Celoxica™

Software-Compiled System Design

# Using Handel-C with DK

Ashley Sutcliffe

Version 1.0

# Nomenclature

- DK
  - DK stands for Design Kit
  - This is the tool, including the GUI, the simulator and the hardware compiler
- Handel-C
  - Handel-C is the programming language
  - For hardware design
  - Not Handle-C!
  - Handel was the favourite composer of one of the early designers
- PDK
  - PDK stands for Platform Developer's Kit
  - This is a package of libraries, tools and source code to help users design using Handel-C and target supported hardware platforms
- FPGA
  - Field Programmable Gate Array

# Course Goals

- Basic Course
  - Be familiar with the main features of Handel-C and DK
  - Able to start writing Handel-C designs
- Advanced Course
  - Be familiar with the advanced features of Handel-C and DK
  - Able to design and debug efficient, flexible Handel-C designs
- This course does not replace the manual!
  - We won’t be covering the entire language exhaustively
  - You won’t see every feature of DK

# Design Experience

- Some of experience of the following is advantageous, but not necessary
  - C programming
  - FPGA design
- You don’t need to be an expert in any of these
- This course covers the basics of FPGA design
- Note for software programmers
  - Handel-C is a *hardware* design language
  - Some of the optimisations and tricks for ANSI-C programming are not appropriate for Handel-C
- Note for hardware designers
  - Handel-C is different to traditional HDLs
  - Take the time to learn the Handel-C way of designing, otherwise you may not perceive the full advantage of Handel-C

Celoxica

Software-Compiled System Design

# Using Handel-C with DK

Basic Course

Ashley Sutcliffe

# Goals for the Basic Course

- Understand the building blocks of Handel-C
- Know how to build, simulate and debug your design
- Know how to build a design for hardware
- Know how to use the advanced language features of Handel-C
- Be able to design and implement a complete design in Handel-C that will both simulate and run in an FPGA

Celoxica™

Software-Compiled System Design

# Handel-C Building Blocks

Handel-C timing, parallel and sequential code, loops and conditions

# Handel-C Concepts

- Handel-C is a C-like language
  - ANSI-C syntax and semantics
  - Extensions and restrictions for hardware design
- Designed for synchronous hardware design
  - Optimised for FPGAs
  - Everything that simulates will compile to hardware
- Extensions to C allow you to produce efficient hardware
  - `par` to introduce parallelism
  - Arbitrary word widths
  - Synchronisation
  - Hardware interfaces

# A Simple Handel-C Program

```
set clock = external;                               // Set clock source

void main()                                         // Entry point for the design
{
    static unsigned 32 a = 238888872, b = 12910669; // Input variables
    unsigned 32 Result;                             // Variable for result

   interface bus_out() OutputResult(Result);        // Output the result to pins

    while (a != b)
    {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }

    Result = a;                                     // Set the output variable
}
```

Software-Compiled System Design

# Handel-C Timing

- Handel-C is implicitly sequential
- Each assignment takes one clock cycle

```
a = b;       // clock cycle 1
a = a + 1; // clock cycle 2
```

- Delay statement to do nothing for a clock cycle

```
a = b;       // clock cycle 1
delay;       // clock cycle 2
a = a + 1; // clock cycle 3
```

- No side-effects in expressions

```
a = b++;    //not allowed
```

  - Breaks the timing model of each assignment taking a clock cycle
  - Anything with side-effects can be written without them

# Variables

- Basic type is the integer
  - No floating point type in Handel-C
- Integers can be either signed or unsigned
  - Signed numbers are stored in two's complement format
- Can be any width

```
signed int 8 a;    // signed 8 bit variable "a"
int 8 a;           // can omit the signed keyword
unsigned int 8 a;  // unsigned 8 bit variable
unsigned 8 a;      // can omit the int keyword
```

  - LSB is stored in bit 0, MSB in bit n-1, where n is the width of the word
- Pre-determined widths available
  - `char (8), short (16), long (32), int32 (32), int64 (64)`
  - Can specify unsigned
- Behave like registers (often referred to as registers)
  - Take new value on the clock cycle following an assignment

# par Statement

- Expresses what should happen in parallel
- Everything in the subsequent block happens in parallel

Sequential Block

```
// 3 Clock Cycles
{
    a=1;
    b=2;
    c=3;
}
```

Parallel Block

```
// 1 Clock Cycle
par
{
    a=1;
    b=2;
    c=3;
}
```

- Seq statement says that a section will be sequential
  - This is just for clarity
  - You can leave the seq out

# Parallel and Sequential Examples

```
unsigned 4 a,b;

seq
{
    a = 1;

    par
    {
        a = a + 1;
        b = 5;
    }

    par
    {
        b = b + 1;
        a = b;
    }
}
```

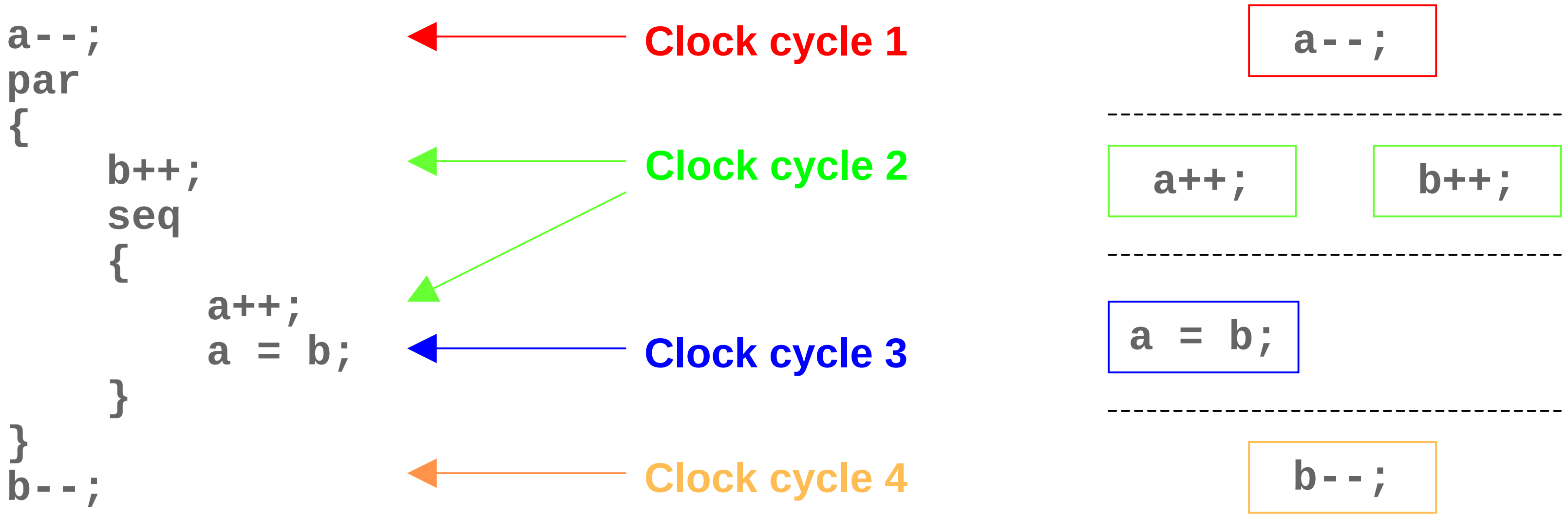
Clock cycle 1: a = 1

Clock cycle 2: a = 2, b=5

Clock cycle 3: a = 5, b=6

# `par` Completion

- `par` block completes when longest path completes



```
a--;
par
{
    b++;
    seq
    {
        a++;
        a = b;
    }
}
b--;
```

Clock cycle 1

Clock cycle 2

Clock cycle 3

Clock cycle 4

```
a--;
----------------------------------
a++;        b++;
----------------------------------
a = b;
----------------------------------
b--;
```

- The `par` statement can be used to express both coarse and fine grained parallelism
  - Individual statements in parallel
  - Functions in parallel

# par Examples

- Can read from variable in parallel
  - Simply wires the two variables together

```
par
{
    b = a;
    c = a;
}
```

- Can’t write to same variable in parallel
  - Undefined value will be written

```
par
{
    a = b; // won’t work
    a = c; // won’t work
}
```

- No need to use temporary variables to swap values

```
unsigned 4 a, b;
par
{
    a = b;
    b = a;
}
```

# Conditional Branching

- Control the flow of your program
- Conditions are evaluated in 0 clock cycles
- `if`
  - Exactly like C
  - Can use a `delay` statement in the else clause to balance execution time

```
if (a == 0)
    a++;
else
    delay;
```

# `for` Loops

- Syntax

  `for(initialisation; test; increment) body`
- All expressions optional
- `initialisation` takes a clock cycle
- `test` evaluated before each iteration of body
- `increment` expression takes a clock cycle at the end of each execution of body
- `for` is not recommended for general use
  - Use `while` or `do…while`
  - `increment` can always be done in parallel with body

# while Loops

- while

  `while(condition) body`

  - condition evaluated before each execution of body
  - while statement terminates if condition evaluates to zero
- do…while

  `do body while(condition);`

  - Always executed at least once
- while(1)
  - Run forever
- Advantage over for
  - Can place initialisation and increments of loop counters in parallel with other code, either inside the body or elsewhere
- Efficient use of loops discussed in advanced course

# Loops – Combinational Cycles

- Every branch within a loop must take at least 1 clock cycle

```
while(1)
{
    if (x)
        delay;
}
```

- If x is not true, the loop would execute in zero clock cycles, creating an invalid circuit
- Sometimes DK will issue a warning, such as
  - *Breaking combinational cycle (while statement) - may alter timing*
  - Fix the cause – this way you know the exact behaviour
- Often DK will produce an error message
  - Expanding the error message, you can follow the route of the combinational cycle to find the error

# Handel-C State Machine

- The state machine is implicit
- Constructed from conditional branches, loops, sequential blocks and parallel blocks
- Handel-C produces a one-hot state machine
- You can produce a very complex state machine with ease
- You can then tweak this machine as much as you want very simply
- The final result is easy for others to understand
- Warning
  - It is very easy for HDL designers to lapse into explicitly writing their state machines, which is possible in Handel-C
  - You will not get the full power of Handel-C if you do this
  - Take the time to learn the Handel-C way of doing things

# Key Points

- Assignments take one clock cycle
- Variables behave like registers
- Par block completes when longest path completes
- Can’t write to the same variable in parallel
- Conditions are evaluated in zero clock cycles
- Use `while` loops instead of `for`
- Don’t be tempted to write your own state machine in Handel-C

Celoxica™

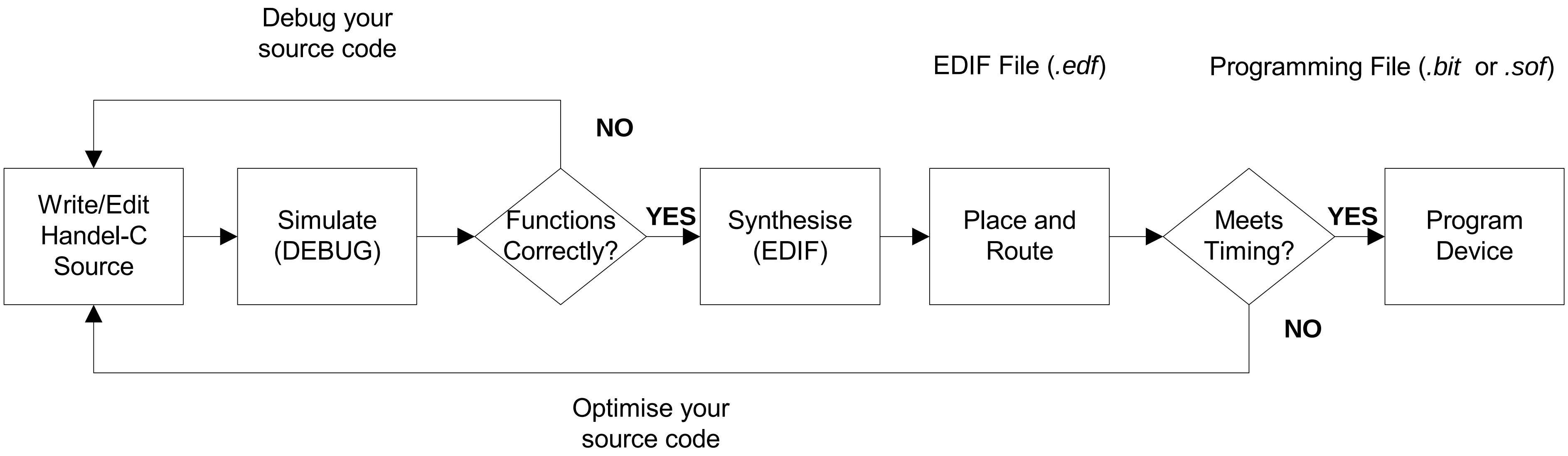
Software-Compiled System Design

# DK Basics

Introduction to DK, tour of the GUI, simulating and debugging your code

# DK Basics

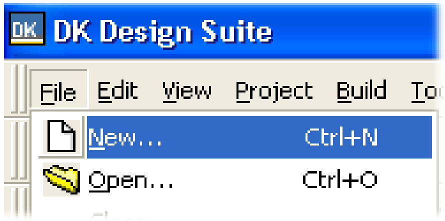
- DK is an Integrated Development Environment (IDE)
  - Familiar look and feel
  - Compile, simulate and debug
  - Compile to hardware (synthesise)
  - Run other tools, e.g. place and route tools, software compilers
- Simplified DK design flow:

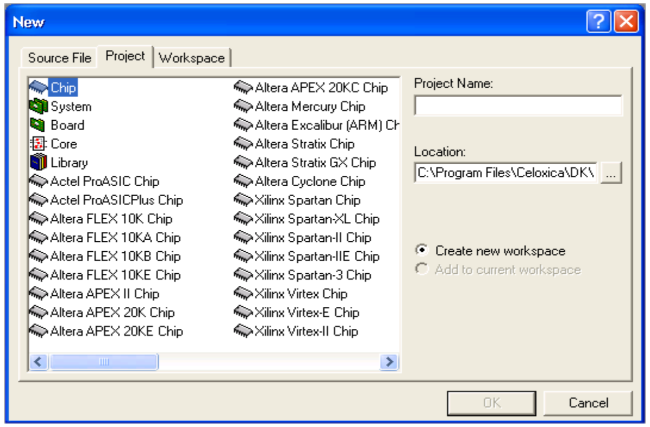


Software-Compiled System Design

# Creating Projects and Workspaces

- A workspace is a collection of projects
- A project can contain any number of source files
- Click new on the file menu
- Either create a new, empty workspace
  - You can then add projects later
- Or create a new project, which will create a new workspace automatically
  - If you already have a workspace open, you can select to add this new project to the current workspace
- Enter a name for the project and where you want to save it
- Select what type of project you want to create





Project Name:

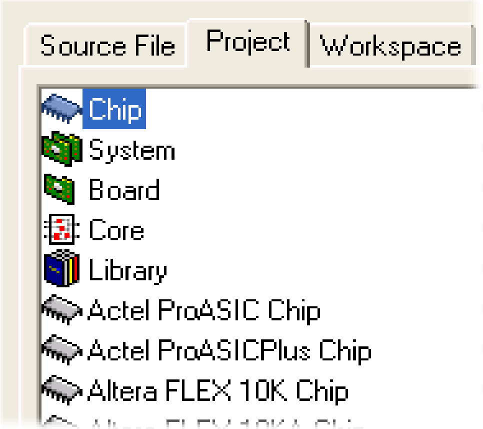
Location: C:\Program Files\Celoxica\DK\

● Create new workspace
○ Add to current workspace

Celoxica
Software-Compiled System Design

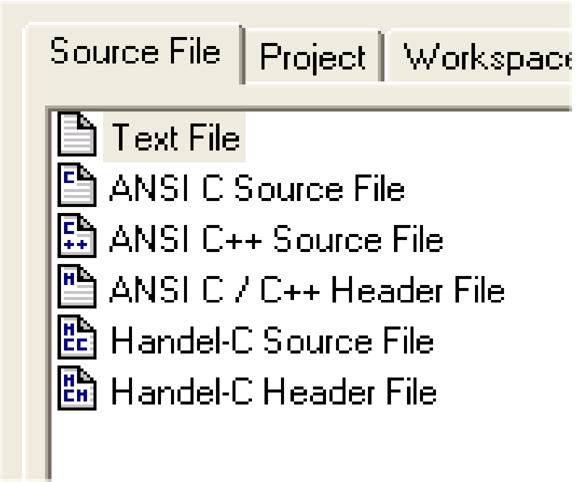
# Project Types

- Chip
  - Generic chip - does not use device-specific resources
- Specific chip
  - Targeted towards a particular device
  - Uses device-specific resources
- Core
  - Discrete piece of code e.g. a filter, a FIFO
  - Targeted towards a particular device architecture
- Library
  - Defines functions that can be used in other projects
  - Like a lib file in C
  - Can be generic, which allows user to target simulation, EDIF, VHDL and Verilog
  - Can be limited to a particular output e.g. simulation
- Other
  - Board can contain multiple chip projects
  - System can contain multiple board projects



