VM LABS

LLAMA: An Optimizing Assembler for NUON

User's Manual

Version 2.83

VM Labs, Inc.
520 San Antonio Rd
Mountain View, CA 94040
Tel: (650) 917 8050
Fax: (650) 917 8052

Copyright notice

This is a preliminary specification. VM Labs reserves the right to make changes to any and all of the interfaces described in this document.

Contents

1. Introduction

LLAMA is an assembler for the NUON media processor. Besides the normal abilities of an assembler, it also has an instruction scheduler and optimizer built into it, so that it can (optionally) optimize your code for you.

1.1 Usage

```
llama [-b][-B addr][-chip version][-compiled]
    [-cprecedence][-c#]
    [-Dsymbol=value][-e errfile][-f FMT][-g-old]
    [-i incfile][-Ipath]
    [-jJUMP][-M#][-nolisp][-nologo][-o outfile]
    [-O#][-r#][-v] file.s [file2.s ...]
```

All of the file names given after the flags are assumed to be input files, and they are processed one after another, the output being collected into a single output file.

1.1.1 Flags

-b Assume condition codes need not be preserved across branches. Useful only when optimization is taking place (`-O1` or higher). Without this flag, LLAMA will assume that condition codes set by an instruction may be used in some instruction after a branch (including a `jsr`). If the `-b` flag is given, then LLAMA assumes that the code following any branch will not use condition codes generated before the branch. Older versions of the compiler did not do any instruction scheduling in branch delay slots, and the `-b` flag was useful for these. This is no longer the case, and in fact we recommend that you not use `-b` any more.

-B address Specifies a base address at which to load overlay sections and sections without an explicit `.origin` directive. All such sections will be placed one after another, starting at this address. The default for the load address is the base of system RAM (`0x80000000`). This is usually a poor choice, since the BIOS jump tables go here, so use of the `-B` option is almost always necessary when the assembler is performing the link and not all sections have a `.origin`.

Note that this option has no effect for relocatable COFF output files, since for such files it is the linker's responsibility to assign load addresses. The use of the linker for this purpose is strongly recommended. The assembler should be used to perform linking only for trivial programs.

Note also that the `.linkbase` directive may also be used to set the base load address.

-chip option Selects the version of the NUON chip for which assembly is targetted. The available options are `alpha`, `oz`, and `aries`. The `alpha` chip is supported only if LLAMA was built with special options (it is very obsolete and no units are available, probably). The `oz` chip is the BETA silicon, and is basically identical

to `aries` (from an assembler instruction point of view) except that in the `oz` chip if an RCU no-op instruction is needed to hold some `dec` instructions then a version of the `addr` instruction is synthesized.

The default of `aries` should be used for all actual production use.

-compiled Indicates that the code being assembled was generated by a compiler. This supresses warnings about performance issues (*e.g.* when a `nop` needs to be inserted for padding) which may be of interest to assembly language coders but is irrelevant for compiled code.

-cprecedence Forces the parser to use C operator precedence rules everywhere. LLAMA's default operator precedence rules (section 2.6.7) are not the same as those of C, although it does (always) use C rules when parsing C preprocessor directives (section 6.2).

-cNUM Assemble the program to run in cache. This affects how instruction padding is done, since for code running in the cache no packets may extend across a cache line boundary. `NUM` is the size of cache lines, in bytes; the default value for this (if just `-c` with no `NUM` is given) is 32. Code compiled for a given instruction cache size may be run if the hardware cache line size is set larger than the `-c` option specified, but will not work correctly if the line size is smaller.

See also the `.cache` and `.nocache` directives.

-Dsymbol=value Defines a symbol. If the value is omitted, a default value of `0xdeadbeef` is assumed.

-e errfile Output an error file suitable for use by the lisp emulator. This causes error messages to go into the file "errfile" instead of to the standard error output, and additionally causes the syntax of the error messages to conform to what the emulator wants (they're lisp expressions, and are only vaguely human-readable).

-fFMT Specify output file format. LLAMA can produce output in a variety of formats. The currently supported values for `FMT` are:

coff Create a COFF load file for the VM Labs linker to use. No options are currently available for this output format. Note that the COFF files created by `-fcoff` are not directly executable by the debugger; they must be linked first. To create an executable COFF directly, use the `-fecoff` output format.

ecoff Create an executable COFF load file. The resulting object file is not linkable, and must not contain any unresolved references.

list Create an assembly listing; equivalent to `-fasm,bin,expand-includes`. Note that listing files are not very useful if the resulting code is passed to the linker, since the assembler does not know where the code will be placed in memory.

asm(,bin)(,expand-syms)(,expand-includes)(,expand-all) Create an assembly language output file. This option is useful for inspecting what the optimizer would do to a program, or for assisting in hand optimization of code (see Assembly Language Optimization (section 4.3)). Several options may be given after the keyword `asm`; these must be seperated by commas. The options are:

`bin` – annotate the assembly language with the binary forms of instructions

`expand-includes` – with this option, the contents of all of the files included via `.include` directives are output; otherwise, just the `.include` itself is output

`expand-syms` – expand all symbols to their final definitions. The default is to pass symbol definitions through unchanged, e.g. for the code:

```
foo = 10
bar = r1
add #foo,bar
```

to be passed through more or less verbatim. If the expand-syms option is given, then the symbols will be expanded in place, and so llama will produce the output

```
add #10,r1
```

for the input above.

`expand-all` – expand symbols and `.module` directives. Has the same effect as `expand-syms` plus `expand-includes`, and in addition removes all `.module`, `.import`, and `.export` directives from the code. Labels are output prefixed with their module name, e.g.

```
    .module foo
mumble:
     add r0,r0
```

will be output as:

```
__foo__mumble:
     add r0,r0
```

Global labels (defined by `.import` and `.export`) will not have their module names prefixed.

veri(,width=nn)(,segheader)(,rom)(,prefix=pppp) Create a verilog load file for hardware simulation. Normally this will be a 128 bit wide file, with section starting addresses given explicitly with verilog's notation. Several options may be given after the keyword `veri`; these must be separated by commas. The options are:

`segheader` – causes a header to be prepended to each section. This header consists of the section starting address, and the section length (including the 8 byte header), as two 32 bit numbers. Without this option,

no header is prepended, and instead a verilog command is output which specifies the load address.

`width=nn` – causes `nn` bits of data (rounded up to the nearest byte) to be output per line. The default is 128 bits (the width of instruction ram).

`rom` – same as `segheader,width=8`; outputs a file suitable for loading into a (simulated) ROM.

`prefix=pppp` – causes the arbitrary string `pppp` to be output at the beginning of each line of data output. The default is to set the prefix string to a tab character. The `prefix` option must be the last option for `-fveri` that is given (since the prefix string may itself contain commas).

binary Create a raw binary output file. No references to unresolved symbols are permitted. Note also that the assignment of addresses should be such that there are no "gaps" in the output, since the assembler cannot include any gaps in the output file.

m68k Create a dump of data formatted for assembling with a 68000, Coldfire, or similar assembler. This option is equivalent to the combination of options: `-fveri,segheader,width=32,prefix=' .dc.l 0x'`.

mpo Create a .MPO load file for the debugger to use. No options are currently available for this output format, which is now considered obsolete (the COFF file format is preferred).

-g-old Output version 1.xxx debugging data. This option only has effect with COFF output formats, and causes the resulting COFF files to have debugging information compatible with old (pre September 1998) versions of puffin. The resulting COFF files will probably not be useful with GDB or with newer versions of puffin. This option is present only for compatibility with ancient tools; do not use it any more.

