typically be done with the following command:

```
setenv VSDSP_DIR /vsdsp/config
```

where the directory mentioned should be the correct directory. In a DOS window under Windows 95 or Windows NT the command is:

```
set VSDSP_DIR="C:\vsdsp\config"
```

All keywords may be written in either upper-case or lower-case. For the sake of clarity all of them are written in upper-case in this document.

3.3 Options

Zero or one link options (`-o`, `-ol`, `-m` and `-mf`) can be defined for one compilation. If none of the link file options is defined, the compiler will try to replace the source file name (e.g. `test.s`) with a corresponding object file name (e.g. `test.o`), and write the output in COFF format.

```
-o objFile
    Define the COFF object file to write the generated program
    to. If the program has no unresolvable references, the
    resulting file will be executable, otherwise it can be used
    by the VS_DSP linker to get an executable program.
-ol lodFile
    Output to the given lod file.
    The program must be executable, i.e. there must not be any
    unresolvable references.
-m  Use the default memory description file when compiling.
    An output file is created for each section of the program,
    named <sectName>.m, where <sectName> is the name of the
    program section.
    The program must be executable, i.e. there must not be any
    unresolvable references.
-mf memFile
    Use the given memory description file, otherwise like -m.
-v  Verbose mode on: tell about the compilation after finished.
-V  Verbose mode off: be quiet if nothing was wrong with
    the compilation. (default)
-l listFile
    Define a file where to output a verbose listing of the
    program. If neither this nor a -L is defined, no list file
    is written.
-L  Like -l, but write the listing to stdout.
-c confFile
    Define a hardware configuration file where to read the
    hardware configuration from. If not defined, the standard
    hardware configuration file is used. If no hardware
    configuration file is found, a warning is displayed and an
    internal default configuration is used.
-w  Turn warning of unused labels on.
-W  Turn warning of unused labels off. (default)
-I dir
    Add a directory before the current directory but after all
    other -I directories to the #include <file> search tree.
    A '/' may end the directory name, but is not required. This
    option may be repeated.
-i dir
    Add a directory to the end of the #include <file> search
```

```
    tree. A '/' may end the directory name, but is not
    required. This option may be repeated.
-D ppSym
    Defines preprocessor symbol ppSym. This can be used for
    conditional compilation with the preprocessor directive
    #ifdef. A value can be defined by using an equal sign
    (sym=value).
```

3.4 Examples

`vsa -o test.o test.s -L`
Reads test.s and creates a COFF file called test.o. Will also send a listing of the file to `stdout`.

`vsa -o test.o test.s -I first -i second_last/ -I second -i last`
Reads test.s and creates a COFF file called test.o. The search path for `#include <file>` commands is: "`first/`", "`second/`", "`./`", "`second_last/`", "`last/`".

3.5 Preprocessor

When `vsa` is started, the input is first fed through a C-like preprocessor. The C preprocessor reads its input, divides it into tokens, parses the input, replaces macros and definitions, inserts whitespace characters between the tokens, and outputs the result one character at a time to the main compiler.

It is important to understand that the preprocessor works on a token basis. A token is any reserved word or identifier, a number, a string, or a special character like '#', ',', '(', or '.'. A whitespace is always added between tokens, and a line is always converted by the preprocessor as follows:

Original:
`ldc 0,d1`

Output from the preprocessor:
`ldc 0 , d1`

If adding spaces between tokens is for some reason not desirable (like in some macros), a '#' character between two tokens can be added to force them not to be separated:

Original:
```
ld#c 0#,#d#1
```

Output from the preprocessor:
```
ldc 0,d1
```

The example above is not a particularly great one, but this feature can be really useful in some macros where the programmer wants to, for instance, define a register name that is to be used by the macro:

```
// Macro scales Reg1 up by n-bits. _ScaleUpCoef is used
// by this macro
#macro ScaleUp Reg1, Reg2, _ScaleUpCoef
        MULSS Reg1#1, _ScaleUpCoef
        ADD NULL, P, Reg2
        MULUU Reg1#0, _ScaleUpCoef
        ADD NULL, P, Reg1
        OR Reg2#0, Reg1#1, Reg1#1
#endm

        ScaleUp d,a,c1 // A valid call to the macro
```

Please notice that at this moment the parsing of '#' characters is only done at the output stage of the preprocessor.

3.5.1 Preprocessor directives

A preprocessor directive always starts with a '#' as the first non-blank character on the line, and it takes up the whole line. At this moment, the following preprocessor directives are supported:

`#include "file.i" | #include <file.i>`

`#include "file.i"`
Includes `file.i` in the current working directory as if the code in `file.i` was written instead of the `#include` command.

`#include <file.i>`
As the previous form, but may search the file from several directories. First, directories added with the command line directive `-I` are searched. Then the current directory is searched. Finally the directories added with `-i` are searched.

`#define x | #define x y`

`#define x`

Defines the preprocessor symbol `x`. Any instance of `x` in normal code will be removed.

`#define x y`

Defines the preprocessor symbol `x` to be replaced by `y`. If `y` is an expression, it is a good idea to enclose it in a pair of parenthesis, like `#define base (old+7)`.

`#undef x`

Undefines the preprocessor symbol `x`.

`#if expr`

Tests for preprocessor expressions. If the expression is true, the following commands and directives upto the next `#else` or `#endif` are performed. For more information on how expressions can be made, see section 3.5.3.

`#ifdef x`

Tests, whether preprocessor symbol `x` is defined or not. If it is, the following commands and directives upto the next `#else` or `#endif` are performed.

`#ifndef x`

Tests, whether preprocessor symbol `x` is defined or not. If it isn't, the following commands and directives upto the next `#else` or `#endif` are performed.

`#else`

Reverses the effect of an earlier `#ifdef`, `#ifndef` or `#else`. Note, that it is allowed to have multiple `#else`s for one `#ifdef` or `#ifndef`.

`#endif`

Ends a condition set up by an earlier `#ifdef` or `#ifndef`.

`#macro name [par1 [,par2 [,par3 ...]]]`

`#macro name`

Defines the macro symbol `name` and begins macro recording. Any instance of `name` in normal code will be replaced by the contents of the macro.

`#macro name par1 [,par2 [,par3 ...]]]`

Defines the macro symbol `name` and begin macro recording. Any instance of `name` with correct arguments in normal code will be replaced by the contents of the macro.

Example:

```
#macro multiply a,b
        (a*b)
#endm
```

Which will be called as follows:

```
multiply 2,3
```

A macro can consist of several lines. However, the last line feed is taken away from the macro. This will result in the fact that one-line macros like the one above can be used as a part of a program line. In the example case the macro would evaluated as `(2*3)` without a line feed.

When calling a macro, the parameters are not surrounded with parenthesis. Because of this, the end of the parameter list must be recognized differently. A macro parameter list is terminated by any of the following events:

- An end-of-line character is reached.
- A semicolon (";") is reached.
- The preprocessor is parsing the last parameter, and a comma (",") is reached.

It is not possible to define a macro inside a macro.

`#endm`

Ends macro recording.

3.5.2 Preprocessor constants

By default, the preprocessor has some predefined constants, that reflect the configuration of the core. Programmers can use these values to check whether the algorithm is suitable for the given core or to make it possible to write scalable code.

`COREVERSION`
The core version number from *hw_desc*.

`DATAWORD` (8..64)
The datapath size.

`DATAADDRESS` (8..23)
Address size (<= `DATAWORD`).

`PROGRAMWORD` (32)
Instruction size.

`PROGRAMADDRESS` (11..20)
Program address size (<= `DATAWORD`).

`GUARDBITS` (0..16)
Guard bits for accumulators.

`LOOPREGS` (0..8)
Number of loop register sets.

`ADDRESSMODE` (0..7)
Address mode. The following bits are defined:

- bit 0 = modulo (if set, `ADDRESSMODE_MODULO` is set to 1, otherwise 0)
- bit 1 = bitrev (if set, `ADDRESSMODE_BITREV` is set to 1, otherwise 0)
- bit 2 = <reserved>

3.5.3 Preprocessor #if and expressions

The VSA preprocessor's #if directive can handle complex expressions that can be used for conditional compilation. An #if expression has the following form:

```
expr    : and-expr
        | or-expr ".or." and-expr
        ;

and-expr: not-expr
        | and-expr ".and." not-expr
        ;

not-expr: compare-expr
        | not-expr ".not." compare-expr
        ;

compare-expr
        : bin-or-expr
        | compare-expr ".gt." bin-or-expr
        | string       ".gt." bin-or-expr /* Don't use */
        | compare-expr ".gt." string /* Don't use */
        | string       ".gt." string
        | compare-expr ".ge." bin-or-expr
        | string       ".ge." bin-or-expr /* Don't use */
        | compare-expr ".ge." string /* Don't use */
        | string       ".ge." string
        | compare-expr ".lt." bin-or-expr
        | string       ".lt." bin-or-expr /* Don't use */
        | compare-expr ".lt." string /* Don't use */
        | string       ".lt." string
        | compare-expr ".le." bin-or-expr
        | string       ".le." bin-or-expr /* Don't use */
        | compare-expr ".le." string /* Don't use */
        | string       ".le." string
        | compare-expr ".eq." bin-or-expr
        | string       ".eq." bin-or-expr
        | compare-expr ".eq." string
        | string       ".eq." string
        | compare-expr ".ne." bin-or-expr
        | string       ".ne." bin-or-expr
        | compare-expr ".ne." string
        | string       ".ne." string
        ;
```

```
bin-or-expr
        : bin-and-expr
        | bin-or-expr '|' bin-and-expr
        ;

bin-and-expr
        : add-expr
        | bin-and-expr '&' add-expr
        ;

add-expr
        : mul-expr
        | add-expr '+' mul-expr
        | add-expr '-' mul-expr
        ;

mul-expr
        : neg-expr
        | mul-expr '*' neg-expr
        | mul-expr '/' neg-expr
        | mul-expr '%' neg-expr
        ;

neg-expr
        : prim-expr
        | '~' prim-expr
        | '-' prim-expr
        ;

prim-expr
        : '(' or-expr ')'
        | pos-int-const
        ;
```

As can be seen from the syntax, the expression types are introduced in an ascending precedence order. Thus, an `add_expr` is of lower precedence than a `mul_expr`.

Notice:

- Preprocessor directives are not short boolean evaluated.
- An empty #if line is evaluated as FALSE.
- All truth expressions evaluate to either 0 or 1, where 0 is FALSE and 1 TRUE. This will not change in the future so it may be used by the programmer. Example: To see if at least 2 out of 3 conditions are true, the following expression can be used:
  #if ((f1 .eq. f2) + (f3 .eq. f4) + (f5 .eq. f6)) .ge. 2
- Unlike all other preprocessor directives, #If expressions are evaluated for macros and preprocessor replacements. However, the author strongly discourages redefining the word "if". In that case, preprocessor behaviour is undefined.

The following example code demonstrates how to use the #if directive:

```
#define one 1
#define two 2
#define sum (one+two)

#if sum .ge. 3 // Evaluates: (1+2) .ge. 3 -> 1, i.e. TRUE
        /* TRUE */
#else
        /* FALSE */
#endif

#if lfsr .gt. reference // Untrue, 'l' is before 'r'
#endif

#if string .lt. 3 // Dont compare names to numbers
        // with .gt. .le. .lt. .le.
#endif

#if string .ne. 3 // Always true, since string is always
// different from any number
// (unless string #defined to be 3)
#endif
```

3.6 Program lines

A program line consists of an optional label and an optional directive or an optional command line. This may be followed by an optional to-end-of-line comment.

A command line consists of one or more instructions that can be inserted in one VS_DSP instruction word. The instructions are separated with a semicolon.

The following lines are syntactically valid program lines:

```
.sect code,MyProgram
multi:
        add null,p,a
myJump: ldc 0x7fff,c0 // Starts the subroutine
                        // Intentionally empty line
        sub NULL,ONES,a0; ldx (i6)+1,c0
.end
```

3.7 Comments

VSA understands two kinds of comments: Block comments and to-end-of-line comments. C++ -conventions are used for the comments.

