Writing Efficient C Code for the LatticeMico8 Microcontroller

The LatticeMico8 soft processor is an easy-to-use, free embedded microcontroller optimized for implementation in Lattice FPGAs and PLDs. The core consumes minimal device resources—less than 200 Look Up Tables (LUTs) in the smallest configuration—while offering a broad feature set. It is ideal for implementing control processing, control communications, state machine replacement, and simple user interfaces. The small size of the microcontroller core forces its subsystems to have limited resources. The LatticeMico8 architecture only supports relative branches (branches from the current position) with offsets of +/-2K. The compiler can convert C and/or ASM code into an application executable as long as it can guarantee that all branches are within the +/-2K range.

Writing good, efficient C code that is readable and maintainable is an extremely important goal for any C programmer. This user guide introduces the reader to techniques for reducing code and processing time in functions. These guidelines can be used in most cases to optimize the code to avoid inconsistent or incorrect results.

Considerations for Using the LatticeMico8 GCC Port

LatticeMico8 GCC port has certain limitations related to the microcontroller's architecture. Developers should consider the following:

- No floating point library, so that it can fit in to the limited resources of LatticeMico8.
- All branches and procedure calls are relative to the current position and the maximum offset is +/-2K, which means that a branch can go to 2K instructions backwards or forwards.

  This limitation directly affects the size of the largest application that can be compiled. If the compiler, during the link phase, determines that a branch

or procedure call wants to jump to a location that is larger than +/-2K, it will not create an application executable; it will exit with link errors.

- Limited support for nested functions.

  The procedure call stack of LatticeMico8 is finite and can be programmed to be 8, 16, or 32. This limits the number of nested functions that the C application can contain.

- Function pointers are not supported. These are declarations of the type: `void (*func)()`
- The LatticeMico8 GCC port does not contain a C Standard Library implementation such as Newlib, libc, or glibc. This means that the C developer has no access to built-in implementations of such functions as `printf()`, `memcpy()`, `srccpy()`, etc.

Guidelines for Optimization

The GNU C Compiler (GCC) is quite good at generating efficient object code. It will normally optimize away any C code that it determines will not affect the output. If the developer does not want a particular piece of C code to be optimized away, he must write the C code in a way that prevents GCC from removing it from the final application.

A good example is a typical delay loop written in C. The typical delay loop consists of a loop that decrements a delay element, such as counter, on every loop iteration and exits the loop only when the delay element reaches a predetermined level. Normally, delay loops are used in C code to introduce a delay. GCC will normally optimize away this delay loop, because it does not see it impacting the output of the application other than adding a time delay. The way to ensure that this delay loop is not optimized away by GCC is to either declare the delay element in C as “volatile,” meaning that the delay element's value is not just controlled by the C code but is also affected by external elements; or write the delay loop, or part of it, in assembly.

Regardless, LatticeMico8 GCC provides a number of optimization techniques that impact the size and/or performance of the final application executable. Sometimes these optimization techniques go too far by aggressively optimizing away C code and affecting the intended functionality. It is recommended that the developer always check the compiled object code to ensure that the GCC optimization used has not adversely impacted the functionality.

- Use the compiler optimization (`-O1`, `-O2`, `-O3`, `-Os`). Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. However care must be taken to ensure that the compiler is not allowed to change the functionality of the code during optimization.

  Usage Example:

  ```
  lm8-elf-gcc -Os -o hello_world.elf hello_world.c
  ```

- -O1,The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- -O2, Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed trade-off. As compared to -O, this option increases both compilation time and the performance of the generated code.
- -O3, Turns on all optimizations specified by -O2 and also turns on the `-finline-functions`, `-funswitch-loops`, `-fpredictive-commoning`, `-fgcse-after-reload`, `-ftree-vectorize`, and `-fipa-cp-clone` options
- -O0, Reduce compilation time and make debugging produce the expected results. This is the default.
- -Os, Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

- Use the “volatile” key word to preserve variables that should not be optimized away by the compiler.

  Usage Example:

  ```
  volatile unsigned char i;
  ```

- Use the “noinline” attribute to prevent the compiler from inlining the callee function (merging the callee function into the caller function).

  Usage Example:

  ```
  void foo () __attribute__ ((noinline));
  ```

- Use the smallest variable type that meets the maximum size requirements for your code. If the maximum value for a variable will be 255, use an “unsigned char” instead of an “unsigned int.” An “unsigned int” takes up 2 bytes of storage, whereas an “unsigned char” takes up only 1 byte of storage. Also, working with an int uses more instructions than working with a char.