# Creating Source Files

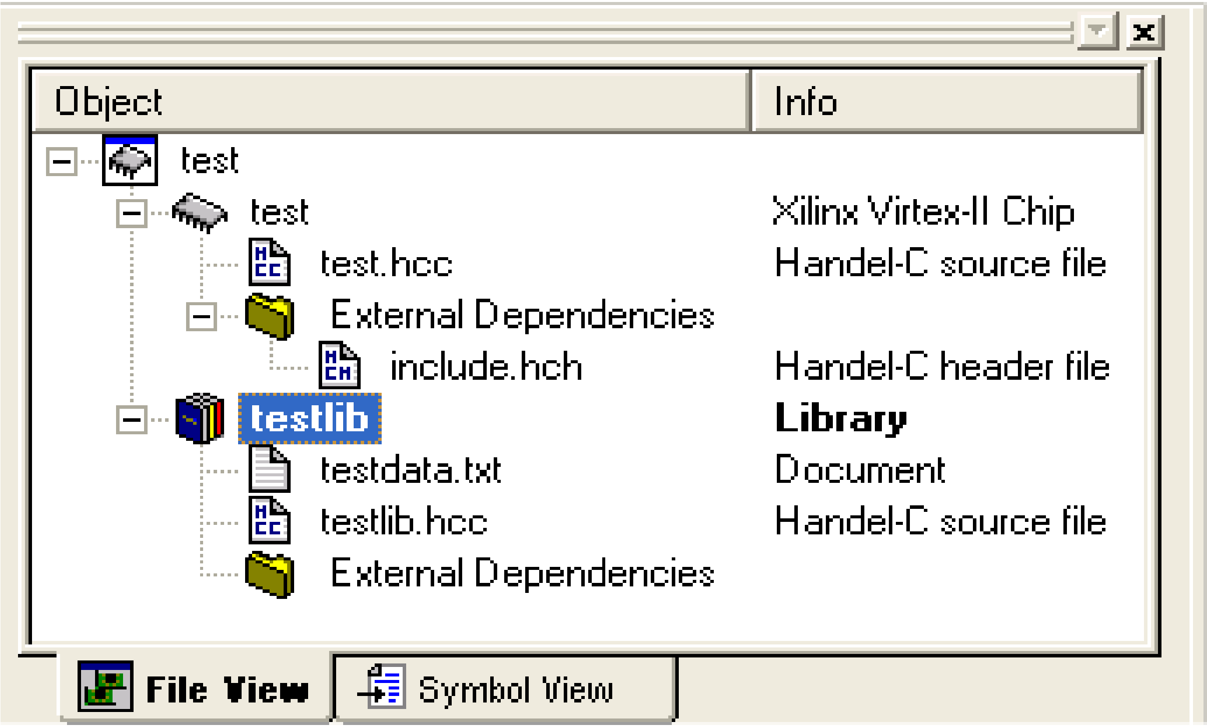
- Enter a name for the source file
- You can select to add the file to the current project
  - You don't need to add header files to the project
- Text files are useful for test data input and output in your project
  - DK will prompt you to reload if the contents change
- ANSI-C or C++ code can be built within DK using an external compiler e.g. MS Visual C++, gcc



Text File
ANSI C Source File
ANSI C++ Source File
ANSI C / C++ Header File
Handel-C Source File
Handel-C Header File

# Workspace View – File View

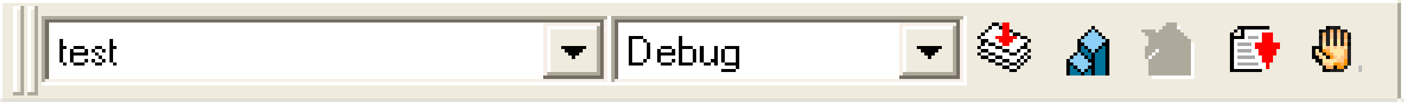
- **Lists all your projects and source files**
- **If you include header files, they will appear in** External Dependencies



| Object | Info |
|---|---|
| test | |
| test | Xilinx Virtex-II Chip |
| test.hcc | Handel-C source file |
| External Dependencies | |
| include.hch | Handel-C header file |
| testlib | **Library** |
| testdata.txt | Document |
| testlib.hcc | Handel-C source file |
| External Dependencies | |

# Building Your Design

- Select the project and configuration that you wish to build
  - Debug is the default simulation build configuration



test

Debug

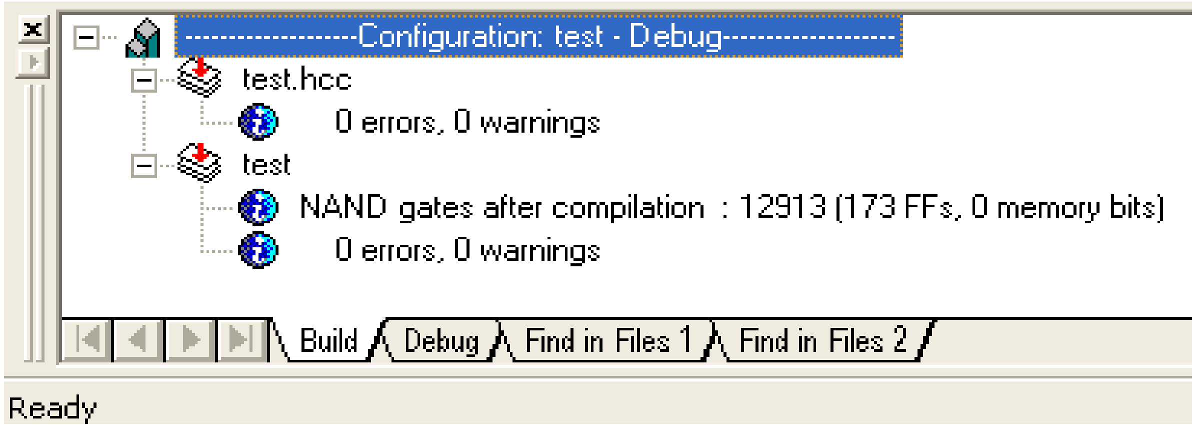
- The build button builds the entire project
  - Compiles all source files that have changed since the last build
- You can compile a single file using the compile button
- When a build is in progress, the Cancel Build button appears
  - This won’t cancel any post build steps (for example place and route tools)
- To recompile all source files, select Build → Rebuild All from the menu bar
- To clean all intermediate files, select Build → Clean

# Build Process

- **When you click on** Build, **DK builds your project in two stages**
- **Compilation stage**
  - **All source files are first compiled to object code**
- **Link stage**
  - **All object code and libraries are linked together**
  - **This is where you will get width and type errors between source files and library code**

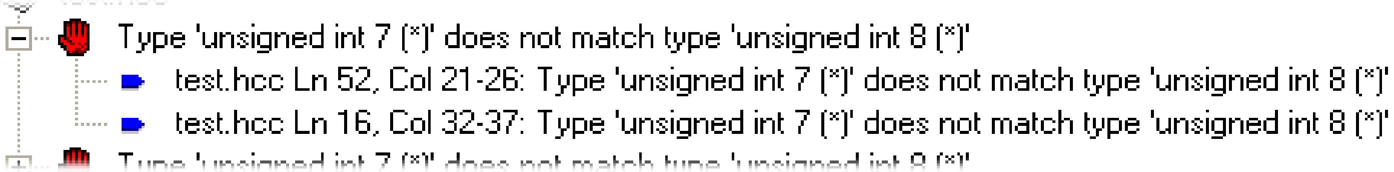
# The Build Output Window

- The build tab shows you build success and error messages



```
------------------Configuration: test - Debug------------------
test.hcc
    0 errors, 0 warnings
test
    NAND gates after compilation  : 12913 (173 FFs, 0 memory bits)
    0 errors, 0 warnings
```

- Double click on any errors to jump to the file and line in the code window
- Some errors are expandable to get more information



```
Type 'unsigned int 7 (*)' does not match type 'unsigned int 8 (*)'
    test.hcc Ln 52, Col 21-26: Type 'unsigned int 7 (*)' does not match type 'unsigned int 8 (*)'
    test.hcc Ln 16, Col 32-37: Type 'unsigned int 7 (*)' does not match type 'unsigned int 8 (*)'
Type 'unsigned int 7 (*)' does not match type 'unsigned int 8 (*)'
```

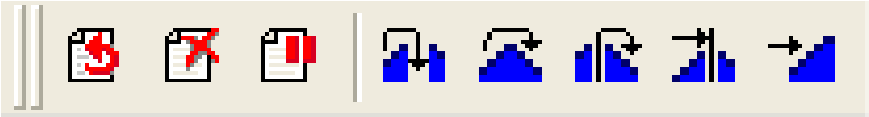
# Simulating Your Design

- Finest resolution of simulation is the clock cycle
  - No idea of absolute time
- Fast simulation
  - Depends on design complexity
- Functional debug
- Does not guarantee correct sub-clock cycle timing
  - e.g. for peripheral interfaces, interfaces to HDL blocks
  - It is essential to apply timing constraints and make sure they are met when building for hardware

# Debugging Your Design

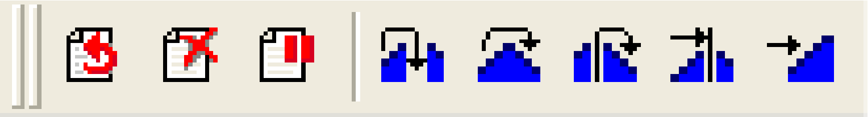
- **To start a debug session, build for** Debug **and click either** Run**,** Step Into **or** Run to Cursor
- **A DOS command box will appear**
  - **This is where any stdio input and output will be performed**
  - **You can safely minimise this or send it to the background**
  - **NOTE: Closing this box ends the simulation**
- **Clicking the** Run **button makes the simulator run freely**

- **The** Debugger **toolbar contains the rest of the simulator control buttons**



- **You can pause the simulator while it is running freely**
- **The** Restart **button resets the simulator to its initial state and pauses at the beginning of the simulation**

# Stepping through Your Code



Step Into **takes you to the first statement within a function call, or to the next clock edge in the current thread**

Step Over **skips to the end of the clock cycle following a function call, or to the next clock edge in the current thread**

Step Out **skips to the end of the clock cycle following the current call to a function, or makes the simulator run freely**

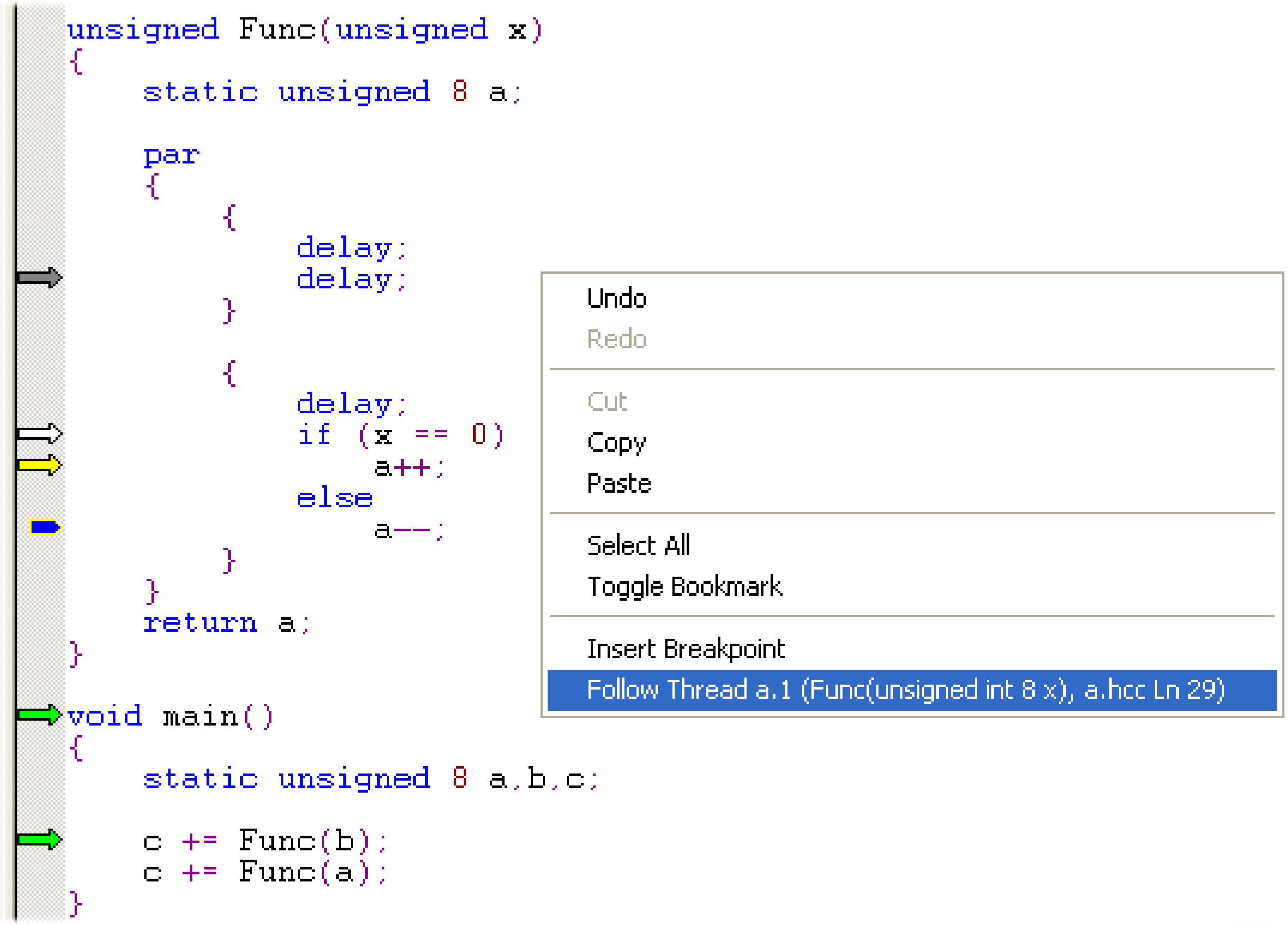
Run to Cursor **will run the simulator to the current cursor position in the debug window, or make the simulator run freely**

Advance **will advance the simulator to the next execution point, rather than a full clock cycle**

# Code Window

- When the simulation is not running freely, various arrows in the left margin of the code window show the current state of the simulation
- The yellow arrow shows the current point of execution in the current thread
- The white arrows show all the code that executed on the same clock cycle in the current thread, including control logic
- The green arrows show the origin of calls to the current function/macro in the current thread
  - You can right click on this arrow and follow another thread within this function/macro
- The grey arrows show the information for the above but for the other threads in the design
  - You can right click on the line and follow the thread

