Celoxica

DK2

Handel-C code optimization

Authors: SB

Document number: 1

Customer Support at http://www.celoxica.com/support/

Celoxica in Europe
T: +44 (0) 1235 863 656
E: sales.emea@celoxica.com

Celoxica in Japan
T: +81 (0) 45 331 0218
E: sales.japan@celoxica.com

Celoxica in Asia Pacific
T: +65 6896 4838
E: sales.apac@celoxica.com

Celoxica in the Americas
T: +1 800 570 7004
E: sales.america@celoxica.com

Contents

Conventions

A number of conventions are used in this document. These conventions are detailed below.

Warning Message. These messages warn you that actions may damage your hardware.

Handy Note. These messages draw your attention to crucial pieces of information.

Hexadecimal numbers will appear throughout this document. The convention used is that of prefixing the number with '0x' in common with standard C syntax.

Sections of code or commands that you must type are given in typewriter font like this:

```
void main();
```

Information about a type of object you must specify is given in italics like this:

```
copy SourceFileName DestinationFileName
```

Optional elements are enclosed in square brackets like this:

```
struct [type_Name]
```

Curly brackets around an element show that it is optional but it may be repeated any number of times.

```
string ::= "{character}"
```

1 Tutorial: Handel-C code optimization

The following examples illustrate different methods of optimizing Handel-C code to produce smaller and faster designs. A basic knowledge of Handel-C is assumed, and some knowledge of digital electronics and design techniques will also be helpful.

New users are recommended to work through the following topics in order:

Timing and area efficient code (see page 4)

Loops and control code (see page 9)

2 Timing and area efficient code

A common goal in digital hardware design is to produce circuits which are small and run at a high clock rate. As Handel-C as a higher level language than HDLs, a new user may sometimes be unclear as to how to produce optimal designs. The following sections illustrate a Handel-C coding style which will usually result in area efficient and fast designs. The **TutorialFIR** workspace illustrates the use of some of the following techniques for generating efficient designs in Handel-C.

2.1 Complex statements

When DK compiles Handel-C code for hardware implementation, it generates all the logic required to execute each line of code in a single clock cycle. Therefore, the more complex a line of code is, the longer it will take to execute, and the lower the design clock rate will be. Some of the operators which produce complex hardware are division, multiplication, addition/subtraction and shifting by a variable. The complexity also depends on width of the operands – larger variables need more hardware. The example code below shows a mixture of simple and complex statements:

```
unsigned 16 a, b, c, d;
a = b + c;
a = d + c;
a = d >> 2;
a = ((b << c) + (b * d);
```

The first three lines of code are quite simple, but the fourth is very complex. The clock rate of the whole design will be limited by the fourth line, so it would be better to break it up into several simpler statements:

```
unsigned 16 temp1, temp2;
par
{
    temp1 = b << c;
    temp2 = b * d;
}
a = temp1 + temp2;
```

Although the modified code will take two cycles to execute instead of one, this will be better overall, as the whole design will now be able to run at a higher clock rate. In many designs, it is possible to use *pipelining* to hide this extra cycle. The use of pipelining is explained in the Advanced Optimization tutorial.

A further issue with complex statements is the use of signals. The code below shows the complex statement from the previous example implemented using signals:

```
signal unsigned 16 temp1, temp2;
par
{
    temp1 = b << c;
    temp2 = b * d;
    a = temp1 + temp2;
}
```

Although this code still has the complex statement broken into three parts, because `temp1` and `temp2` are signals, all the operations must still be performed in one clock cycle. This is because signals do not store the values assigned to them, so the results from the first two lines of code are fed straight into the third line in the same cycle.

In summary:

- Avoid division wherever possible, and use shifts or subtracts instead.
- Break complex statements up into several simpler statements
- Remember that signals “stack up” the complexity of the lines of code which write and read them in parallel.