-i file Process file "file" as though it had been the object of a `.include` directive at the beginning of the first input file. Any number of `-i` options may be given on the command line (*i.e.* you may use this mechanism to include an arbitrary number of files).

-I directory Add the given directory to the path for `.include` directives. Paths added on the command line will be searched immediately after the current directory, in the order they are given. See the discussion of the `.include` directive for details on the search path.

-jlocal

-jextern Specifies whether `jmp` and `jsr` instructions are to be assumed to be in local MPE memory (`-jlocal`) or to be in external RAM (`-jextern`). The default is `-jlocal` unless the `-c` option has been given, in which case it is `-jextern`.

-Mnum Print a dependency list on the standard output, suitable for inclusion in a Makefile. This option overrides the normal output process; if `-M` is given, then

no output (other than the dependency list) is produced and no error messages are produced. An optional number may appear after the `-M` option to give the maximum length of any lines which will be output; this may produce more readable dependency files or may be necessary for some tools. Thus, `-M80` will tell llama to break lines as near to column 80 as it can. These broken lines will be terminated by a backslash followed by a newline (so that they may be used in makefiles). `-M` by itself is equivalent to `-M1000000`.

-nolisp Removes all before and after methods from the code. Useful for disabling debugging information.

-nologo Supresses the printing of the version identification information which is normally printed by LLAMA when it starts up.

-Osize Requests that the optimizer favor smaller code size over faster execution. This option has no effect on `-O1`, and does not always result in smaller code being output by `-O2`; it is more of a hint than a command!

-O*n* Gives the optimization level. `-O0` results in no optimization (the default). `-O1` results in fast but not particularly clever optimization. `-O2` results in slower but fairly good optimization. Levels `-O3` and higher may sometimes produce slightly better code, but take an enormously greater length of time to run.

If you want to use optimization, we recommend level `-O2` for final production, and level `-O1` for common use (because it is much faster).

Note that the use of any optimization level above `-O0` will result in code that may be exceedingly difficult to debug. See the section on optimization (section 4) for more details.

-o outfile Specifies the name of the output file. If this is not given, then output will be sent to the standard output of the process.

-r revision Specifies the default assembly language syntax to be accepted, corresponding to the revision level of the documentation. See also the `.revision` directive.

-v Verbose flag: prints some (possibly) interesting statistics about the input and output.

2. The LLAMA Assembly Language

2.1 Instructions

The instruction mnemonics supported by LLAMA are the ones given in the *MMP–L3B Specifications*, with the following exceptions:

addr The operand for the immediate form of the `addr` instruction should always be given as a `16.16` fixed point number. Older versions of the manual were ambiguous on this point. For backwards compatibility, the numbers `-16` to `+15` can be interpreted as `(-16<<16)`, `-15<<16`, etc., if the `-r19` command line option or `.revision 19` directive are included in the code.

The default revision is now 21, which causes no mapping to be applied to immediate operands of `addr`.

dotp,mul_p,mul_sv In older versions of the assembly language manual the immediate shift values for these instructions were given as `0`, `8`, `14`, and `16`, rather than `16`, `24`, `30`, and `32`. Code using these old shift values may be assembled by giving the `-r19` command line option, or the `.revision 19` directive.

ld_io,st_io The assembler accepts these as synonyms for `ld_s` (or `ld_v`) or `st_s` (`st_v`). Use of this form allows backwards compatibility with applications written for alpha silicon (which did not allow `ld_s` or `st_s` operations to I/O space). It also may aid readability, since it helps to clearly distinguish "ordinary" memory accesses and accesses to I/O registers.

In addition, the assembler understands that the `_io` forms of instructions are references to local memory. These forms may be used for all local memory accesses (even those that are not I/O registers) and this may help the assembler issue more useful warnings about the cache bug which is triggered when a local memory access follows a load instruction which causes a cache miss.

General For most instructions with a three operand form having two source registers and a destination register, LLAMA will also accept a two operand form in which the second source register is identical to the destination. For example, it is legal to write `addm r0,r2` instead of `addm r0,r2,r2`.

2.1.1 Instruction packets

Instruction packets are delimited with "{" and "}". For example,

```
{  sub r0,r1
   mul r2,r3
   rts eq,nop
}
```

is a packet containing three instructions: an ALU instruction, multiply unit instruction, and ECU instruction. All three instructions will execute simultaneously. See the

MMP–L3B Specifications for further details, and for the restrictions which apply to instruction combinations within a packet.

2.1.2 Before and after methods

Besides instructions, a packet may also contain one `before` or one `after` directive. Each such directive is followed by a debugger script expression (referred to as a before method or after method). `before` and `after` directives do not actually generate code, but they do cause debugging information to be output which instructs the debugger to place a breakpoint on the instruction packet containing the directive, and to execute the corresponding debugger script code either before (for the `before` directive) or after (for the `after` directive) the packet executes.

Not all versions of the Puffin debugger properly support `before` and `after` directives.

2.2 Equates

Synonyms may be defined for registers, condition codes, and expressions. The = operator is used for all of these. For example, the following are all legal directives:

```
myreg = r1
x = $1000
x2 = 2*x + 1
```

In a disassembly listing generated by the `-flist` format, the calculated values for equates to expressions will be displayed on the left, preceded by an = sign.

The scope of equates (and other symbols) may be controlled with the `.module` directive.

2.3 Labels and Symbols

Identifiers consist of arbitrary sequences of letters, digits, underscores, or dollar signs, beginning with a letter or underscore. Case is significant in identifiers; that is, the symbol `Foo` is considered different from the symbol `foo`.

A label is an identifier followed by a colon, and must appear before any instruction or directive on a line.

An identifier followed by two colons creates a global symbol; that is,

```
foo::
    .dc.s 1
```

is equivalent to:

```
    .export foo
foo:
    .dc.s 1
```

2.4 Local Symbols

A local symbol or label is specified as a backquote character ‘ followed by an identifier as specified above. The scope of a local symbol extends only between the nearest global labels, or within a macro body.

For example, the two definitions of ‘skip below represent distinct labels, and do not cause any conflict or warning about redefinition (because the intervening global label sign causes a new scope to begin).

```
main:
    cmp  #0,r0
    bra  ne,‘skip,nop
    mv_s #1,r0
‘skip:
    add  r0,r1
    rts  nop

sign:
    mv_s #1,r1
    cmp  #0,r0
    bra  le,‘skip,nop
    mv_s #-1,r1
‘skip:
    rts nop
```

2.5 Comments

LLAMA accepts two kinds of comments: block comments and line comments. Block comments are identical to those found in the C programming language; they begin with the two character sequence /* and end with the next occurence of the sequence */. Block comments may not be nested.

Line comments begin with either ; or // and extend to the end of the current line.

2.6 Expressions

There are a variety of kinds of expressions which LLAMA can accept; they are outlined below. Also, any valid expression enclosed in parentheses is a valid expression as well.

LLAMA will evaluate expressions as fully as possible when they are first encountered. Some components of an expression (for example, labels) cannot be evaluated until a second pass, since their values are not known during the initial pass. This is true even of labels which occur earlier in the source file than the expression being evaluated, because optimization may cause code (and hence label values) to change. This should mostly be transparent to you, except that you cannot use such

a non-constant value in the controlling expression of a `.if` or `.while` statement. For example:

```
foo:
 .if foo > $20100300
   .dc.s 1
 .else
   .dc.s 2
 .endif
```

is invalid, and will produce an error.