Other Recommendations

Use Pointers Instead of Subscripts Pointers should be used instead of subscripts. This enables the compiler to perform the calculations for accessing the elements in the array of elements or structures at compile time instead of at run time.

For example, the following could cause several multiplies to occur each time any element is accessed.

```
aStruct[Subscript].elementName1
```

This results in a calculation for each access of every element.

```
address = &aStruct +
(Subscript*sizeof(aStruct))+offset(elementName1)
```

Using pointers as follows,

```
structPntr=&aStruct[Subscript];
```

will result in “multiply” at the structure level and not at the element level and access would become

```
structPntr->elementName1 …
```

This allows the compiler to do all of the calculations at compile time for the offsets in each event, resulting in only one multiply for each subscript, and not for each element.

Try to Use Small Operands LatticeMico8 is an 8-bit micro processor. Using 16-bit or 32- bit numbers require additional instructions. Therefore, use larger operands only when necessary. No float variables are available. When dealing with math, it is extremely important to be aware of the possibilities of wrap-around conditions.

Watch Out for ANSI Size Promotions By ANSI definition, all values used in a calculation will be promoted to the size of the largest value. Therefore, even if only one of the variables in the calculation is defined as UNIT16, all UINT8 variables will be promoted to UINT16. Note that when constants are defined, their default size will be “int”.

Re-use Temporary Variables The scratchpad memory in the LatticeMico8 is 32 bytes, so it should be a regular practice to reuse any available temporary variables. This will also make for better register use.

Use Table Lookups for Math Table lookups should be used where possible for complex math, such as CRC calculations. This can also save processing time.

Use “shift” Use “shift” instead of “divide” or “multiply” wherever possible. It is typically much quicker, when dealing with power of two division, to shift left/ right by the appropriate number of bits rather than calling a “divide” routine.

Use “if” instead of “modulo” Statements Substituting an “if” statement instead of “modulo” statement results in a more efficient, faster code. For example, the following

```
if (++count >= moduloMax) count = 0;
```

is much faster than

```
count = ++count % moduloMax;
```

Use Code Space Effectively Choice of memory model or code space (large -32 bits, medium -16 bits, or small -8 bits) impacts code size, since registers are used as page pointers (for medium and large memory models); and fewer registers are available to the compiler for passing parameters between functions.

Function Call Hints It is a good practice to pass as few parameters as possible to a function, because it takes time to load and read each parameter. Whenever there is a need to pass large amounts of data, pointers should be used instead of passing data. This minimizes the amount of data passed on the stack.

Minimize Use of Nested Functions A nested function is a function that is lexically (textually) encapsulated within another function (in other words, functions calling functions calling functions…). Although use of nested functions is supported, it should be minimized because of call stack limitations. In some situations, the fixed size of the call stack might produce run time exceptions.

Use Inline Assembly Use inline assembly routines to implement functions that can be written more compactly in assembly. Inline assembly routines are written as function within a C function. Inline assembly routines are handy, speedy, and useful in programming. Inline assembly is important primarily because of its ability to operate and make its output visible on C variables. Because of this capability, "asm" works as an interface between the assembly instructions and the "C" program that contains it.

You can find more information about GCC's inline assembly feature from http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

Use Built-in Functions There are some built-in functions that can be used:

- Import/Export functions. These functions are the only method of input and output for the processor. If a memory-mapped concept (mapped locations should be declared volatile) is not used, then anything not considered memory in the FPGA will require access using these built-in functions. There are 32 location addresses available:
  - `__builtin_import(port_address)`

    This function will generate the" import" or "importi" instruction to bring a byte of data in from the FPGA or PLD. The result of the "import" instruction is the returned value.
  - `__builtin_export(value, port_address)`

    This function will generate the "export" or "exporti" instruction to put a byte of data in the FPGA or PLD.
  - Interrupt Function

    ```
    void __IRQ (void) __attribute__ ((interrupt))
    ```

    The compiler will generate code to setup an interrupt stack frame suitable for an interrupt handler , The function implementing the interrupt handler should be named __IRQ, so the interrupt handler links correctly. Only one interrupt function is available.

When possible, avoid using an interrupt routine; it is expensive in terms of memory use and performance.

Experiments with the Code

As stated before, the GNU C Compiler is very good at optimization. This section is written from the perspective of a novice using this port of the compiler. It should help the reader understand what to watch out for when working with the GCC. These issues will be somewhat common to all of the GCC ports. Below are some examples related to the optimization issues when working with the compiler for LatticeMico8.