Block comments begin with a '`/*`' and they end with a '`*/`'. Block comments can not be nested.

To-end-of-line comments start with a '`//`' and continue until the end of the line.

3.8 Directives

`.SECT stype , name | .SECT stype`

Defines a new section. `stype` may be one of the following: `CODE`, `CODE_FAR`, `DATA_X`, `DATA_Y`, `DATA_XY`, `DATA_FAR`. If no name is defined, it will be the same as the type. If a section with the given name has already been defined, no new section is created, but the following code is output to the end of the section.

There must always be a `.SECT` directive before any code or data is written.

`.IMPORT label [,label [...]]`

Tells the compiler these labels are externally defined. This has no effect on lod or M files.

`.EXPORT label [,label [...]]`

Tells the compiler these labels are to be shown externally. This has no effect on lod or M files.

`.WORD expression | string , ...`

Tells the compiler there is raw data that must be inserted as is. The range of the words is -(1<<(NO_OF_BITS-1)) .. ((1<<NO_OF_BITS-1)-1). Strings are not null-terminated automatically. Example:

```
.WORD 17, -0b1011011, @+2, -2.2, @+(lbl1-lbl2), 0xFA
.WORD "Hello, world!\n\0"
```

`.UWORD positive_integer | string , ...`

Tells the compiler there is raw data that must be inserted as is. Note that only constant values (not expressions) are allowed. The range of the words is 0 .. ((1<<NO_OF_BITS)-1). Strings are not null-terminated automatically. Example:

```
.WORD 17, 0b1011011, 0xFA, "Hello, world!\n\0"
```

`.ORG positive_integer [, positive_integer]`

Tells the compiler to what address the next program or data word is to be compiled. The first value gives the link (virtual) address, and the second, optional value gives the load (physical) address. The link address must match the possibly defined alignment.

`.ALIGN positive_integer`
Specifies which alignment to use for a section and the next data generated into a section. The value must be a power of two, i.e. 1, 2, 4, 8, 16, 32,... If used before any data is generated into a section, the section start will be aligned according to the value. If some data is already generated into the section, generates enough empty space (if needed) so that the next code/data will be correctly aligned.

`.FRACT positive_integer`
Tells the compiler how many bits are to be used as the fractional part of fractional numbers. The default value is (NO_OF_BITS-1), i.e. 15 in the default core.

`.ZERO expr`
Adds given number of zero-valued data words.

`.BSS expr`
Reserves the requested number of words from a section. If a section does not contain initialized data, a COFF BSS section is generated, which uses no file space other than the section header.

`.END`
Every program must end with this directive, otherwise a warning is displayed. No lines after this one are read from the input file.

3.9 Expressions

Whereever the "VS_DSP Specification Document" mentions a `constant`, it can be replaced by an expression in VSA. An expression has the following form:

```
expr32  : expr
        | LO ( expr )
        | HI ( expr )
        ;
expr
        : bin-or-expr
        | label
        | label +- ( shift-expr )
        | label +- ( label - label )
        | label +- pos-int-const
        | label - label
        ;
```

```
bin-or-expr
        : bin-xor-expr
        | bin-or-expr '|' bin-xor-expr
        ;
bin-xor-expr
        : bin-and-expr
        | bin-xor-expr '^' bin-and-expr
        ;
bin-and-expr
        : shift-expr
        | bin-and-expr '&' shift-expr
        ;
shift-expr
        : add-expr
        | shift-expr "<<" add-expr
        | shift-expr ">>" add-expr
        ;
add-expr
        : mul-expr
        | add-expr +- mul-expr
        ;
mul-expr
        : neg-expr
        | mul-expr * neg-expr
        | mul-expr / neg-expr
        | mul-expr \% neg-expr
        ;
neg-expr
        : prim-expr
        | - neg-expr
        | ~ neg-expr
        ;
prim-expr
        : int-const
        | fract-expr
        | ( bin-or-expr )
        | ( label - label )
        ;
```

As can be seen from the syntax, the expression types are introduced in an ascending precedence order. Thus, an `add-expr` is of lower precedence than a `mul_expr`.

The label pairs in the `label - label` subexpressions must be from the same section. A special label name '@', that refers to the location of the current code word, is allowed instead of the first `label` in `expr`.

Examples of valid expressions are:

- `27`
- `0x3b + (1-5)*17`
- `lab1 - (6 | 8)`
- `lab1 + (23.6-7 >> 2) // Dangerous mixing of int and float`
- `lab1 + ((lab2 - lab3)*-5)`
- `lo(200000)`

3.10 Labels

Labels are used in defining symbolic names for addresses. A label definition looks like the following:

- Format 1 - `label_name:`
- Format 2 - `$positive_integer:`

3.10.1 Label format 1 - label_name:

A label of this kind may begin with one of the following letters: "A-Z", "a-z", "." or "_", followed by zero or more letters from the group: "A-Z", "a-z", ".", "_", "0-9".

A label definition can be in the beginning of any program line or alone on its own line. A label may also be used as a beginning of a constant expression. (see also Expressions).

An example fragment of a short program using labels:

```
        j myLoop
        nop
dblLoop:
        ... do something ...
        j dblLoop
        nop
myLoop: ldc 1,i0
        ...
```

An example of a program calling a subroutine:

```
        j store
        ldc @+1,LR0 /* Set return address in delay slot */
        ...
store:
        add NULL,p,a ; ldx (i6)+1,null
        stx a0,(i6)+1 ; sty a1,(i6)
        jr
        stx a2,(i6) /* Remember the delay slot! */
```

3.10.2 Label format 2 - $positive_integer:

A label of this kind begins always with a "$", followed by a positive integer number.

The second format can be used when the programmer wants to make a quick loop or something similar without reserving any global names for local labels. This format creates a local label name that is valid only until the next label of format 1. An example:

```
myFirstLab:    // Starts a new local label scope
    ...code...
$1:            // Definition of local label
    ...code...
    j $1       // Jump two steps backwards
mySecondLab:   // A new scope is defined
    ...code...
    j $1       // Jump two lines down to the next $1
    ...code...
$1:            // Here the new $1 is defined
```

If a local label is defined before the first proper label, the name will be generated as follows: `section_name + "@@" + label_number`

Otherwise, the label is generated using the following formula: `last_label_name + "@" + label_number`

Thus, the first $1 would translate in the above example to `myFirstLab@1` and the second to `mySecondLab@1`.

3.11 How the compiler works internally

This section discusses in depth how the compiler works internally. By reading this chapter the reader may avoid some common mistakes.

3.11.1 Steps needed to compile a program

This section goes through the steps made by the compiler to compile a whole program. The operations here are described in the order they happen in the compiler.

The command line options are read and processed.

The hardware configuration is read. If no configuration file is defined in the command line options, the hardware configuration is read from the file `hw_desc`, or if that fails, `$(VSDSP_DIR)/hw_desc`. On error, an error message is shown.

If a source file name is defined, the source file is opened for reading. If an error in opening the source file is encountered, an error message is shown and the compiler exits.

If a listing file name is defined, the listing file is opened for writing. If an error in opening the list file is encountered, a warning message is shown.

The internal compiler data structures, the lexical analyzer and the preprocessor are initialized. While initializing, the possible compiler extensions are read from the file `extensions`, or if that fails, from `$(VSDSP_DIR)/extensions`. At this point both upper and lower case versions of the extended commands are added as preprocessor macros.

The source file is read and compiled into memory. So far the expressions are not evaluated. Thus "3-5" is stored as a subtraction expression, not as the result "-2".

After the whole source file is read or too many errors are encountered, the lexical analyzer and preprocessor are reset.

If there were no errors this far, all expressions are resolved. If a faulty expression is found, or the output is something other than coff and the program can't be made executable, an error message is shown. If warning for unused labels is enabled, the warnings are shown.

If there were no errors this far and an output file name has been defined, the destination file is opened for writing. If an error in opening the destination file is encountered, an error message is shown.

If there were no errors this far the object file is written. If writing or closing the object file fails, an error message is shown.

The final message is shown, which tells either of the completion or failure of the compilation.

If the source file name was defined, the source file is closed.

The compiler internal structures are freed.

The compiler exits.

3.11.2 Steps needed to compile one program line

This chapter discusses in depth how a single line is compiled with the main emphasis on the preprocessor. The operations here are described in the order they happen in the compiler.

A line from the source file is read by the preprocessor.

The source line is broken into tokens with the help of the lexical analyzer. The lexical analyzer removes all comments at this point.

One of the following actions take place:

- If in normal mode, and the line starts with a '#', the preprocessor symbols are gone through. If the preprocessor symbol is '#macro, macro mode is initialized. However, if the preprocessor is in skip mode, only #ifdef, #ifndef, #else and #endif directives are handled. They, in turn, can set or unset the skip mode.
- If in normal mode, but the line doesn't start with a '#', the tokens are gone through one by one, and preprocessor symbols are looked for. If any are found, they are replaced with their corresponding data. If any replacement took place, the operation is redone. If it has to be redone more than 64 times, the compiler gives an error message. However, If the preprocessor is in skip mode, the line is cleared with the exception of the finishing line feed character before processing.
- If in macro mode, and the line reads "#endm", the last line feed character of the current macro is stripped, the current macro is closed down, and normal mode is resumed.
- If in macro mode, and the line doesn't read "#endm", the line is recorded as a macro line and all used macro parameters are recognized and remembered.

The program line tokens are again converted to text, and output from the preprocessor.

3.12 Hints

3.12.1 Fractional numbers

It is very easy to make errors using fractional numbers. Every fractional number is converted immediately to an integer corresponding to its value. Thus, the result of 6.0/2.0 is not 3.0, but 3. The result of 6.0/2 is 3.0. Also, it must be noted that fractional numbers work correctly only if they are held at the calculating range at all times. Thus, if the upper limit for a fractional number is < 4.0, an expression 6.0/2 will lead to an incorrect value.

3.13 A compilable example program

Here is a short example of a stand-alone, ready-to-compile program.

```
.sect code
    // Calculate a single-sample symmetrical FIR
    ldc 0x200,mr0                          // integer mode

    ldc vector,i0
    ldc 1,i1
    ldc vectorend-1,i2
    ldc -1,i3
    ldc multiplier,i4
    ldc 1,i5
    ldc (vectorend-vector)/2-1,c0

    sub a,a,a;ldx (i0)*,b0;ldy (i2)*,b1 // delay line vals
    add b0,b1,d1;ldx (i4)*,d0           // multiplier

    loop c0,@+3
    mul d1,d0 ; ldx (i0)*,b0;ldy (i2)*,b1
    //----
    add b0,b1,d1 ;ldx (i4)*,d0
    mac d1,d0,a ; ldx (i0)*,b0 ; ldy (i2)*,b1
    //----
    add a,p,a  // add the last multiplication result
               // A = 218 (0xda)
    j 0xface   // End vssim simulation
    nop

.sect data_x
```

```
.export multiplier
multiplier:
    .word 1,2,3,4,4,3,2,1

.sect data_xy
.export vector
    .word 0
vector:
    .word 1,20,3,-4,5,16,27,58
vectorend:
   .word 0 // this word is read, but not used

.end
```

Chapter 4

VSAR - VS_DSP Archiver

4.1 Synopsis

`vsar` [txsprvud] *archive* [*files*]

4.2 Description

`vsar` is an archiver for common object file format (COFF) object files generated by `vsa` to be used in the VS_DSP software development environment. The VS_DSP linker (`vslink`) can directly use the archive files created by `vsar` as link libraries.

There are also other tools for object file management: `vslink` (object module linker), `vssym` (symbol lister) and `vsomd` (object module disassembler).