2.6.1 Numbers

LLAMA accepts integer and floating point numbers in a variety of input formats. Integer numbers in bases other than 10 may be specified in one of the following ways:

1. As a hexadecimal number prefixed by `$`.
2. As a hexadecimal number prefixed by `0x`.
3. As a binary number prefixed by `0b`.
4. As a base `r` number prefixed by `r$`, where `r` is expressed as a decimal number.

Floating point numbers must be specified in decimal, and may have an optional exponent given after `e` or `E`, *e.g.* the number `100.0` may also be specified as `1.0e2`.

Associated with every number is a scale. This is the number of bits to the right of the decimal point in the binary representation of the number. The default, usual scale is 0, so that for example the numbers `1`, `1.1`, and `1.9` are all represented by the binary bit pattern `0b0001`. Numbers with a scale other than 0 are created with the `fix` function; for example, the binary bit pattern of `fix(1.0,2)` is `0b100`. Scaled numbers are particularly useful for doing fixed point arithmetic.

Note that the `fix` function does not do any rounding.

2.6.2 Boolean operators

In general, any defined, non-zero value is treated as `true`, and a zero or undefined value is `false`. It is best not to rely on any use of undefined values; use the `defined` function if you are in doubt as to whether a symbol may be defined or not.

`v1 || v2` produces `v1` if it is `true`, otherwise produces `v2`.

`v1 && v2` produces `false` unless both `v1` and `v2` are `true`, in which case it produces `v2`.

`!v1` is `false` if `v1` is `true`, and vice-versa.

2.6.3 Comparison operators

These all produce `1.0` if true, and `0.0` if false. Comparisons with undefined or external symbols will produce an error.

`v == v` tests for equality.

`v != v` tests for inequality.

`v < v` and `v <= v` test for “less than” and “less than or equal to”, respectively.

`v >= v` and `v > v` test for “greater than or equal to” and “greater than”, respectively.

2.6.4 Bitwise operators

All numbers are converted to integer before being used in these operations. Fixed point numbers are scaled appropriately as part of this conversion.

`v1 | v2` is a bitwise logical or.

`v1 ^ v2` is a bitwise exclusive or.

`v1 & v2` is a bitwise and.

`~v1` finds the bitwise complement of `v1`.

2.6.5 Shift operators

All numbers are converted to integer before being used in these operations. Fixed point numbers are scaled appropriately as part of this conversion.

`v1 << v2` shifts `v1` left by `v2`. If `v2` is negative, then `v1` will actually be shifted right by `-v2`.

`v1 >> v2` shifts `v1` right by `v2`. If `v2` is negative, then `v1` will actually be shifted left by `-v2`.

2.6.6 Arithmetic operators

These operators work directly on the floating point representations of the numbers, so they are *not* converted to integer. Note in particular that, for example, `1/2` is `0.5`, NOT `0`.

`v1 + v2` finds the sum of `v1` and `v2`. The scale of the result is the larger of the scales of `v1` and `v2`, so that the expression `fix(2.0,10) + fix(2.0,16)` is equal to `fix(4.0,16)`.

`v1 - v2` finds the difference of `v1` and `v2`. If `v1` and `v2` are both symbols in the same section, then their difference is an absolute constant. The scale of the result is the larger of the scales of `v1` and `v2`, so that `fix(2.0,10) - fix(1.0,16)` is equal to `fix(1.0,16)`.

`v1 * v2` finds the product of `v1` and `v2`. The scale of the result is the sum of the scales of `v1` and `v2`, so that the expression `fix(2.0,10) * fix(2.0,8)` is equal to `fix(4.0,18)`.

`v1 / v2` finds the (floating point) result of dividing `v1` by `v2`. If an integer result is desired, use the `floor` function. The scale of the result is the larger of the scales of `v1` and `v2`, so that `fix(1.0,10) / fix(2.0,8)` is equal to `fix(0.5,10)`.

`-v` takes the negative of `v1`. The scale of the result is the scale of `v1`.

2.6.7 Operator precedence

The precedence of the operators, in order from lowest to highest, is:

```
||
&&
==  !=
>=  <=  >  <
|
^
&
-  +
*  /
<<  >>
! ~ - (unary operators)
```

(Here operators with the same precedence are listed on the same line). Thus, for example,

```
a || b+c*d&e
```

is parsed as:

```
a || ( (b+(c*d)) & e ).
```

It is often a good idea not to rely on operator precedence, but instead to use explicit parentheses if you are at all in doubt as to how an expression will be parsed.

Please note that LLAMA's operator precedence differs from that of C and C++. This may be a problem for expressions involving the shift operators and the bitwise logical operators (`|`, `&`, and `^`). The `-cprecedence` option may be used to change the operator precedence to match that of C.

While processing a C compatible directive like `#define`, LLAMA will always use C's operator precedence rules. This could lead to some confusion, so it is recommended that you put parentheses in to avoid any ambiguity.

2.6.8 Functions

The following functions are built in to the assembler. They all accept floating point parameters and produce a floating point result. They generally preserve the scale of their first arguments. For example `cos(fix(0.5,16))` produces the same result as `fix(cos(0.5),16)`. An exception is the `float` function, which ignores its input scale and always produces an integer with scale 0 (the 32 bit integer which is the IEEE single precision representation of the floating point input number).

Function Name	Meaning
`abs(x)`	absolute value
`acos(x)`	arccosine
`addrof(x)`	returns the address of an I/O register
`asin(x)`	arcsine
`atan2(x1,x2)`	arctangent
`ceil(x)`	rounds x upwards to the nearest integer
`cos(x)`	cosine
`defined(sym)`	returns 1.0 if `sym` is defined, `0.0` otherwise
`dtor(x)`	convert degrees to radians
`exp(x)`	*e* raised to the x
`fix(x,fracBits)`	convert real to fixed with fracBits fraction bits
`float(x)`	IEEE single precision binary representation of x
`floor(x)`	rounds x downwards to the nearest integer
`integer(x)`	converts a fixed point number to its integer representation
`log(x)`	log (base is *e*)
`log2(x)`	log (base is 2)
`pi()`	pi
`pow(x,power)`	raise x to power
`rottor(x)`	convert rotations to radians
`rtod(x)`	convert radians to degrees
`rtorot(x)`	convert radians to rotations
`sin(x)`	sine
`sqrt(x)`	square root
`tan(x)`	tangent

2.7 Predefined Symbols

In addition to the standard set of registers, I/O addresses, and so on, LLAMA predefines the following symbols:

Symbol Name	Value
`LLAMA_VERSION`	the version number of the running assembler
`LLAMA_OPTIMIZE`	the level of optimization specified with `-O`, or 0 if not optimizing
`NUON_VERSION`	0 for alpha, 1 for beta silicon

Moreover, there are some special “macro” like facilities similar to ones provided by the C preprocessor:

Symbol Name	Value
`__FILE__`	a string containing the name of the file currently being processed
`__LINE__`	a number giving the current line number

Unfortunately, the `__LINE__` macro is of limited use; it probably would be nice to have a way to make it into a string.

2.8 Conditional Assembly

LLAMA supports conditional assembly via the `.if` and `.while` directives. Both of these work in a very straightforward way. Both directives are followed by a boolean expression; this is simply an ordinary expression which is evaluated. If it evaluates to a non-zero value, it is considered to be `true`. If it evaluates to 0, it is considered to be `false`. If the expression cannot be evaluated (for example, if it contains an undefined symbol) then it is considered to be `false`. Note that at assembly time, the values of labels are *not* known; labels are not resolved until a later pass. Therefore, any expression containing a label will evaluate to `false`, or will produce an error. For example,

```
foo:
    .if foo
        .dc.s 1
    .else
        .dc.s 2
    .endif
```

will always output the constant 2.