Maximum Optimization Built in

Consider the following code:

```
#define BUF_SIZE  18

static char *
inttostr (int i, char buf[BUF_SIZE])
{
    unsigned int ui = i;
    char *p = buf + BUF_SIZE-1;

    *p = '\0';
    if (i < 0)
        ui = -ui;
    __builtin_export((unsigned char)(ui>>8), PORT10);
    __builtin_export((unsigned char)(ui), PORT10);
    do
    {
        *--p = '0' + ui % 10;
        __builtin_export(*p, PORT01);
    }
    while ((ui /= 10) != 0);
    if (i < 0)
        *--p = '-';
    return p;
}

int
main ()
{
    /* volatile */ char buf[BUF_SIZE], *p;
    char i;

    p = inttostr (-1, buf);
    if (*p != '-')
    {
        __builtin_export(1, PORT30);
    }

    __builtin_export(0, PORT30);
    return 0;
}
```

The objective of the function `inttostr` is to convert a number into a positive integer string. There is only one call to the function and only one possible result. The following assembly is generated upon compilation:

```
0000004e <main>:
  4e:01 20 00 movi  R00,0x00
  51:01 2d 00 movi  R13,0x00
  54:02 e0 50 exportR00,10
  57:01 20 01 movi  R00,0x01
  5a:01 2d 00 movi  R13,0x00
  5d:02 e0 50 exportR00,10
  60:01 20 31 movi  R00,0x31
  63:01 2d 00 movi  R13,0x00
  66:02 e0 08 exportR00,1
  69:01 20 00 movi  R00,0x00
  6c:01 2d 00 movi  R13,0x00
  6f:02 e0 f0 exportR00,30
  72:01 21 00 movi  R01,0x00
  75:03 90 00 ret   75 <main+0x27>
```

Notice that the generated code does not make any function calls, and the "export" instructions are inline in the main function itself. The compiler has properly analyzed the code and has reduced it to a straight line of code. The "do/while" loop is executed only once, and the compiler has properly analyzed it and has generated the proper code. When debugging, it is important to understand that the structure of the code may be different between the C source and the generated assembly.

If the developer does not want the function "inttostr" to be inlined in to function "main," he should use the attribute "noinline":

```
static char *inttostr (int i, char buf[BUF_SIZE])
__attribute__((noinline));
```