4.3 Options

```
-t List the specified files in the order in which they
   appear in the archive.
-x Extract the specified archive members into the
   files named by the command line arguments.
-s Recreate symbol table. Obsolete option - this is
   done automatically each time the archive is changed
   or the symbol table does not exist.
-p Write the contents of the specified archive files
   to the standard output.
-v Provide verbose output.
-r Replace or add the specified files to the archive.
-u Update files. When used with the -r option, files
   in the archive will be replaced only if the version
   in the archive is older.
-d Delete files from archive.
archive
   The object library to be operated upon.
files
   Object files to add/remove/update/extract.
```

The options are generally compatible with the standard unix `ar` tool.

See the `vslink` manual page for a simple example on how to use `vsar`.

Chapter 5

VSLINK - VS_DSP Linker

5.1 Synopsis

`vslink [-rksv] [-p` *I*`[,`*X*`,`*Y*`]] [-L` *libpath*`] [-o` *ofile*`] [-l` *libname*`] [-c` *linkcmd*`]` *files*

5.2 Description

`vslink` is a linker for common object file format (COFF) object files generated by `vsa`. The linking process combines object files and routines from a link library and creates an executable file.

There are also other tools for object file management: `vsar` (object file archiver), `vssym` (symbol lister) and `vsomd` (object module disassembler).

5.3 Options

```
-r   Incremental linking - preserve relocations
-k   Keep relocation information
-s   Strip symbols
-v   Verbose, give extra information about linking
-L libpath
     Add a directory to the library search path
-o target
     Set the output filename, default is "a.coff"
-l library
     Define a link library to use
```

```
files
     Object files to be linked
-p page
     Physical address change to another page
-c linkcmd
     Override the default linker command file vslink.cmd
```

Linking process starts from the object files defined in the command line. Sections that have the same name are merged, provided that their flags match, i.e. they are of the same type. Local symbols are removed. New sections are created as needed.

If any symbol references remain unresolved, link libraries are used to resolve them. Link libraries are used in the order they are defined in the command line. They are searched from the directories defined with the `-L` option, under the name `lib`*libname*`.a`. Only directories defined this way are searched. If no directories are added, the current working directory is searched.

In the case of non-incremental linking, i.e. vslink is producing an executable file, the sections are relocated, so that they don't overlap. First all fixed sections (sections with the assembler `.org` directive) are allocated from their respective memories (the memory description file defines available memory). If fixed sections are successfully allocated, other, non-fixed sections are allocated starting from the first free memory block big enough in the order they are defined in the memory description file. The sections are then relocated to their new starting addresses. In incremental linking this isn't done.

In incremental linking, the resulting object file is then saved. In non-incremental linking, the object file should now be executable. Because relocations are not needed anymore, they are removed in this stage, unless the `-k` option is used to prevent this (needed if symbol information is required in the simulator disassembly). Also if the `-s` option is given, symbol information is stripped.

With the `-p` option you can change the physical start (load) address of an executable program. Code and data page numbers can be set independently.

5.4 mem_desc

```
MEMORY
{
 page 0: code:   origin = 0, length = 400h
         return: origin = faceh, length = 1, option = "quit"
 page 1: data_x: origin = 0, length = 400h
 page 2: data_y: origin = 0h, length = 400h
         io:     origin = 7000h, length = 8, option = "vsstdio"
}
```

Memory description file is not used when doing incremental linking. In non-incremental linking, the file is first searched from the current working directory. If it doesn't exist, or can't be read, the version in the directory pointed by the environmental variable VS-DSP_DIR is used.

Only entries having no option-field are used by the linker.

Memory block start addresses (`origin`) and lengths (`length`) do not currently have restrictions except that they may not overlap. However, the linker expects an entry to be either near or far memory, so if you have memory in e.g. 0x8000-0x18000, you should divide this to two memory block entries: one 0x8000-0x10000, the other 0x10000-0x18000.

Core parameters are compared when linking, if they exist in the object files or link library objects. Some backward compatibility is taken into account. Consider the case when an object file is added to another object file. The file to be added can have more restrictive parameters, i.e. the number of registers can be smaller, and some options that are selected in the original file may be disabled. The requirements are that the register numbers match and in essence all the features used in the object file are available in the original object file.

Currently the link library routines use on-demand loading of object files. This means that not the whole link library needs to be loaded into memory if just some of its parts are used. If an object file in a link library defines a symbol that is needed to resolve an undefined reference, the whole object is taken from the library.

If there are sections with the same name, but different types, a new section is created, and a dollar-sign and a number is appended to the original name.

The allocation order of sections is

1. Fixed sections - sections containing the .org directive
2. Near sections - sections that are inside the 16-bit address space
3. Far sections - sections that are inside the 32-bit address space

The allocation of memory is performed in the same order as the memory areas are defined in the memory description file. Allocation uses first-fit.

Far code containing loop instructions can not be allocated inside the first 0x3100 words of each far code memory page. This is why sometimes the linker may give an out of memory error although at first glance there seems to be enough memory left.

Forcing Far Code to External Memory

If you want to have far code to be primarily allocated into far (external) memory, put the far memory area first in the mem_desc file. If you want all near functions automatically allocated into internal memory, put that memory area next. Only if the internal memory becomes full, will the linker begin to use the external near memory.

```
MEMORY
{
  page 0: code_far: origin = 10000h, length = 30000h
          int_prog: origin = 2000h, length = f00h
          near_ext: origin = 4000h, length = c000h
  page 1: int_x:    origin = 0000h, length = 4000h
          intx2:    origin = 4000h, length = 8000h
          const_x_far:   origin = f00000h, length = 80000h
  page 2: int_y:    origin = 0000h, length = 4000h
}
```

The behaviour is a little different if the far code memory block is declared later than near code memory in mem_desc. Because near sections are allocated first, they will be located as before, but now the remaining near code memory is filled with far code. However, this is not usually beneficial as the near code memory is usually internal RAM and the far code memory is usually FLASH. RAM contents must be loaded at startup, which slows down the boot and requires some extra FLASH space.

If internal memory is required for a function becaue of the execution speed, the linker command file `vslink.cmd` can be used.

5.5 vslink.cmd

A linker command file can be used to map and force sections into specific memory areas as found in `mem_desc`. This file is searched from the current working directory and then from the configuration directory (VSDSP_DIR).

A hash mark at the start of a line marks a line as a comment. Two commands are currently supported. `map` gives a preferred memory area for a section (`map section_name memory_area`) and tries another memory area only if this fails. `force` performs similarly, except that the linking process fails if the section can't be placed into the specified memory area.

```
force data_x data_x
force data_y data_y
force const_y const_y
force const_x const_x
force appcode int_prog
map Interrupt int_prog
map main code_far
```

5.6 An Example

The following example consists of four source files and a makefile.

Makefile

```
exe.o: libtest.a startup.o main.o
        vslink -vk startup.o main.o -o exe.o -L./ -ltest

libtest.a: routine1.o routine2.o
        vsar ruv libtest.a routine1.o routine2.o

main.o: main.asm
        vsa main.asm -o main.o

startup.o: startup.asm
        vsa startup.asm -o startup.o

routine1.o: routine1.asm
        vsa routine1.asm -o routine1.o

routine2.o: routine2.asm
        vsa routine2.asm -o routine2.o

clean:
        rm -f *.o libtest.a *~
```

startup.asm

```
.import _main
.sect code
.org 0
reset:
        J       _main
        LDC     0x300,i6       // Setup stack

.org 8
interrupt:
        /* prologue */
        LDX     (i6)+1,NULL     // increment stack pointer
        STX     LR1,(i6)
        /* Interrupt routine here */
        /* epilogue */
        LDX     (i6)-1,LR1      // decrement stack pointer
        RETI
        NOP
.end
```

main.asm

```
.export _main
.import storeP
.import restoreP

.sect   code
_main:
        LDC     -1,b0
        LDC     1234,b1
        CALL    storeP          // store P-register
        MUL     b0,b1


        CALL    restoreP        // restore P-register
        MAC     b1,b1,a

        J       0xface          // exit simulator
        NOP
.end
```

routine1.asm

```
.export storeP
.sect code
storeP:
    ADD NULL,p,a ; LDX (i6)+1,NULL // Free stack pos.
    JR
    STX a0,(i6) ; STY a1,(i6)
.end
```

routine2.asm

```
.export restoreP
.sect code
restoreP:
        LDX (i6)-1,a0 ; LDY (i6),a1
        JR
        RESP a0,a1
.end
```

The executable file is built by the `make` tool. First, the files `routine1.asm` and `routine2.asm` are compiled (`vsa`) and they are used by the VS_DSP archiver tool (`vsar`) to create a link library.

The files `startup.asm` and `main.asm` are compiled next, and finally all routines are linked together with `vslink`, producing an executable file `exe.coff`. A sample `make` run is shown below.

```
iir 101% make
```

```
vsa routine1.asm -o routine1.o
vsa routine2.asm -o routine2.o
vsar ruv libtest.a routine1.o routine2.o
vsar: reading libtest.a
r - routine1.o
r - routine2.o
vsar: writing libtest.a
vsa startup.asm -o startup.o
vsa main.asm -o main.o
vslink -vk startup.o main.o -o exe.o -L./ -ltest
vslink: I'm verbose.
vslink: I have 2 files to process.
vslink: main.o: section types for code don't match
vslink: creating new section "code$0"
vslink: undefined symbols, using link libraries
vslink: trying library .//libtest.a
vslink: found symbol storeP from library test
vslink: test: section types for code don't match
vslink: merging data to section "code$0"
vslink: found symbol restoreP from library test
vslink: test: section types for code don't match
vslink: merging data to section "code$0"
vslink: binding fixed section code to 0:0x0..0xc
vslink: binding section code$0 to 0:0xd..0x1f
vslink: keeping relocations
vslink: writing exe.o
```

Chapter 6

VSOMD - VS_DSP Object Module Disassembler

6.1 Synopsis

`vsomd` [-o *listfile*] *file*

6.2 Description

`vsomd` is an object module disassembler for common object file format (COFF) object files generated by `vsa` to be used in the VS_DSP software development environment.

There are also other tools for object file management: `vslink` (object module linker), `vssym` (symbol lister) and `vsar` (archiver).

6.3 Options

```
-o listfile   Redirect the output to this file
-n            No addresses
file          Object file to be disassembled
```

Contents of the named object file or library/archive are disassembled and printed either to the standard output or to the file specified with the `-o` option. Instruction and data word addresses can be suppressed with the `-n` option. This in especially convenient when comparing two object files, when most of the code matches, but addresses don't.

Chapter 7

VSSYM - VS_DSP COFF Symbol Lister

7.1 Synopsis

`vssym` [-q] [-Q] [-o *listfile*] *file*

7.2 Description

`vssym` is a symbol lister for common object file format (COFF) object files generated by `vsa` to be used in the VS_DSP software development environment.

There are also other tools for object file management: `vslink` (object module linker), `vssym` (symbol lister) and `vsar` (archiver).

7.3 Options

```
-q            Quiet, does not print relocation information
-Q            Really quiet, only print total memory usage
-o listfile   Redirect the output to this file
-n            No addresses
file          Object file to be disassembled
```

Relocation and symbol information of the named object file or library/archive are printed to the standard output or to the file specified with the `-o` option. Relocation information can be left out with the `-q` option.

7.4 Output Format

1. Section-by-section listing of section headers and relocation entries

```
Section 3: vaddr 0x1d00, paddr 0x1d00, size 0x04f, flags 0x240, name int_x
```

   - the section number
   - virtual link address - where the code can be run in
   - physical load address - where to load the code to
   - the section size in words
   - section flags (code, data x/y, fixed, far, bss)
   - and section name

```
Relocations
1: type 0, vaddr 0x000d, symbol [ 9], r_curval 0x1d00 (_intVectors)
2: type 0, vaddr 0x0016, relative to section 6
```

   - relocation number
   - type - 0 = 16-bit, 1 = 32-bit low part, 2 = 32-bit high part
   - the offset of the word to relocate from the start of the section
   - (a) either a section reference, or
     (b) a symbol reference - the symbol number, value, and name are shown

2. Symbol listing

```
[Index]   Value      Size    Type  Bind  Other Section Name
[1]     |      7489|       0|NULL |GLOB |0    |3      |_readyQueue
[2]     |      8594|       0|NULL |GLOB |0    |6      |_AddTimer
[3]     |      8429|       0|NULL |GLOB |0    |6      |[_SetTaskPri]
```

   - symbol index - the same value is used in relocations
   - symbol value
   - object size - 0, size information is not currently supported
   - symbolic type - NULL, type information is not currently supported
   - scope info - LOCAL means a local symbol (eliminated when linking)
   - number of auxiliary entries - 0, not currently supported
   - the number of section defining the symbol, or DEBUG / UNDEF / ABS
   - and the symbol name

   The symbol name is included in brackets if it is not referenced from that object file.