LLAMA does not have a `.ifdef` directive; in its place use the `defined` function. For example:

```
    .if defined(bar)
        .dc.s 1
    .else
        .dc.s 0
    .endif
```

will output `1` if `bar` is defined, and `0` otherwise.

The expression in a `.while` directive is re-evaluated each time through the loop. The value of the expression must be changed by some directives inside the loop. In order to prevent infinite loops, LLAMA will abort any loop that executes more than 65536 times and will print an error message.

Example:

To generate a table containing the numbers from 1 to 100, you could use:

```
_ii = 1
.while _ii <= 100
    .dc.s _ii
    _ii = _ii + 1
.end
```

At the end of this `.while` loop, the variable `_ii` will have the value 101.

Note that the `-fasm` output format expands the results of conditonal assembly, so no `.if` or `.while` loops will be preserved when using this format. See limitations of assembly language output (section 4.3.1) for more details.

2.9 Macros

The macro facility in LLAMA is still under development, and is in a preliminary state right now. There are no facilities for token pasting in macros, so it is not possible (for example) to use a macro to construct an identifier from expressions. It is also not possible to use a macro with a variable number of arguments. Expansion of macro arguments does not take place within `before` or `after` methods, or inside of strings.

We hope to correct these problems in a future release of LLAMA. In the meantime, macros can be used for relatively simple purposes, for example to provide aliases for some instruction forms or to encapsulate simple sequences of instructions.

Example: a macro to hide the `nop` form of jump:

```
.macro jump place
    jmp place,nop
.mend
```

Example: a macro to set the value of a variable:

```
.macro setit foo,x
foo = x
.mend
```

Example: a macro to shift left by a positive constant, or right by a negative constant:

```
;; a macro to shift a register left by the
;; constant value "x"
;; if x < 0, shift it right instead
.macro shli x,rj
.if (x < 0)
    lsr #-x,rj
.else
    lsl #x,rj
.endif
.mend
```

Note that in the last example, the macro must be used like:

```
shli 5,r0
```

rather than like:

```
shli #5,r0
```

because the latter form would cause the `.if` within the macro body to expand to:

```
.if (#5 < 0)
```

which is not syntactically legal.

Syntactically, the formal argument name within a macro is simply an identifier. Macro argument names are looked for before opcode or symbol lookup, so it is possible for a macro argument to have the same name as an existing opcode or symbol. Beware of code like:

```
.macro jump_and_inc addr
{   addr #2,rx
    jmp addr,nop
}
.mend
```

Here the string *addr* will always be replaced with the argument to the macro, and hence the instruction `addr` will not be found; most likely a syntax error will result.

Note that the `-fasm` output format prints the results of macro expansion, so no macro definitions or invocations will be preserved when using this format. See limitations of assembly language output (section 4.3.1) for more details.

Note also that the scope of local symbols (section 2.4) (those beginning with a backquote character) is restricted to the macro body. This can make local symbols quite useful within macros.

3. Directives

The assembler recognizes a number of directives. The `before` and `after` directives were discussed earlier in the section on instruction packets (section 2.1.2). These directives *must* occur inside of instruction packets.

The `.if`, `.else`, and `.endif` directives may appear either inside or outside of instruction packets. All other directives discussed below must occur outside of instruction packets.

3.1 .align

`.align` *modulo*
Sets location to next multiple of *modulo* bytes. The number *modulo* will in fact be rounded up to the next largest power of 2, so that

```
.align 5
```

is identical in effect to

```
.align 8
```

Alignments stricter than 32768 bytes are not supported.

3.2 .align.s

`.align.s`
Sets location to the next scalar (32 bit) boundary. Same as "`.align` 4".

3.3 .align.sv

`.align.sv`
Sets location to the next small vector (64 bit) boundary. Same as "`.align` 8".

3.4 .align.v

`.align.v`
Sets location to the next vector (128 bit) boundary. Same as "`.align` 16".

3.5 .alignlog

`.alignlog` *n*
Sets location to the next multiple of 2^n bytes. Same as "`.align 1<<`n".

3.6 .ascii

`.ascii` *"string"*
Defines an ASCII string. Within the string, the following escape sequences are recognized:

`\"` a double quote

`\\` a backslash

`\nnn` where `nnn` is 3 octal digits: the character with the given (octal) ASCII representation

A trailing zero, if desired, must be explicitly given with `\000`.

3.7 .asciiz

`.asciiz` *"string"*
Defines a zero terminated ASCII string; other than the automatically added terminating zero, this is identical to the `.ascii` directive.

3.8 .binclude

`.binclude` *"filename"*
Includes the contents of *filename* as literal binary data. The program counter is aligned to a multiple of 4 bytes before the file is included, and the file is padded with 0's to occupy a multiple of 4 bytes in the output. See the discussion of `.include` for details on the search path for included files.

3.9 .bss

`.bss`
Sets current output to the `bss` section (section 3.49). Same as "`.section bss`".

3.10 .byte

`.byte` *data* [,*data* ...]
Defines one or more items of single byte (8 bit) data. Same as `.dc.b`.

3.11 .cache

`.cache`
Forces padding to be inserted to allow code to run from cache. This occurs even if no `-c` flag was given on the command line. `.cache` overrides any previous `.nocache` directive. The padding used is taken from the `-c` flag on the command line; if no explicit padding size was specified (or no `-c` was given) then a default cache line size of 32 bytes is assumed.

The `.cache` directive also causes the assembler to assume that any undefined (external) symbols are in cached memory. For example, `jmp` or `jsr` instructions where the target is at an unknown address will be assembled in the long form which permits an arbitrary 32-bit address.

3.12 .comm

`.comm` *symbol*,*length* [,*alignment*]
Declares a named common area. *symbol* will be created as a global symbol, and the linker will put it in the common area. The size of the symbol (amount of space reserved for it) will be determined by the linker as the maximum of the *length* operands given in all of the `.comm` directives in the files linked together. The alignment of the symbol will also be determined as the maximum of all of the *alignment* directives specified. Omitting *alignment* from a `.comm` directive is the same as specifying an alignment of 1 byte.

Note that *alignment* must be a power of 2, and may be at most 32768 bytes.

3.13 .data

`.data`
Sets current output to the `data` section (section 3.49). Same as "`.section data`".

3.14 .dc.b

`.dc.b` *data* [,*data* ...]
Defines one or more items of single byte (8 bit) data.

3.15 .dc.s

`.dc.s` *data* [,*data* ...]
Defines one or more items of scalar (32 bit) data. Alignment is forced to a scalar boundary.

3.16 .duc.s

`.duc.s` *data* [,*data* ...]
Defines one or more items of scalar (32 bit) data. Alignment is not forced to any boundary.

3.17 .dc.sv

`.dc.sv` *data* [,*data* ...]
Defines small vector (64 bit) data. Each element of the small vector is a 16 bit word, so `.dc.sv` is similar to `.dc.w`, except that alignment is forced to a 64 bit boundary instead of a 16 bit boundary.

Note that although each small vector contains 4 elements, `.dc.sv` does not actually check the number of elements given.

3.18 .dc.v

`.dc.v` *data* [,*data* ...]
Defines vector (128 bit) data. Each element of the vector is given as a scalar (32 bit) word.

Note that although each vector contains 4 elements, `.dc.v` does not actually check the number of elements given. Thus, it is exactly the same as the `.dc.s` directive, except that alignment is forced to a vector boundary.

3.19 .dc.w

`.dc.w` *data* [,*data* ...]
Defines one or more items of word (16 bit) data.