The following assembly is generated upon compilation:

```
0000004e <inttostr>:
  4e: 00 a8 f5  addi  R08,0xF5
  51: 00 e9 ff  addic R09,0xFF
  54: 01 0c 40  mov R12,R08
  57: 01 0d 48  mov R13,R09
  5a: 03 80 d8  call 2e2 <__prologue_save_r18>
  5d: 01 12 00  mov R18,R00
  60: 01 0e 08  mov R14,R01
  63: 01 10 00  mov R16,R00
  66: 01 11 08  mov R17,R01
  69: 01 0a 10  mov R10,R02
  6c: 01 0b 18  mov R11,R03
  6f: 00 aa 11  addi  R10,0x11
  72: 00 eb 00  addic R11,0x00
  75: 01 22 00  movi R02,0x00
  78: 01 0d 58  mov R13,R11
  7b: 02 e2 56  sspi R02,R10
  7e: 01 23 00  movi R03,0x00
  81: 03 80 8c  call 225 <__cmphi2>
  84: 03 30 09  bnc 9f <inttostr+0x51>
  87: 01 24 00  movi R04,0x00
  8a: 01 25 00  movi R05,0x00
  8d: 01 00 20  mov R00,R04
```

```
  90: 01 01 28  mov R01,R05
  93: 00 00 80  sub  R00,R16
  96: 00 41 88  subc R01,R17
  99: 01 10 00  mov R16,R00
  9c: 01 11 08  mov R17,R01
  9f: 01 00 80  mov R00,R16
  a2: 01 01 88  mov R01,R17
  a5: 01 22 08  movi R02,0x08
  a8: 03 80 59  call 1b3 <__lshrhi3>
  ab: 01 2d 00  movi R13,0x00
  ae: 02 e0 50  export R00,10
  b1: 01 2d 00  movi R13,0x00
  b4: 02 f0 50  export R16,10
  b7: 00 aa ff  addi  R10,0xFF
  ba: 00 eb ff  addic R11,0xFF
  bd: 01 00 80  mov R00,R16
  c0: 01 01 88  mov R01,R17
  c3: 01 22 0a  movi R02,0x0A
  c6: 01 23 00  movi R03,0x00
  c9: 03 80 70  call 219 <__umodhi3>
  cc: 01 02 00  mov R02,R00
  cf: 00 a2 30  addi  R02,0x30
  d2: 01 0d 58  mov R13,R11
  d5: 02 e2 56  sspi R02,R10
  d8: 01 2d 00  movi R13,0x00
  db: 02 e2 08  export R02,1
  de: 01 00 80  mov R00,R16
  e1: 01 01 88  mov R01,R17
  e4: 01 22 0a  movi R02,0x0A
  e7: 01 23 00  movi R03,0x00
  ea: 03 80 4b  call 1cb <__udivhi3>
  ed: 01 10 00  mov R16,R00
  f0: 01 11 08  mov R17,R01
  f3: 01 22 00  movi R02,0x00
  f6: 01 23 00  movi R03,0x00
  f9: 03 80 7e  call 273 <__ucmphi2>
  fc: 03 1f e9  bnz b7 <inttostr+0x69>
  ff: 01 00 90  mov R00,R18
 102: 01 01 70  mov R01,R14
 105: 03 80 60  call 225 <__cmphi2>
 108: 03 30 06  bnc 11a <inttostr+0xcc>
 10b: 00 aa ff  addi  R10,0xFF
 10e: 00 eb ff  addic R11,0xFF
 111: 01 20 2d  movi R00,0x2D
 114: 01 0d 58  mov R13,R11
 117: 02 e0 56  sspi R00,R10
 11a: 01 00 50  mov R00,R10
 11d: 01 01 58  mov R01,R11
 120: 01 0c 40  mov R12,R08
 123: 01 0d 48  mov R13,R09
 126: 03 80 c2  call 36c <__epilogue_restore_r18>
 129: 00 a8 0b  addi  R08,0x0B
 12c: 00 e9 00  addic R09,0x00
 12f: 03 90 00  ret 12f <inttostr+0xe1>

00000132 <main>:
 132: 00 a8 ea  addi  R08,0xEA
 135: 00 e9 ff  addic R09,0xFF
 138: 01 2c 12  movi R12,0x12
```

```
 13b: 01 2d 00  movi R13,0x00
 13e: 00 8c 40  add  R12,R08
 141: 00 cd 48  addc R13,R09
 144: 03 80 96  call 306 <__prologue_save_r11>
 147: 01 20 ff  movi R00,0xFF
 14a: 01 21 ff  movi R01,0xFF
 14d: 01 02 40  mov R02,R08
 150: 01 03 48  mov R03,R09
 153: 03 8f a9  call 4e <inttostr>
 156: 01 0d 08  mov R13,R01
 159: 02 e0 07  lspi R00,R00
 15c: 02 20 2d  cmpi R00,0x2D
 15f: 03 00 04  bz 16b <main+0x39>
 162: 01 20 01  movi R00,0x01
 165: 01 2d 00  movi R13,0x00
 168: 02 e0 f0  export R00,30
 16b: 01 20 f6  movi R00,0xF6
 16e: 01 21 ff  movi R01,0xFF
 171: 01 02 40  mov R02,R08
 174: 01 03 48  mov R03,R09
 177: 03 8f 9d  call 4e <inttostr>
 17a: 01 0d 08  mov R13,R01
 17d: 02 e0 07  lspi R00,R00
 180: 02 20 2d  cmpi R00,0x2D
 183: 03 00 04  bz 18f <main+0x5d>
 186: 01 20 02  movi R00,0x02
 189: 01 2d 00  movi R13,0x00
 18c: 02 e0 f0  export R00,30
 18f: 01 20 00  movi R00,0x00
 192: 01 2d 00  movi R13,0x00
 195: 02 e0 f0  export R00,30
 198: 01 21 00  movi R01,0x00
 19b: 01 2c 12  movi R12,0x12
 19e: 01 2d 00  movi R13,0x00
 1a1: 00 8c 40  add  R12,R08
 1a4: 00 cd 48  addc R13,R09
 1a7: 03 80 a3  call 390 <__epilogue_restore_r11>
 1aa: 00 a8 16  addi  R08,0x16
 1ad: 00 e9 00  addic R09,0x00
 1b0: 03 90 00  ret 1b0 <main+0x7e>
```