3. code and data memory summary

```
total code memory size 0x0729 (1833) words
total    X memory size 0x1f6e (8046) words
total    Y memory size 0x1d05 (7429) words
```

Chapter 8

COFF2LOD - COFF-LOD Converter

8.1 Synopsis

`coff2lod [-a] [-q] [-z] [-l` *exec*`] [-e` *entry*`] [-p` *dev*`] [-x` *crystal*`]`
`[-s` *speed*`] [-ts` *targetspeed*`] [-sm` *speedmultiplier*`]`
`[-c basic|gps|pem|ngps] [-b` *I-mirror-base*`]`
`[-bc` *boot-character*`] [-bc any|none] [-end` *string*`] [-o` *file*`]`

8.2 Description

`coff2lod` is a COFF to LOD converter program. It can also be used to upload COFF executable to a VS_DSP chip using a serial port. After the executable is sent, the program goes into receive mode, where all bytes received are printed on the screen in hexadecimal or ascii (`-a` option). With the `-q` option the program exits after sending the executable.

The speed multiplier option (`-sm`) can be used with serial cards that have 2x or 4x speed options. The target speed option (`-ts`) together with chip type (`-c`) and crystal (`-x`) options can be used to change the default bootup serial speed. E.g.

```
coff2lod -s 9600 -c basic -x 26000 -ts 115200 -l flash.coff
```

8.3 Options

```
-a        ascii mode -- output the serial data as ASCII
-q        quit program after sending the COFF file
-z        do not convert/send zeros
-l        file to convert/send
-e        execution address
-p        port/device number
-s        serial port initial speed (default: 57600)
-sm       serial port speed multiplier (1, 2, or 4)
-ts       serial port target speed
-x        clock crystal in kHz
-bc       value which is expected from chip, or 'any', or 'none'
-c        chip type - 'basic', 'gps', 'pem', or 'ngps'
-b        sets the mirror base for 'gps'
-end      quits when string received in ascii mode, 8 chars max
-o file   output filename if conversion is requested
```

Chapter 9

VSSIM - VS_DSP Simulator

9.1 Synopsis

`vssim` [`-m` *mem-desc*] [`-h` *hw-desc*] [`-l` *executable*] [*lod-file*`.lod`] [*coff-file*`.coff`] [*coff-file*`.cof`] [`-c` *cmd-file*] [`-log` *log-file*] [`-prof` *prof-file*] [`-rt` *rtrace-file*] [`-i` *interrupt-mode*] [`-r` *rounding-mode*]

9.2 Description

`vssim` simulates the VS_DSP core and RAM memories connected to it. Core parameters can be changed by using a hardware configuration file. Different memories are defined in a memory description file. LOD-format or COFF-format executables generated by the `vsa` assembler can be loaded into the simulator. Cycle-based simulation can be controlled both interactively and using a command script file. A command log can be used so that the simulation actions can be recorded and the simulation rerun in batch mode later.

9.3 Options

```
-m mem-desc
   Define the memory configuration file
-h hw-desc
-hw hw-desc
   Define the core configuration file
-l executable
   Object file to be loaded. In order to be executable,
```

```
   the file must not contain any unresolved references
-c cmd-file
   Command script file to execute
-log log-file
   Command log file
-prof prof-file
   Execution profile file
-rt rtrace-file
   Register trace file (used by VLSI Solution for hardware
   verification)
-i interrupt-mode
   Interrupt mode selection
-r rounding-mode
   Rounding mode selection
```

Three different interrupt handling modes can be selected with the `-i` option. If mode is 0, interrupt requests happening during interrupt handling will be lost. If mode is 1, interrupt requests happening during interrupt handling will cause one interrupt only (edge-triggered mode). If mode is 2, each interrupt request happening during interrupt handling will cause one interrupt (level-triggered mode). Note that interrupts and resets are handled equally in this mode. If mode is 2, resets and interrupts will both be queued, so that it is possible that a reset is performed on the wrong cycle.

Four rounding modes can be selected with the `-r` option. This option overrides the setting in the hardware configuration file, and the possible setting in the automatically loaded executable. The rounding modes are 0 - truncation, 1 - rounding, 2 - convergent-to-0 and 3 - convergent-to-1.

`vssim` also supports auto-detection of parameters. If a command line parameter has a suffix "`.lod`", it is assumed to be a LOD-file and is automatically read in before the simulation starts. Similarly, for suffix "`.coff`", "`.cof`" or "`.o`", a COFF object file is assumed. If the suffix is "`.cmd`", a command file is assumed.

If no command file is given with `-c` option, nor through the auto-detection feature, the simulation commands are read from the standard input and the simulation is considered interactive. Interactive simulation has higher fail level, thus redirecting the input of the simulator from a file is **not** the same as to give a command file name to the simulator.

9.4 Environment

Memory description file and hardware description file are first searched from the current working directory. If they don't exist, or can't be read, the versions found in the directory pointed by environmental variable `VSDSP_DIR` are used. If the files can't be located in any of these places, default core parameters are used.

If a LOD-file or a COFF object file is defined in the command line, core parameters are taken from there and only the memory description file is read.

9.5 Files

The file formats handled by the simulator are the hardware description file (hw_desc), the memory description file (mem_desc), the LOD and COFF executable files, the simulator command file, the simulator log file, the profiler output file and the register trace file.

9.5.1 mem_desc

```
MEMORY
{
 page 0: code:   origin = 0, length = 400h
         return: origin = faceh, length=1, option = "quit"
 page 1: data-x: origin = 0, length = 400h
 page 2: data-y: origin = 0h, length = 400h
 page 3: infil:  origin = 4800h, length=1, option = "<indata"
         outfil: origin = 4801h, length=1, option = ">results"
         io:     origin = 4810h, length=8, option = "vsdspio"
}
```

The default name for the memory description file is `mem_desc`, but any name can be used, provided that the name is given with the `-m` option on the command line. If the memory description file is not found in the current working directory, it is searched for in the directory pointed by the `VSDSP_DIR` environmental variable.

Each memory entry in the memory configuration file consists of a maximum of five fields. The first one, `page`, defines the memory page for that entry. The second field defines a logical name for the entry, e.g. `datax`. The third field, `origin`, sets the starting address, and the fourth field, `length`, defines the length of the memory block. An optional field, `option`, may be used to define special functions for memory entries. As can be seen, `page` definition can be omitted when the page does not change.

Currently three special functions are available:

- `quit`
- `>`
- `<`
- `vsdspio`

`quit` defines an end-address for program execution. If instructions are fetched from memory address that has this option, the simulation is stopped with success return value. This option is only available in page 0 of the program memory space.

An option starting with < defines an input file, and option starting with > defines an output file. These options are only available in pages 1, 2, and 3. Pages 1 and 2 are data memories X and Y, respectively. Page 3 refers to both of them, meaning that page 3 can be accessed from both X- and Y-bus.

`vsdspio` is a special module that provides stream output and cycle counter. Functionality is as follows:

```
Address   Read             Write
0x0       data = Cycles    Cycles = data
0x1       reads stdin      writes to stdout as ASCII
0x2       reads stdin      writes to stdout as decimal
0x3       random value
0x4..7    reserved
```

New `option` types can be created for memory-mapped I/O devices.

Memory block start addresses (`origin`) and lengths (`length`) do not currently have restrictions except that they may not overlap.

9.5.2 hw_desc

```
# core parameters for default version
dataword        16 // Datapath size
dataaddress     16 // Address size (<= dataword)
programword     32 // Instruction size (only 32 now)
programaddress  16 // Program address size (<= dataword)
multiplierwidth 16 // Multiplier input width (not used)
guardbits        8 // guard bits for accumulators
indexregs        8 // Number of address/modifier registers
aluregs          8 // Number of ALU registers
modifieronly     0 // 0=interchang. - I0(M1)<>I1(M0) etc.
                   // 1=odd mod-only I0->I1(M0) I2->I3(M2)
                   // 2=separate modif. regs I0->M0 I1->M1
loopregs         1 // loop registers sets (0, 1 valid)
addressmode      1 // 0= +-m only, 1,2 = +-n%, 3 = bit-rev
modemask         0x077f

version          2 // 0 for version 1, 2 for v2
```

The hardware description file defines the target VS_DSP architecture. This file is read by both the assembler and simulator. Architecture restrictions must be obeyed.

9.5.3 LOD files

The simulator reads LOD-format executable files generated by the assembler. LOD-files contain the core parameters that were used in the assembler stage. If a LOD file is given as an argument to the simulator, core parameters from the LOD file are used. If a LOD file is loaded with the "load" command, the core parameters that are active at that time are used. If the active parameters and the parameters in the LOD file do not match, the loading of the LOD file will fail. LOD files are typically generated for use with a bootloader and not for simulation purposes.

A LOD file typically looks like the following:

```
_START test.asm 2.0 32 16 16 16 8 8 0 0 8 16 0 8 8 1 1
_COMMENT vsa 2.0, (C)1995-99 VLSI Solution

_SYMBOL X l1           00000000
_SYMBOL Y l2           00001000
_SYMBOL P p1           00020028
_SYMBOL P p2           00020026

_DATA X 0000
1122 3344 5566 0000 1000 FFFF

_DATA Y 1000
1001 2002 3003
_SYMBOL P sub1         0002002E

_DATA X 0000
1122 3344 5566 0000 1000 FFFF

_DATA Y 1000
1001 2002 3003

_DATA X FFFF
AA55 5AA5 AABB CCDD

_DATA X 1BFFE
1234 5678

_DATA P 20000
00000010 30F30024 00000040 28800186 00000080 000000C0 28800256 00000041
00000081 00000043 00000082 29800B80 00000140 40800024 00000000 288004D5
40900024 001DDDC0 001DDDC1 00000081 288005D4 60900024 61000024 00000040
00000081 40140024 42260024 00000440 00000881 4132400F 00000052 29800B80
00000024 BC820024 00000024 28800985 00000045 00000085 00000004 00000400
00000801 40100024 F1000024 64980024 28800A00 002AF350 20000000 60900024

_END 20000
```

9.5.4 COFF files

The simulator also reads COFF executable files generated by the assembler. COFF files contain the core parameters that were used in the assembly stage. If a COFF file is given as an argument to the simulator, core parameters from the COFF file are used. If a COFF file is loaded with the "coff" command, the core parameters that are active at that time are used. If the active parameters and the parameters in the COFF file do not match, the loading of the COFF file will fail. It is recommeded to use COFF instead of LOD for the simulator, because the relocation information in the COFF files make the disassembly of the code much more readable.

9.5.5 Command files

Command files work in the same way as the interactive mode. The default fail level is just lower, so that for any error, the reading of the command file is stopped and an error value is returned. Command files typically look like the following:

```
failat 20
si 9
si 10
x 0x3f00
s 8
s
s
r
s 4
r
s 5
r
s
r
x 0x3f00
s 10
quit
```

9.5.6 Log file

By default no log file is generated. With the `-log` option it is possible to define a log file for the simulation. Each command that is entered is recorded, whether the simulator is run in interactive or batch (command script) mode. The log file is directly usable as a command file. The log file does not include the simulation output.

9.6 Profiler listing file

There are two ways to generate a profiler listing file. The first one is to give `vssim` the command line option `-prof` and a filename. Execution data is collected during the simulation, and when the user exits the simulator (using the `quit` command or when an end of file is reached) a profiler listing is generated into the named file.