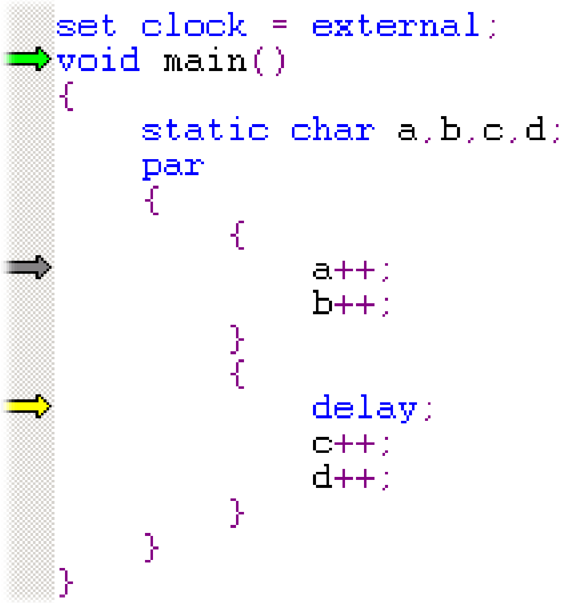
# Code Window - 2



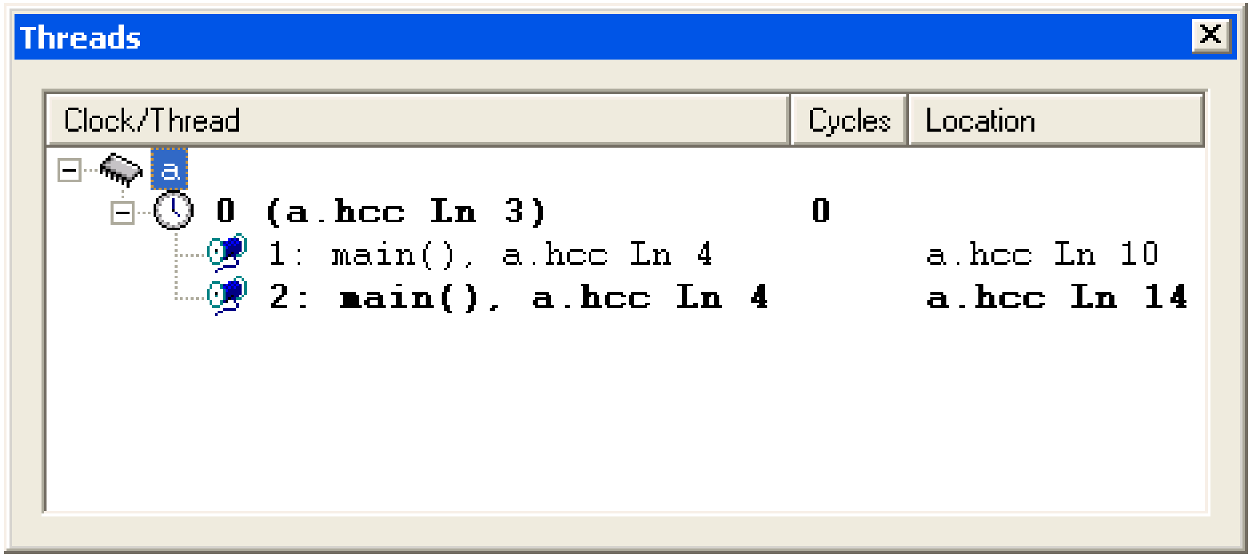
```
unsigned Func(unsigned x)
{
    static unsigned 8 a;

    par
    {
        {
            delay;
            delay;
        }

        {
            delay;
            if (x == 0)
                a++;
            else
                a--;
        }
    }
    return a;
}

void main()
{
    static unsigned 8 a,b,c;

    c += Func(b);
    c += Func(a);
}
```

# Threads Window

- Shows all the currently executing threads, grouped by clock
- The current clock cycle for each clock is in the Clock/Thread column
- The current clock and thread are in bold



```
set clock = external;
void main()
{
    static char a,b,c,d;
    par
    {
        {
            a++;
            b++;
        }
        {
            delay;
            c++;
            d++;
        }
    }
}
```

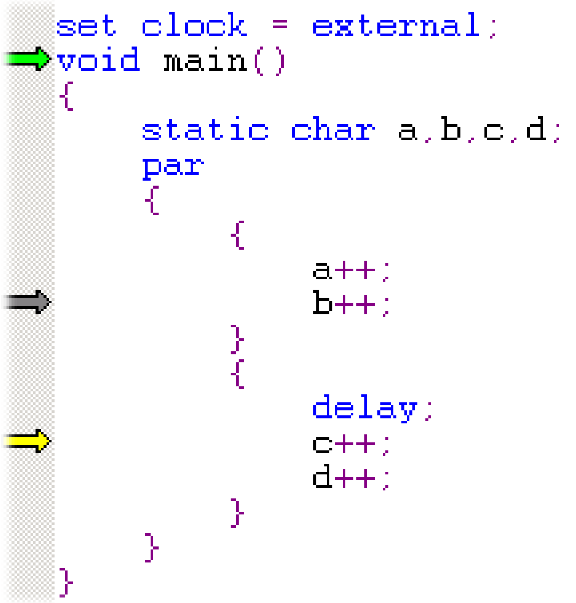


Threads

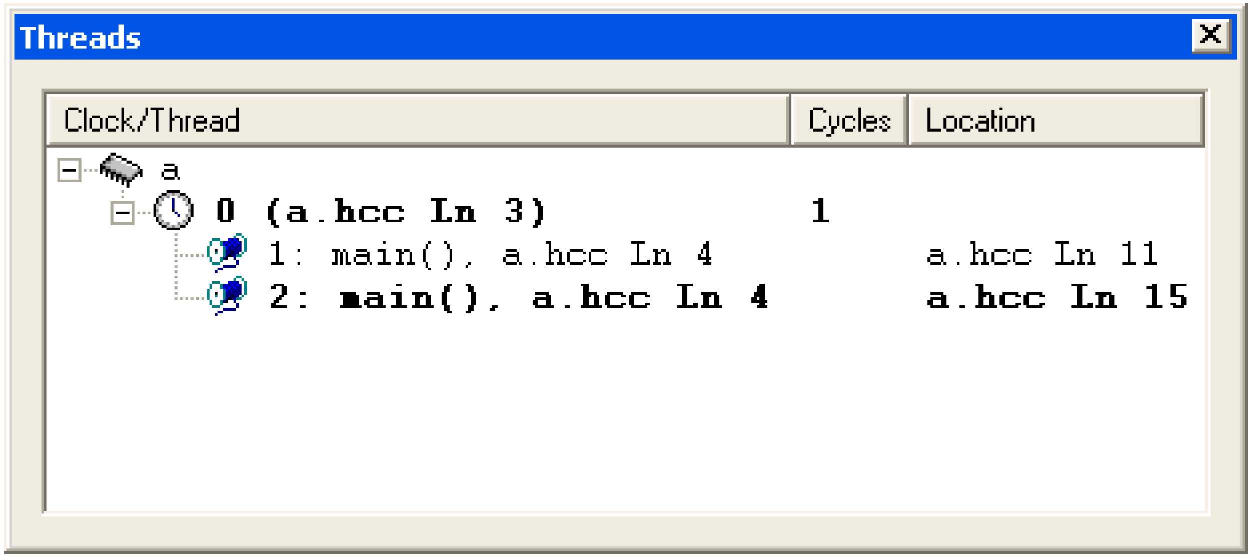
| Clock/Thread | Cycles | Location |
|---|---|---|
| a | | |
| **0 (a.hcc Ln 3)** | **0** | |
| 1: main(), a.hcc Ln 4 | | a.hcc Ln 10 |
| **2: main(), a.hcc Ln 4** | | **a.hcc Ln 14** |

# Threads Window - 2

- You can see the function name, line number and filename for the current statement in each of the threads
- By right clicking on a thread, you can either follow that thread or jump to the location of the statement in the file



```
set clock = external;
void main()
{
    static char a,b,c,d;
    par
    {
        {
            a++;
            b++;
        }
        {
            delay;
            c++;
            d++;
        }
    }
}
```



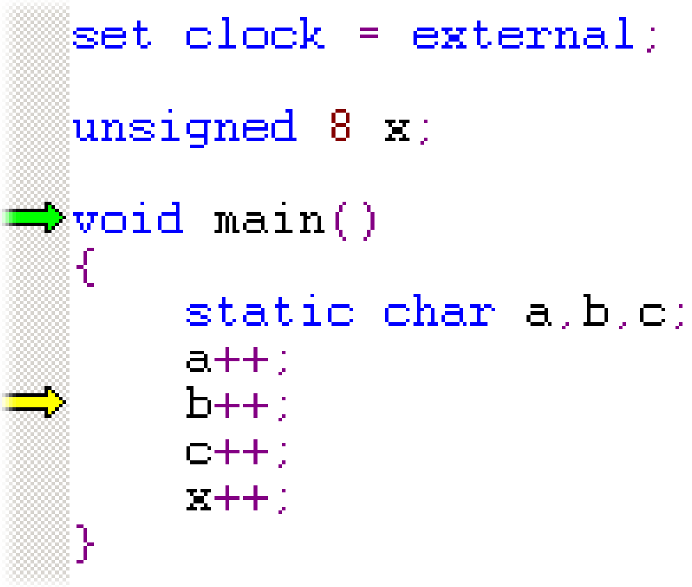
Threads

| Clock/Thread | Cycles | Location |
|---|---|---|
| a | | |
| 0 (a.hcc Ln 3) | 1 | |
| 1: main(), a.hcc Ln 4 | | a.hcc Ln 11 |
| 2: main(), a.hcc Ln 4 | | a.hcc Ln 15 |

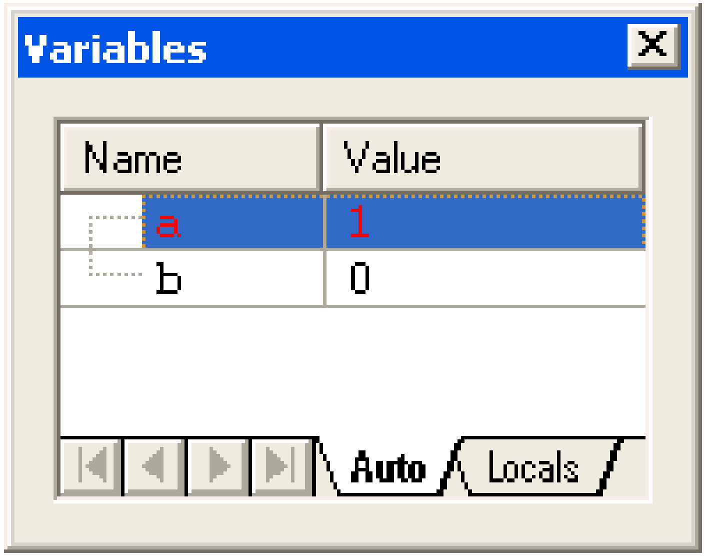
# Variables Window

- Auto **shows the variables used in the current and previous statements of the current thread**
  - **If you have just changed thread, then it shows you the variables used in the previous statement in the thread you have just switched from**
- Locals **shows you the variables in the scope of the current function or macro, including the global scope**
- **Variables show the value at the end of the current clock cycle**
  - **NOT the value at the current execution point**
- **Values will appear in red if they have changed from the previous clock cycle**
  - **If the variable is assigned to with the same value, it will not go red**
- **You can change the base of the display by right clicking in the window and selecting the appropriate option**

# Variables Window - Auto



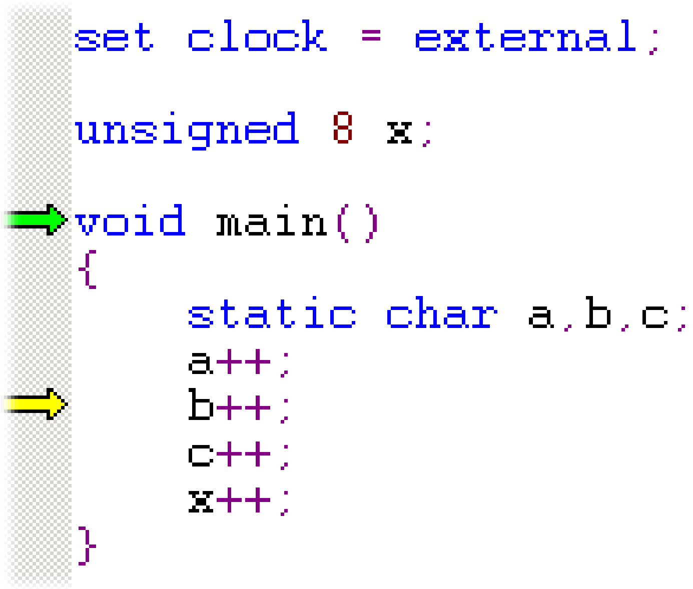
```
set clock = external;

unsigned 8 x;

void main()
{
    static char a,b,c;
    a++;
    b++;
    c++;
    x++;
}
```



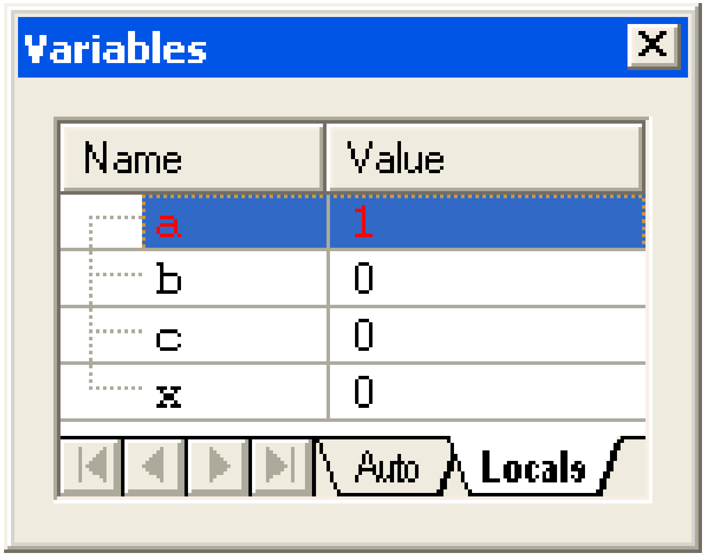
Variables

| Name | Value |
|---|---|
| a | 1 |
| b | 0 |

# Variables Window - Locals



```
set clock = external;

unsigned 8 x;

void main()
{
    static char a,b,c;
    a++;
    b++;
    c++;
    x++;
}
```



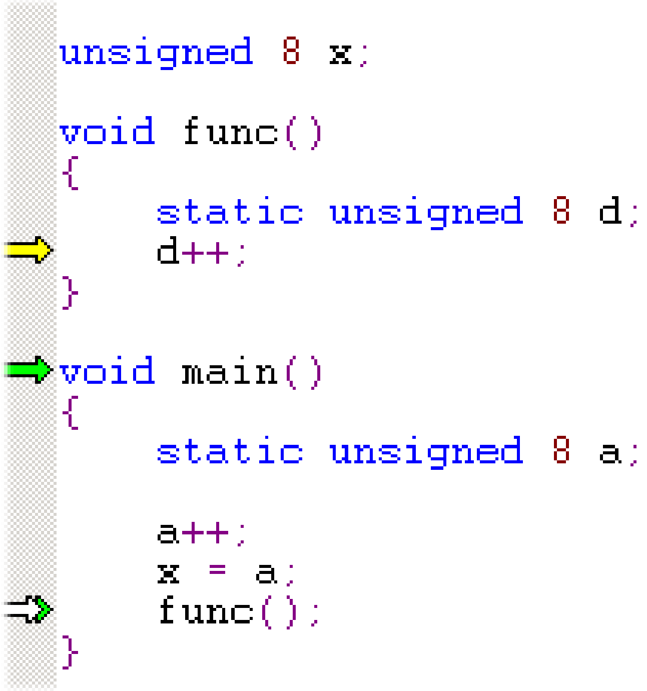
Variables

| Name | Value |
|---|---|
| a | 1 |
| b | 0 |
| c | 0 |
| x | 0 |

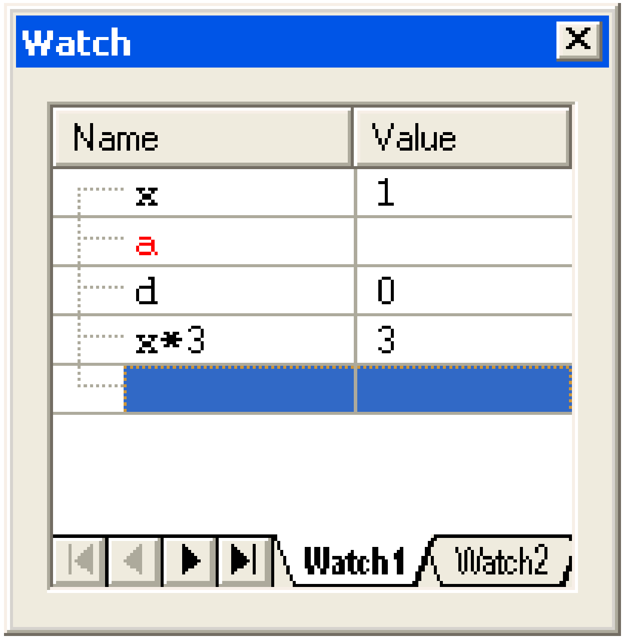
Celoxica
Software-Compiled System Design