2.2 Arrays and memories

Handel-C supports arrays used in the same way as in C, however, there are implications resulting from the way arrays are implemented in hardware. An array can be seen as a collection of variables which can all be accessed in parallel, with elements either specified explicitly or indexed by a variable. Explicit access to individual array elements is efficient, but indexing through an array can generate significant amounts of hardware, particularly if it is done from more than one point in the code. Arrays are good for implementing shift registers and allowing initialization of the contents of every element in a single cycle, as shown below. This use of arrays is efficient, as every element is specified explicitly.

```
unsigned 8 Array[4];
par  /* Initialise array in single cycle */
{
   Array[0] = 23;
   Array[1] = 25;
   Array[2] = 26;
   Array[3] = 29;
}
while (1)
{
    par  /* Move data through shift register */
    {
       Array[0] = Input;
       Array[1] = Array[0];
       Array[2] = Array[1];
       Array[3] = Array[2];
    }
}
```

RAM and ROM

If random access into an array can not be avoided, it is better to use a RAM instead, simply by adding the `ram` keyword at the start of the array declaration:

```
ram unsigned 8 Memory[4];
```

This will create a more efficient structure in hardware, but will now be limited to a single access per clock cycle. The `rom` keyword can be used if a read-only memory is required, and can be declared as static to allow initialization:

```
static rom unsigned 8 Memory[4] = {23, 25, 26, 29};
```

Block Memory

Many FPGAs have more than one method of implementing memories, optimized for different sizes. You should investigate the memory types available on their target FPGA when choosing how they should implement them, but typically memories larger than a few kbits should use the `{block = 1}` setting to make use of an FPGA's support for larger memory structures, as shown below. The number of block memories available on an FGPA is limited, so you should plan which parts of their design need to make use of them.

```
ram unsigned 32 BigMemory[1024] with {block = 1};
```

Multi-port memory

If memory needs to be accessed more than once per clock cycle, it is possible to use multi-ported memories. The number and type of ports depends on the type of FPGA being targeted, but most support two ports (see DK online help for detail). When a dual-port RAM is built in distributed memory (without {block = 1}), it will take up twice the amount of hardware as a single-ported memory. However, the block memories in many FPGAs are already dual-ported, so if {block = 1} is used, the dual port memory may take up no further hardware.

Dual port memories are useful for the design of FIFO buffers, increasing the read/write bandwidth and interfacing between clock domains (as the two ports can run at different clock rates).

Timing efficient use of memories

As a memory of any sort includes addressing logic, there is always an inherent delay in accessing it for a read or write operation. Because of this, a memory access should be regarded as a complex operation to include in a statement, so the points explained in the section on ***Complex statements*** (see page 4) should be taken into account. In general it is best to use three single registers for the address, input and output of a memory, and to re-use these registers whenever the memory is accessed at different points in the code, as shown below:

```
ram unsigned 16 Memory[64];
unsigned 16 MemoryDataIn, MemoryDataOut, a, b, c;
unsigned 6 MemoryAddress;

par
{   /* set up data and address first */
    MemoryDataIn = a * b;
    MemoryAddress = c * 3;
}

par
{   /* access memory, and set up next address */
    Memory[MemoryAddress] = MemoryDataIn;
    MemoryAddress = c * 5;
}

a = Memory[MemoryAddress];   /* access memory again */
```

2.3 Macro proc vs. function

The main difference between a `macro proc` and a function in Handel-C is the number of copies of hardware that result. Placing a block of frequently used code in a function means that one copy of the code will exist in the hardware, and every time the function is called this single copy of the code will be used. A `macro proc`, however, builds a fresh copy of the code every time it is called. This means that if the code block needs to be called several times in parallel, a single function can not be used, as multiple copies of the code are required. To cater for this situation, arrays of functions can be declared, build a specified number of copies, which can then be called in parallel. However, a further consideration is that multiple sequential calls to a single function will result in complex circuitry at the entry and exit points of the function, leading to the following trade-offs:

- a function may take up less space than a `macro proc`.
- using a `macro proc` will generally result in a higher clock rate

Overall, the best practice is to use `macro procs` by default, as they are easier to design with and result in higher performance. If there is a particularly large (in hardware terms) block of code that is used infrequently but in several places, it may be a candidate for implementation in function. Another alternative, however is to implement a client-server architecture, as described in the Advanced Optimization tutorial, as it offers other advantages.

2.4 Static initialization

For a variable to be initialized in Handel-C it must either be declared as global or static. As assigning a value directly to a variable takes a clock cycle, static initialization can be used to save cycles and increase the performance in a design. Note that variables should always be initialized before use in any case, as their values can not be assumed to be zero at startup.

3 Loops and control code

The following sections illustrate how to code efficient loops and other control structures in Handel-C, optimizing for both area and timing efficiency.