The last executable file loaded to the simulator must still be available for the profiling listing generation to be accurate or even possible.

Another way to control the profiling is the `profile` command. When using this without arguments profiling is toggled on and off. Using the command with a filename argument generates the profiling list into that file. Note that profiling increases the memory usage and decreases the speed of the simulation.

The generated profile listing contains:

1. General information about the executable
   name, number of sections and symbols
   `vsmpg.bin: includes optional header, 68 sections, 113 symbols`

2. Disassembly of sections in the order they are in the COFF file

   - for each instruction:
     `67968 0.169% 0xb304 0xfe028445 MULSS A1,A0; LDY (I1)+1,C1`
     - the number of cycles taken for executing the instruction, decoding the next instruction and fetching the instruction after that
     - percentage of the whole simulation
     - instruction address in hexadecimal
     - instruction code in hexadecimal
     - disassembly of the instruction
   - summary after each detected code block:
     `---> 7053804 17.516% 4248 1660.500 6 _synth_1to1`
     - total cycles spent in the code block
     - percentage of the whole simulation
     - the number of times entered
     - the average number of cycles spent
     - the number of cycles / function length
   - summary for each section:
     `===> 7053804 17.516% [synth_1to1]`

  - total cycles spent in the section
  - percentage of the whole simulation time

3. Operation usage
   for each operation type encountered in the simulation:
   `Unary ALU 433279 1.076%`

   - the operation type
   - number of times executed
   - the percentage of these operations per simulation cycles

   And a summary line giving the total number of operations and simulation cycles. Note that wait states affect the operations per cycle count as well as the percentage values.

4. Parallel operations utilization
   for each operation shows what other operations are in parallel

5. Jumps taken and not taken
   for each jump condition (and loop end address matches)

   - the jump condition
   - the number of times jump taken
   - the percentage of jumps taken for this jump condition
   - the number of times jump not taken
   - the percentage of jumps not taken for this jump condition
   - the total number of times this jump condition was used

6. Approximate call graph
   for each section or function

   - function name
   - s: average cycles spent in the function
   - the number of times called
   - t: average cycles spent, including small-memory subroutine calls
   - for each subroutine called
     - subroutine name
     - average cycles spent, including subroutine calls
     - average number of times called per parent function

The call graph only has approximate numbers for mainly two reasons. Current interrupts cause some instructions to be fetched more times than they are actually executed, which may confuse the calculation of how many times a function is entered. One other reason is that some code (for example in the link libraries) uses fallthrough to the following function, which also confuses the per-function cycle counting.

A shortened example profiler output follows.

```
vsmpg.bin: includes optional header, 68 sections, 113 symbols

                .sect code,synth_1to1
                .org 0xb2ae

_synth_1to1:
  4248  0.011% 0xb2ae 0x36130024   LDX (I6)+1,NULL
  4248  0.011% 0xb2af 0x3e12b817   STX MR0,(I6)+1; STY I7,(I6)
...

  4248  0.011% 0xb300 0x62340024   SUB B0,B1,B0
  4248  0.011% 0xb301 0x242ccd8e   LOOP LS,0xb336
  4248  0.011% 0xb302 0xf4004091   MOVE B0,I1
 67968  0.169% 0xb303 0x33100441   LDX (I3)+1,A0; LDY (I1)+1,A1
 67968  0.169% 0xb304 0xfe028445   MULSS A1,A0; LDY (I1)+1,C1
...

  4248  0.011% 0xb3b3 0x20000000   JR
  4248  0.011% 0xb3b4 0x36f29817   LDX (I6)-1,MR0; LDY (I6),I7
 --->  7053804  17.516%   4248 1660.500     6 _synth_1to1
 ===>  7053804  17.516% [synth_1to1]

Operation Usage
===============
Unary ALU                   433279   1.076%
Binary ALU                12079337  29.995%
Conditional Jump (Jcc)     1180567   2.932%
Cond Jump LR0 (JRcc)        735208   1.826%
Jump LR1 (RETI)                  0   0.000%
Multiply-Accumulate        1898233   4.714%
Multiply                   1923733   4.777%
X-Move                    19424553  48.234%
Y-Move                     7963587  19.775%
Load Constant              5104524  12.675%
Loop                        119234   0.296%
Restore P                        0   0.000%
MOVE Reg                   1429511   3.550%
CALLcc                      454418   1.128%
```

```
Custom                              0   0.000%
** No Operation              1027448   2.551% **
52746184 operations executed in 40271697 cycles (1.31 op/cycle)

Parallel Operation Utilization
==============================
     UALU BALU  Jcc JRcc  MAC  MUL MV X MV Y  LDC LOOP MOVE CALL
UALU100%   0%   0%   0%   0%   0%  15%  12%   0%   0%   0%   0%
BALU  0% 100%   0%   0%   0%   0%  27%   6%   0%   0%   5%   0%
Jcc   0%   0% 100%   0%   0%   0%   0%   0%   0%   0%   0%   0%
JRcc  0%   0%   0% 100%   0%   0%   0%   0%   0%   0%   0%   0%
MAC   0%   0%   0%   0% 100%   0%  53%  45%   0%   0%   0%   0%
MUL   0%   0%   0%   0%   0% 100%  49%  20%   0%   0%   1%   0%
MV X  0%  17%   0%   0%   5%   5% 100%  29%   0%   0%   0%   0%
MV Y  1%   9%   0%   0%  11%   5%  72% 100%   0%   0%   0%   0%
LDC   0%   0%   0%   0%   0%   0%   0%   0% 100%   0%   0%   0%
LOOP  0%   0%   0%   0%   0%   0%   0%   0%   0% 100%   0%   0%
MOVE  0%  41%   0%   0%   0%   2%   0%   0%   0%   0% 100%   0%
CALL  0%   0%   0%   0%   0%   0%   0%   0%   0%   0%   0% 100%

Jumps Taken / Not Taken
=======================
ALWAYS   T  518472 100.0%  +  N       0   0.0%  =   518472
CS       T     364   0.7%  +  N   53822  99.3%  =    54186
ZS       T   99037  48.9%  +  N  103519  51.1%  =   202556
LT       T  213571  85.7%  +  N   35755  14.3%  =   249326
LE       T     289  22.7%  +  N     982  77.3%  =     1271
CC       T    5536  18.6%  +  N   24198  81.4%  =    29734
NC       T     413  21.2%  +  N    1534  78.8%  =     1947
ZC       T  272710  80.9%  +  N   64225  19.1%  =   336935
GE       T   56330  62.3%  +  N   34110  37.7%  =    90440
GT       T  149685  99.7%  +  N     431   0.3%  =   150116
LOOPEND  T 1071668  90.0%  +  N  119160  10.0%  =  1190828

Approximate Call Graph
======================
df_to_short      s  18.13     23    t  18.13
df_mul           s  32.20    261    t  32.20
df_add           s  24.47    260    t  24.47
_sqrt            s 153.25      4    t 1230.22
  df_div           316.04 3.0
  df_add            24.47 3.8
  df_lt             11.60 0.8
_atan            s  35.00      2    t  46.60
  df_lt             11.60 1.0
```

9.6.1 Register trace file

A register trace file can be generated with the `-rt` option. A single line of output is generated for each simulated cycle. This option is mainly used for co-verification with other simulator tools.

The register trace output includes the states of fixed registers in a binary notation in the following order: `A0, A1, B0, B1, C0, C1, D0, D1, LR0, LR1, MR0, MR1, LC, LS, LE, I0, I1, I2, I3, I4, I5, I6, I7, P, PC`. All of the registers are printed with the configured number of bits. Even if the register does not exist, it is still displayed.

9.7 Commands

All numerical values for commands may be given in decimal, octal (denoted by a leading zero) or in hexadecimal (denoted by leading '0x'). Wherever an address is required, a symbol (label) can be used. However, the symbol must be defined in the corresponding memory space (program, X or Y memory). For example, for a command which deals with the program memory, the symbol must be defined in the program address space.

register `=` *value*
: Assigns a value into register. The change in the register contents is visible immediately, i.e. the write is analogue to a execute-cycle store stage. Also, the program counter (`PC`) and the pseudo-register `cycles` can be written.

memspace`:`*address* `=` *value*
: Assigns a value into a memory location. Different memory spaces are `X` for X-memory, `Y` for Y-memory, and `P` for program memory. The change is visible immediately. If the address is a memory-mapped I/O device, everything goes exactly the same way as the normal instruction execution.

```
> x 0 1
        X: 0x0000:  0x0000
> x:0 = 0x1234
> x 0 1
        X: 0x0000:  0x1234
```

`p` *register*

Prints the contents of one register. The register name must be one of the registers in the core, or the pseudo-register `cycles`. The P-register can be accessed only as a single register, not in separate high and low parts.

```
> p cycles
cycles: 0
> p i4
I4: 233495261
> p le
LE: -1
```

`help [`*command*`]`

Gives information on command or the command list.

`quit`

Quits the simulator

`failat` *level*

Failat sets the minimum return value that causes the simulation to quit. To abort simulation on warnings, "`failat 5`" can be used. Default fail level in scripts is 10 (errors). Fail level 20 is used in interactively controlled simulations.

`fill` *page*`:`*start end+1 value*

Fill, as the name implies, fills memory from `start` to `end` with `value`. `page` must be one of `P`, `X` and `Y`.

`run` *command-file* `[`*arguments*`]`

Executes commands from a file. If the command file can not be found from the current directory, the file is searched from a directory indicated by the environmental variable `VSDSP_CMD_DIR`. Returns to interactive mode or to the previous script file when the file end, quit command, or the quit address is reached. Return codes bigger than the failat level cause recursive exit from every script file.

The optional arguments can be used in the script file. Only whole words can be replaced. For example `test.cmd`:

```
load test.coff
watch regs
s %1
```

When this command file is invoked from the simulator by `run test.cmd 10`, the script loads the COFF file, sets the right watch mode, and then simulates for 10 cycles. `%1` is the first argument passed to the script, `%2` is the second, and so on. If the argument does not exist, no replacement is done.

`run` can be omitted, if the command file name does not equal any of the internal commands or their synonyms. In this case the command file is first searched from

the current working directory, then from the VSDSP_CMD_DIR. If the command file is not found, ".cmd" is appended to the name, and that file is searched from the previously mentioned directories.

`load` *executable*

Reads in a LOD or COFF format file. If the architecture parameters defined in the hardware description file and the parameters in the LOD/COFF-file do not match, warnings are generated for each mismatching parameter.

`coff` *coff-file*

Reads in a COFF file. If the architecture parameters defined in the hardware description file and the parameters in the COFF file do not match, warnings are generated for each mismatching parameter.