# Watch Window

- Here you can enter variables and expressions of which you wish to see the value
- The value will be blank if the variable goes out of scope



```
unsigned 8 x;

void func()
{
    static unsigned 8 d;
    d++;
}

void main()
{
    static unsigned 8 a;

    a++;
    x = a;
    func();
}
```



Watch

| Name | Value |
|---|---|
| x | 1 |
| a | |
| d | 0 |
| x*3 | 3 |

# Breakpoints

- You can set a breakpoint on any statement in your code by clicking the Breakpoint button
- This causes a purple circle to appear in the margin on the line of the cursor in the code window

```
b++;
delay;
}
```

- When the simulator reaches that line of code, the simulation will pause and the current status of the simulation will be shown
- Clicking the breakpoint button again when the cursor is on that line if code will remove the breakpoint
- In the Edit → Breakpoints dialog box, you can set conditions on your breakpoints
  - e.g. To break after n clock cycles or when a variable reaches a certain value

# Key Points

- **Create a workspace and a project**
- **Select the appropriate project type**
- **Create a Handel-C source file**
- **Build for simulation**
- **Step through your code using the debug buttons**
- **Follow the arrows to see the current simulation state**
- **See which threads are currently executing in the** Threads **window**
- **Look at the contents of variables in the** Variables **and** Watch **windows**
- **Use breakpoints to stop the simulation at specific points**

Celoxica™

Software-Compiled System Design

# Handel-C Operators

Description of the Handel-C operators

# Operators

- Operator groups
  - Relational
  - Boolean
  - Bitwise
  - Shift
  - Bit manipulation (Handel-C specific)
  - Arithmetic
- Operator precedence
  - Defined in manual
  - Good practice to use explicit brackets
    - Easy to make mistakes if you don’t
    - Maintainable – easy to understand

# Relational Operators

- Used in conditional statements
- Both operands must have same type
- Type of the result is unsigned `1`
- `!=, ==`
  - e.g `if (a == b)`
  - Cheap to evaluate
  - Map well into FPGA architecture
  - == easily mistaken for =
    - DK will give an error message
    - *Handel-C does not support this form of statement*
- `>, <, >=, <=`
  - e.g `if (a >= b)`
  - Relatively expensive to evaluate, especially between two variables – discussed in detail in advanced course

# Boolean Operators

- &&, ||, ! (AND, OR, NOT)
- Both operands must have same type
- Type of the result is unsigned 1
- Easy to confuse && and &

```
c = 0xF && 0xF; // result = 1
d = 0xF & 0xF;  // result = 0xF
```

  - Likewise for || and |
- Test result against 0 to be clear
  - Won't make more logic

```
if (!a)
if (a == 0)
```

# Bitwise Operators

- Both operands must have same type
- Result has same type as inputs
- ^, |, &, ~  (XOR, OR, AND, NOT)
- Examples

```
a = 0xFF & 0xF0; // result = 0xF0
a = 0xFF | 0xF0; // result = 0xFF
a = 0xFF ^ 0xF0; // result = 0x0F
a = ~0xFF;       // result = 0x00
```

- Often used with constants to mask bits

```
unsigned 8 a, b;
b = 0xF1;
a = b & 0xF0     // result = 0xF0
```

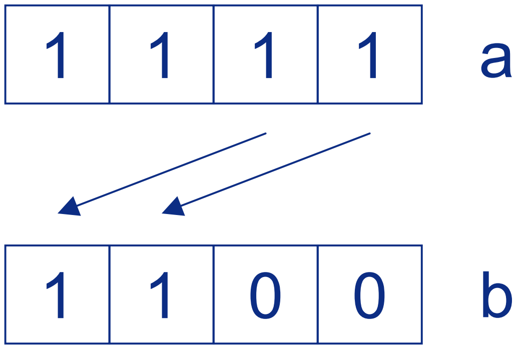
# Shift Operators

- Result has same type as input
- Left << and right >> shifts
- Shifting left
  - Top bits of the variable are lost
  - Bottom bits are filled with zeros
- Shifting right
  - Unsigned
    - Top bit are filled with zeros
    - Bottom bits are lost
  - Signed
    - Top bits are filled with copies of the original top bit
    - Bottom bits are lost
- Shifting by a constant is implemented as wires so is very cheap in hardware

# Shift Operators - 2

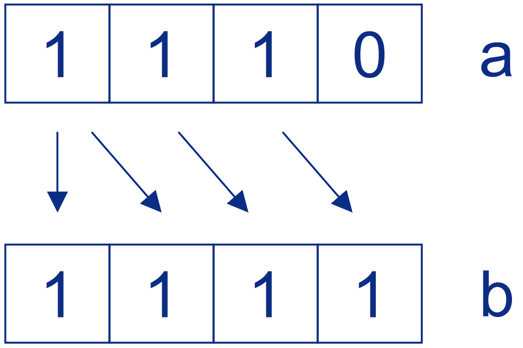
- Unsigned shifts

```
static unsigned 4 a = 0xF;
unsigned 4 b;
b = a << 2; // result = 0xC
b = a >> 1; // result = 0x7
```



- Signed Shifts

```
static signed 4 a = -2;
signed 4 b;
b = a << 1; // result = -4
b = a >> 1; // result = -1
```



# Shift Operators - 3

- You can shift by a variable
  - Be aware that this can be inefficient
  - Implemented as multiplexer
- The width of b needs to have a width equal to log2(width(a)+1) rounded up to the nearest whole number
- Example

```
unsigned 8 a;
unsigned 4 b;
a = a << b;
```

# Bit Manipulation

- In hardware, you have control over individual bits
  - Therefore new operators provided in Handel-C
- Very cheap in hardware
  - All implemented as simple wires
- Take <-, Drop \\

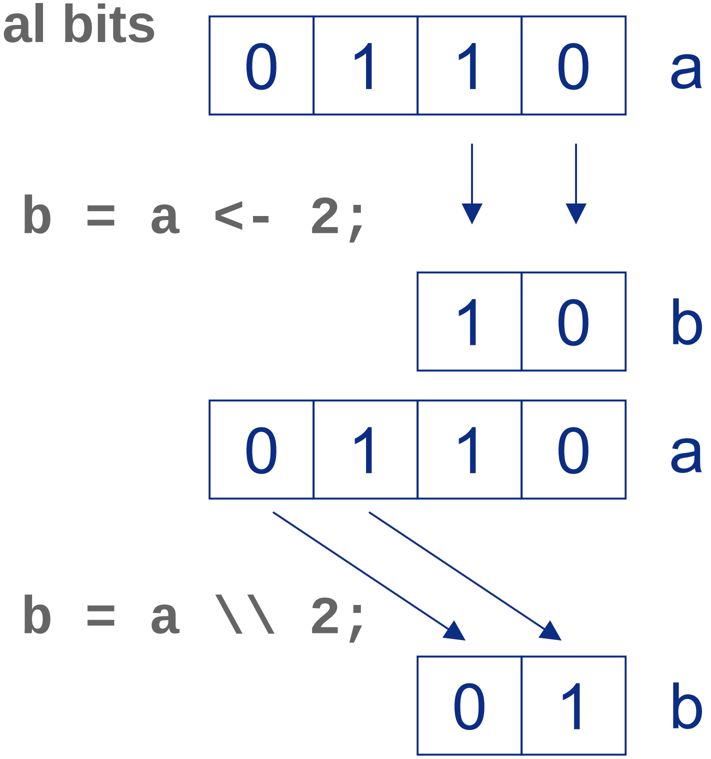
```
expression <- n bits
```

  - Width of result = n

```
expression \\ n bits
```

  - Width of result = width of expression - n
  - n must be compile-time constant

```
unsigned 4 a;
unsigned 2 b;
a = 6;
b = a <- 2; // take bottom 4 bits of a and assign them to b
b = a \\ 2; // drop bottom 4 bits of a and assign rest to b
```



Celoxica
Software-Compiled System Design

# Bit Manipulation - 2

- **Bit selection**

  `expression[n]`

  - n must be a compile-time constant
  - Width of result = 1
- **Range selection**

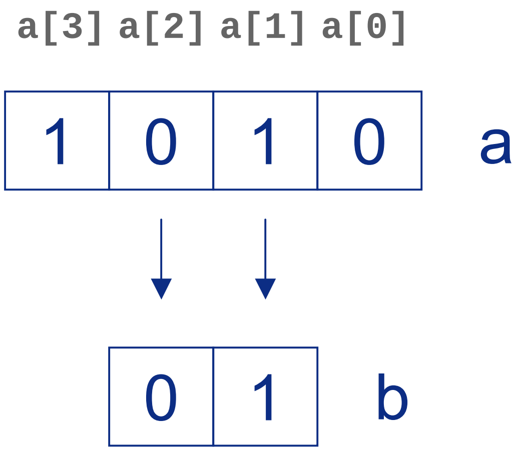
  `expression[n:m]`

  - n and m must be compile-time constants
  - n ≥ m
  - Width of result = (n-m)+1
- **Concatenation @**
  - Concatenate two words together to make a new one

  `expression a @ expression b`

  - Width of result = width of a + width of b

```
unsigned 4 a;
unsigned 2 b;
a = 10;
b = a[2:1];
```



a[3] a[2] a[1] a[0]

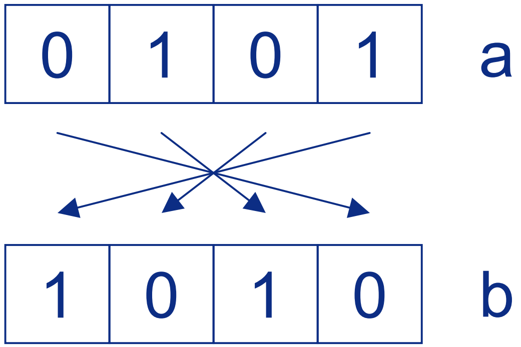
1 0 1 0 a

0 1 b

# Bit Manipulation - 3

- Can combine any of the bit manipulation operators together
  - E.g. bit-reversal

```
unsigned 4 a, b;
a = 5;
b = a[0]@a[1]@a[2]@a[3];
```

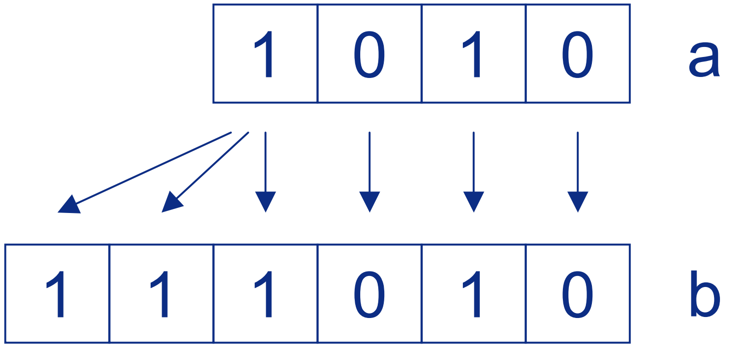


0 1 0 1 a

1 0 1 0 b

  - E.g. sign-extension

```
signed 4 a;
signed 6 b;
a = -6;
b = a[3]@a[3]@a;
```



1 0 1 0 a

1 1 1 0 1 0 b

# Bit-Masks versus Bit-Manipulation

- C method of bit masks still works and is efficient
- Handel-C method is perhaps clearer relationship to hardware
- May want to maintain C-like code
- C method is often much more verbose
- Example: circular shift (rotate)
  - Shift left 1 and put the value of bit 7 into bit 0

```
b = a[6:0]@a[7];
c = (a<<1)|(a>>7);
```

- Example: Concatenate selections
  - Concatenate bottom 4  bits of b to top 4 bits of a

```
d = b[3:0]@a[7:4];
e = ((a&0xF0)>>4)|((b&0x0F)<<4);
```

# Arithmetic Operators

- Both operands must have same type
- Result has same type as inputs
- Add +, Subtract -, Multiply *, divide /, modulus %
- Multiply
  - To contain largest result, user must make width of result equal to sum of widths of inputs
  - Zero-pad for unsigned or sign extend inputs for signed
  - Remember that signed variables are stored in two's complement format

```
unsigned 8 Result;
unsigned 4 OpA, OpB;

Result = (0@OpA) * (0@OpB);
```

# Arithmetic Operators - 2

- Add
  - To contain largest result, user must make width of result equal to width of widest input + 1
  - Zero-pad for unsigned or sign extend inputs for signed

```
signed 5 Result;
signed 4 OpA, OpB;

Result = (OpA[3]@OpA) + (OpB[3]@OpB);
```

- / and % produce deep logic
  - Consider pipelined implementations
  - Can use retiming, discussed later

# Key Points

- Don’t rely on operator precedence – use brackets
- Bit manipulation and shifts are cheap in hardware – just wires
- ANSI-C bit-masking is just as efficient as using Handel-C bit manipulation operators
- Extend width of addition and multiplication operands to store the largest possible result
  - Multiplication: width of result equal to sum of widths of inputs
  - Addition: width of result equal to width of widest input + 1

Celoxica™

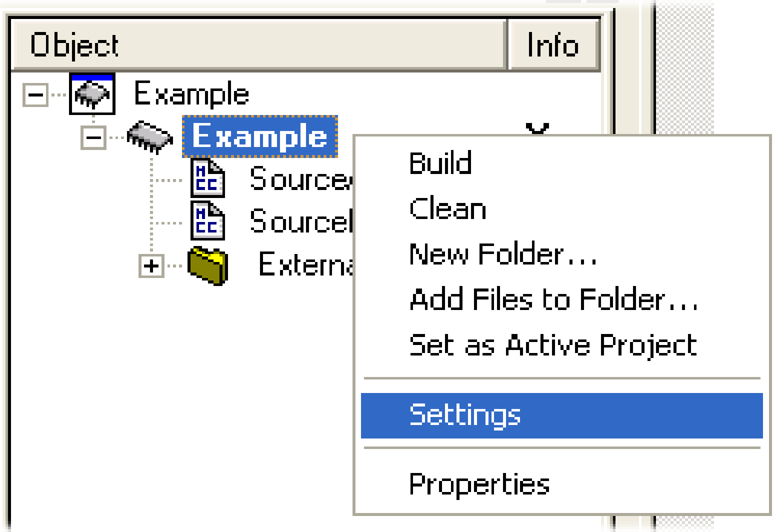
Software-Compiled System Design

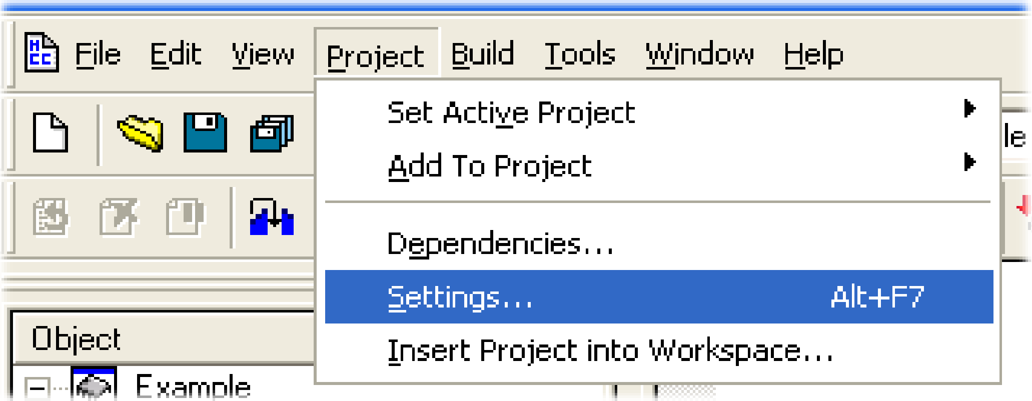
# DK Options

Project Settings, Compiling to hardware

# Project Settings

- **To set options for a project, you can load the** Project Settings **dialog box either by**
  - **Right clicking on the project and selecting** Settings
  - **Selecting** Project → Settings **from the menu bar**