3.1 Cycle efficiency of loops

As Handel-C is very close to C, it is common to port code directly from C to Handel-C, modifying it to add parallelism. There are several areas where common coding styles in C will not produce the most efficient hardware design in Handel-C, and in the area of control statements it is the `for()` loop which is not ideal.

`for()` loops are supported by Handel-C, but because of the control portion of the loop typically contains an assignment, it must use a clock cycle. This is because the Handel-C timing model requires every assignment to take a single clock cycle. The result is that `for()` loops have a single clock cycle overhead, so the example below takes 20 cycles to execute, rather than 10:

```
for (i = 0; i < 10; i++)
{
    a[i] = 0;
}
```

To improve the performance, a `while()` loop should be used instead, as shown below. In this example the loop will now take 11 clock cycles instead of 20. In practice it may be possible to initialize `i` to zero in parallel with an earlier operation, effectively reducing the number of cycles taken from 11 to 10.

```
i = 0;
while (i < 10)
{
    par
    {
        a[i] = 0;
        i++;
    }
}
```

3.2 Timing efficiency of loops and control code

The section on ***Complex statements*** (see page 4) explained that they can result in a design having a low clock rate. When control code, such as a `while()` loop or an `if()...else` statement, is used, the logic implementing the control must be execute in the same clock cycle as the first line of code which actually operates on data. For example, the code shown below would have a low clock rate, even though the operation a++ is simple, because the condition for the while loop is so complicated.

```
unsigned 8 a;
unsigned 32 b, c, d;

while (((b * c) + d) > (d - b))
{
    a++;
}
```

The same principle applies to `for()` loops, `if()…else` statements, conditional assignments, and any other control code which might be used. To increase the performance in loops, there are a couple of simple techniques to use:

- Test for equality (==, !=) where possible, rather than using comparisons (<, <=), as this produces smaller and faster hardware.
- Set a single bit variable within the loop, and test it in the loop control.

The code below illustrates the use of the second of these two techniques, using the complex example above:

```
static unsigned 1 Test = 1;
unsigned 8 a;
unsigned 32 b, c, d;

while (Test == 1)
{
    par
    {
        a++;
        Test = ((b * c) + d) > (d - b);
    }
}
```

3.3 Avoiding combinatorial loops and control code

A combinatorial loop is a series of logic components connected in a loop with no latches or delay elements inserted. Combinatorial loops are typically generated from `if()` statements and `while()` loops in Handel-C, as shown in the example code below:

```
while (Wait == 1)
{
   a = 0;
}
if (Wait == 1)
{
   a = 0;
}
```

The `while()` loop shown can generate a combinatorial loop as it may take zero cycles to execute. Similarly, the `if()` statement could take zero or one cycle to execute, depending on the value of `Wait`. Code which causes combinatorial loops is bad for two reasons:

- The number of cycles to execute the code at runtime is unclear, making the design difficult.
- As the code could take zero cycles to execute, the Place and Route tools will assume the worst case when calculating the maximum clock rate, which will be the time taken to execute the `if()` or `while()` condition *added to* the time taken to execute the following line of code.

To avoid these problems for `while()` loops, ensure that they always take at least 1 cycle to execute, by carefully selecting the condition or by using `do...while()` instead. For `if()` statements, always include an `else` block which takes at least 1 cycle to execute, as shown below:

```
if (Wait == 1)
{
   a = 0;
}
else
{
   delay;
}
```

Note that this also applies to `switch()` statements, where a `default` case should always be included, even if it only contains a single `delay` statement.

3.4 Nested control

Using nested `if()` statements, or long chains of `if()...else()` blocks can result in a design having a low clock rate. This is because the worst case is that all the nested conditions must be executed in a single cycle, so the delay can become significant. If possible, Handel-C code should be written to avoid nesting control statements more than a few layers deep. If this is not avoidable, there are two options for reducing the impact:

- Ensure that the first line of code to be execute after nested control statements is relatively simple, so as not to adversely affect the clock rate.
- Break up the nesting of control statements by executing a line of code in the middle:

```
if (a == 1)
    if (b == 1)
    {
        x = 0; // execute code here to break up nested control
        if (c == 1)
            if (d == 1)
                e = 0;
            else
                delay;
        else
            delay;
    }
    else
        delay;
else
   delay;
```

4 Index