3.20 .duc.w

`.dc.w` *data* [,*data* ...]
Defines one or more items of word (16 bit) data. Alignment is not forced to any boundary.

3.21 .ds.b

`.ds.b` *num*
Defines space for *num* bytes of data. No alignment is imposed.

3.22 .ds.s

`.ds.s` *num*
Defines space for *num* scalars of data. Each scalar is 32 bits. Output is aligned to a 32 bit boundary.

3.23 .ds.sv

`.ds.sv` *num*
Defines space for *num* small vectors of data. Each small vector is 64 bits. Output is aligned to a 64 bit boundary.

3.24 .ds.v

`.ds.v` *num*
Defines space for *num* vectors of data. Each vector is 128 bits. Output is aligned to a 128 bit boundary.

3.25 .ds.w

`.ds.w` *num*
Defines space for *num* words of data. Each word of data is 16 bits. Output is aligned to a 16 bit boundary.

3.26 .else

`.else`
Optionally goes with `.if`; terminates the "if" part of the conditional assembly and starts a new part with the opposite sense. See the section on conditional assembly (section 2.8) for details.

3.27 .end

`.end`
Closes a `.if` or `.while`. See the section on conditional assembly (section 2.8) for details.

3.28 .endif

`.endif`
This is a synonym for `.end`. It is intended to be used to terminate a `.if` block, but it can actually be used anywhere `.end` can be. We do not recommend its use for anything but terminating a `.if` (with optional `.else`) block.

3.29 .error

`.error` *"msg"*
Specifies an error condition, and prints the string *msg* as an error message. This is

commonly used in `.if` blocks to detect conditions which will prevent proper assembly. For example, if some code requires version 2.60 or later of LLAMA it would be a good idea to prefix it with a test like:

```
.if LLAMA_VERSION < 2.60
.error "llama 2.60 or later required for this code."
.endif
```

3.30 .export

`.export` *name* [,*name* ...]
Declares one or more names as global and visible to other object files and modules (see `.module`). A label may also be exported by following it with two colons ("::") instead of one.

3.31 .float

`.float` *num* [,*num* ...]
Defines one or more four byte IEEE single precision floating point numbers.

3.32 .include

`.include` *"filename"*
`.include` *"prefix" "filename"*
Includes the source file *filename*. The following directories are searched, in the order listed:

1. The current directory.
2. In any directories specified by `-I` options on the command line, in the order given on the command line.
3. In the directories specified in the `LLAMA_PATH` environment variable, if one is found. This is the same as PATH – a semicolon separated list of directories.
4. In the `include` subdirectory of the directory pointed to by VMLABS, if that environment variable is defined.
5. Under Windows: in the `..\include` directory relative to where LLAMA is on the disk. That is, if LLAMA is started as `C:\VMLABS\BIN\LLAMA.EXE`, then the directory `C:\VMLABS\INCLUDE` is searched for include files.

If a *prefix* string is given, then the debugger will show that prefix in front of the file name. This is useful when the same file is to be included multiple times, for example in different overlays; the prefixes can then be used to distinguish the different instantiations of the file.

3.33 .if

`.if` *bool-expr*
If false, assembly is suppressed until the next matching `.else` or `.end`. See the section on conditional assembly (section 2.8) for details.

3.34 .ifdef

`.ifdef` *SYMBOL*
This is an alternate form of `.if` and is exactly equivalent to:

```
.if defined(SYMBOL)
```

3.35 .ifndef

`.ifndef` *SYMBOL*
This is an alternate form of `.if` and is exactly equivalent to:

```
.if !defined(SYMBOL)
```

3.36 .import

`.import` *name* [,*name* ...]
Declares one or more names as global and imported from another module or object file. This directive is required to import names from another module (see `.module`) but is optional when referring to symbols from a different object file.

3.37 .lcomm

`.lcomm` *symbol*,*length* [,*alignment*]
Declares a named common area in the `.bss` section. The symbol is entered as local to this object file (not global). The alignment of the symbol (in bytes) may be explicitly specified. If no alignment is specified, then scalar alignment is assumed, unless *length* is a known constant less than 3, in which case the length determines the alignment.

3.38 .linkbase

`.linkbase` *expr*
`.linkbase` *expr*,*maxsize*
Defines the base address to be used for sections without a `.origin` directive. In this respect it is similar to the `-B` command line option. However, the `.linkbase` directive also has an optional argument specifying the maximum total size of all sections placed starting at the given address. If the total size of overlays (and any other

sections whose load time addresses are determined by the assembler, rather than given explicitly) exceeds the specified maximum, an error is generated.

Please note that this directive is not passed to the linker, so `.linkbase` has no effect if LLAMA is not doing the link step (for example, if a non-executable COFF file is being created which will be passed to a linker).

3.39 .macro

`.macro` *name* [*param1* [,*param2* ...]]
Begins a macro definition. See the macros section (section 2.9) for details.

3.40 .mend

`.mend`
Ends a macro definition. See the section on macros (section 2.9) for details.

3.41 .module

`.module` *name*
`.module`
Creates a new namespace. Symbols defined within a module will be prefixed by the module name, so that they do not conflict with other symbols with the same name in other modules. If *name* is omitted, then the symbols will have global scope again. A `.module` directive lasts until the end of the current file, or until the next `.module` directive. Symbols from a module that are intended to be used by other modules must be made global with a `.export` directive, and to reference symbols from another module a `.import` directive must be used.

To refer in the debugger to symbol `foo` from module `bar`, use the name `bar%foo`.

3.42 .nocache

`.nocache`
Directs that padding should no longer be inserted in code to allow it to run in cached mode. Obviously, one should take care not to use this directive in code that may be run in a cached processor! This directive overrides any previous `.cache` directive or `-c` command line switch, and is in turn overridden by a following `.cache` directive.

`.nocache` also causes the assembler to assume that any undefined (external) symbols are in local memory. This applies to both data and instruction references.

3.43 .nooptimize

`.nooptimize`
Begins a section of code that the assembler should not attempt to optimize. This

section is terminated by a matching `.optimize` directive. The `.nooptimize` and `.optimize` directives nest, so that two consecutive `.nooptimize` directives require two `.optimize` directives before optimization will again be permitted.

3.44 .nowarnings

`.nowarnings`
Disables printing of (most) warnings. This may be used to suppress warnings in code which is correct but which for some reason LLAMA is suspicious about. For example, this might be useful to suppress warnings like "instruction with no effect" in code fragments which are being used for binary patches or overlays. Warnings must be turned on again by the `.warnings` directive. It is usually a poor idea to suppress warnings, so the section between `.nowarnings` and `.warnings` should be kept as small as possible. LLAMA will issue a warning for any `.nowarnings` directive which does not have a corresponding `.warnings` following it.

3.45 .optimize

`.optimize`
Ends a section of code that should not be optimized, and may allow optimization to begin again. Has no effect unless a `-O` flag was given to LLAMA. That is, `.optimize` states that optimization is permitted, but it does not actually force optimization to occur; only the `-O` command line flag activates optimization.

3.46 .origin

`.origin` *expr*
Specifies the origin for a section. At most one `.origin` directive should be present in any given section.

If a section is not given a `.origin`, then it will either be output as a relocatable section (if the output format is relocatable, for example if the `-fcoff` option was given) or else it will be assigned a load address by the assembler. The `-B` option or `.linkbase` directive may be used to control where such sections will be loaded.

The `local_ram` section has a default origin of `0x20100000`. Similarly, the `instruction_ram` section has a default origin of `0x20300000`.