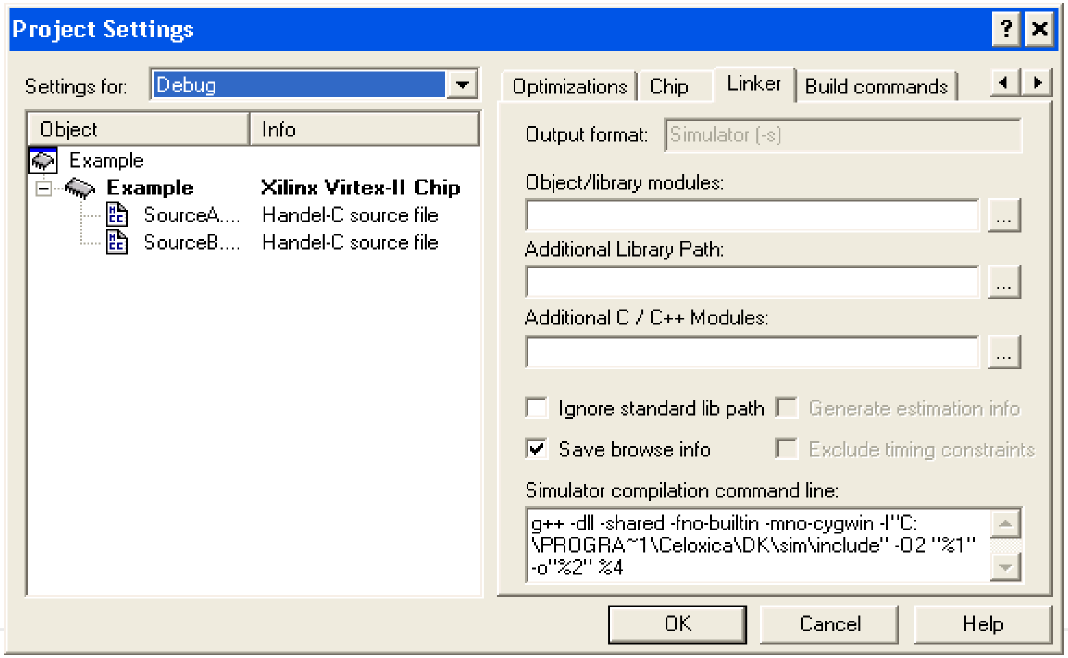




Celoxica
Software-Compiled System Design

# Project Settings

- **For each configuration of a project, you can set various options**
- **The options available on each tab depend on the configuration type**
- **Tabs are** General, Preprocessor, Debug, Synthesis, Optimisations, Chip, Linker **and** Build Commands



Settings for: Debug

| Object | Info |
|---|---|
| Example | |
| Example | Xilinx Virtex-II Chip |
| SourceA.... | Handel-C source file |
| SourceB.... | Handel-C source file |

Output format: Simulator (-s)

Object/library modules:

Additional Library Path:

Additional C / C++ Modules:

☐ Ignore standard lib path ☐ Generate estimation info

☑ Save browse info ☐ Exclude timing constraints

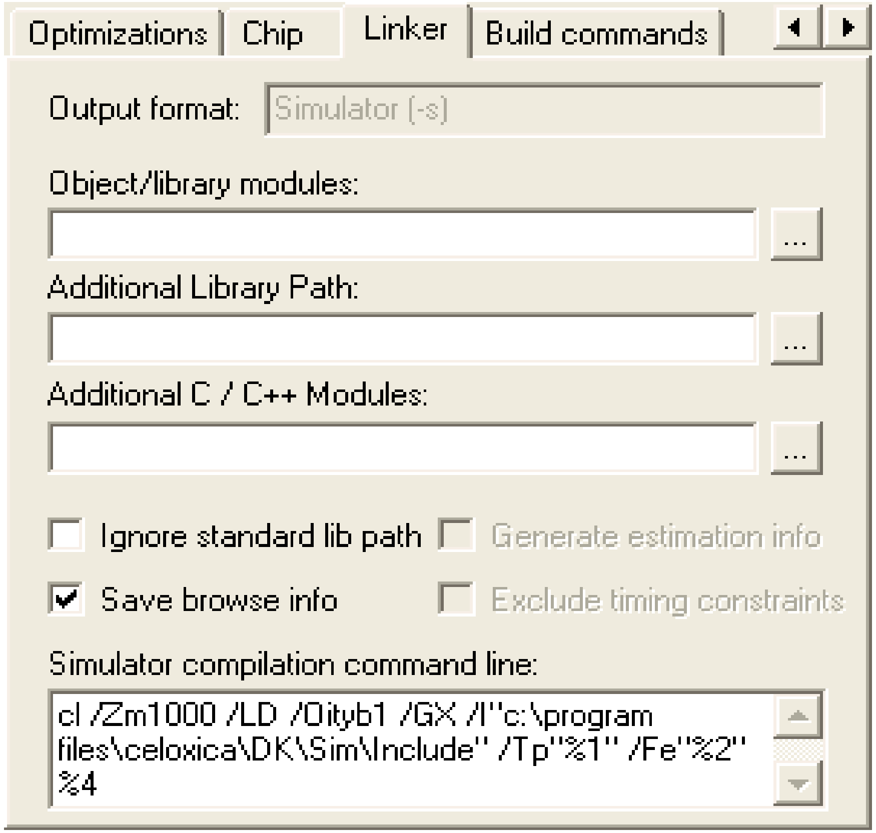
Simulator compilation command line:

```
g++ -dll -shared -fno-builtin -mno-cygwin -I"C:
\PROGRA~1\Celoxica\DK\sim\include" -O2 "%1"
-o"%2" %4
```

Celoxica™
Software-Compiled System Design

# Linker Tab

- Specify the library files (.hcl) that you want to link to
- For simulation, specify any C/C++ library files (.lib) that you want to link to
- If you forget to link to your library files, you will get error messages like this:
  - *example.hcc Ln 59, Col 9-28: Unresolved identifier : 'PalFrameBuffer16Run'*



Output format: Simulator (-s)

Object/library modules:

Additional Library Path:

Additional C / C++ Modules:

☐ Ignore standard lib path
☐ Generate estimation info
☑ Save browse info
☐ Exclude timing constraints

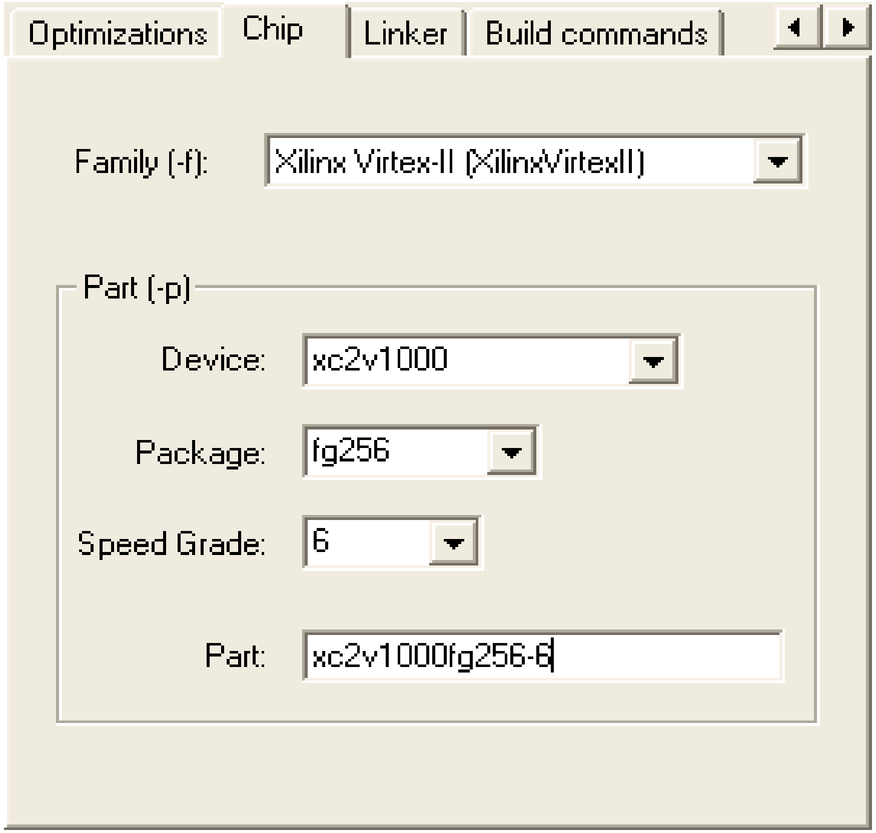
Simulator compilation command line:

```
cl /Zm1000 /LD /Oityb1 /GX /I"c:\program
files\celoxica\DK\Sim\Include" /Tp"%1" /Fe"%2"
%4
```

Celoxica
Software-Compiled System Design

# Chip Tab

- Without a specific chip set
  - Retiming tool will not be available
  - ALU mapping tool will not be available
- Choose the correct speed grade and package



Family (-f): Xilinx Virtex-II (XilinxVirtexII)

Part (-p)

Device: xc2v1000

Package: fg256

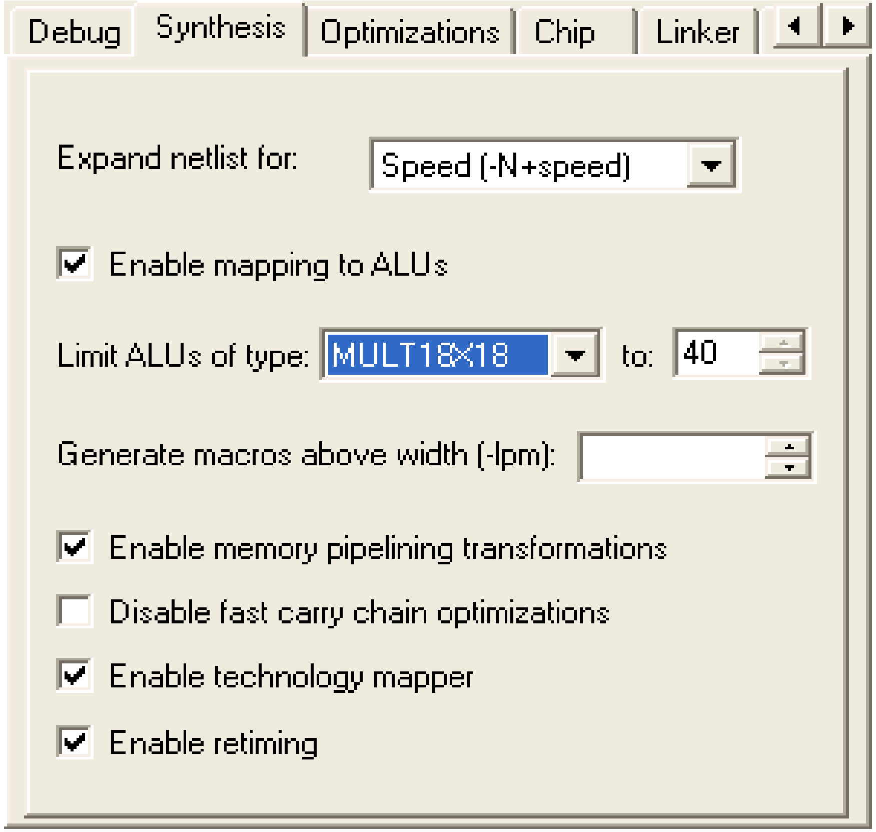
Speed Grade: 6

Part: xc2v1000fg256-6

Celoxica
Software-Compiled System Design

# Synthesis Tab

- **These are general options about how your design will be mapped to hardware**
- **ALU mapping options will be greyed out if**
  - **No chip part selected**
  - **Chip part does not have ALU resources**
- Enable technology mapper **and** enable retiming **will be greyed out if the selected chip is not supported**
- **More detail about these options later**



Expand netlist for: Speed (-N+speed)

☑ Enable mapping to ALUs

Limit ALUs of type: MULT18X18 to: 40

Generate macros above width (-lpm):

☑ Enable memory pipelining transformations

☐ Disable fast carry chain optimizations

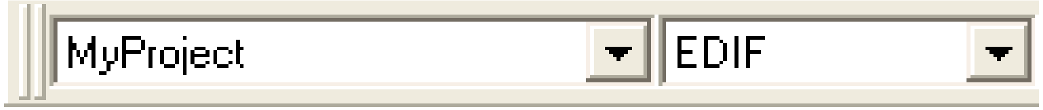
☑ Enable technology mapper

☑ Enable retiming

Celoxica
Software-Compiled System Design

# Compiling to Hardware

- Select a hardware configuration from the drop down box



MyProject

EDIF

- Two routes to HW
  - EDIF (a netlist description format), where DK does all the synthesis
  - RTL VHDL/Verilog where third-party tool does synthesis
- EDIF is recommended route for FPGA
  - Offers tightest integration
  - Fast and easy to use
  - Produces good results
  - Easier to debug for timing issues
  - VHDL/Verilog for use with favoured synthesis tool, simulation tool, combining into larger HDL project

# Place and Route

- What is place and route?
  - Automatic process to place the logic components and determine a path between them through the dedicated routing resources
- We recommend *always* using timing constraints
- Xilinx
  - DK produces an EDIF file (*.edf*) and a timing constraints file (*.ncf*)
  - Xilinx ISE software is used to place and route
  - *edifmake.bat* (supplied with PDK), Project Navigator
- Altera
  - DK produces an EDIF file (*.edf*), a TCL script (*.tcl*) and memory initialisation files (*.mif*)
  - Quartus-II software is used to place and route
  - *sofmake.bat* (supplied with PDK)
  - Remember MIF files!
  - Remember to run TCL script to assign pins

# Creating EDIF Output

- All Handel-C programs must have
  - A main function

    ```
    void main(void)
    ```

    - This is the start point for the design
    - No integer return value in Handel-C
  - A clock specification
    - For example:

    ```
    set clock = external “A12” with {rate = 50};
    ```

    - “A12” is the name of a clock pin on the device
    - 50 is the clock rate in MHz of the input clock
      - This passes on timing constraints to the place and route tools
  - Output interfaces in order to synthesise
    - For example:

    ```
    interface bus_out() OutputBus(OutputPort);
    ```

    - Without it, the design will be optimised away to nothing because your design would sit inside the FGPA and never affect the outside world

Software-Compiled System Design

# Key Points

- Don’t forget to link to your libraries
- Select a specific chip to get the best hardware results
- EDIF is the recommended route to hardware
- *Always* use timing constraints
- Must have output interfaces to produce any hardware output
- Specify a clock rate to apply timing constraints

Celoxica™
Software-Compiled System Design

# Talking to the Outside World

Input and Output interfaces, PDK Introduction

# Interfaces

- 3 basic types
  - Bus for interfacing to external devices via pins
  - Port for when Handel-C is not top-level module in a design
  - User-defined for talking to external code (e.g. VHDL, Verilog, EDIF) with Handel-C as the top-level
- Interface declarations appear with variable declarations before any statements
- All interfaces have the same basic syntax

```
interface InterfaceType(InputsToDK) InstanceName(OutputsFromDK);
```

  - Each interface type has restrictions to the inputs and outputs
- Only signed and unsigned types may be passed over interfaces
- Ports and user-defined interfaces are dealt with in the advanced section of the course

# Output Buses

```
interface bus_out() Name(OutputPort) with {data = PinList};
```

- You can have only one port in `OutputPort`
- Example

  ```
  unsigned 4 a;
  interface bus_out() MyBus(unsigned 4 Out = a) with {data = {“A1”,
     “A2”, “A3”, “A4”}};;
  ```

- An expression must be assigned to the output port
  - You don’t assign to the port itself
  - The output of the interface is the value of this expression at all times
- You can specify pins if you know them
  - Using specification `{data = PinList}`
  - Bit order is left to right, so `“A1”` is bit 3 and `“A4”` is bit 0
  - Place and route tools just choose random pins otherwise

Software-Compiled System Design

# Input Buses

```
interface bus_clock_in(InputPort) Name() with {data = PinList};
```

- You can have only one port in `InputPort`
- Example

  ```
  interface bus_clock_in(unsigned 8 In) MyBus();
  unsigned 8 a;
  a = MyBus.In;
  ```

- Attaches flip-flops that are clocked by the Handel-C clock to the input pins
  - Accessing `Name.InputPort` reads the value from these flip-flops
  - Using a clocked interface avoids many potential timing issues with directly accessing the pins
- Other input bus types
  - `bus_latch_in, bus_in`

# Tri-State Buses

```
interface bus_ts(InputPort) Name(OutputPort, ConditionPort);
```

- A tri-state bus is a bi-directional bus, where the same set of pins is used for both input and output
- You can have only one port in `InputPort` and `OutputPort`
- `ConditionPort` is used to switch between input and output
  - Write mode when `ConditionPort` = 1
  - Make sure that you don’t write to the interface when the condition is 0
- Example

```
unsigned 8 a, b;
static signal unsigned 1 Write = 0;
interface bus_ts(unsigned 8 In) MyBusTS( unsigned 8 Out = a, unsigned 1 OE
  = Write);
Write = 1;      // write a to the bus
b = MyBusTS.In; // read from the bus into b
```

