>: FREQUENTLY ASKED QUESTIONS//

>: Programming in Handel-C:

CONTENTS

1 How should I write Handel-C for efficient synthesis?

The simpler and clearer your code, the better the chance there is for optimization by the compiler. Long and complicated statements can often produce deep and slow logic. If nothing else, code written in this way will be harder to understand.

For a fast implementation remember to reduce logic depth and to pipeline the algorithm. The "timing and efficiency information" section of the Language reference manual provides advice on these topics. There are a number of further tips in the online help and this FAQ. Read on!

1.1 Can you explain pipelining code?

A classic way to increase clock rates in hardware is to "pipeline". A pipelined circuit takes more than one clock cycle to calculate any result but can produce one result every clock cycle. In order to pipeline, you should split up a calculation into a number of shallow logic stages. Note that a pipeline is a little more involved than simply performing partial calculations on successive clock cycles. Using pipelining, each of these logic stages is executed in parallel. The input data for each stage is taken from the partial result of the preceding stage, which was calculated on the preceding cycle. Thus it takes a number of cycles to "load" the pipeline before data is ready for calculation at each stage. The trade-off is increased latency for a higher throughput, in other words you have to wait more clock cycles for the first result but after the pipeline is full the time (in ns) between successive results is shorter (this is because the logic depth has been reduced and a faster clock can be used). The online help for the compiler describes a pipelined multiplier in Handel-C.

1.2 What's the most efficient way to implement loops?

The for() statement adds one cycle to the loop for incrementing the counter. You can use while() instead of for() statements and increment the counter in parallel to the body of the code. Not only will this clarify the number of clock cycles taken by the loop but will reduce the number of cycles per loop by one.

1.3 When should I use switch instead of if..else?

Switch and break have the potential to produce shallower code than if....else. A parallel comparator is generated in hardware for switch whereas if..else generates a priority tree for each choice. Remember that the case expressions in the switch must be compile time constants. To ensure correct timing, ensure that there is a default case with specified delay in if...else and switch constructs.

1.4 When should I use conditional expressions as opposed to if...else?

Conditional expressions have the potential to produce less logic. Be careful when using conditional expressions with RAMs. For example the code x = y>z ? RamA[1] : RamA[2]; does not execute correctly because of the multiple use of the RAM in the expression. The Handel-C simulator will give the wrong answer if it encounters it. The solution is to re-write the code using if...else. In this particular case it is possible to rewrite the code as follows: x = RamA[y>z ? 1 : 2]. Here, there is only a single access to the RAM so the problem does not occur.

1.5 How do I use RAM's effectively?

- You should remember to make full use of the whole width of RAM locations if possible, especially if dealing with off-chip and block RAMs. This makes more efficient use of the storage available and as much data is accessed in one access cycle as possible.

- For on-chip RAM try and reduce the number of address locations - remember that the address locations can be any width but by reducing the number of addresses lines reduces the about of logic built.
- Access RAMs from the minimum number of locations in the source code - this avoids duplication of address logic and associated routing around the chip.
- Arrange RAMs in such a way that the number of addresses is a power of 2 - this is the most efficient way of using address logic.
- Do not use large arrays of registers unless you need random access to many elements in parallel. The storage density is much lower than for distributed ram.
- Remember to fully utilize on chip block RAMs if they are available on the target device. This can significantly reduce the usage of device CLBs.

1.6 When should I use macro procs instead of functions?

Macro procs have become largely obsolete by inline functions but not by ordinary (reused) functions. Functions provide for more portable code and are the preferred method of writing in Handel-C. Macro procs are included in the language primarily for compatibility with earlier versions of the language. Macro procs may provide an area saving for small blocks of code and do allow for more extensive parameterization of the code. For example, macro procs are very useful for functions that will work on different width arguments.

Macro procs share datapath if possible, but duplicate control path for each call. Function calls share control path and datapath. In some cases the overhead of a function call may be larger than the duplicated control logic; for example when the subroutine has a small amount of control logic and is only used a few times.

1.7 How do I make the best use of multiple clock domains?

Use multiple clock domains only when unavoidable. Remember that there are a limited number of clock propagation nets in each device, and communication between domains usually takes a long time (full handshaking requires 4 clock cycles in the slowest clock domain).