`r`

Register dump displays all registers, program counter, cycle count and the state of the pipeline (the address from where the instruction was originally fetched and the disassembled instruction).

```
> r
  A1  : 0x00fa  A0  : 0x00fa  B1  : 0x0100  B0  : 0x00ff
  C1  : 0x0000  C0  : 0x0000  D1  : 0x003f  D0  : 0xffff
  LR0 : 0xd52a  LR1 : 0x0000  MR0 : 0x0200  MR1 : 0x0000
  LC  : 0x002e  LS  : 0xd520  LE  : 0xd52a
  I0  : 0x9b07  I1  : 0x401c  I2  : 0x9b0c  I3  : 0x84d2
  I4  : 0x4033  I5  : 0xa975  I6  : 0x4038  I7  : 0x92ac
  IPR0: 0x0000  IPR1: 0x0000
  P   : 0x00fffa00 =~ 16775680
  A   : 0x00fa00fa =~ 16384250
  B   : 0x010000ff =~ 16777471
  C   : 0x00000000 =~ 0
  D   : 0x003fffff =~ 4194303
  PC  : 0x0 0xd523  Cycles: 0x0017200b = 1515531  Operations: 326373
Last Exec: 0xd520 LDC 0x100,B1
Next Exec: 0xd521 LDX (I3),B0 ; LDY (I4)-3,NULL
Fetched:   0xd522 MULUU B1,B0 ; LDX (I4)+3,A1
```

`d [`*start-address* `[`*end-address*`]]`

Displays program memory with disassembled instructions and symbol information. If the end address is omitted, 8 entries are displayed. If both the start and the end addresses are missing, the next 8 addresses are displayed.

```
> d 0
code:
    0x0000:  0x0000800a  LDC 0x200,MR0
    0x0001:  0x00000010  LDC 0x0,I0       /* vektori1 */
    0x0002:  0x00000051  LDC 0x1,I1
    0x0003:  0x00000012  LDC 0x0,I2       /* vektori2 */
    0x0004:  0x00000053  LDC 0x1,I3
    0x0005:  0x00000584  LDC 0x16,C0      /* vektori1 + 0x16 */
    0x0006:  0x6cc30a2b  SUB A,A,A ; LDX (I0)*,B0 ; LDY (I2)*,B1
@000:
    0x0007:  0x24000244  LOOP C0,0x9      /* @000 + 0x2 */
```

```
> d
    0x0008:  0xfe350a2b  MULSS B0,B1; LDX (I0)*,B0; LDY (I2)*,B1
    0x0009:  0x54330a2b  MAC B0,B1,A; LDX (I0)*,B0; LDY (I2)*,B1
    0x000a:  0x28000000  J 0x0000
    0x000b:  0x4cb20024  ADD A,P,A
    0x000c:  0x283eb380  J 0xface
    0x000d:  0x00000024  NOP
    0x000e:  0x00000000  LDC 0x0,A0
    0x000f:  0x00000000  LDC 0x0,A0
>
```

The disassembly only shows symbols in operations such as jumps and LDC, if relocations are present in the loaded object file. LOD-files and COFF executables linked without the "-k" option do not have relocation information.

```
> d @000
@000:
    0x0011:  0x15000267  LOOP A3,0x13     /* @000 + 0x2 */
    0x0012:  0x00000000  NOP
    0x0013:  0x38400000  SUB A0,ONES,A0
    0x0014:  0x3f100000  SUB A3,A0,A1
    0x0015:  0x73500000  CMPZ A1,A1
@001:
    0x0016:  0x18000345  JN A1,0x001a    /* @001 + 0x4 */
    0x0017:  0x00000000  NOP
@002:
    0x0018:  0x18000301  J 0x0018    /* @002 */
```

`x [`*start-address* `[`*end-address*`]]`

Displays X-memory with symbol information. If the end address is omitted, 8 entries are displayed. If both the start and the end addresses are missing, the next 8 addresses are displayed.

```
> x 0x5fd 0x602
        X: 0x05fd:  0x0000
        X: 0x05fe:  0x0000
        X: 0x05ff:  0x0000
        X: 0x0600:  <unconnected>
        X: 0x0601:  <unconnected>
```

`y [`*start-address* `[`*end-address*`]]`

Displays Y-memory with symbol information. If the end address is omitted, 8 entries are displayed. If both the start and the end addresses are missing, the next 8 addresses are displayed.

`xy [`*start-address* `[`*end-address*`]]`

Displays X and Y-memory with symbol information. If the end address is omitted, 8 entries are displayed. If both the start and the end addresses are missing, the next 8 addresses are displayed.

```
> xy 0 3
X: vector1
Y: vector2
0x0000:  X: 0x0001 /*     1 */
         Y: 0x0001 /*     1 */
0x0001:  X: 0x0002 /*     2 */
         Y: 0x0002 /*     2 */
0x0002:  X: 0x0003 /*     3 */
         Y: 0x0003 /*     3 */
>
```

`g` *address*
Sets a new value for program counter and clears the pipeline. Other register contents remain unchanged.

`s` [*cycles*]
Simulates the defined number of cycles. If there is an error, the simulation is stopped. If the cycle parameter is omitted, only one cycle is simulated, which corresponds to a single-step operation. The information displayed during the simulation can be changed with the "`watch`" command.

The simulation is cycle-accurate: the pipeline, change-of-flow instructions, loops and interrupts are simulated. Register contents change at the end of each cycle, so that both the old value of a register can be used and the register can be updated in parallel operations within the same instruction.

`sb` *address* [*times*]
Sets a breakpoint where the simulation will be temporarily halted. If the `times` parameter exists, the execution continues normally the first `times` number of times the address is reached.

`rb` *address* [*address**]
Removes one or more previously set breakpoints.

`lb`
Lists all breakpoints.

`sr` *cycle-offset*
Schedules a reset. The cycle offset defines how many cycles it takes until the reset becomes active. The reset is detected on the next cycle. If there is another reset or interrupt active (the first five cycles), the reset is lost. Reset must be scheduled for five consecutive cycles to be certain that it will be detected, if other interrupts are scheduled. If both reset and interrupt activates on the same cycle, reset is given precedence and the interrupt is lost.

`si` *cycle-offset* [*loop-time*]
Schedules a one-shot or a timer interrupt. The cycle offset defines how many cycles it takes until the interrupt request becomes active. The interrupt is detected

on the next cycle. If loop time is omitted, a one-shot interrupt is scheduled. Otherwise the interrupt is rescheduled immediately when it becomes active. Loop times shorter than 6 cycles are not allowed.

```
> watch execute interrupts
> si 2
> s 8
Executing: 0x0005 SUB A0,A0,A0
Executing: 0x0006 NOP
Executing: 0x0007 JZC 0x0005       /* test */
Interrupt cycle 1
        LR1 = fetch address
Executing: 0x0008 SUB A0,A0,A0
Interrupt cycle 2
        0x0008 SUB A0,A0,A0 not change-of-flow inst
        -> 0x0009 JZS 0x0005     /* test */ cancelled
Executing: 0x0009 NOP
Interrupt cycle 3
Executing: 0x0008 SUB A0,A0,A0
Interrupt cycle 4
Executing: 0x0009 JZS 0x0005     /* test */
Z flag (potentially) changed by previous instruction
1 error executing instruction
Interrupt cycle 5
Executing: 0x000a NOP
Interrupt idle cycle
>
```

`li`

Lists all active interrupts. Loop times are displayed for timer interrupts.

```
> si 5
> si 10 20
> li
Interrupts/resets:
cycle: 0x00000012 interrupt
cycle: 0x00000017 interrupt looptime: 20
```

`watch [(`**on** | **off**`)] [`**execute**`] [`**decode**`] [`**fetch**`] [`**interrupts**`] [`**regs**`]`
Defines the output during simulation. Default is **regs** and **interrupts**.

```
off        - no output
on         - pipeline and interrupt actions
execute    - execute-stage
decode     - decode-stage
fetch      - instruction fetch
interrupts - interrupt actions
regs       - show registers each time simulation stops
```

`watch` **reg** *regname* [**on**] [**off**] [**write**] [**read**] [**Tvalue**] [**break**]
`watch reg` sets a register under scrutiny.

```
off    - no watch
write  - watch writes to the register
read   - watch reads from the register
Tvalue - watch register value, T is one of <, >, =, !.
         e.g. "watch reg i0 =0x200 break"
break  - cause the simulation to be stopped
         when write/read reported
```

`watch` **mem** *memspace*`:`*address* [**on**] [**off**] [**write**] [**read**] [**Tvalue**] [**break**]
`watch mem` sets a memory location under scrutiny.

```
off    - no watch
write  - watch writes to the memory location
read   - watch reads from the memory location
Tvalue - watch memory location value, T is one of <,
         >, =, !. e.g. "watch mem x:0x10 =0x200 break"
break  - cause the simulation to be stopped
         when write/read reported
```

`write` *memspace*`:`*start end filename*
Writes memory into a file in ASCII format, one value per line. Different memory spaces are `X` for X-memory, `Y` for Y-memory, and `P` for program memory. If the address is a memory-mapped I/O device, everything goes exactly the same way as the normal instruction execution.

`read` *memspace*`:`*start filename*
Reads data into memory from a file, which is in ASCII format, one value per line. Different memory spaces are `X` for X-memory, `Y` for Y-memory, and `P` for program memory. If the address is a memory-mapped I/O device, everything goes exactly the same way as the normal instruction execution.

`where`
Outputs the name of the section where the execution is currently proceeding. Also displays the last non-local symbol before the current program counter value.

`skip`
Executes the code until a breakpoint, error or a return (JR) to the same execution level (subroutine-wise). `Skip` is useful in skipping subroutines without single-stepping.

`profile [`*filename*`]` [**on**] [**off**]
Without the filename toggles profiling on/off. All previously gathered profiling information will be lost when profiling is turned on. If a filename is given, a profiler listing is created into that file, when the profiling is on.

`dump [`*filename*`]`
Creates a snapshot file of the state of the simulator memory, registers and pipeline.

`undump [`*filename*`]`
Reads a snapshot file of the state of the simulator memory, registers and pipeline. The simulator configuration must be the same it was when the dump was created, otherwise the behaviour of the simulator is undefined.

`echo [`*text*`]`
Echoes its arguments to the standard output.

Chapter 10

Installing and First Steps Using VS_DSP Tools

10.1 Overview

This chapter presents the necessary steps the user must take before he can use the VS_DSP tools to compile and link programs and libraries. Also, a programmer new to the architecture is led step-by-step through the first compilations and software simulations.

The chapter assumes the reader knows ANSI C language and has some idea of what assembly language looks like. It is also assumed that such terms as ”compiler”, ”linker” and ”profiler” are familiar.

This is a very brief chapter and does not go into smallest details. While this necessarily causes some important things to be omitted, this chapter should be able to serve as a crash course to VLSI Solution's VS_DSP architecture. Further details should be looked for in VS_DSP reference manuals.

10.2 Installing the VS_DSP Tools

To install the VS_DSP tools the following steps need to be taken:

Copy the files from the distribution directory `"bin"` to a directory on your hard disc (e.g. `C:\VSDSP\` on a Windows-based computer or `/usr/local/bin/vsdsp/` on a Unix-based computer). Copy also the subdirectories, especially `"config"`.

If there is no path to the directory, add it (Windows/DOS: add directory to PATH command of AUTOEXEC.BAT, Unix: Add directory to your appropriate shell startup

script).

Now that the path is set, you should be able to run the VS_DSP programs. Try this by giving the command `"vsa"`, which should without any command line options give the following message: `"*** ERROR: No input file defined!"`

Now it is time to set the configuration directory. This is done by setting the environmental variable VSDSP_DIR (Windows/DOS: Add 'SET VSDSP_DIR C:\VSDSP\CONFIG', Unix 'setenv VSDSP_DIR /usr/local/bin/vsdsp/config' (slightly dependent on the shell used)).

If using a Windows system, reboot the machine to activate the changes, on Unix restart your shell or run your shell's startup-script.

After these operations, you might want to copy all the files from directory `src` to your working directory. Later we'll be editing these files. Note, that the source files may be in Unix ascii format with only an LF indicating a new line instead of Windows' CRLF, so some old editors may have problems with them.

10.3 Compiling the First Assembly Program with VSA

In directory `src`, which you should copy to your working directory, there are several test programs. One of them is `simple.s`. This is a very simple assembly language program that first copies 4 data words from one location to another, and then copies yet another 16 data words with a different method. Refer to the source code for further details.

If running under Unix, have a look at `Makefile` (it may also be useful under Windows for those who have a version of GNU Make utility installed). Change the directory for the VS_DSP tools to reflect where you have the tools, and command `"make"`. This should compile and link all the example programs mentioned in this chapter.

Under Windows, or if you want to try compiling compile programs by hand in Unix, command `"vsa -o simple.cof simple.s"`.

Now you have created a COFF object file `simple.cof`.

10.4 Linking the First Assembly Program with VSLINK

To get a running program, you have to link the program to an executable binary file. This is done as follows under Windows: `"vslink -k -L C:\VSDSP\libc16 -lsim -o simple.bin C:\VSDSP\libc16\_startup2.o simple.cof"`. Under unix, change the directory name to the location where you put your VS_DSP software tools.

If all went well, you should now have a properly compiled executable COFF file called `simple.bin`. The binary is now linked with a simple startup code from library file `_startup2.o`, which provides a 256-word stack and the `exit();` routine.