- Other tri-state bus types
  - `bus_ts_clock_in, bus_ts_latch_in`

# Platform Developer's Kit

- PDK has been conceived to accelerate the design process
  - Lets you concentrate on implementing your algorithms rather than spend your time on low-level complexities
- PDK offers three layers of functionality
  - Platform Abstraction Layer (PAL): API for portable projects
  - Platform Support Libraries (PSL): board specific support
  - Data Stream Manager (DSM): integration between processors and FPGAs
- PDK also includes
  - Support for co-simulating Handel-C with VHDL, Verilog, SystemC and MATLAB designs
  - Support for our reconfigurable platforms

# Platform Abstraction Layer (PAL)

- Provides portability between platforms
  - Application Programming Interface (API)
  - A common interface to device-specific resources
- Advantages
  - Designer can concentrate on algorithms
  - Designs are easily portable between platforms
- PAL Kit
  - Templates, examples and tutorials for writing your own PAL implementations
- Supported resources include
  - External memory, video input, video output, audio I/O, Ethernet
- Supported platforms
  - RC100, RC200, RC203, RC300, RC1000, RC2000, Altera’s NIOS Development Board
  - PalSim – a generic simulation platform
    - □ Allows you to synthesise and simulate identical code
    - □ Supports RAM, LEDs, video output, 7-segment displays

# Standard Library

- A PSL which provides useful macros for Handel-C design
- Usage
  - Include *stdlib.hch*
  - Link to *stdlib.hcl*
- Useful macros include

```
adjs(Expression, Width)
```

  - □ Adjusts width of signed expression up or down
  - □ Sign extends MSBs of expression when expanding width

```
log2ceil(Constant)
```

  - □ Calculates $\log_2$ of a number and rounds the result up
  - □ Useful to determine the width of a variable needed to contain a particular value

```
signed ((log2ceil(15))+1) a;  // a has width 5
signed 10 b;
b = adjs(a, 10) * adjs(a,10); // sign-extend a to width 10
```

# Using PDK

- Celoxica can provide training on using the various features of PDK
  - This course will not go into any of the features in great detail
- Some of the exercises in this course will use the PAL API
  - We can concentrate on learning Handel-C and DK features rather than on low-level details
- You can use some of the useful features from the standard library in your code
- The exercises should contain enough information for you to complete the tasks that use PDK
  - Refer to the documentation if you need to know more about how to use any of the features in PDK

# Key Points

- Value in variable assigned to output interface will be output every clock cycle
- Use `bus_clock_in` to avoid timing issues on input pins
- Don’t write to a tri-state bus when you should be reading
- PDK provides abstractions for various types of interfacing
- PAL provides board-level abstraction
- Standard library provides a number of useful macros

Celoxica

Software-Compiled System Design

# Bulk Storage and Further Control Statements

Arrays, RAMs, multi-port RAMs, switch statements, jump statements, conditional operator

# Bulk Storage

- Frequently use collections of the same type of object
- Handel-C provides arrays in the same way as ANSI-C
- Often need efficient random access for hardware design
- Often only one value per clock cycle needed
- FPGAs provide dedicated on-chip RAM resources
  - Block RAM or distributed RAM on Xilinx FPGAs
  - Tri-Matrix RAM or LUT ROM on latest Altera FPGAs
- Handel-C therefore provides RAMs and ROMs

# Arrays of Variables

```
unsigned 8 a[n];
```

- Collection of variables – not a single chunk of memory
- Can be multi-dimensional

  ```
  unsigned 8 cube[3][3][3]; // 27 8-bit registers
  ```

- First index is 0, last is n-1
- The variable you use to index it must have the correct width
  - i.e. $\log_2 n$
- Using a constant as an array index has no logic overhead
  - No different to using individual named variables
- Using a variable as an array index can be inefficient
  - Multiplexers between registers
  - Use a ROM/RAM
    - ROMs/RAMs not inferred from arrays

# RAMs and ROMs

```
ram unsigned 8 a[n];
```

- Can only read from or write to one location in a clock cycle
  - Shared address bus for read and write
- ROM has no write data bus
- Built out of dedicated RAM resources
  - Uses distributed RAM by default (for Xilinx)
  - Use `with {block = “X”}` to use specific RAM resource

  e.g. `ram unsigned 8 a[64] with {block = “M512”}; // uses an M512 block in an Altera Stratix`
  - Use `with {block = 1}` to allow the place and route tools to choose an appropriate resource
- Can have a RAM of structures, as long as structure does not contain channels, semaphores or other RAMs

# Arrays versus RAMs

- Use arrays for
  - Parallel access to all elements, for example in pipelined FIR
- Use RAMs for
  - Random access
  - Large data storage
- Use ROM for
  - Decoding or encoding signals, for example seven segment display
  - Lookup table of coefficients
  - Lookup table for results like sine/cosine/tangent, using input as address
- Can’t read, modify and write to a RAM in the same clock cycle

```
ram unsigned 8 MyRAM[16];
MyRAM[0] += 2;            // won’t work
```

- Experiment with a small design to see logic size/depth tradeoffs

# Multi-Port RAM

```
mpram
{
    wom unsigned 4 Write[32]; // write only port
    rom unsigned 8 Read[16];  // read only port
}
```

- Devices have entirely independent read/write ports
  - Can use both ports on the same clock cycle
  - Read/write configuration depends on device
  - Virtex-II  has two read/write ports on Block RAM, one read/write port and one read port on distributed dual-port RAM
  - Stratix has two read/write ports on M4K and M-RAM, one read and one write port on M512 blocks
- Port dimensions can be different
  - Again, permissible configurations depend on device
  - See DK and vendor manuals on how the data maps between the ports
- Block resources have built-in logic for address and data
  - Use it, even if only to reduce multiplexing/routing on address and data bus
- Example: line buffers in image processing
  - One pixel in and one out every clock cycle

# Multi-Port RAM Example

- Xilinx Block RAM, dual port

```
mpram
{
    wom unsigned 16 Write[256];
    rom unsigned 8 Read[512];
} Buffer with {block = 1};

par
{
    while(1)
    {
        Buffer.Write[AddressIn] = DataIn[0]@DataIn[1];
    }
    while(1)
    {
        DataOut = Buffer.Read[AddressOut];
    }
}
```

# Switch Statements

```
unsigned 5 a, b;
switch(a)
{
    case 0: a++; break;    // break, so no fall-through
    case 1: b=~b;          // no break, so falls through to case 2
    case 2: b++; break;    // b++ happens if a is 1 or 2
    default: delay; break; // delay so that switch takes >=1 cycle
}
```

- All cases must be constant
- All cases evaluated in parallel
- Cases fall through until a break is encountered
  - e.g. if a is equal to 1 in the example above, b = ~b happens in clock cycle 1 and b++ in clock cycle 2
- If no matches are found, then the default statement is executed
  - Consider using a `delay` statement to balance execution time

Software-Compiled System Design

# Jump Statements

- `continue` takes you directly to the next iteration of a loop
- `break` exits enclosing `while`, `switch` or `for`
- `goto` takes you directly to a labelled statement
  - `goto` *still* considered harmful
  - Difficult to maintain
- `continue`, `break` and `goto`
  - Take no clock cycles to evaluate
  - All add control paths that can affect timing

# Conditional Operator

- `condition ? expression1: expression2`
- If `condition` evaluates to 0, then the result is `expression2`, otherwise the result is `expression1`

  e.g. `c = a > b ? a : b; // Get maximum`
- Assignment takes one clock cycle to complete
  - Conditional statement is an expression and can have no side effects
- Both expressions are evaluated in parallel
- Selects which result to read
  - Very subtle difference in meaning to an `if` statement
    - □ An `if` statement selects which statement to execute
  - Be careful of parallel reads from components, e.g. RAM read:

  e.g. `c = a > b ? myRAM[a] : myRAM[b]; // won’t work`

# Key Points

- Avoid random access to arrays
- Use a RAM for random access
- Use an array for parallel access to variables
- Choose the appropriate RAM resource for your data
- Use a multi-port RAM if you need access in two places
- Use both ports of a block RAM if possible
- Use a `delay` statement in the default of a `switch`
- `goto` *still* considered harmful
- Beware using conditional operator with RAM access

# Further Handel-C Types

Structures, signals, channels and semaphores

# Structures

- Allows you to group related information together

```
struct VGADisplay
{
    unsigned 10 CurrentX, CurrentY;
    unsigned 8 Red, Green, Blue;
};
struct VGADisplay MyVGADisplay; // declare an instance
```

- A structure can contain any number of instances of any other types including other structures
- Use the . operator to access elements from the structure

```
Red = MyVGADisplay.Red;
```

# Initialising Data

- FPGAs have the ability to give variables and memories a value at start-up
- You can specify this in Handel-C using initialisers
- `static unsigned 8 a = 8;`
- No overhead in logic
- Static and global variables only
  - *test.hcc Ln 6, Col 17-23: Illegal initializer*
- Can initialise structures, arrays, ROMs and RAMs
- Value can be a compile-time evaluated expression
- If a variable is not initialised you *cannot* assume that it will be zero on start-up
- If no initialiser is specified, a static variable will be initialised to 0

# Initialising Data - 2

```
unsigned 8 x;

static struct
{
    unsigned 8 a;
    unsigned 8 b[2][4];
} MyStruct = { 3, {{0,1,2,3}, {4,5,6,7}}};

x = MyStruct.a; // x = 3
x = MyStruct.b[1][2]; // x = 6
```

# Signals

```
signal unsigned 8 a;
```

- A signal behaves like a wire
  - Takes the value assigned for the current clock cycle only
- Default value is undefined
  - May get different behaviour in simulation compared to hardware
- Declare as static to give a default value
  - Will take this value if not assigned to in a clock cycle
  - A static signal without an explicit initialiser will default to 0

```
static signal unsigned 8 a = 8;
```

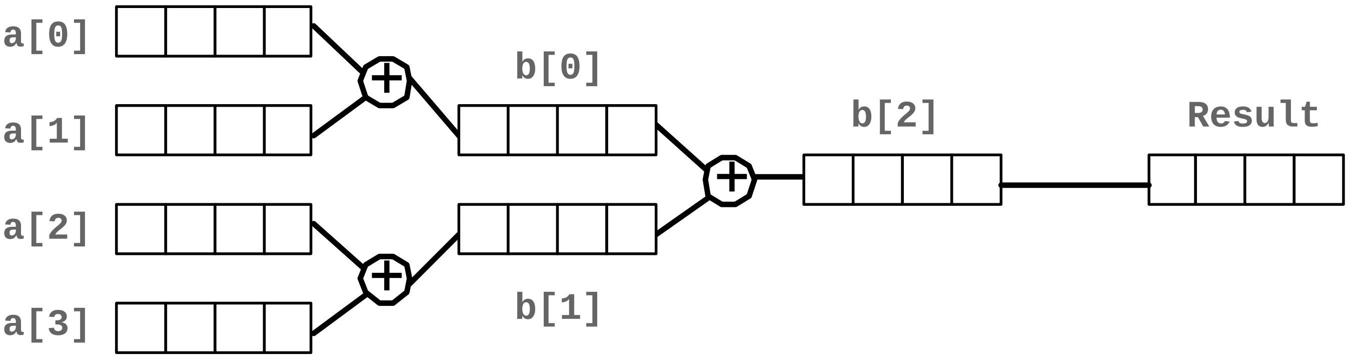
- Assignment evaluated before read in a clock cycle
- You can use the `signal` qualifier on an array declaration to create an array of signals

# Signals - Example

```
signal unsigned 8 SignalA;
static signal unsigned 8 SignalB = 5;
static unsigned 8 VarX = 1, VarY = 2;
par
{
    SignalA = VarX * 2;    // Assign a value to SignalA
    VarX = SignalA;        // Use the value from SignalA
    VarY = SignalA + 1;    // Use it again here
}                          // VarX = 2, VarY = 3
VarX = SignalB;            // Use the default value of SignalB
                           // VarX = 5
VarY = SignalA;            // Won’t work: SignalA is not assigned to
                           // VarY is UNDEFINED
```

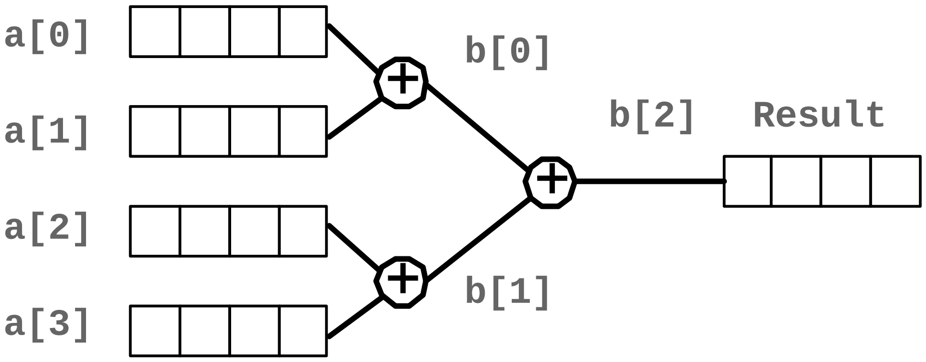
# Register Implementation

```
unsigned 4 a[4];
unsigned 4 b[3];
unsigned 4 Result;
while(1)
{
   par
   {
        b[0] = a[0] + a[1];
        b[1] = a[2] + a[3];
        b[2] = b[0]+b[1];
        Result = b[2];
   }
}
```



# Signal Implementation

```
unsigned 4 a[4];
signal unsigned 4 b[3];
unsigned 4 Result;
while(1)
{
   par
   {
        b[0] = a[0] + a[1];
        b[1] = a[2] + a[3];
        b[2] = b[0]+b[1];
        Result = b[2];
   }
}
```



Celoxica
Software-Compiled System Design

# Signals – Combinational Cycles

- Cannot have circular dependencies

```
signal unsigned 8 a;
signal unsigned 8 b;
par
{
    a = b;
    b = a;
}
```

```
par
{
    if (a)
        b = 1;
    else
        delay;

    if (b)
        a = 1;
    else
        delay;
}
```

- Cannot assign signal to itself

```
a++;
```

- Error message: *Design contains an unbreakable combinational cycle*

# Synchronisation and Communication

- Many programs have independent processes running in parallel
- They often need to communicate and synchronise with each other
- They often need to share resources such as functions and RAMs
- This type of code can be complicated and convoluted to write
- Handel-C has two features to make these problems easier to solve
  - Channels to communicate between processes
  - Semaphores to control access to critical sections

# Channels

- Blocking communication between two sections of code
  - Both sides block until the other is ready

```
chan unsigned 8 ChannelA; // ChannelA is an 8 bit channel
unsigned 8 VarX;
ChannelA ! 3;             // send 3 down ChannelA
ChannelA ? VarX;          // read from ChannelA into b
```

- Channel communication takes a clock cycle
- Can only read and write once to a channel in parallel
  - Simulation can detect parallel channel accesses
- Width of channel can be inferred like a normal variable
- Can use zero width for synchronisation only
- Channels between clock domains discussed later
- chanin and chanout for debug
  - Default output to Debug tab of Output window
  - Can attach to text files

# Channel Example

- Two processes running in parallel

```
    chan unsigned 4 MyChan; // Channel between the two processes
```

```
static unsigned 4 Val = 1;
while(1)
{
    Val = Val[2:0]@Val[3];
    MyChan ! Val; // Send
    delay;
}
```

```
unsigned 4 Count;
while(1)
{   // wait 0 or more cycles
    while (Count != 1)
    {
        Count--;
    }
    MyChan ? Count; // Receive
    delay;
}
```

- Always happen on the same cycle

# Prialt

```
chan unsigned 8 ChannelA, ChannelB;
unsigned 8 VarX;
prialt
{
    case ChannelA ! 3: DoSomething(); break;
    case ChannelB ? VarX: DoSomethingElse(); break;
    default: Otherwise(); break;
}
```

- Similar syntax to `switch` statement
- Selects a channel event
  - First ready is read from or written to
- Priority order from top to bottom
  - More cases gives greater logic depth
- Blocks unless there is a default
- The channel communication takes a clock cycle
- No fall-through between cases permitted – must use break for every case
- Cannot have same channel in two cases

# Semaphores

```
sema MySema;                // MySema is a semaphore
while(trysema(MySema) == 0) // Wait until I can take it
    delay;
CriticalSection();          // Execute my critical section
releasesema(MySema);        // Release the semaphore
```

