Celoxica

DK2

Handel-C advanced 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 advanced optimization

The following examples illustrate advanced methods of optimizing Handel-C code to produce smaller and faster designs. This builds on the content of the Code Optimization Tutorial, which should be studied first. Two main techniques are covered; pipelining and client-server architectures. A thorough 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:

2 Pipelining

A simple technique for increasing the clock rate of a Handel-C design is to split complex operations over several cycles. However, this results in more cycles being required to perform the operation. Pipelining splits operations up in the same way, but achieves the same data throughput as the original circuit.

The following code illustrates a complex expression that is might result in a low clock rate for the design it is included in:

```
while (1)
{
    a = (b + c) * (d + e);
}
```

This can be split into two operations to calculate the sums, which can be executed in parallel, and then the following cycle the multiplication can be performed. This will result in each line having a shallower logic depth, allowing a higher clock rate.

```
while (1)
{
    par
    {
        sum1 = (b + c);
        sum2 = (d + e);
    }
    a = sum1 + sum2;
}
```

However, the original operation took only one cycle, and the modified version takes 2 cycles. If all three lines of code are executed in parallel, a two stage pipeline will be formed, as shown below:

```
while (1)
{
    par
    {
        /* pipeline stage 1 */
        sum1 = (b + c);
        sum2 = (d + e);

        /* pipeline stage 2 */
        a = sum1 + sum2;
    }
}
```

The behaviour and timing of the code is as follows:

- After the first clock cycle:
  - new values for the additions are calculated and stored in sum1 and sum2.
  - the value in a will be undefined, as it depends on sum1 and sum2 for its inputs, and they were undefined at the start of the cycle.
- After the second clock cycle:
  - another set of new values for the additions are calculated and stored in sum1 and sum2.
  - The multiplication has been performed, using the values of sum1 and sum2 generated in the previous clock cycle, and the result is stored in a.

The behaviour in the second cycle is then repeated in all following cycles, providing that the data in b, c, d and e is valid on every cycle.

The result is that the block of code will be capable of running at a higher clock rate when implemented in hardware, at the expense of results being delayed by one cycle. As long as new inputs are presented every cycle, there will be a new result every cycle, after the initial one cycle delay. Hence the pipeline has a latency of one cycle and a throughput of one result per cycle.

3 Pipelines and replicators

Parallel and sequential replicators can be used in Handel-C to build complex program structures quickly and allow them to be parameterized. Replicators are used in the same way as `for()` loops, except that during compilation they are expanded so that all iterations are implemented individually, and can be executed sequentially or in parallel. So, the following code:

```
par (i=0; i<3; i++)
{
    a[i] = b[i];
}
```

expands to:

```
par
{
    a[0] = b[0];
    a[1] = b[1];
    a[2] = b[2];
}
```

If a seq had been used instead of the par, the expanded code would have been executed sequentially instead of in parallel.

Replicators are useful for implementing algorithms which access iterate over an array or bitwise across several variables. A good example is a pipelined multiplier where the number of pipeline stages is equal to the width. The input data and a sum are passed through each stage, the inputs being shifted and added to the sum as required. The code below implements a pipelined multiplier with a user-defined data width.

```
#define WIDTH 8
unsigned WIDTH sum[WIDTH];
unsigned WIDTH a[WIDTH];
unsigned WIDTH b[WIDTH];
while(1)
{
    par
    {
        sum[0] = ((a[0][0] == 0) ? 0 : b[0]);

        par (i=1; i<=(WIDTH-1); i++)
        {
            sum[i] = sum[i - 1] + ((a[i][0] == 0) ? 0 : b[i]);

            a[i] = a[i - 1] >> 1;
            b[i] = b[i - 1] << 1;
        }
    }
}
```

The first line of code inside the `while(1)` loop sets the value of `sum[0]`, then the replicated `par` moves the shifted inputs through the `a[]` and `b[]` arrays, and the results through the `sum[]` array. The final result is available in the last element of the `sum[]` array, after a latency equal to the width of the input data. The workspace **TutorialMult** contains a copy of this code set up for simulation, using `chanin` to get input data from two files, and `chanout` to write data to another file.

4 Client-server architecture

When an operation or device driver is particularly complex or requires significant resources when implemented in hardware, it may not be efficient to use it repeatedly in different locations in a Handel-C program. A client-server architecture puts all the complexity into a "server" process which runs indefinitely, and provides a "client" API through which you can gain access to the resources of the server. The end result is similar to using a *function* in Handel-C, but allows more control, as you can specify an API and devise methods of handling multiple simultaneous requests for access to the resource.

The following examples illustrate how to implement client-server architectures, first using a simple divide operation, then a more complex Flash Memory driver:

Client-Server divide example (see page 8)

Flash Memory client-server example (see page 10)

4.1 Client-server divide example

A simple example of a client-server architecture can be based on a divider, which inherently requires a large amount of hardware. If the divider is used several times throughout a program, but never more than once simultaneously, then it can be implemented in a server process as follows.