3.47 .overlay

`.overlay` *section_name*
`.overlay` *section_name*, *maxsize*
This specifies a section which will be used as an overlay. The initial (load time) location of the overlay in SDRAM or system RAM will be assigned by the linker (for COFF output files), or may be specified with the `-B` flag or `.linkbase` directive for

other output formats. The run-time code location must be given with a `.origin` directive. See Overlays (section 5) for more information on overlays.

A maximum size that will be allowed for an overlay may optionally be specified as a second argument for `.overlay`. If the size of the overlay exceeds this maximum size, then an error will result. Size checking is performed only when the assembler is producing executable output; for relocatable COFF object files the linker must be used to check section sizes.

The `.overlay` directive is identical to the `.section` directive, except that it may also arrange for overlay specific debugging information to be placed in the output file (if the output file format supports this).

The *section_name* argument may have at most 8 characters. This is a restriction imposed by the COFF file format, but it applies regardless of what kind of output is actually being produced.

It is not necessary for all code and/or data in an overlay to be contiguous in the source code; for example, it is perfectly legal to have code like:

```
.overlay foo,128
sub r1,r0

.overlay bar
mv_s #0,r0

.overlay foo
cmp #0,r0

.overlay bar
st_s r0,bar_data
```

The above snippet of code has the same effect as:

```
.overlay foo,128
sub r1,r0
cmp #0,r0

.overlay bar
mv_s #0,r0
st_s r0,bar_data
```

3.48 .revision

`.revision` *number*

Specifies the version of the instruction set mnemonics (section 2.1) and forms to use. This is useful mainly for supporting old code which used different conventions for the NUON assembly language.

`.revision 19` specifies the initial version of the assembly language, as specified in revisions 19 and 20 of the assembly language manuals.

`.revision 20` is the same, except that the immediate shift values 8, 14, and 16 used by the `dotp`, `mul_p`, and `mul_sv` instructions should have 16 added to them (to make the shifts relative to a complete 32 bit number). The old shift values are still accepted but give warnings. In addition, warnings are issued for `addr` instructions with small immediate constants.

`.revision 21` is similar to `.revision 20`, except that warnings are no longer issued for `addr` instructions with small constants; instead, these are assembled to actually add the literal immediate used. This will break old code which expects the small constants to be shifted left by 16 before use. Also, the small vector shift value 16 is now interpreted as a 32 bit shift, so it now means the same as the old shift value of 0.

The assembler now (as of version 2.20) defaults to `.revision 21`.

3.49 .section

`.section` *name*
`.section` *name*, *maxsize*
Sets current output to the section named *name*. This may be one of LLAMA's predefined sections (which are `text`, `data`, `bss`, `dtram`, `dtrom`, `iram`, `irom`, and `sdram`) or it may be a completely new name, in which case a new section will be created. Note that this means that a spelling mistake in a `.section` directive will not be flagged by the assembler, but could result in unexpected behaviour.

A maximum size that will be allowed for a section may be optionally be specified as a second argument for the `.section` directive. If the size of the section exceeds this maximum size, then an error will result. Size checking is performed only when the assembler is producing executable output; for relocatable COFF object files the linker must be used to check section sizes.

Only the first 8 characters of the *name* argument are significant. However, for backwards compatibility certain long names are automatically translated into shorter names, as follows:

Long Name	Short Name	Meaning
`instruction_ram`	`iram`	MPE local instruction RAM
`instruction_rom`	`irom`	MPE local instruction ROM
`local_ram`	`dtram`	MPE local data RAM
`local_rom`	`dtrom`	MPE local data ROM
`external_ram`	`sdram`	SDRAM

Please note that the `sdram` section is not understood by the linker; it is placed into SDRAM automatically only if the assembler is doing the link.

The C compiler makes use of the following sections:

Name	Meaning
`text`	program code and constants
`data`	initialized data
`bss`	uninitialized data
`comm`	uninitialized data (obsolete, use `bss` instead)
`heap`	memory to be allocated by malloc
`ctors`	C++ constructor functions
`dtors`	C++ destructor functions
`intdata`	data in internal MPE memory

User defined sections will probably be used most often for overlays (section 5). In this case one should use the `.overlay` variant of the `.section` directive, since it sets some flags in the object file which the debugger can use.

It is not necessary for all code and/or data in a section to be contiguous in the source code; for example, it is perfectly legal to have code like:

```
.section foo
sub r1,r0

.section bar
mv_s #0,r0

.section foo
cmp #0,r0

.section bar
st_s r0,bar_data
```

The above snippet of code is identical to:

```
.section foo
sub r1,r0
cmp #0,r0

.section bar
mv_s #0,r0
st_s r0,bar_data
```

3.50 .segment

`.segment` *name*
`.segment` *name*, *maxsize*
This is a synonym for `.section`.

3.51 .start

`.start` *symbol*
Specifies a starting location for execution (*i.e.* the program's entry point). This directive has useful effects only for the MPO output format (and hence it is obsolete). For COFF output formats the execution start address is set by the linker.

3.52 .text

`.text`
Sets current output to the `text` section. Same as "`.section text`".

3.53 .warn

`.warn` *"message"*
Prints the indicated message on the standard error output as a warning. This is similar to `.error`, but does not cause assembly to be halted.

3.54 .warnings

`.warnings`
Re-enables warning messages if they have previously been suppressed by a `.nowarnings` directive. No matter how many `.nowarnings` appeared before, `.warnings` forces warnings to be printed again.

3.55 .while

`.while` *expr*
While the expression is true (nonzero), repeatedly assembles lines up to the next `.end` directive. See the section on conditional assembly (section 2.8) for details.

3.56 .word

`.byte` *data* [,*data* ...]
Defines one or more items of single word (16 bit) data. Same as `.dc.w`.

4. Optimization

4.1 General

If LLAMA is given the `-O` option, it will optimize the input code. To do this, it tries to move instructions as far ahead in the instruction stream as possible. It never moves instructions backwards, only forwards; this simplifies some of the algorithms, but obviously this does cause some optimizations not to be performed. The main goal of LLAMA has been safety (it should not take a valid assembly language program and produce an invalid one) rather than performance, and so some optimizations are foregone for safety's sake.

When moving instructions, LLAMA will also try to substitute equivalent instructions that use a different function unit. For example, it will interchange the equivalent forms of `add` and `addm`, `sub` and `subm`, and `copy` and `mv_s`. It will also substitute `sub rn,rn` for `mv_s #0,rn`, and `mul #1,rn,>>#-A,rn` for `lsl #A,rn`.

LLAMA keeps track of dependencies, including condition codes produced. It will never move an instruction past another instruction which changes a resource which the first instruction depends on, nor will it schedule instructions so that resources conflict (e.g. placing a `mv` instruction immediately after a `ld`). It also tries to detect such conflicts in the input file and issue warnings about them; however, if the input code contains such conflicts, it is possible that LLAMA may not notice and will reproduce them in the output.

LLAMA is not very clever about branches, and is extremely cautious about preserving register contents and condition codes across branches and subroutine calls. Some of its cautiousness can be relaxed with the `-b` flag. If `-b` is given, then LLAMA assumes that the code following any branch will not use condition codes generated before the branch. This is in fact a common case for hand written assembly code.

The difference can be shown by example. Given the input sequence:

```
{   add  #2,r2
    mv_s r1,r0
}
{   mv_s r2,r3
    jsr  foo
}
    mv_s r4,r5
    mv_s r6,r7
```

LLAMA will ordinarily not be allowed to change any of the `mv_s` instructions into a `copy` instruction, because theoretically the `foo` subroutine might use some of the flags set by the `add` instruction. The `-b` flag tells LLAMA that it's OK to assume that flags will not be used after any branch, and so in the example above it will be permitted to change `mv_s` instructions into `copy` instructions.