1.8 Multiplication can produce a lot of hardware. Do you have any tips?

Remember that shift left < < and shift right operators > > can be used for multiplying powers of two. The optimizer will automatically implement these operations for multiplication by a constant.

You can always pipeline the algorithm. See the online help for more information.

2 How does the compiler optimize my code?

Since we are compiling from software to hardware, we are able to benefit from both traditional software compiler techniques and logic optimization techniques. In this section we describe the general framework for compilation from the abstract syntax tree to target hardware and provide detail for some of the optimization techniques.

The input code is first transformed to an abstract syntax tree. During this transformation the compiler performs width inferencing, constant folding and other syntactic expansions (such as macro expressions, replicator unfolding etc.). The abstract syntax tree is then compiled to a high-level netlist containing coarse blocks of functionality. Optimizations are applied to this high-level Netlist before it is expanded to a technology specific lower-level netlist. This format is able to take advantage of specific architectural features of the target device,

such as fast carry chains, on-chip RAM, dedicated multiplier circuitry and so forth. Further optimization is then performed on the low-level netlist.

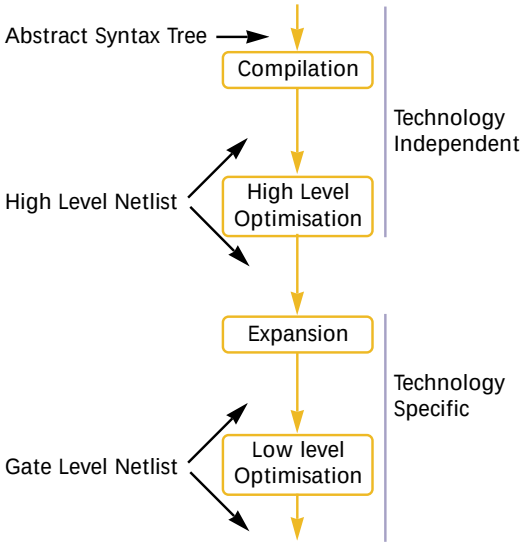


Figure: Compilation flow after parsing.

2.1 What optimizations will the compiler perform automatically on my code?

Three of the simplest optimization techniques used are described in more detail below:

2.1.1 Re-Writing

Re-writing involves transformations to the gate-level netlist that leverage logical equivalences. For example, a constant '1' input to an AND gate is removed (figure (a)). This sort of situation arises when constants are used in expressions such as

```
x = y + 7;
```

Chains of inverters can be optimized to reduce gate count, and flip-flops with constant outputs can be optimized to a logical '1' or '0'. If the output of a gate is not used, then that gate can be removed since it will have no effect on the circuit operation (figure (b)). Additional checks are performed to remove blocks of circuitry that form a 'loop' but are not connected to an external pin (figure (c)). This optimization is similar to dead-code elimination in a traditional compiler, except that we can remove hardware that is not connected to an output pin even if it is 'executed' at some point.

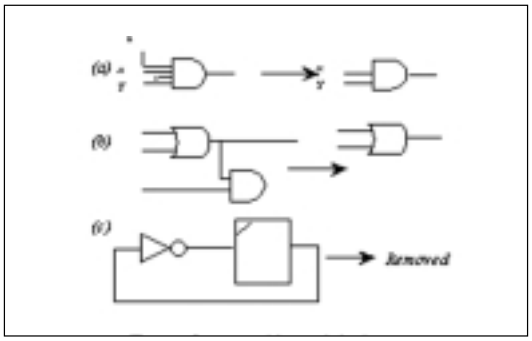


Figure: Some re-wiring optimizations.

2.1.2 Conditional Re-writing

An extension of re-writing is to apply test patterns to gates to infer impossible conditions (such as a gate output being both '0' and '1' at the same time) - since these conditions can never exist the compiler can optimize away any circuitry that contributes to such cases - see figure below for an example.

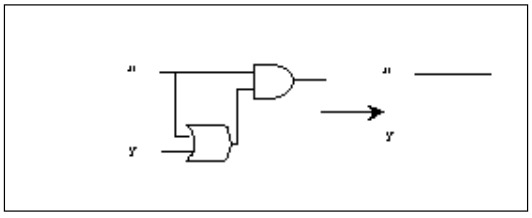


Figure: Conditional re-wiring.