Create a data structure which will be used to access the server:

```
struct _DivideStruct
{
    unsigned 16 InputA;
    unsigned 16 InputB;
    unsigned 16 Result;
};
typedef struct _DivideStruct DivideStruct;
```

Now create a server process:

```
macro proc DivideServer(DividePtr)
{
    /* perform divide operations forever */
    while(1)
    {
        DividePtr->Result = DividePtr->InputA / DividePtr->InputB;
    }
}
```

and a client API macro:

```
macro proc Divide(DividePtr, a, b, ResultPtr)
{
    /* send data to the divide server */
    par
    {
        DividePtr->InputA = a;
        DividePtr->InputB = b;
    }

    /* wait one cycle for divide to be performed */
    delay;

    /* send back the result of the divide */
    *ResultPtr = DividePtr->Result;
}
```

Note that because the server will take a cycle to calculate the result and store it in the data structure, a delay is inserted in the client macro to allow for this. Other methods include using channels in the data structure to transfer data in and out of the server, or using a shift register filled with "valid" bits in the server, so the client macro can tell when the result is valid.

The server does not exit, so should be run in parallel with the main program, and the client macro can be called as many times as required (sequentially) without imposing large hardware overheads. If more than one divide needed to be carried out in parallel it is possible to run two servers, and explicitly call them using the client macro in parallel. For this to operate correctly, two differently named data structures must be used and passed to the respective server and client macros.

The **TutorialClientServer** workspace contains two projects, the first using normal divide operators, and the second using the client-server architecture described above. Compile them both for **EDIF** output, with the Technology Mapper and Logic Estimator enabled, and compare the output. The client-server version takes significantly less hardware.

4.2 Flash memory client-server example

The operation of flash memory is more complicated than asynchronous RAM. It is organized into blocks of data. An entire block must be erased before any locations within it can be programmed.

This example is based on the Intel flash memory part 28F640J3A, which has a capacity of 64 Mbits, organized as 64 blocks. You can obtain the data sheet for this part from http://developer.intel.com.

The 28F640J3A has an internal state machine that you must program to perform device operations. The device has the following connections:

- 23 bit address bus (input)
- 16 bit data bus (bi-directional)
- chip enable pins (input)
- reset pin (input)
- output enable pin (input)
- write enable pin (input)
- status pin (output)
- byte enable pin (input)

The device can operate in 16 bit data or 8 bit data mode. You select the mode using the byte enable input. In 16 bit mode the Least Significant Bit (LSB) of the address bus is discarded. This example uses the device in 16 bit mode so the byte enable is deactivated by wiring high (it is active low) and only the most 22 most significant bits of the address bus are used. Each block inside the flash device contains 128 Kb. In 16 bit mode the blocks are 64 Kwords long. Of the total 23 address bits, the block address is given by the most significant 6 bits and the address within a block is given by the least significant 17 bits.

The API requires functions for reading, writing and erasing data from the Flash device. Although the device also features operations for querying device identity and locking blocks of data (to prevent them from being erased) these are not essential for the operation of the device.

This example implements the interface translation code that converts API functions into device operations using a server process that runs in parallel with an application. The API functions act as clients to the server. The server is implemented using a non-terminating loop inside a macro procedure. The API functions and the server use shared variables and a channel to communicate. These are collected together inside a structure and passed as a parameter to the API functions and the server.

Here are the prototypes for the read, write and erase API functions:

```
/*
 * Read datum from specified Address in flash into (*DataPtr)
 * Parameters:  FlashPtr : input of type (Flash)*
 *              Address  : input of type (unsigned 22)
 *              DataPtr  : input of type (unsigned 16)*
 */
extern macro proc FlashReadWord (FlashPtr, Address, DataPtr);


/*
 * Write a datum from Data into Flash at specified Address
 * Parameters:  FlashPtr : input of type (Flash)*
 *              Address  : input of type (unsigned 22)
 *              Data     : input of type (unsigned 16)
 */
extern macro proc FlashWriteWord (FlashPtr, Address, Data);


/*
 * Erase data from the block in the Flash referenced by BlockNumber
 * Parameters:  FlashPtr    : input of type (Flash)*
 *              BlockNumber : input of type (unsigned 6)
 */
extern macro proc FlashEraseBlock (FlashPtr, BlockNumber);
```

The macro procedure containing the server has the following prototype:

```
/*
 * Run the Flash device driver server
 * Parameters:  FlashPtr  : input of type (Flash)*
 *              ClockRate : clock rate in Hz
 */
extern macro proc FlashRun (FlashPtr, ClockRate);
```