- sema is a type
  - There is no width or sign to a semaphore
- `trysema()` tests and takes the semaphore if it can
  - It is an expression, so therefore takes no clock cycles
  - Returns 1 if successful
  - Implement a loop until it returns 1 to block
- `releasesema()` releases the semaphore
  - Takes one clock cycle: semaphore is free to be taken on following clock cycle
  - Can be called in parallel with the last statement in the critical section
  - Can be called in same clock cycle as taking it

# Semaphores - 2

- Use one semaphore to protect each resource
  - Either declare it globally or pass it as an argument to each process sharing the resource
- Undefined priority – program appropriately
  - Don’t rely on the order
    - You could get different behaviour between simulation and hardware
    - Behaviour could change due to seemingly unrelated changes in the code
  - Be careful that processes are not starved
    - One process can release the semaphore and then immediately take it again if it has a higher priority than other waiting resources

# Key Points

- Initialise data to save logic and clock cycles
- A signal behaves like a wire
- Channels offer blocking communications
- Semaphores have undefined priority – program appropriately

# Advanced Types

Pointers, type-casting, arrays of other objects, enumeration constants and bit fields

# Pointers

- Declaration: `unsigned 8 *a;`
  - a is a pointer to a variable of type `unsigned 8`
- Address of operator: `a = &b; // b is an unsigned 8`
  - a now points at b
- Dereference: `*a = 3;`
  - Set the variable at which a is pointing to 3 (i.e. `b = 3`)
- Useful for data abstraction
- Mapped to multiplexers
  - Apart from compile-time evaluated pointers
  - A pointer that points only at the contents of a single RAM will have no multiplexer - just like using an explicit address
  - Incrementing a pointer that points to an array is the same as using a variable as an index to the array, which can be inefficient

# Pointer Examples

```
unsigned 8 *a; // a is a pointer to an unsigned 8
unsigned 8 b;
unsigned 8 *c;
unsigned 8 d[8];
a = &b;        // set a to the address of b
*a = 3;        // b = 3
c = a;         // pointer c = pointer a
*c = 4;        // b = 4
c = &(d[0]);   // set c to the address of the first element of d
*c = 1;        // d[0] = 1
c++;           // increment c
*c = 2;        // d[1] = 2
a = &(d[0]);   // set a to the address of the first element of d
a[2] = 3;      // d[2] = 3
```

# Pointer Examples - 2

```
ram char RamA[16];
ram char RamB[16];

static char *Ptr = &RamA[0]; // Initialise the pointer

*Ptr = 1;                    // RamA[0] = 1
Ptr++;                       // Ptr now points at RamA[1]
*Ptr = 2;                    // RamA[1] = 2
Ptr = &RamB[0] + 5;          // Ptr now points at RamB[5]
*Ptr = 3;                    // RamB[5] = 3
Ptr[5] = 4;                  // RamB[10] = 4
Ptr = RamA;                  // Ptr now points at RamA[0]
*(Ptr+10) = 5;               // RamA[10] = 5
```

# Pointer Notes

- Cannot point to a bit-selection

```
unsigned 8 *a;              // a is a pointer to an unsigned 8
unsigned 16 b;
a = &(b[11:4]);             // Won't work
```

- You can initialise a pointer in the same way as a variable

```
unsigned 8 b;
static unsigned 8 *a = &b; // Initialise a to point at b
```

- A pointer that is only ever assigned to point to one variable will have no logic overhead
- Unlike ANSI-C, Handel-C pointers don't have values that you can look at
  - You can see what something is pointing at in simulation, but you don't see an actual address

# Pointer Notes - 2

- Take care not to dereference un-initialised pointers

```
unsigned 8 *a;
*a = 3;          // Error: Write using un-initialised pointer
```

  - Simulation error message: *Write using un-initialised pointer*
  - Undefined behaviour in hardware

- Take care not to increment pointer off the end of an array or RAM

```
unsigned 8 *a;
unsigned 8 b[4];
a = b+3;         // A points to the last element in b
a++;             // Increment A off the end of b
*a = 3;          // Won't work: Attempt to dereference A
```

  - Read of dereferenced pointer will get undefined value
  - Attempt to write to dereferenced pointer in simulation may corrupt simulation

# Void Pointers

- Void pointers are very useful for data abstraction
  - Libraries can manipulate data without caring what the actual type of data is

```
void *a;
unsigned 8 b;
unsigned 16 c;
a = (void *) (&b);
*((unsigned 8 *)a) = 3;
```

- Void pointer can only point to one type in Handel-C

```
a = (void *) &c; // not allowed
```

  - *Type 'unsigned int 8 (*)' does not match type 'unsigned int 16 (*)'*

# User Type Names

- Like C, Handel-C allows you to define your own type names
- It doesn't create new types
  - It just gives a name to an existing type
  - You can still use a variable of the base type
- Makes the purpose of a structure clearer
- For example

```
typedef struct
{
    ram unsigned 8 Buffer[16];
    unsigned 4 Head;
    unsigned 4 Tail;
} CircularBuffer;
```

  - Referring to `CircularBuffer` makes the functionality clearer than using some complicated structure definition everywhere
- Useful shorthand for complicated declarations

# Type Casting

- There is no automatic type conversion in Handel-C
  - This means that you cannot assign a variable directly into another which is not of the same type
  - You have to explicitly tell the compiler that you know what you are doing
- You can cast between `signed` and `unsigned` for variables of the same width
- For constants, you can cast to the appropriate width
  - e.g.

```
unsigned 16 a;
unsigned 4 b;
a = ((unsigned 8) 0xf) @ b @ ((unsigned 4) 4);
```

# Arrays of Other Objects

- You can have an array of any of the following in Handel-C
  - Channels

    ```
    chan unsigned a[16]; a[15] ! 3;
    ```

  - Pointers

    ```
    unsigned *a[16]; a[15] = &b; *(a[15]) = 3;
    ```

  - Structures

    ```
    struct {int x; int y;} a[16]; a[15].x = 3;
    ```

  - Semaphores

    ```
    sema a[16];
    if (trysema(a[0]))
    {
      delay;
      releasesema(a[0]);
    }
    ```

# Arrays of RAMs

```
ram unsigned 8 RAMArray[NumRAMs][NumEntries];
```

- Declares an array of `NumRAMs` RAMs, each with `NumEntries` entries
- If you have several dimensions, the right-hand dimension is the number of elements in each RAM
  - e.g. `ram int 8 Array[2][3][4][8]; // 2*3*4=24 RAMs, each with 8-entries`
- Avoid using random access on the array dimensions
- Example: line buffers for image processing
  - 5 by 5 window required into VGA (640x480) image
  - 4 line identical buffers required

  ```
  ram unsigned 24 LineBuffers[4][640-5];
  ```
- If you want a multi-dimensional RAM
  - Use concatenation of coordinates in 1D RAM
  - Example: 2-bit, 32x16 state storage

  ```
  ram unsigned 2 States[32*16];
  unsigned 5 X; unsigned 4 Y;
  State = States[Y@X];
  ```

# Enumeration Constants

- Instead of using `#defines` for different values, for example days of the week, you can use enum

```
enum weekdays {MON, TUES, WED, THURS, FRI};
enum weekdays MyEnum; // instance
```

- By default, the first name in the enum has value 0 and the subsequent names increment in steps of 1 from this value
- You can set explicit values on elements

```
typedef enum {NO= 0, YES = 1} yesno;
```

- Subsequent names without values will increment from the previous value

```
typedef enum {NO= 0, YES = 2, MAYBE } yesno; // MAYBE = 3
```

- You can give the enum names a width and sign

```
typedef enum {MON = (unsigned 3) 0, TUES, WED, THURS, FRI} weekdays;
```

# Bit Fields

```
struct
{
    unsigned statusA : 1;
    unsigned statusB : 2;
    signed statusC : 3;
} StatusFlags;
```

- Useful for maintaining C syntax
- Behave exactly as in C
- Bit fields have no address, so you cannot point to them

# Key Points

- A pointer that is only ever assigned to point to one variable will have no logic overhead
- Don’t dereference un-initialised pointers
- Don’t increment pointer off the end of an array or RAM
- A `void` pointer can only point at one type
- Use `typedef` to make the purpose of data structures clear
- Avoid random access on array dimension of array of RAMs
- An array of RAMs is not a multi-dimensional RAM
- Bit fields are useful for maintaining C syntax

Celoxica™
Software-Compiled System Design

# Functions and Macros

Functions in their various flavours, macro procedures, macro expressions

# Functions and Macros

- Functions and macros encapsulate the implementation of a task into a named block
- Abstraction
  - A well-encapsulated function hides how something is implemented
  - PAL provides an excellent example of this
- Readability
  - The meaning of the code is clearer when discrete, well-named functions are used
- Modularity
  - Breaks the code into discrete components that can be maintained and tested without having to either change or understand the whole design
- Re-use
  - Within the same project and in other projects
  - Again, PAL provides a good example
- Sharing
  - Code that is encapsulated in a function or macro can be easily shared between multiple processes

# Functions

```
int MyFunction(int a)
{
    static unsigned b = 1;
    b++;
    return a+b;
}
```

- Shared functionality by default
  - Only one block of hardware created
  - Multiplexes inputs – potential logic overhead
- Static and automatic variables
  - Automatic variables cannot be initialised
  - Only static variables keep their values between calls to the function
  - Initialisation occurs only once at start-up
  - Relying on automatic variables retaining their values between calls is unsafe

# Function Flavours

- Inline functions

  `inline ReturnType Name(ParameterList)`

  - Creates a new copy of the function for each call
  - Very similar to a macro procedure

- Arrays of functions

  `ReturnType Name[n](ParameterList)`

  - Creates n copies of a function
  - Each function in the array is shared
  - You can share each of the copies between a number of processes

# Function Returns

- Assignment of return value takes a clock cycle

```
unsigned MyFunction(unsigned A)
{
    A++;                      // clock cycle 1
    return A;                 // clock cycle 2
}

unsigned 8 a, b;
b = MyFunction(a);              // assignment on clock cycle 2
```

  - Consider passing a reference to the variable to save a clock cycle

- Return statement cannot be in par block

```
par
{
    A++;
    return A;                 // not allowed
}
```

Software-Compiled System Design

# Function Examples

- Two multipliers

```
unsigned 8 a,b,c,d,e,f;
a = b*c;
d = e*f;
```

- One multiplier

```
unsigned Mult(unsigned A, unsigned B)
{
    return A*B;
}
unsigned 8 a,b,c,d,e,f;
a = Mult(b,c);
d = Mult(e,f);
```

# Function Examples - 2

- Not allowed (with definition of Mult function from previous slide)

```
par
{
  a = Mult(b,c); // won’t work
  d = Mult(e,f); // won’t work
}
```

- Two multipliers

```
inline unsigned Mult(unsigned A, unsigned B)
{
    return A*B;
}
unsigned 8 a,b,c,d,e,f;
a = Mult(b,c);
d = Mult(e,f);
```

# Function Examples - 3

- Two shared multipliers

```
unsigned Mult[2](unsigned A, unsigned B)
{
    return A*B;
}

unsigned 8 a,b,c,d,e,f,w,x,y,z;

par
{
    a = Mult[0](b,c);
    d = Mult[1](e,f);
}
par
{
    a = Mult[1](w,x);
    d = Mult[0](y,z);
}
```

# Function Parameters

- Strongly typed
  - Makes sure that you pass the expected type of variable
- Call by value
  - Reads original variable in first clock cycle
  - Operates on a copy in subsequent clocks
  - Assignment to arguments does not affect original values
  - Can introduce combinational delays on paths that are never taken
  - You cannot pass a semaphore or a channel by value
- Call by reference
  - No copying overhead (unless you change the pointer)
  - Changes to the variable being pointed to affects original variable
  - Note: changes to the pointer value itself are NOT made to the original pointer – this is passed by value
  - To change a pointer, you need to pass a pointer to a pointer

# Call by Value Example

```
unsigned Change(unsigned x)
{
    x++;          // increment copy of argument a
    return x;     // return the new value of the copy
}

static unsigned 8 a = 0, b;
b = Change(a);    // pass the value of a
```

- End result: a = 0, b = 1

# Call by Reference Example

```
unsigned Change(unsigned *x)
{
    (*x)++;      // increment the target of x
    return *x;   // return the value of target of x
}

static unsigned 8 a = 0, b;
b = Change(&a); // pass a pointer to a
```

- End result: a = 1, b = 1

# Call by Reference Example 2

```
unsigned Change(unsigned *x)
{
    x++;                         // increment a copy of the pointer
    (*x)++;                      // increment the target of x
    return *x;                   // return the value of target of x
}

static unsigned 8 a[2] = {1,2};
static unsigned *Ptr = a;        // Ptr points to a[0]
unsigned b;

b = Change(Ptr);
(*Ptr)++;                        // Increment the target of Ptr
```

- End result: a[0] = 2, a[1] = 3, b = 3

# Function Prototypes

- You can declare a function before you define its implementation
- Allows you to group the declarations together
  - Don’t have to worry about the order of use
- You can put all of your declarations in a header file
  - The implementation will be provided by a source file or a library

```
void FunctionName(Parameter-list); // prototype
void FunctionName(Parameter-list)  // definition
{
   Statements
}
```

# C Pre-Processor Language

- Handel-C includes the C pre-processor language
- Allows you to
  - Include header files using #include
  - Define constants, names and macros using #define
  - Control what gets built using #if, #ifdef, #ifndef, #else, #endif
- Useful tips
  - Use #if 0 to exclude sections of code from compilation
    - □ Better than using comments as a #if can include any other code, including other #ifs and comments
  - Use pre-processor guards to prevent multiple declarations of header file contents from multiple inclusions

```
#ifndef HEADER_GUARD
#define HEADER_GUARD
// Header body
#endif
```

# Macros

- Macro procedures are groups of statements somewhat like functions
- Macro expressions are shorthand complex expressions
- Factors common to both types of macros
  - No typing on arguments
  - Parameterisable
  - Can recurse at compile-time
    - Beware of infinite recursion!
  - Create one piece of logic for each call
- Additional functionality to C pre-processor language
  - Can be prototyped like functions
  - Clearer syntax

# Macro Procedures

- Alternative to functions

```
macro proc Change(A)
{
    A++; // increments A
}
```

- Produces a new copy of the logic for each call
- No typing on parameters
  - Very versatile
- No return value
  - Pass variable to hold the result as an argument
- Macro procedures operate on the original variables
- No overhead of copying data
  - Just like calling by reference
  - Still good practice to pass pointers to make meaning clear

# Macro Procedures - 2

- Can pass in expressions as arguments
- Can pass in constants as arguments and use for
  - Width
  - Array/RAM dimensions
  - Example:

```
macro proc Change(A) // No typing on arguments, no return type
{
    (*A)++;          // Increment target of A
}
unsigned 8 a;
Change(&a);          // Pass a pointer to make the meaning clear
```

# Macro Expressions

- **Non-Parameterised**

  ```
  macro expr Name = Expression;
  ```

  - Can be used like `#define`
  - e.g. `macro expr Size = 8;`
  - Can be used for more complicated expressions
  - e.g. `macro expr Sum = a+b;`

- **Parameterised**

  ```
  macro expr Name(Arguments) = expression;
  ```

  - Can be used to express complicated expressions on the input arguments
  - No type checking on the arguments

    ```
    macro expr Add(a, b) = a+b;
    unsigned 8 x, y, z;
    z = Add(x,y);
    ```

# Macro Expressions - 2

- Either compile-time or run-time evaluated

```
macro expr Add(a,b) = a+b;
unsigned (Add(3,4)) x, y, z;
x = Add(y,z);
```

- Compile-time evaluated expressions can be used for
  - Constants
  - Word widths
  - Array/RAM/ROM dimensions
- Run-time example: sign extension

```
macro expr SignExtend(X, Width) = ((signed (Width –
  width(X))((X)[width(X) - 1] ? ~0 : 0)) @ (signed) X;
```

# Key Points

- Functions are shared by default
- Inline functions create a new copy of the logic for each call
- Arrays of shared functions allow sharing of a number of copies
- Return statement takes a clock cycle
- Functions operate on a copy of the arguments
- Use call by reference to operate on the original variables
- Macros create a new copy of the logic for each call
- No return value from a macro procedure
- Macro procedures operate on the original variables
- Macro expressions can be compile-time or run-time evaluated
- Macro expressions can be used to calculate widths