2.1.3 Common Sub-Expression Elimination

This is a classic optimization from software compilation techniques and is also applicable at the hardware level. The example given in the figure below shows how two identical gates with identical inputs are optimized.

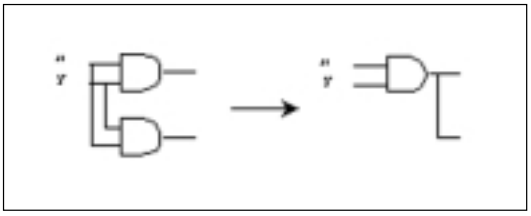


Figure: Common sub-expression elimination.

2.2 What optimizations should I not rely upon the compiler to perfom?

The compiler doesn't attempt to do several types of optimization. No rescheduling can be performed on user code (this follows from the timing semantics of the language which gives the user explicit control over the timing of their code).
Secondly, the compiler makes no attempt to dynamically reuse hardware. The user must explicitly reuse the hardware for example:

```
void main(void)
{
   ...
   a = b+c;
   d = e+f;
   ...
}
```

This will instantiate two adders:

```
int add(x, y)
{
   return x+y;
}
```

```
void main(void)
{
   ...
   a = add(b,c);
   d = add(e,f);
   ...
}
```

This will reuse the same adder.

The compiler also makes no attempt at traditional state machine minimization. However, the state machines generated by the compiler are already very small.

2.3 Does the compiler optimize away parts of an array that are never accessed? What about a RAM/ROM?

If an array is being indexed by constants, then the compiler knows at compile time which locations are not being used. Some parts of RAMs may get optimized away according to the granularity of the RAM blocks used (e.g. 1 LUT or 1 block). Parts of ROM with trivial contents (e.g. all zeros) may get optimized away.

In the case of a RAM/ROM or Array being indexed with a variable, then the compiler cannot know at compile time which items will never be accessed, so it cannot optimize them away.

3 Tell me more about checks the compiler performs

3.1 Will the compiler generate an error if I have a shared function, channel, RAM or ROM and I try to use it in different branches of my code in the same clock cycle?

At compile time, the compiler cannot test every single possible state of the program to check whether parallel accesses to a shared resource will ever happen. In hardware, there is no logic implemented

to check for parallel accesses to a single shared resource, so the programmer has to ensure that there no parallel accesses can occur.

3.2 What about during simulation?

The simulation tool does not currently give any errors when illegal parallel accesses to a shared resource take place, but this will be implemented at a later date.

3.3 Does the compiler generate an error if I access a location that does not exist in a RAM/ROM?

If an array or RAM/ROM is being indexed by constants, then the compiler can check at compile time whether or not the index is illegal. In the case of a RAM/ROM or array indexed with a variable, then the compiler cannot know at compile time what values this index will take, so no error can be generated. In hardware, there is no checking of the validity of an index and so the data read from an element that does not exist will be undefined. In simulation, the simulation tool does not currently give any errors when a non-existent element is accessed, but this will be implemented at a later date.

4 How does the Handel-C simulator work?

The high-level netlist description of the Handel-C algorithm is expanded into logically equivalent blocks of C code which are combined to produce a sequential program that is compiled by a standard C compiler.

5 What about timing simulation?

The Handel-C simulator is clock-cycle based (as Handel-C itself is), so the Handel-C simulator can give no information on absolute timing in seconds, only numbers of clock cycles.

P&R tools can produce a post-layout timing annotated EDIF file. This can be simulated with (for example) the Xilinx Foundation simulator. Timing annotated VHDL files can also be produced for use with e.g. ModelSim.

6 Do you have any tips on place and route strategies?

Use the timing analyzer to find the critical paths in your program. If the -g compile option is used net names are annotated with the Handel-C file and line number that generated them.

7 Do you have examples of arithmetic operations mapped to different architectures?

The Handel-C compiler uses architectural features such as fast carry chains automatically. It is useful to be familiar with FPGA design techniques, for example constant coefficient multipliers map well to look up tables. In Handel-C these can be implemented as ROMs.