10.5 Running the First Assembly Program with VSSIM

Now that you have an executable program, you can start VSSIM by commanding `"vssim -m mem_desc.sim -l simple.bin -prof simple.prof"`. You should get approximately the following output:

```
VSSIM 2.0 May 14 2002 16:10:57 (C)1995-2002 VLSI Solution
Reading COFF file
simple.bin: includes optional header, 7 sections, 4 symbols
Section  1: stack_x                 1:0x0000..0x0100 [257] fixed
Section  2: stack_y                 2:0x0000..0x00ff [256] fixed
Section  3: reset                   0:0x0000..0x0001 [2] fixed
Section  4: startup                 0:0x0002..0x000d [12]
Section  5: main                    0:0x000e..0x001b [14]
Section  6: const_x                 1:0x2000..0x2013 [20]
Section  7: data_Y                  2:0x2000..0x2013 [20]
```

Here you can see all the data sections, their types (0=code, 1=X, 2=Y), their link addresses, their lengths, and whether they are compiled to fixed or relocatable addresses.

If you are interested in the inital state of all the hardware registers, you may try the command `"r"`. You will be presented with the following kind of output:

```
  A2  : 0x00  A1  : 0x1acc  A0  : 0x1acc
  B2  : 0x00  B1  : 0x1acc  B0  : 0x1acc
  C2  : 0x00  C1  : 0x1acc  C0  : 0x1acc
  D2  : 0x00  D1  : 0x1acc  D0  : 0x1acc
  LR0 : 0xdadd  LR1 : 0xdadd  MR0 : 0x0000  MR1 : 0x0000
  LC  : 0x0000  LS  : 0x0000  LE  : 0xffff
  I0  : 0xdadd  I1  : 0xdadd  I2  : 0xdadd  I3  : 0xdadd
  I4  : 0xdadd  I5  : 0xdadd  I6  : 0xdadd  I7  : 0xdadd
  A2  : 0x00  B2  : 0x00  C2  : 0x00  D2  : 0x00
  P   : 0x00000000 =~ 0
  A   : 0x001acc1acc =~ 449583820
  B   : 0x001acc1acc =~ 449583820
  C   : 0x001acc1acc =~ 449583820
  D   : 0x001acc1acc =~ 449583820
  PC  : 0x0 0x0000  Cycles: 0x00000000 = 0  Operations: 0
Last Exec: 0x0000 LDC 0x0,A0       /* section startup */
Next Exec: 0x0000 LDC 0x0,A0       /* section startup */
Fetched:   0x0000 LDC 0x0,A0       /* section startup */
```

Here you can see the full state of the VS_DSP processor core. First all 16-bit arithmetic registers (A0..D1) and their guard bit registers (A2..D2) are shown, then the return pointers (LR0/1) and mode registers (MR0/1), then the loop (LC, LS, LE) and address registers (I0..I7), then the guard bit registers again and the multiplication result register (P). Then arithmetic register values are reshown as combined values (A2, A1 and A0 combined to a 40-bit full-length register A, and so on), followed by the program counter. Finally the processor pipeline is shown, and as no instructions have yet been executed, it is full of zero instruction words.

Now, to run the processor for one clock cycle, command `"s"`. In this reset cycle all registers are cleared, and by executing the `"s"` command a couple times more, you will see that your first assembly language commands start executing.

Now, if you want to take a look at the X memory constant data area labeled readArea in the source code, you may do so by issuing the command `"x readArea"`. The output should look as follows:

```
readArea:
        X: 0x0100:  0xface /* -1330 */
        X: 0x0101:  0xcafe /*-13570 */
        X: 0x0102:  0xcace /*-13618 */
        X: 0x0103:  0x1234 /*  4660 */
        X: 0x0104:  0x0001 /*     1 */
        X: 0x0105:  0x0002 /*     2 */
        X: 0x0106:  0x0003 /*     3 */
        X: 0x0107:  0x0004 /*     4 */
```

Here you can see 8 addresses with their contents shown in both hexadecimal and decimal notation. If you want to see further, command `"x"` without any parameters.

Now if you want to run a few more steps, command `"s 5"`. This will run the processor for 5 clock cycles. To finish the simulation, command `"s 1000000"`. Now the simulator tries to run 1000000 cycles, but before you ever get close to that, you'll get the following message:

```
Fetched from unconnected memory at 0x001a
Last Exec: [000001] 0x001a J 0xffff
[... register dump ...]
```

The simulator does not accept illegal read, write or execute operations, and by putting one operation like that just before the end of the code, one can make sure the simulation stops when the whole program is executed.

For more details about the numerous commands available in VSSIM, type `"help"`. For a particular command, e.g. *watch*, type `"help watch"`.

While you exit the simulator with the command `"exit"`, an execution profile file called `"simple.prof"` is generated. In this profile file you can read some interesting statistics about your program run. See section 11.3 or more details.

10.6 Compiling, Linking, Running the First C Program

Now that you hopefully feel yourself comfortable compiling simple assembly language programs, we take a step further and try compiling a C source code program. For this task there is an example program called `prime.c` that calculates which numbers between 2 and 4096 are prime numbers, i.e. divisible only by themselves and 1.

For further details about ANSI C language see *Brian W. Kernighan / Dennis M. Ritchie: The C Programming Language - Second Edition / ISBN 0-13-110362-8*.

Again, we'll start by compiling. This time the appropriate command is `"vcc -IC:\VSDSP\libc16 -O -fsmall-mem -o prime.cof prime.c"`. The output of VCC looks approximately like this:

```
Successfully compiled 164 lines (46 in source file)
(150 118 115 115 115)
C 115 CF 0 X 61 Y 256 F 0
```

The first number on the second line (150) tells how many code words the original length of the compiled program was, and each successive number tells how much was left after each recursive optimization step. The number of optimization rounds varies with code size, and the more there is code, the more steps are generally done.

The last line of the compiler output tells the final size of the object file (in 32-bit code and 16-bit data words). It is interpreted as follows: C = code, CF = far (32-bit) code, X = X memory, Y = Y memory, F = far (32-bit) X memory.

The C compiler has now created a file called `prime.a` and sent that file forward to the assembler, which in turn outputs `prime.cof`. If you want to see what kind of assembly language code the compiler generated, take a look at `prime.a`.

The C program is linked like the assembly language program: `"vslink -k -L C:\VSDSP\libc16 -lsim -o prime.bin C:\VSDSP\libc16\_startup2.o prime.cof"`. Again, remember to change the directory name according to the location where you put your VS_DSP software tools.

Now you can simulate the system as you did with the earlier program `simple`. One interesting command you might like to try in the simulator is `"where"`, which tells you which C function you are executing at the moment.

10.7 Compiling and Linking Multiple Object Files

To compile multiple object files into one executable program, all you really have to do is first to separately compile the C and/or assembler source files and then link them together.

You can try a multiple source file compilation by first compiling the C source file `"vcctest.c"` to `vcctest.cof`, then assembler source file `"vcctestasm.s"` to `vcctestasm.cof`. Just do it as we did with the earlier examples.

Now, linking these two modules together is easy. Just say `"vslink -k -L C:\VSDSP\libc16 -lsim -o vcctest.bin C:\VSDSP\libc16\_startup2.o vcctest.cof vcctestasm.cof"`. All the COFF files presented to VSLINK are compiled in the same destination file.

Actually, the program you have created is the same program that is used in optimization examples in section 11.7. Now, with the help of the profiler, you can try to make the same speed measurements. After that, you might want to change the programs slightly or start creating your own programs.

However, before creating your own C programs, it is recommended to read the chapter 11.

Chapter 11

VCC Programming Tips

11.1 General

VCC is a powerful ANSI C compiler that has the potential of creating very efficient and small code. However, to gain its full potential it is good for the programmer to have some knowledge of its inner workings and to understand what kind of code is most suitable for the compiler. By slightly adjusting one's code, even five-fold speed increments can be achieved. Usually also the program code size is reduced.

For optimal performance, one who has never worked with a two-data-bus DSP architecture should pause to consider the possibilities the VS_DSP architecture gives: in one clock cycle, two memory read/write operations may be executed as long as the operations operate on different data buses, i.e. the other is done to/from X memory (compiler default) and the other to/from Y memory (must be switched on by the user).

11.2 Pointers, X and Y Memory

Normal variables and tables can easily lie in either X or Y memory; they can be put where there is space. However, with pointer variables one must take care to keep the pointer pointing to the correct kind of memory. For instance, the following code is invalid and will not compile:

```
void MyFunc(void) {
  static int __y table[256];  /* Table put to Y memory       */
  int *p = table;             /* WRONG! Pointer to X memory! */

  [... code ...]
}
```

The correct way to write the function is as follows:

```
void MyFunc(void) {
  static int __y table[256];  /* Table put to Y memory   */
  int __y *p = table;         /* RIGHT: Pointer in Y mem */
                              /* points to Y memory      */

  [... code ...]
}
```

However, there are times when one wants the pointer to be in other memory, and the data to be in other memory. If we absolutely want the previous pointer `p` to be in X memory, we can write the code as follows:

```
void MyFunc(void) {
  static int __y table[256]; /* Table put to Y memory       */
  int __y * __x p = table;   /* Also RIGHT:Pointer in X mem */
                             /* points to Y memory          */

  [... code ...]
}
```

11.3 Using the Profiler

The VSSIM simulator's built-in profiler is a very powerful tool when optimizing C or assembly code. To activate the profiler use the option `-prof filename.prof` when starting VSSIM. Alternatively you can use the command `profile` after starting the simulator.

11.3.1 Reading Profiler Output

When exiting VSSIM, the profiler output is written to the named file. Profiler output will be provided for each and every instruction word executed. Often, however, the user is only interested in the total execution times for whole functions. This can be achieved by searching for those lines in the profile file that have the string `" ---"` in them (Use e.g. *grep* under Unix, and *find* under Windows/DOS. Since the profile file is in a line-based text format, any general-purpose text-search program will do). The output of the search is as follows:

```
 --->        4   0.019%      3    1.33        2 reset
 --->        7   0.034%      1    7.00        1 startup
```

```
--->        4   0.019%      1    4.00        0 _exit
--->        0   0.000%      0    0.00        0 _Void
--->     3100  15.089%      1 3100.00       77 _MemCpy0
--->     2587  12.592%      1 2587.00       69 _MemCpy1
--->     2072  10.085%      1 2072.00       64 _MemCpy2
--->      539   2.624%      1  539.00       18 _MemCpy3
--->      539   2.624%      1  539.00       18 _MemCpy4
--->      321   1.562%      1  321.00        7 _MemCpy5
--->     3338  16.247%      1 3338.00      115 _StrCpy0
--->     2571  12.514%      1 2571.00      122 _StrCpy1
--->     1037   5.047%      1 1037.00       61 _StrCpy2
--->     1037   5.047%      1 1037.00       61 _StrCpy3
--->      447   2.176%      1  447.00       10 _FIR
--->     2658  12.937%      1 2658.00       22 _main
--->      272   1.324%      1  272.00       16 _AsmFIR
```

The first number (3100 for *_MemCpy0()*) tells the total number of clock cycles spent in the function.

The second number (15.089 %) tells the relative clock count spent in the function (total = 100%).

The third number tells (1) how many times the function was called.

The fourth number (3100.00) tells the average clock cycle count for the function.

The last number (77) tells the so-called "goodness" number for the function. It is the number of clock cycles spent in the function divided by its length. Thus, the higher the number, the more is achieved by putting that function to lower-power or faster memory.

11.3.2 Profiler Feedback to Compiler

After running the profiler, its output can be used to recompile the code to be more efficient. By adding the option `-fprof filename.prof` to VCC, it uses the output of the previous profiler run to optimize branches according to real data.

If more than one simulation output is present, multiple `-fprof` options may be used.

11.4 FIR Filters and C

Writing a FIR filter is one of the tasks that might be best left done in the assembly code level. However, even a FIR can be written in C. Below is one example. As usual with DSPs, the other FIR data source is assumed to be in X memory and the other in Y memory.