Note that gcc does instruction scheduling into delay slots, so in fact it is dangerous to use `-b` with compiled code!

4.2 Optimization of Binary Code

There are two ways to use LLAMA to optimize your code. LLAMA can simply optimize the code as it assembles, and produce optimized binary output (this will be the default action if you give a `-O` flag); or, you can specify that you want to see the resulting (optimized) assembly language by giving the `-fasm` option to LLAMA.

Producing optimized binary code is certainly the simplest method, and is suitable for production use on already debugged code. If asked to optimize code that is being produced in a binary output format (*e.g.* `-fmpo` or `-fcoff`) then LLAMA will perform the optimizations automatically and produce an optimized binary executable. Note that because optimization typically involves moving instructions to new packets, and sometimes eliminating instructions or packets altogether, the debugger is likely to become quite confused if asked to single step through an optimized function. For this reason, if you plan to debug the code you should either forgo optimization, or do the optimization "by hand" by asking for assembly language output with `-fasm` and then merging LLAMA's changes back into your code, as described below.

4.3 Assembly Language Optimization

You can use LLAMA as a tool to assist you in optimizing your assembly language code. If LLAMA is given the `-fasm` output format option, then it will produce a (somewhat readable) assembly language output version of its input. You can look at this code, compare it to your original, and use LLAMA's suggestions to improve your code.

There are several options which can be helpful in this task. The `-i` option allows you to include a file, which will not appear in the resulting assembly language output. This allows you to optimize a small piece of a larger program. For example, suppose that `project.s` is the top level file for some project, which consists of:

```
.include "baz.i"
.include "bar.s"
.include "quux.s"
```

If you wish to optimize just the file `quux.s`, you should use the command line:

```
llama -fasm -O2 -o quux_new.s -i baz.i
      -i bar.s quux.s
```

This tells LLAMA to include the `baz.i` and `bar.s` files before attempting to process `quux.s`. If we didn't do this, then there would likely be undefined symbols (anything defined in `baz.i` or `bar.s` that was used in `quux.s`).

Another way of avoiding such undefined symbols is to use the `-D` directive to define the symbol. For instance, if in the example above the only symbol from `baz.i` or `bar.s` that was used in `quux.s` was a symbol called `baz`, with some constant value, then we could have written:

```
llama -fasm -O2 -o quux_new.s -Dbaz quux.s
```

If we knew that, for example, `baz` had the value 0, then we could have written:

```
llama -fasm -O2 -o quux_new.s -Dbaz=0 quux.s
```

The `-D` option only works for constant value symbols; it is not possible to define a register symbol in this way. If no explicit value is given with `-D`, LLAMA assumes a value of `$deadbeef`; this is because small values (especially 0) may be treated specially by the assembler, so in the absence of an explicit value LLAMA must assume that the symbol may require 32 bits to hold it.

4.3.1 Limitations of assembly language output

When doing assembly language output, LLAMA actually assembles the program into an internal format, optimizes it, and then disassembles the result. This has a number of consequences which you should be aware of:

1. LLAMA tries to keep comments with the instructions to which they refer, but it isn't always successful at this. You should double check that the comments in the output code make sense, and that none are missing.

2. Conditional assembly directives, and macro definitions are not preserved, and are instead expanded into their final form. So, for example, the code fragment:

   ```
   .if 1
       add #1,r0
   .else
       sub #2,r0
   .endif
   ```

   will be output by LLAMA as simply:

   ```
       add #1,r0
   ```

3. Constants in `.dc.s` and similar directives may be expanded to their final form, *e.g.*

   ```
   foo = $10
   .dc.s foo
   ```

   may be output as:

   ```
   .dc.s 16
   ```

For all of these reasons, it is important that you examine LLAMA's output and merge its suggested changes into your source code (or vice-versa), rather than simply having LLAMA overwrite the original source.

5. Overlays

The LLAMA assembler and the VM Labs linker can work together to create code overlays. The general procedure may be summarized as follows. For each piece of code or data that is to be overlaid, create a new assembler section using the `.overlay` directive. The overlay section must have a unique name of 8 characters or less. Each overlay section should have a `.origin` directive giving the location where the code is to run (NOT where it is to reside in memory while not active). The linker can then be used to link these sections into RAM or ROM. Note that:

1. The linker doesn't care where the overlays will actually run; it is only concerned with their "load time" location (where the code lives when it isn't active).
2. The assembler doesn't care about "load time" location, only run time.
3. Some piece of code is going to have to do the actual loading of the overlay code into an MPE, and start the overlay running.

The last task is simplified somewhat by the linker's automatic variable facility. For any section `foo`, the linker creates symbols named `_foo_start` and `_foo_size` which give the start address and size for the section as it appears in the load map (*not* the run time addresses). Also note the prepended underscore; this is for C compatibility.

Example:

Suppose we have two pieces of code, with associated data, which will run in an MPE and which will be loaded into MPE instruction memory at address `0x20300800`. Then we might do something like the following:

```
.overlay code1
.origin $20300800
;; put first bit of MPE code here

.overlay data1
.origin $20100000
;; put data for first bit of MPE code here

.overlay code2
.origin $20300800
;; put second MPE overlay code here

.overlay data2
.origin $20100000
;; put data for second MPE overlay here

...

;; here is where we load overlay 1
```

```
;; the load_overlay function takes
;; the following parameters:
;;    r0 == external address of code
;;    r1 == size of code (in bytes)
;;    r2 == run time address for code


;; import symbols defined by linker
.import _code1_start,_code1_size
.import _data1_start,_data1_size

mv_s  #_code1_start,r0
mv_s  #_code1_size,r1
mv_s  #$20300800,r2
jsr   load_overlay,nop

mv_s  #_data1_start,r0
mv_s  #_data1_size,r1
mv_s  #$20100000,r2
jsr   load_overlay,nop

....

;; A similar procedure would be used for
;; the code2 and data2 sections
```

Note also that although we chose fixed addresses for our overlays in the example above, the `.origin` directive can be used with labels. The only restriction is that code overlays must start on a vector boundary.

6. Interfacing C and Assembly

6.1 Calling Conventions

Assembly language functions which are to be called from C must obey the following guidelines:

1. The first ten arguments are passed in registers `r0` through `r9`. Other arguments are passed on the C stack (pointed to by `r31`.
2. The C stack pointer is general purpose register `r31`. The hardware stack pointer `sp` is reserved for interrupt purposes. Note that the C stack grows downwards (predecremented), and must always be vector aligned. Assembly language code must preserve the `sp` register, and may use `push` and `pop` instructions (but should not assume that the stack is larger than 128 bytes).
3. The function's return value should be placed in `r0`. 64 bit values are returned in `r0` and `r1`.
4. The function may modify general purpose registers `r0` through `r11`, and general purpose register `r29`. The other general purpose registers (`r12` through `r28` and `r30` through `r31`) must be preserved.
5. I/O registers concerned with bus transfers (the main bus DMA registers, other bus DMA registers, and comm bus registers) may be modified.
6. The `rc0` and `rc1` registers may be modified by the called function, and need not be preserved.
7. All other registers should be preserved, including (in particular) the `sp` and `acshift` registers.
8. C callable functions must be assembled to use the cache, in its default configuration (32 byte lines). This may be done with the `-c` flag to LLAMA, or by putting the `.cache` directive into the code. No direct references to MPE instruction RAM or MPE data RAM should be made without care being taken to flush the cache first. This is particularly of concern when the assembly language code wishes to issue DMA commands, since these always use an MPE internal address.
9. If the assembly language function is to call a C function, it must be aware that the C function may change registers `r0` through `r11`, and also `r29` (which is used by the C compiler to perform linkage).