8 How do I make use of the carry chain logic in Xilinx Virtex devices?

Handel-C will automatically make use of the carry chain logic if the target FPGA family supports it. There is also the option to disable use of fast carry chains by the compiler, which can sometimes improve routability.

9 Is it possible to 'program' and select the voltage spec of the I/O pins (e.g. 3V or 5V output) using Handel-C.

Yes, Select I/O support is implemented. Any of the I/O standards available on Virtex, SpartanII and Altera APEX devices can be selected via the specification 'standard' on bus interfaces. One exception, however: differential standards (such as LVDS and LVPECL) are not supported yet. Support for these will be added at a later date.

10 Can I make explicit use of features available in the FPGA, for example Shift Register Look-Up-Tables?

The compiler will automatically make use of many of the features on the FPGA, but you can explicitly instantiate any of the components on the FPGA directly in Handel-C. This is done in the same way as integrating CoreGen modules, using the "interface" construct. See the Handel-C Language Reference Manual for details on the syntax of declaration and definition of interfaces.

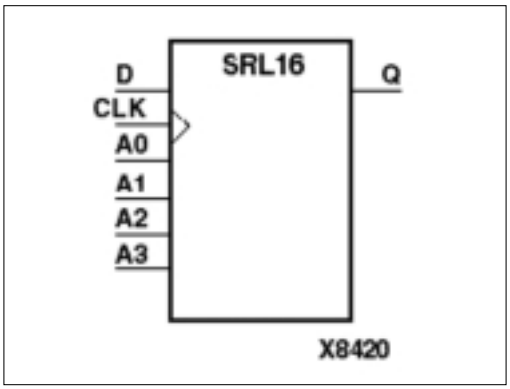
Here's an example using a 16-Bit Shift Register Look-Up-Table (LUT) on a Xilinx SpartanII, Virtex or Virtex2 part:

```
void main()
{
   unsigned 1 DIn, X;
   unsigned 4 AIn, Y;
   unsigned 1 DataOut;

   /*declare the interface to the Shift
   Register Look-Up-Table*/
   interface SRL16(unsigned 1 Q)
   MySRL(unsigned 1 D = DIn,
   unsigned 1 CLK = __clock,
   unsigned 4 A = AIn);

while(1)
{
      par
   {
      DataOut = MySRL.Q; //read data out
      DIn = X; //put data into D
      AIn = Y; //put data into A
      }
   }
}
```

cont'd

On every clock cycle, the value of DIn & AIn is passed to the inputs of the SRL16. The output is available every clock cycle as MySRL.Q.

The name of the primitive, in this case SRL16, will be propagated through to the place and route tools, which will recognize primitives for the target device. In this example, the __clock keyword tells the Handel-C compiler to route the clock for the current clock domain to this location.



11 Do the pin numbers in Handel-C correspond to pin numbers on the device itself?

Yes, the pin numbers used in Handel-C correspond to the pin numbers specified in the FPGA data book.

12 Is it possible to declare an interface with no pins and decide which pins to use at the place and route stage?

Yes, the pin specification on bus interfaces may be omitted to declare buses without pins. This may be useful to find the optimum pin locations at the place and route stage.

13 How is synchronization performed between clock domains using Handel-C channels and how predictable is the timing?

Handel-C channel communication between clock domains uses standard four-phase handshaking. The associated number of clock cycles and variation of clock cycles depends on clock skew and the relative clock frequency.

14 How do I set a default for a signal in a structure?

The initialization of members of a structure is done when an instance of a structure is created e.g.

```
typedef struct{
   signal unsigned int 1 Strobe1, Strobe2;
   unsigned int 8 Counter;
} SomeStuff;
SomeStuff MyInstance = {0,0,0};
```

The default values for the signals are set to 0 and the counter member variable is initialized to 0.