In this example the result is shifted down by 13 bits.

Also, it is assumed that the length of the FIR filter is a multiple of 8. This way loop unrolling will make the code quite a bit faster.

The example C code is as follows:

```
int FIR(register __y int *s1, register int *s2,
        register int len) {
    register int i;
    register long res = 0;

    len /= 8;

    for (i=0; i<len; i++) {
        res += *s1++ * *s2++;   res += *s1++ * *s2++;
        res += *s1++ * *s2++;   res += *s1++ * *s2++;
        res += *s1++ * *s2++;   res += *s1++ * *s2++;
        res += *s1++ * *s2++;   res += *s1++ * *s2++;
    }

    return (int)((res << 3) >> 16);
}
```

This code actually performs a filter operation at 1.875 clock cycles per FIR stage. I.e. it reads two data words, multiplies them and adds to the intermediate result storage in 1.875 clock cycles on average.

Because FIR filters are so widely used, below is presented a fully optimized assembly version of the same filter. The only functional difference with this filter is that its input doesn't necessarily have to be a multiple of 8. However, it must be greater than 0.

This version requires 16 clock cycles per call, plus exactly 1 clock cycle per FIR point. The example is as follows:

```
  .sect code,AsmFIR
  .export _AsmFIR
_AsmFIR:
  ldx (I6)+1,NULL;            // Move stack pointer up
  stx i1,(i6)+1; sty i3,(i6); // Store i1 and i3 to stack
  stx MR0,(I6); sty a1,(i6);  // Store mode register and a1
  ldc 0x200,MR0 // Integer mode, no saturation

  ldc 1,i1                    // Set i1 to 1, later to be used
                              //   as post-mod register for i0
  add a0,ONES,a0; mv i1,i3    // Subtract 1 from a0, i1->i3
  ldy (i0)*,d1 ; ldx (i2)*,d0 // Load from (i0) and (i2).
```

```
                                //   '*' = registers postmodified
                                //    by i1 and i3, respectively
  loop a0,LoopEnd-1             // Set A0+1 loops
  and a,NULL,a                  // Clear 'a' in loop's delay slot
LoopStart:
  mac d1,d0,a; ldy (i0)*,d1 ; ldx (i2)*,d0
                                // Multiply and ACcumulate, get
                                //   next datas to d1 and d0,
                                //   post-modify i0 and i1,
                                //   all this in 1 clock cycle!
LoopEnd:
  add a,p,a; ldy (i6),i3        // Copy multiplier result to A
  add a,a,a; ldx (i6)-1,i1      // Begin shifting the result and
  add a,a,a                     //   restoring the stack
  add a,a,a
  add a1,null,a0                // Result is put to a0

  jr                            // Restore final registers in
  ldx (i6)-1,MR0; ldy (i6),a1 //   jr's delay slot

.end
```

To get this filter working, the following prototype needs to be introduced to the calling C interface:

```
int FIR(register __i0 __y int *s1, register __i2 int *s2,
        register __a0 int len);
```

It is always a good idea to explicitly introduce the registers used for parameters if register parameters are used. This way software compatibility with future versions of the compiler can be assured.

Also note that if stack parameters are used, they are put to the stack in reverse order, i.e. the last first. Thus, the first parameter is closest to the stack pointer inside the function.

11.5 File Naming Conventions

It is suggested that the following file name postfixes are used:

- .s or .asm : Hand-written assembly source files
- .i : Hand-written assembly include files
- .coff or .o : COFF object files

- .c : C source code files
- .h : C include files
- .a : Assembly source files created by the compiler (this is the default)

11.6 How to Optimize Your C Code for Speed

11.6.1 Automatic Variables

As with any C compiler, one may introduce new variables in functions or at any new program scope. An example is below:

```
int MyFunc(void) {
    int a, b, tmp, myT2, q, w, n, i, r, t, t2, t3, tab[256];
    char *s, *d;

    [... code ...]
}
```

There are several things that can be made better in this function.

First, as these variables are allocated from the stack, the table `tab[256]` may make the stack too big. Unless you are absolutely sure there is actually enough space, the table should be introduced as "static". This way it will be allocated in *bss* data section instead of stack.

Then, one might consider to have some variables locked into dedicated registers if they are used often enough. If, say, `i`, `q` and `tmp` are referenced very often, they could be allocated as register variables. Of the eight 16-bit arithmetic registers, almost always four to six may be allocated to registers. Of the four 16-bit address registers usually two may be allocated to variables. So, we'll allocate hardware registers for these three variables.

Stack variables are allocated by default from X memory. By changing half of them to Y memory the code gets smaller and faster. The compiler can parallelize operations better if variables used very near each other in code are in different memory pages.

Also, because of architecture limitations, if there are more than 7 stack variables on the same memory page (X or Y), the code will get slower, and if there are more than 15, the code will get a lot slower. By spreading stack variables across X and Y memory the needed amount of stack is minimized and the code is smaller and faster.

So, a properly optimized version of the function would look like this:

```
int MyFunc(void) {
    register int tmp, q, i; /* 3 register variables      */
    int a, b, myT2, w, n;   /* 4 variables to x space    */
    int __y r, t, t2, t3;   /* 5 variables to y space    */
    static int tab[256];    /* static = no stack space   */
    char *s, *d;            /* 2 more to x space (tot 6) */

    [... code ...]
}
```

11.6.2 Parameters

If some of the parameters are used very often in a function, it might be a good idea to have them as register parameters. This way accessing them is much faster. For instance, a function performing the same as <string.h>'s strcpy() could be implemented as follows:

```
void StrCpy2(register char *d, const register char *s) {
    while ((*d++ = *s++))
        ;
}
```

Four address registers of eight are available. Normally register names don't have to be used, unless register parameters are used while interfacing to assembler language functions.

11.6.3 Local Scopes

Using local scopes the amount of stack space and registers required may be greatly reduced. The following is an example of code that can be made more register efficient with scopes:

```
int MyFunc(register __a0 int n, register __a1 int base,
           register __i0 int *p) {
    register int i, res;     /* Ok, but not optimal */

    if (*p > 0) {
        res = 0;
        for (i=0; i<n; i++)
            res += *p++;
    } else {
        res = base-*p++;
        res *= *p;
```

```
    }

    return res;
}
```

The variable `i` is needed only in the first branch. Thus, for the other branch one register would be saved by introducing `i` only in the branch where it really is used.

```
int MyFunc(register __a0 int n, register __a1 int base,
           register __i0 int *p) {
    register int res;        /* Only one register required  */

    if (*p > 0) {
        register int i;      /* Now 'i' takes up a register */
        res = 0;             /* only in this branch         */
        for (i=0; i<n; i++)
            res += *p++;
    } else {
        res = base-*p++;
        res *= *p;
    }
    return res;
}
```

In this particular simple example the optimization doesn't necessarily save anything, but with more complex codes the differences can be quite dramatic.

11.6.4 Shifting

A good thing to remember is that since the VSDSP doesn't have a barrel shifter in core version 2, shifting 32-bit numbers is relatively slow (up to 31 clock cycles), unless one shifts exactly 16 bits to the left or right. Thus, if one wants to shift a 32-bit fixed-point number and assign it to a 16-bit value, it is better to shift as little as possible to the nearest 16-bit jump point.

Thus, if many over 16-bit shifts are expected for 32-bit numbers, the following code makes shifts faster.

```
    int toShift = something;
    long source = something_else, dest;

    if (toShift >= 16)
        dest = (long)((int)(source>>16) >> (toShift-16);
    else
        dest = source >> toShift;
```

Because shifting 16-bit numbers can be easily implemented with the multiplier, the price for shifting a 16-bit number is much lower, and in the first branch the first 16 bits are shifted very quickly. Note, that the compiler *can* optimize constant shifts, so if the shift amount is a constant, no playing around like this is needed.

11.7 An Example on How to Optimize Your Code

As an example on how C code may be optimized to better fit VCC and the VSDSP architecture, a simple function is presented below. It does the same things as <string.h>'s memcpy function. The function is called with two string pointers, and a word count number. In the test runs, 256 words were copied.

11.7.1 MemCpy0, 40 words, 3100 clocks (12.11 clocks/word)

```
void MemCpy0(char *d, const char *s, int n) {
    int i;
    for (i=0; i<n; i++)
        d[i] = s[i];
}
```

This might be the most straightforward way to write a memory copying function. However, 12 clock cycles for copying one data word can by no means be called efficient.

11.7.2 MemCpy1, 37 words, 2587 clocks (10.11 clocks/word)

```
void MemCpy1(char *d, const char *s, int n) {
    int i;
    for (i=0; i<n; i++)
        *d++ = *s++;
}
```

In this function, array indexing has been replaced with a simpler pointer arithmetic. This way 2 clock cycles are saved per one copy operation and we are down to 10 clocks per copy operation.

11.7.3 MemCpy2, 32 words, 2072 clocks (8.09 clocks/word)

```
void MemCpy2(char *d, const char *s, int n) {
    register int i;
```

```
    for (i=0; i<n; i++)
        *d++ = *s++;
}
```

Register parameters are a way to greater efficiency: by replacing the loop variable `i` with a register version, we gain another two clock cycles. This leads logically to the next version.

11.7.4 MemCpy3, 29 words, 539 clocks (2.11 clocks/word)

```
void MemCpy3(char *d, const char *s, int n) {
    register int i;
    register char *D = d;
    register const char *S = s;
    for (i=0; i<n; i++)
        *D++ = *S++;
}
```

This is the smallest version of MemCpy(), only 31 program words long. Here we start by copying the parameters to register pointer variables. Pointer referencing through these new variables `D` and `S` is very fast, and now we have a very powerful routine, where we are down to only two clock cycles per copy operation. And, if both the source and destination addresses are in the same type of memory, be it X or Y memory, this is the theoretical minimum we can get. It cannot be improved, not even by hand-coding the function in assembly language.

However, as we see in the next example, the source and destination registers don't necessarily need to be in the same type of memory.

11.7.5 MemCpy4, 29 words, 539 clocks (2.11 clocks/word)

```
void MemCpy4(char __y *d, const char *s, int n) {
    register int i;
    register char __y *D = d;
    register const char *S = s;
    for (i=0; i<n; i++)
        *D++ = *S++;
}
```

This is very similar to the previous routine. The only real change is that the destination buffer `d` is in Y data memory instead of X data memory. MemCpy4() has exactly the same size and speed as MemCpy3(), but it gives way to a further optimization in MemCpy5().

11.7.6 MemCpy5, 44 words, 321 clocks (1.25 clocks/word)

```
void MemCpy5(register __i0 char __y *d,
             register __i1 const char *s, register __a0 int n) {
    register int i;

    /* If n is not divisible by 8, copy modulo amount */
    for (i=0; i<(n&7); i++)
        *d++ = *s++;

    /* Divide n by 8, because 8 words are copied in one loop */
    n /= 8;

    /* A loop unrolled by a factor of 8 */
    for (i=0; i<n; i++) {
        *d++ = *s++;    *d++ = *s++;
        *d++ = *s++;    *d++ = *s++;
        *d++ = *s++;    *d++ = *s++;
        *d++ = *s++;    *d++ = *s++;
    }
}
```

This is the ultimate example, where we spend only slightly over 1 clock cycle per one read/write operation. The secret here is loop unrolling.

First, if the number of words to be copied cannot be divided by 8, we copy the odd words (say, if n is 126, we copy 6 words). Then, we divide n by 8 (leaving n=15 after the division). Then, we make 8*n copy operations (8*15 = 120).

The idea with loop unrolling is that the compiler can interleave consecutive simple statements so that it can read the next data in the same clock cycle it is writing the previous data. Of course this requires that the source and destination data to be in different memory buses, i.e. that the other is in X and the other in Y memory. However, this is how one can create extremely fast routines even with C. This particular copier could be made faster by 0.25 clocks/cycle with hand-optimized assembler, but most of the time the result isn't worth the trouble.