The structure that contains variables shared between the server and API functions also contains expressions for the interfaces to the device. The advantage of this is that the same API functions and server code can be used to control multiple 28F640J3A flash memory devices at the same time. A different copy of the structure is created for each device and then the server is run multiple times in parallel with the application, once for each device. The structure has the following definition:

```
struct _Flash
{
        interface bus_ts (unsigned DataIn) *DataBus
                         (unsigned DataOut, unsigned OE);
        interface bus_clock_in (unsigned Input) *StatusBus ();
        unsigned  1 CEn;
        unsigned  1 WEn;
        unsigned  1 OEn;
        unsigned  1 DataOE;
        unsigned 22 Addr;
        unsigned 16 Data;
        unsigned  1 ByteEnable;
        unsigned 22 APIAddress;
        unsigned 16 APIData;
        unsigned  6 APIBlockNumber;
        chan unsigned 3 APICommand;
};
typedef struct _Flash Flash;
```

The declaration and the definition of the structure type are placed in separate files to indicate that the structure should be treated as opaque. Putting expressions for the interfaces inside the Flash structure separates the interface definitions that connect to the flash device from the implementation of the device driver. The interfaces will now be defined in the context of an application or PSL that uses the device driver to control a specific 28F640J3A flash memory device.

The members of the Flash structure each have the following purpose:

DataBus	Connects the server to the input expression of the flash data bus interface
StatusBus	Connects the server to the input expression of the flash status bus interface
CEn	Connects the server to the output expression on the flash chip enable pin
WEn	Connects the server to the output expression on the flash write enable pin
OEn	Connects the server to the output expression on the flash output enable pin
DataOE	Connects the server to the output enable expression on the flash data bus (enables output from the FPGA/PLD to the device)

Addr	Connects the server to the output expression on the flash address bus
Data	Connects the server to the output expression on the flash data bus
ByteEnable	Connects the server to the output expression on the flash byte enable pin
APIAddress	Shared between the API clients and server, used to communicate the address for a read or write operation
APIData	Shared between the API clients and server, used to communicate the Data for a read or write operation
APIBlockNumber	Shared between the API clients and server, used to communicate the block number for a block erase operation
APICommand	Used by the API clients to send commands to the server

A single instance of the Flash structure can be declared, initialized and connected to pins by a call to the FlashInit() macro, which is declared as follows:

```
extern macro proc FlashInit (FlashPtrPtr,
                             FlashAddrPins,
                             FlashDataPins,
                             FlashChipEnablePins,
                             FlashOutputEnablePin,
                             FlashWriteEnablePin,
                             FlashStatusPin,
                             FlashByteEnablePin,
                             FlashEraseEnablePin);
```

The APICommand channel can take any of three values equivalent to the different operations, these values are defined using the following macro expressions.

```
static macro expr FlashAPICommandReadWord   = 1<<0;
static macro expr FlashAPICommandWriteWord  = 1<<1;
static macro expr FlashAPICommandEraseBlock = 1<<2;
```

The commands are decoded inside the server with a switch-case construct. Using the series $x=2^n$ to generate values for each branch assists the DK compiler in optimizing away branches that are never used. This is desirable since a programmer may not use all of the available operations in an application.

The server process has the following skeleton structure:

```
macro proc FlashRun (FlashPtr, ClockRate)
{
   // Initialization sequence
   unsigned 3 Command;

   do
   {
      FlashPtr->APICommand ? Command;
      switch (Command)
      {
         case FlashAPICmdRead:
            // Read sequence goes here
            FlashPtr->APICommand ! 0;
            break;
         case FlashAPICmdWrite:
            // Write sequence goes here
            break;
         case FlashAPICmdErase:
            // Erase sequence goes here
            break;
         default:
            delay;
            break;
      }
   }
   while (1);
}
```

The full implementation of the server can be found in the **TutorialFlashRAM** workspace.

The API functions have the following implementation:

```
macro proc FlashReadWord (FlashPtr, Address, DataPtr)
{
        static unsigned 3 Dummy;

        par
        {
                FlashPtr->APICommand ! FlashAPICommandReadWord;
                FlashPtr->APIAddress = Address;
        }
        FlashPtr->APICommand ? Dummy;
        *DataPtr = FlashPtr->APIData;
}
macro proc FlashWriteWord (FlashPtr, Address, Data)
{
        par
        {
                FlashPtr->APICommand ! FlashAPICommandWriteWord;
                FlashPtr->APIAddress = Address;
                FlashPtr->APIData = Data;
        }
}

macro proc FlashEraseBlock (FlashPtr, BlockNumber)
{
        par
        {
                FlashPtr->APICommand ! FlashAPICommandEraseBlock;
                FlashPtr->APIBlockNumber = BlockNumber;
        }
}
```

When the flash device driver is used, the application programmer must:

- Declare a variable of type `(Flash *)`
- Call `FlashInit()` with appropriate parameters to build interfaces to the correct pins and to create and initialize a `Flash` structure.
- Run the `FlashRun()` server process in parallel with the application

Alternatively, the calls can be put inside a PSL that is configured for a specific platform.

5 Index