15 I want to use a function more than once in parallel, can I do this without typing in a seperate copy?

Handel-C provides a construct that can generate arrays of functions. Function arrays allow a designer to have control over exactly how many duplicates of a piece of hardware are generated e.g.

```
// Create an array of five multiplier functions
unsigned int Multiplier[5](unsigned int
InputA, unsigned int InputB)
{
   return InputA * InputB;
}
// To use the functions, the index specified will
indicate which copy of the // function is used
Thing = Multiplier[0](Var1, Var2);
```

16 I have a register that requires mutual exclusion protection, how can I do that?

Handel-C supports mutual exclusion protection using the sema keyword e.g.

```
unsigned int 1 ProtectedBit;  // The
protected register
sema BitProtection;
macro proc SetBit(Value)
{
   while(trysema(BitProtection)) delay;
   par
   {
      ProtectedBit = Value;
      releasesema(BitProtection);
   }
}
```

17 Can I connect an FPGA device to an open collector bus in Handel-C?

Yes, you can modify the interpretation of the bus_ts interface to perform open collector functionality. The data output is tied to zero and the data register is inverted and connected to the output enable connection of the bus interface. When a '0' is written to the data register the output is enabled and the FPGA sinks the external bus, when a '1' is written the output is disabled and the external bus state remains in its default state ('1' assuming external pull up resistors are present, the FPGA has internal pull up resistors if they are required, add 'with {pull=1}' to the end of the interface definition to add a pull up resistor to the pin). e.g.

```
unsigned int Data;
interface bus_ts(unsigned int 1 DataInput)
DataPorts(unsgined int NormallyTheData=0,
unsigned int 1NowTheData=~Data);
```

18 I need a section of code to be reset if a certain condition occurs when it is executing, how do I do that?

Handel-C provides the try reset mechanism. e.g.

```
try
{
   ... code ...
}
reset (reset-condition/expression)
{
   ... code to execute when a reset occurs ...
}
```

19 I am crossing a clock domain boundary and the clocks are synchronised, how do I avoid the minimum 4 cycle handshaking mechanism used by channels?

Multi-port RAMs can also be used to cross clock boundaries. You can create an mpram with only one storage location, this will build a register than can be written by one clock domain and read from another. You will have to generate your own hand-shaking logic; this can be implemented using more mprams or careful design.

20 I am using interfaces to create instances of external modules, the modules have some circular connections how can I make these connection in Handel-C.

Handel-C, like C, requires definitions to be in sequence. In order to refer to something not yet defined, a declaration (or prototype) must be written. With functions you would use a function prototype to do this, with interface and other components you can use the extern keyword. This will allow a direct connection with no intermediate logic. e.g.

```
// Early reference to Module B, a declaration
extern interface ModuleB(unsigned int 1
ModuleBOutput) ModuleBPorts(unsigned int 1
ModuleBInput);

interface ModuleA(unsigned int 1
ModuleAOutput) ModuleAPorts(unsigned int 1
ModuleAInput=ModuleBPorts.ModuleBOutput);
interface ModuleB(unsigned int 1
ModuleBOutput) ModuleBPorts(unsigned int 1
ModuleBInput =
ModuleAPorts.ModuleAOutput);
```

21 Can I have Handel-C macros in one source file and use them from another?

Yes, this is achieved using the same syntax as for functions. e.g.

source 1:

```
macro expr Add(Number1, Number2) =
(Number1+Number2);
```

source 2:

```
extern macro expr Add();
```

22 I am building a library file. Can it take parameters after it has been built?

A library can support parameters by using Handel-C macros. The source for the library should contain external references to macros that will be defined in the header to accompany the library. e.g.

Library Source (compiled into the .lib file):

```
extern macro expr BufferSize();
ram unsigned int
InternalDataBuffer[BufferSize()];
```

Library Header (To be included into files that will use the library):

```
macro expr BufferSize() = (54);
```

NOTES

>: FREQUENTLY ASKED QUESTIONS//

Celoxica Ltd.
20 Park Gate, Milton Park
Abingdon
Oxfordshire, OX14 4SH
United Kingdom
Tel: +44 (0)1235 863656
Fax: +44 (0)1235 863648

Celoxica, Inc.
900 East Hamilton Avenue
Campbell, CA 95008
USA
Tel: 1 800 570 7004
Tel: +1 408 626 9070
Fax: +1 408 626 9079

Celoxica Japan K.K.
YBP West Tower, 11F
134 Godo-cho, Hodogaya-ku
Yokohama 240-0005
Japan
Tel: +81 (0)45 331 0218
Fax: +81 (0)45 331 0433

Celoxica Singapore
Level 37
50 Raffles Place
Singapore
048623
Tel: +65 320 8735
Fax: +65 320 8730