6.2 Header Files

As of version 2.62, LLAMA can understand a limited number of C programming language constructs and preprocessor directives. This allows certain (carefully constructed) header files to be used by both C and assembly.

The C preprocessor directives understood by LLAMA are blindly translated into LLAMA equivalents. If the resulting expressions are not valid assembly language directives, a syntax error will result.

#define Translated into a symbol definition. Only simple arithmetic expressions may be defined; `#define` may not be used to define macros or strings. A line like:

```
#define FOO
```

is translated by LLAMA into:

```
FOO = 1
```

A line like:

```
#define BAR (BAZ+QUUX)
```

is translated by LLAMA into:

```
BAR = (BAZ+QUUX)
```

Note that for the duration of the line beginning with `#define` LLAMA uses C's operator precedence rules rather than its own. This affects the left and right shift operators and the bitwise and, or, and exclusive or operators.

#else Translated into `.else`.

#endif Translated into `.endif`.

#error Translated into `.error`.

#if Translated into `.if`.

Note that for the duration of the line beginning with `#if` LLAMA uses C's operator precedence rules rather than its own. This affects the left and right shift operators and the bitwise and, or, and exclusive or operators.

#ifdef Translated into `.ifdef`.

#ifndef Translated into `.ifndef`.

#include Translated automatically into `.include`. Note that the search path used by LLAMA may differ from that used by the C compiler; see the discussion of `.include` for how LLAMA finds files. Also note that the argument to `#include` must be in quotes; in other words

```
    #include <foo>
```

will not work.

`#line` This directive is ignored.

`#pragma` This directive is ignored.

Besides the preprocessor directives, LLAMA can also understand a particularly limited version of the C `enum` statement. If every element of the enumeration appears on a separate line, and each one is assigned an explicit value, then LLAMA will create symbols with the names of the enumeration tags and with the assigned values. That is,

```
enum {
  a=1,
  b=2
};
```

will be accepted by LLAMA and translated as:

```
  a = 1
  b = 2
```

However, the equivalent (in C) declaration:

```
enum {
  a=1,
  b
};
```

is not understood by LLAMA and results in a syntax error.

7. Bugs and Shortcomings

7.1 Bugs

These are known bugs, *i.e.* they can cause LLAMA to produce invalid output under certain circumstances.

1. The `st_s` immediate instruction allows for relocation only in the immediate value, not in the address; that is, in

   ```
   st_s #nnnn,labelC
   ```

   the assembler must know the address of labelC.

2. LLAMA does not resolve labels until its final pass; as a result, it must initially assume that operands involving labels may require 32 bits. This can lead to spurious errors about too many instructions in a packet using the 32 bit instruction prefix. It can also (sometimes) lead to a spurious "packet too large" error for packets containing a short (16 bit) branch which are exactly 128 bits in size. These latter errors are very rare.

3. LLAMA assumes that external subroutines do not end with two tick operations. If this is not true of a subroutine, then under certain circumstances LLAMA might re-schedule instructions to use the result of a subroutine before it is ready. Obviously this bug is only a potential hazard if optimization is enabled, and only if LLAMA hasn't seen the actual subroutine definition itself and hence can't determine whether the subroutine contains a two tick operation.

4. Labels may not appear inside of packets. This is in fact a very difficult problem to solve, and it's possible that LLAMA will never allow this, even though (theoretically) the NUON media architecture will allow it.

7.2 Shortcomings

These are annoying problems with LLAMA that are short of actual bugs.

1. `-O2` and higher levels of optimization are slow; `-O3` and higher are excruciatingly slow and not really much better.

2. Although LLAMA will convert `mv_s #0,r0` into `sub r0,r0`, it won't try the obvious alternate `subm r0,r0,r0` for this particular instruction.

3. There are a number of limitations in the `-fasm` output format; see the section on assembly language optimization (section 4.3.1) above.

4. Macros are still quite limited; see the macros (section 2.9) section for some details.

5. LLAMA consumes quite a bit of memory to run; it may take up to 1K bytes of memory for each line it processes (usually it's less than this!).

8. Error Messages

The assembler can generate a number of warning and error messages which may seem cryptic. Here are some explanations of some of these:

8.1 Unable to find previous instruction packet for padding

This warning message from the assembler is harmless. Some instructions must be aligned in particular ways; for example, no instruction can cross a cache line boundary. The assembler forces alignment by inserting padding into instruction packets. This padding uses space, but does not take any time to execute. The operation of padding is normally transparent, but there are some times when the assembler needs to insert padding to force alignment but is unable to find a packet to insert the padding into. For example, this can happen if some data has been inserted in the middle of code. In these circumstances, the assembler is forced to insert a `nop` instruction. The warning informs the user that this has happened. It's useful for an assembly language programmer, since the extra `nop` instruction adds a tick, which could be a problem in a carefully crafted inner loop.

8.2 Cache stall may cause repeated read/write to register

This is a warning about a bug in the beta hardware that can cause problems with accesses to certain volatile registers. If the instruction that causes this warning is in a branch delay slot, move it out of the delay slot. If it is in a large packet, try moving it to a smaller packet or make it an instruction all on its own. If all else fails, insert one or two `nop` instructions before the offending instruction.

8.3 Obsolete instruction form

The syntax for the `addr` instruction changed in order to accomodate some new instruction semantics which became possible late in the design of the chip. The old syntax took

```
addr #1,rx
```

to mean "add 1 in 16.16 fixed point format to `rx`". However, it is in fact possible to add an arbitrary 32 bit constant using `addr`, and so an ambiguity arose; how could we specify adding small literal constants? As of revision 20 of the instruction syntax, the `addr` instruction always takes a 32 bit constant, so the example above should become:

```
addr #1<<16,rx
```

8.4 Obsolete shift

There are several instructions (for example, `mul_sv`) which operate on small vectors, which are the upper 16 bits of each of four consecutive registers. Because only 16 bits were involved in the operation, the original assembly language syntax treated all shifts as being “16 bit”, that is, considering only the bits involved in the operation. However, since the small vectors occupy the *upper* 16 bits of registers, this can be confusing. Other multiply operations have shift counts relative to the full 32 bit registers. For consistency’s sake, and to allow for future expansion of the instruction set, the assembly syntax has been changed for revision 20 of the instruction set to make the small vector operations use full 32 bit shifts. During the switch over the assembler will accept old forms but issue warnings. It is important to correct the warnings, because there is one ambiguity (the old shift of 0 will become 16, which conflicts with the old shift of 16 which has become 32). Follow the assembler’s instructions and your code should be OK.

8.5 Local access following remote load may trigger cache bug

The beta version of the Nuon media processor (aka “Oz”) has a number of cache bugs. One of these is that if a load instruction causes a cache miss and the next instruction is an access to local memory (including any memory-mapped register), then the MPE will hang indefinitely. Since only indirect loads can cause cache misses, the assembler flags any local memory access following an indirect mode if the cache is active (the `-c` flag or a `.cache` directive was given). If you’re sure that the indirect load will not cause a cache miss, then you may be able to indicate this to the assembler by using the `ld_io` instruction instead of `ld_s` or `ld_v`.

Unfortunately, these cache bugs were not fixed in the “Aries” version of the chip, and remain a problem.

Index