Restrictions

Take note of the following restrictions if you place a variable at an absolute address:

- The argument of the `__at()` attribute must be a constant address expression.
- You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses.
- When declared `extern`, the variable is not allocated by the compiler. When the same variable is allocated within another module but on a different address, the compiler, assembler or linker will not notice, because an assembler external object cannot specify an absolute address.
- When the variable is declared `static`, no public symbol will be generated (normal C behavior).
- You cannot place structure members at an absolute address.
- Absolute variables cannot overlap each other. If you declare two absolute variables at the same address, the assembler and / or linker issues an error. The compiler does not check this.
- When you declare the same absolute variable within two modules, this produces conflicts during link time (except when one of the modules declares the variable ’extern’).
- If you use 0 as an address, this value is ignored. A zero value indicates a relocatable section.

2.4 Using Assembly in the C Source: __asm()

With the `__asm` keyword you can use assembly instructions in the C source. Be aware that C modules that contain assembly are not portable and harder to compile in other environments.

Furthermore, assembly blocks are not interpreted by the compiler: they are regarded as a black box. So, it is your responsibility to make sure that the assembly block is syntactically correct.

General syntax of the `__asm` *keyword*

```
__asm( ”instruction_template”
       [ : output_param_list
       [ : input_param_list
       [ : register_save_list]]] );
```

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair.
output_param_list	[[”=[**&**]*constraint_char*”**(**C_expression**)**],...]
input_param_list	[[”*constraint_char*”**(**C_expression**)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.

C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[”*register_name*”],...]
register_name:q	Name of the register you want to reserve.

Typical example: adding two C variables using assembly

```
char a, b;
int result;

void main(void)
{
    a = 3;
    b = 4;
    __asm( ”ADD %0,%1,%2” : ”=r”(result): ”r”(a), ”r”(b) );
}
```

`%0` corresponds with the first C variable, `%1` with the second and so on.

Generated assembly code:

```
main: .type func
    movui r2,0x3
    stb   r2,@gprel(a)(gp)
    movui r3,0x4
    stb   r3,@gprel(b)(gp)
    ADD   r2,r2,r3
    stw   r2,@gprel(result)(gp)
    ret
```

Specifying registers for C variables

With a *constraint character* you specify the register type for a parameter. In the example above, the `r` is used to force the use of registers (Rn) for the parameters `a` and `b`.

You can reserve the registers that are already used in the assembly instructions, either in the parameter lists or in the reserved register list (*register_save_list*, also called ”clobber list”). The compiler takes account of these lists, so no unnecessary register saving and restoring instructions are placed around the inline assembly instructions.

Constraint character	**Type**	**Operand**	**Remark**
R	general purpose register (64 bits)	r0 .. r31	Based on the specified register, a register pair is formed (64–bit). For example r0:r1.
r	general purpose register (32 bits)	r0 .. r31	

Constraint character	Type	Operand	Remark
i	immediate value	#value	
l	label	*label*	
m	memory label	*variable*	stack or memory operand, a fixed address
number	other operand	same as %*number*	used when in– and output operands must be the same Use %*number*.0 and %*number*.1 to indicate the first and second half of a register pair when used in combination with R.

Table 2–3: Available input/output operand constraints for the Nios II

Loops and conditional jumps

The compiler does not detect loops that are coded with multiple `__asm` statements or (conditional) jumps across `__asm` statements and will generate incorrect code for the registers involved.

If you want to create a loop with `__asm`, the whole loop must be contained in a single `__asm` statement. The same counts for (conditional) jumps. As a rule of thumb, all references to a label in an `__asm` statement must be contained in the same statement.

Example 1: no input or output

A simple example without input or output parameters. You can use any instruction or label. Note that you can use standard C escape sequences.

```
__asm( "nop\n\t"
       "nop" );
```

Generated code:

```
    nop
    nop
```

Example 2: using output parameters

Assign the result of inline assembly to a variable. A register is chosen for the parameter because of the constraint `r`; the compiler decides which register is best to use. The `%0` in the instruction template is replaced with the name of this register. Finally, the compiler generates code to assign the result to the output variable.

```
char var1;

void main(void)
{
    __asm( "movui %0,0xff" : "=r"(var1));
}
```

Generated assembly code:

```
movui  r2,0xff
stb  r2,@gprel(_var1)(gp)
```

Example 3: using input and output parameters

Add two C variables and assign the result to a third C variable. Registers are used for the input parameters (constraint `r`, `%1` for `a` and `%2` for `b` in the instruction template) and for the output parameter (constraint `r`, `%0` for `result` in the instruction template). The compiler generates code to move the input expressions into the input registers and to assign the result to the output variable.

```
char a, b;
int result;

void main(void)
{
    a = 3;
    b = 4;
    __asm( ”ADD %0,%1,%2” : ”=r”(result): ”r”(a), ”r”(b) );
}
```

Generated assembly code:

```
main: .type func
    movui r2,0x3
    stb   r2,@gprel(a)(gp)
    movui r3,0x4
    stb   r3,@gprel(b)(gp)
    ADD   r2,r2,r3
    stw   r2,@gprel(result)(gp)
    ret
```

Example 4: reserve registers

Sometimes an instruction knocks out certain specific registers. The most common example of this is a function call, where the called function is allowed to do whatever it likes with some registers. If this is the case, you can list specific registers that get clobbered by an operation after the inputs.

Same as *Example 3*, but now register `R3` is a reserved register. You can do this by adding a reserved register list (`: ″R3″`). As you can see in the generated assembly code, register `R3` is not used.

```
char a, b;
int result;

void main(void)
{
    a = 3;
    b = 4;
    __asm( ″ADD %0,%1,%2″ : ″=r″(result): ″r″(a), ″r″(b) : ″R3″ );
}
```

Generated assembly code:

```
main: .type func
    movui r2,0x3
    stb   r2,@gprel(a)(gp)
    movui r4,0x4
    stb   r4,@gprel(b)(gp)
    ADD   r2,r2,r4
    stw   r2,@gprel(result)(gp)
    ret
```

2.5 Pragmas to Control the Compiler

Pragmas are keywords in the C source that control the behavior of the compiler. Pragmas overrule compiler options.

The syntax is:

```
#pragma pragma-spec [ON | OFF | DEFAULT]
```

or:

```
_Pragma( ″pragma-spec [ON | OFF | DEFAULT]″ )
```

For example, you can set a compiler option to specify which optimizations the compiler should perform. With the `#pragma optimize` *`flags`* you can set an optimization level for a specific part of the C source. This overrules the general optimization level that is set in the C compiler Optimization page in the Project Options dialog (command line option **-O**).

The compiler recognizes the following pragmas, other pragmas are ignored.