Experiments with “volatile” and Other Optimization Workarounds

This sections explores methods to allow the use of size optimization while retaining desired elements of the code structure. The following code listing is used as the starting point.

```
struct fb {
  unsigned x1:1;
  unsigned x2:2;
  unsigned x3:3;
};


struct adjust_template
{
  int kx_x;
```

```
  int kx_y;
  int kx;
  int kz;
};

static struct adjust_template adjust = {0, 0, 1, 1};

void
adjust_xy (int  *x, int  *y)
{
  *x = adjust.kx_x * *x + adjust.kx_y * *y + adjust.kx;
}

/* c-torture/execute/20000113-1.c */
int
foobar (int x, int y, int z)
{
  struct fb a = {x, y, z};
  struct fb b = {x, y, z};
  struct fb *c = &b;

  c->x3 += (a.x2 - a.x1) * c->x2;
  if (a.x1 != 1 || c->x3 != 5)
    return (3);
  return (0);
}


int main ()
{
  unsigned int zz;
  unsigned int *z = (unsigned int *) &zz;

  int x = 1, y = 1;

  *z = -3;
  *z = *z * *z;
  if (*z != 9)
    /*return (1);*/
    __builtin_export(1, PORT30);

  adjust_xy (&x, &y);
  if (x != 1)
    /*return (2);*/
    __builtin_export(2, PORT30);

  if (foobar (1, 2, 3))
    /*return (3);*/
    __builtin_export(3, PORT30);

  __builtin_export(0, PORT30);
  return (0);
}
```

The code was compiled with six different compiler option configurations:

Test Case 1: The code was plain and “out of the box” and returns with an exit code if a problem is detected and with no calls to

`__builtin_export()`, which typically forces the compiler to generate code, except in this case.

Test Case 2: The code had calls to `__builtin_export()`, which in this case had no effect on the code generated compared to the code in the first test case.

Test Case 3: Two automatic variables in “main” are initialized to 1 upon their definition.

Test Case 4: Same as test case 3, but the structure “zz” is also declared as a “volatile” and had no affect on the code.

Test Case 5: The two functions have the attribute of “noinline” and there are no “volatile” variables.

Test Case 6: The two functions have the attribute of “noinline” and there are “volatile” variables.

In test cases 1 and 2, with optimization set to –O2, –O3 or –Os, the “main” function only sets up to exit. All the code has been evaluated to have the same result as to declare everything OK and exit.

In test cases 3 and 4, with optimization set to –O2, –O3 or –Os, the “main” function performs as intended, but functions are implemented inline.

Test case 5 appears to be the best option. The “noinline” attribute forces the compiler to use the function code it has already generated. This makes the main function a little larger to set up for the calls, instead of having the function code “inline”.

Test case 6 produces an output that is noticeably larger. This is due to the use of the functions and the presence of the volatile variables. Similar to test cases 3 and 4, where the use of “volatile” keyword forced the compiler code to preserve the intended functionality of the code; in this case, the code local to the “main” function was forced to be handled by the compiler.

The code generated by compiling test case 6 is the most efficient option in executing the code as written. However, test case 5 generated the most efficient code in terms of size and performance as intended.

The size of the code generated with optimization levels, –O2, –O3 , or –Os are comparable for this test case. Optimization with –Os creates the most compact size. When the test cases are compiled with optimization set to –O1, the size of the generated code is quite close to the result with optimization set to –O2, –O3 and –Os. This is more applicable when we compare the size of the code generated for the “main” function. Compiling with optimization set to –O0 produces a significantly larger code than optimization set to any other level.

The table below shows the number of instructions in the compiler-generated output for different optimization options.

	Number of Instructions Generated Upon Compilation				
Test Case	**-O0**	**-O1**	**-O2**	**-O3**	**-Os**
Complete Program					
Test Case 1: No calls to __builtin_export	1719	278	196	196	190
Test Case 2: Makes calls to __builtin_export	1693	278	196	196	190
Test Case 3: Volatile int x,y	1693	348	288	288	282
Test Case 4: Volatile int x,y,zz	1693	348	288	288	282
Test Case 5: Functions have noinline attribute	1693	278	276	276	270
Test Case 6: Functions have noinline attribute volatile x,y,zz	1693	348	346	346	340
Main Function Only					
Test Case 1: No calls to __builtin_export	630	75	5	5	5
Test Case 2: Makes calls to __builtin_export	604	75	5	5	5
Test Case 3: Volatile int x,y	604	129	71	71	71
Test Case 4: Volatile int x,y,zz	604	129	71	71	71
Test Case 5: Functions have noinline attribute	604	129	75	75	75
Test Case 6: Functions have noinline attribute volatile x,y,zz	604	129	129	129	129

Notes:
Black calls functions and processes as intended.
Blue does not call functions generated but processes as intended.
Red does nothing but exit.

Technical Support Assistance

Hotline: 1-800-LATTICE (North America)

+1-503-268-8001 (Outside North America)

e-mail: techsupport@latticesemi.com

Internet: www.latticesemi.com