TECHNICAL NOTE	ONDEMAND MICROELECTRONICS
Project:	SVENm
Project No:	n.a.
Author:	Ralf Schreier
Document ID:	svn_note031
Last modified:	2006-12-06, 9:56 AM
Version:	1.1
File: Z:\pdoc\note\svm_note31_v1r1x.doc	

Chili ASM short manual

1 Introduction

This document summarizes guidelines for the inline assembly programming.

2 Revisions

Version	Date	Author	Description
1.1	2006-11-23	R. Schreier	Initial version

3 References

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s9

http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_4.html#SEC93

Local symbols for jumps:
http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_chapter/as_5.html#SEC45

4 Assembly programming guidelines

4.1 Data register file issues

4.1.1 Data types & alignment

INT	32 bit
SHORT INT	16 bit
LONG INT	64 bit
CHAR	8 bit
FLOAT	32 bit
DOUBLE	64 bit

- All core registers are 32 bit.
- Memory can be addressed in 1 Byte (= 8 bit) steps.
- 4 and 2 Byte Data Types have to be aligned (%4, %2):

  Int16 src_data[…] __**attribute**__ **((aligned (4)))** = {…};

4.1.2 Register allocation

r0	function return value
r1 - r6	argument register
r7 - r14	callee-saved registers
r15- r54	**caller-saved temporary registers**
r55- r60	reserved global registers (caller-saved temporary registers)
r61	frame pointer
r62	stack pointer
r63	return address

Note 1:
For **inline assembly routines preferably r15 to r60 should be used** as temporary registers because they don't need to be saved in leaf functions (i.e. if no function calls from inside the inline assembly function are issued).

Note 2:
r55 to r60 shall be reserved for global variables. They must be saved to the stack if they are used for other purposes (details in 4.2.5).

4.1.3 Function stack frame

- first 6 Arguments are passed in r1-r6;
- structures are passed on the stack.
- For functions with variable argument count the arguments are passed on the stack.

4.2 Inline assembly syntax

4.2.1 General syntax

```
asm volatile (
        // inline assembler code
        "{%[a] = %[b] + %[c];}\n"
        "{;;;;}\n"
        "{;;;;}\n"
        // output operands
        : [a]"=r"(a)
        // input operands
        : [b]"r"(b),[c]"r"(c)
        // list of clobbered registers
        //:"r16",”r17”
);
```

Notes:

- The **volatile** keyword should be used to avoid moving or deletion of the assembly code within the C-code. This is especially critical if output operands are declared and the assembly routine also modifies memory. (An inline asm function without return values is automatically considered as volatile.)
- Omitting the volatile keyword sometimes leads to better performance (fewer registers are saved on the stack).
- The **output and input operand** lists map the C-environment variables “(var)” to the assembly registers [var]. Names must not be identical and the assembly names should be short and intuitive as the code lines get pretty long.
- Inline assembly does not care about data types and casting. Variables are simply 32 bit numbers in registers and it is up to the programmer to interpret them correctly.
- (handling of long int and doubles?)
- **Writing to input operand registers is not allowed!** It can lead to side effects as the C-compiler is not aware of the modification and the original content of the register might be needed later in the function. Use inout registers when appropriate (4.2.2).
- The **register allocation** of input and output operands is handled by the C-compiler. It is not possible to lock input/output operands to a specific register (exception: 4.2.5). Note: if input operands of a function are passed directly to an inline assembly routine they are usually located in r1 to r6.
- **Temporary registers** used in the inline assembly (preferably r15 to r55) must be listed as clobbered registers. The C-compiler will try to avoid using these registers if possible or save and restore them if necessary. **Missing clobber registers can result in very-hard-to-debug side effects!**

- **Be aware of the implicit register modification of the permute instruction!** Don't forget to clobber the 2nd output register.
- (What is the "memory" clobber good for?)
- Upon **completion of the inline assembler routine**, the compiler does not verify if all pending memory writes are completed. Hence, if the result of a memory write access is needed immediately by the C-environment, the programmer must insert nop statements at the end of the assembly routine (refer to 4.3 for details).
- The **inline assembler code is not optimized or verified** by the compiler. Obsolete empty lines "{}\n" result in nops of all slots. No register modifications are checked for correctness! The code is basically copied into the assembler output file as it is.

4.2.2 Inout operands

- Operands which are used as input and output are declared with "+r" in the output operands section.

```
asm volatile (
        "{%[a] = %[a] + %[c];}\n"
        "{;;;;}\n"
        "{;;;;}\n"
        // output operands
        : [a]"+r"(a)
        // input operands
        : [c]"r"(c)
        // list of clobbered registers
        //:"r16","r17"
);
```

4.2.3 Output memory sections

- Output memory sections should be declared in the output operands section to inform the compiler about modified memory sections. Otherwise the code might be moved or deleted by compiler optimizations.

```
asm volatile (
        "{;;;;}\n"
        "{;;;;}\n"
        // output operands
        : "=m"(*base)
        // input operands
        : [pointer]"r"(base)
);
```

4.2.4 Labels & Directives

- For a complete list of directives (.if, .else, …) refer to

http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_chapter/as_5.html#SEC45

- Labels use the following form:

```
asm volatile (
        "{jump (label1);}\n"
        "{%[a] = %[a] + %[c];}\n" // 1st branch delay instruction
        "{%[a] = %[a] + %[c];}\n" // 2nd branch delay instruction
        "{%[a] = %[a] + %[c];}\n" // 3rd branch delay instruction
        "{%[a] = %[a] + %[c];}\n" // 4th branch delay instruction
        "{%[a] = %[a] + %[c];}\n" // 1st instruction after branch delay
        "{;;;;}\n"
        "label1:"
        "{;;;;}\n"
        // output operands
        : [a]"+r"(a)
        // input operands
        : [c]"r"(c)
);
```

- **Notes:**
  - The Chili DSP core has a branch delay of 4 instructions. Hence, the four lines of code following the jump instruction will be executed before the jump is executed.
  - Labels should *not* be written in curly brackets {}, as this introduces an additional nop instruction in the code.
  - It is of *not* allowed to jump from one inline asm routine to another one as the effects on the C-environment can not be foreseen.

- **If inline assembler code is replicated (i.e. inlining, macros, code unrolling) local symbols must be used.**

- To define a **local symbol**, write a label of the form N: (where N represents any positive integer). To refer to the most recent previous definition of that symbol write Nb, using the same number as when you defined the label. To refer to the next definition of a local label, write Nf - The b stands for "backwards" and the f stands for "forwards".

```
asm volatile (
        "{jump (1f);}\n"
        "{%[a] = %[a] + %[c];}\n"
        "{%[a] = %[a] + %[c];}\n"
        "{%[a] = %[a] + %[c];}\n"
        "{%[a] = %[a] + %[c];}\n"
        "{%[a] = %[a] + %[c];}\n"
        "{%[a] = %[a] + %[c];}\n"
        "1:"
        "{;;;;}\n"
```

```
                // output operands
                : [a]"+r"(a)
                // input operands
                : [c]"r"(c)
        );
```

4.2.5 Variables in specified registers

- GNU C allows you to put a few global variables into specified hardware registers, e.g.

  register Int32 testvar asm ("r60");

- Global register variables reserve registers throughout the program.
- Local register variables in specific registers do not reserve the registers, except at the point where they are used as input or output operands in an asm statement.

- **Note:**
  The compiler does not verify if reserved registers are accessed incorrectly, i.e. clobbering a reserved register does not save and restore the register and it does not issue a warning.

4.3 Mandatory nop operations

- Some instructions are recognized by the pipeline scheduler only after a certain delay. If the result register is read before the scheduler is aware of the instruction, a false value will be returned.

- Mandatory delays must be considered for the following instructions
  - **Port** read to core register: 2 instructions
  - **Mult32**: 2 instructions
  - Permload: ??

4.4 Permute instruction: hints and side effects

- If **multiple perm operations** are issued in the same execution cycle it must be ensured that the (explicit and implicit) output registers do not overlap.

- The permute operation can modify two output registers. In the case where just the first (explicit) output register shall be modified (i.e. permval = 0xAAAAXXXX), the second (implicit) register can be destroyed by side effects if a second instruction form another slot tries to write it in the same execution cycle.

- Hence, the results of a perm operation should always be stored in non-overlapping clobbered registers. Otherwise, if complier compiler-assigned output registers are used, the neighbouring register of an output operand is unknown which can lead to serious side effects.