Celoxica™

Software-Compiled System Design

# Flexible Code

Width inference and calculation, code replication, recursive macros

# Width Inference

- Putting widths on every variable can be tedious
- Often necessary to write library functions that can deal with any width
- You can leave the width undefined
  - Either omit a number or use `undefined`
  - e.g. `unsigned undefined a;`
- Compiler will infer width from
  - Variables whose value you assign into this variable
  - Variables into which you assign the value of this variable
  - e.g.

```
int a, c;
int 6 b;
int 8 d;
a = b; // a’s width will be inferred to be 6
b = c; // c’s width will be inferred to be 6
a = d; // Error: a’s width is now 6, so doesn’t match 8
```

# Width Calculation

- `width` operator gives you the width of a variable

```
int 4 a; int (width(a)) b;
```

  - Useful for generic code that depends on width of inputs and outputs
- Example
  - Multiplication – width of maximum multiplication result = width of a + width of b

```
unsigned 7 a;
unsigned 8 b;
unsigned (width(a) + width(b)) result;
result = (0@a) * (0@b);
```

- Example
  - Work out the width of a variable from the maximum value you want to store
  - `unsigned (log2ceil(640)) a; // standard library macro in PDK`

# Asserts

`assert(BooleanExpr, ConstantExpr, ErrorMessage)`

  - e.g. `assert(width(a) == 4, 0, “width of a is not 4, it is %d”, width(a));`
  - If `width(a)` is 4, then the assert will be replaced with `0;`
  - Asserts must appear after declarations within your code
- Compile-time evaluated expression that prints strings to the output window during compilation
- You can use the constant expression result of the assert
  - e.g. `unsigned (assert (width(a) < 4, width(a), “Wrong width”)) b;`
- Useful compile-time tool for library checking
  - Make sure that the input arguments satisfy certain conditions
- Useful for debugging compilation problems
  - For example width mismatches
  - Insert asserts to see what values are actually being used where

# Code Replication

- It is often necessary to perform the same or similar operations many times
  - This can be tedious to write
  - It needs to modified if you add or remove any copies of the code
- Handel-C allows both parallel and sequential replication
  - Replicates the contents of the block at compile time
- Very useful for building flexible code
  - e.g. Pipelines whose depth or bit width depends on inputs and user parameters
- Parallel replication

```
par (index_Base; index_Limit; index_Count)
{
    Body
}
```

- Sequential replication

```
seq (index_Base; index_Limit; index_Count)
```

# Code Replication

- Example

```
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];
}
```

- Example

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

- Expands to:

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

# Code Replication

- Loop counter is not a variable
  - No variables allowed in index_Base, index_Limit or index_Count
  - Does have a width, inferred from its use
- Nesting replicators
  - Just like nested for loops

```
static unsigned 11 Tree[4][8];

while(1)
{
    par(i=0;i<3;i++)
    {
        par (j=0;j<7;j+=2)
        {
            Tree[i+1][j/2] = Tree[i][j] + Tree[i][j+1];
        }
    }
}
```

Celoxica
Software-Compiled System Design

# Function Implementation

- Inline function and #define

```
#define SIZE 8
#define TYPE signed

inline void Shift(TYPE (*A)[])
{
    par (i = 0; i<(SIZE-1); i++)
    {
        (*A)[i] = (*A)[i+1];
    }
}
```

- Cannot use an argument as a constant in replicated `par`, as a width or for a bit selection
- Cannot deal with arbitrary types

# Macro Procedure Implementation

- Macro procedure

```
macro proc ShiftB(A, Size)
{
    par (i = 0; i<(Size-1); i++)
    {
        A[i] = A[i+1];
    }
}
```

- Can use argument as a constant in `par` replicator
- No type checking, so can deal with any type
  - Could be an array of `signed` or `unsigned`

# Compile-Time Evaluated Expressions

- `select(Condition, Expression1, Expression2)`
  - Used instead of `? :` for compile-time evaluation
  - `Condition` must be a compile-time constant
  - If `Condition` evaluates to 1, then the whole statement is replaced by `Expression1`
  - Otherwise, `Expression2` is used
- Example

```
macro expr MyExpr(a, b) = select(a < b, a++, a--;)
```

  - If `a < b`, then `a++;` will be built, otherwise `a––;` will be built
- Can be used to calculate end condition for compile-time recursion
- No logic overhead

# Compile-Time Evaluated Expressions - 2

```
ifselect (Condition)
    Statement1
[else
    Statement2]
```

- Used instead of `if` for compile-time evaluation
- `Condition` must be compile-time constant
- If `Condition` is true, then `Statement1` will be built
- Otherwise `Statement2` will be built
- Can be used to calculate end condition for compile-time recursion
- No logic overhead

# Compile-Time Evaluated Expressions - 3

- Pipeline example

```
unsigned init;
unsigned q[7];
unsigned 8 out;

par (r = 0; r < 8; r++)
{
    ifselect(r == 0)
        q[r] = init;
    else ifselect(r == 7)
        out = q[r-1];
    else
        q[r] = q[r-1];
}
```

- Expands to:

```
par
{
    q[0] = init;
    q[1] = q[0];
    q[2] = q[1];
    q[3] = q[2];
    q[4] = q[3];
    q[5] = q[4];
    q[6] = q[5];
    out = q[6];
}
```

# Compile-Time Recursive Macros

- Both macro procedures and expressions can be compile-time recursive
  - Run-time recursion is not permitted
- Make sure that infinite recursion is not possible
  - DK cannot determine if the macro just recurses a lot or infinitely
- Example of bit-reversal, used in radix-2 FFT

```
macro expr BitReverseAux(Var, Current) =
        select(Current == width(Var)-1, Var[Current],
        Var[Current]@BitReverseAux(Var, Current+1));

macro expr BitReverse(Variable) = BitReverseAux(Variable, 0);

static unsigned 8 a = 0b10101111, b;
b = BitReverse(a); // b = 0b11110101
```

- Note that this will recurse infinitely if the width of Variable is 0

# Key Points

- You can let DK infer the width of variables
- You can use expressions to calculate widths
- `log2ceil` macro in standard library can be used to calculate width needed
- `seq` and `par` replication allow you to create flexible code very easily
- Compile-time evaluated expressions have no logic overhead
- Take care to avoid infinite recursion in macros

Software-Compiled System Design

# Using Handel-C with DK

Advanced Course

Ashley Sutcliffe

# Goals for Advanced Course

- Understand how Handel-C relates to hardware output
- Know how to debug logic area and delay
- Know how to write efficient Handel-C code
- Understand and be able to program for retiming
- Know how to take advantage of DK optimisations
- Know how to do advanced I/O in simulation

Celoxica™

Software-Compiled System Design

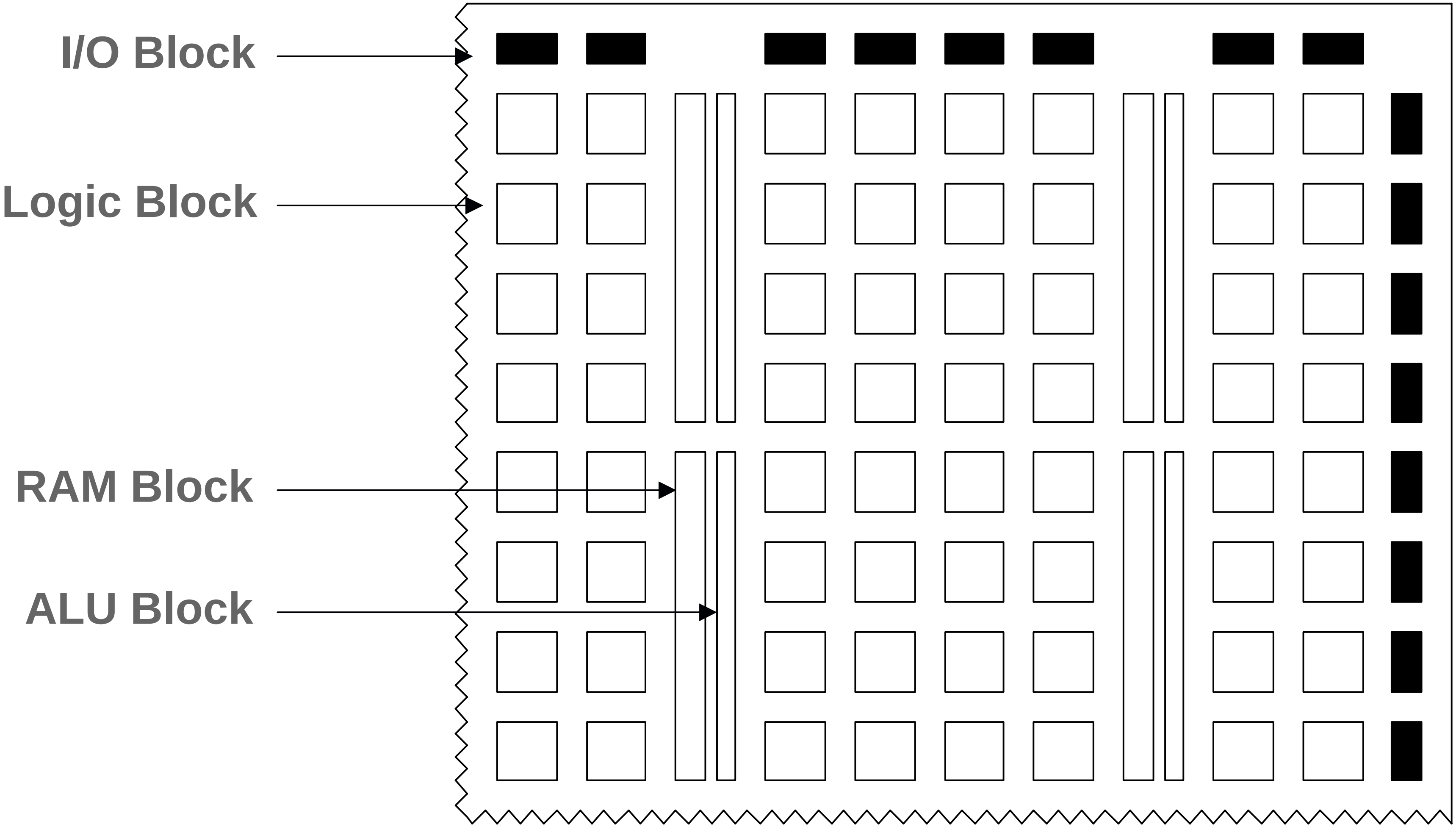
# Inside an FPGA

A look inside a simplified FPGA

# Inside an FPGA

- An FPGA consists of programmable logic blocks and routing
- In most FPGAs, the basic logic block contains a Lookup Table (LUT) and a flip-flop (FF)
  - Logic is implemented in these LUTs, which can be chained together to make complicated logic functions
  - There may well be other logic, such as multiplexers and carry logic
- There are fast, configurable routing resources to connect blocks together
- There are also dedicated low-skew clock nets
- Modern FPGAs also have other important resources
  - Dedicated memory resources
  - Dedicated arithmetic blocks (ALUs)
  - Processor cores
  - Fast serial I/O

# Inside an FPGA - 2



# Inside a Xilinx CLB

- A Xilinx Virtex II Configurable Logic Block (CLB) contains 4 Slices, which each contain 2 LUT/FF pairs
- An Altera Stratix Logic Array Block (LAB) contains 10 Logic Elements (LE), which each contain 1 LUT/FF pair
- Not all devices have the same arrangement

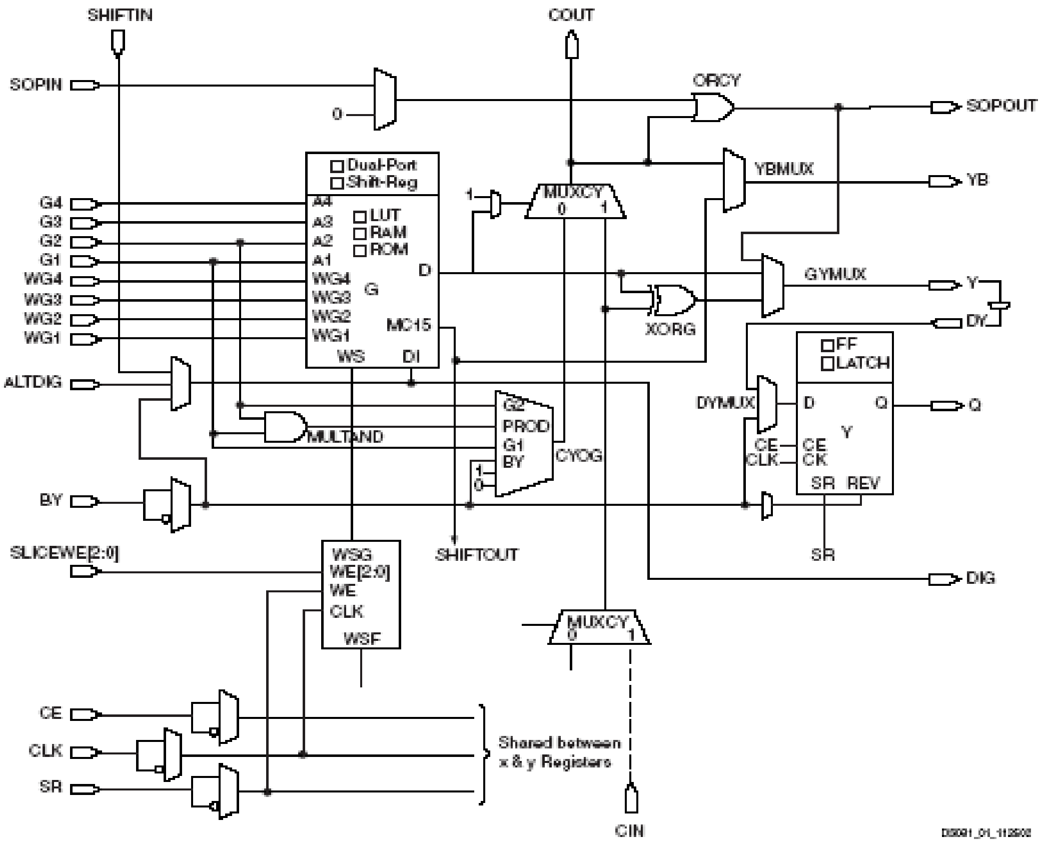


*Figure 16:* Virtex-II Slice (Top Half)

Source: Xilinx Virtex II Data Book

# Place and Route

- Placement is the process of putting individual components in specific locations in the device
- Routing is the process of connecting all the components together in the device
  - Such that the routing delay between components meets timing constraints
- *Always* check timing report
- Look for changes in mapping report
  - Is something being optimised away?
- Learn how to set effort levels

# How Big Is an FPGA?

- Many people talk about “gates” or “ASIC equivalent gates”
- The only absolute figures you can use are
  - The number of LUT/FF pairs
    - CLBs/Slices in Xilinx
    - LABs/LEs in Altera
  - The number of RAM bits
  - The number of hard arithmetic units
- The device data book is your friend
  - This tells you exactly what type of resources are available in your device and how many there are of each type
  - You can download them from the manufacturer’s website
  - The paper copies go out of date: use the latest information from the website

# Will My Design Fit?

- A common question is “How many NAND gates will fit in my FPGA?”
  - The answer is “it depends what they are doing”
  - There is no direct correspondence between NAND gates and LUTs
  - Do not rely on this number to determine if your design will fit
  - The NAND gate count is useful for comparing builds for optimisation purposes
  - The LUT, Flip-Flop and memory counts are the significant numbers
- Just because it fits in terms of LUT and flip-flop count, it may not place or route
  - There may not be enough routing resources
  - e.g. if you are using 99% of FFs and LUTs, the tools might not be able to route your design
- Conclusion: place and route your design before making any firm conclusions about its device utilisation

# How Fast Will My Design Run?

- LUTs, MUXs and other resources are connected together to make up complex logic statements
- Each component has a propagation delay
  - The time taken for the result to stabilise on its outputs after the inputs change
- The longer the longest path through any components between two flip-flops (or other clocked storage elements), the longer the clock period for the design

# How Fast Will My Design Run? - 2

- Routing is also important
  - The route between two components has its own propagation delay
  - The longer the route, the longer the propagation delay
  - The place and route tools try to minimise the propagation delay
  - The closer you are to the limits of the chip, the harder it is to route
  - You cannot accurately predict the routing delay before P&R
- To determine the maximum clock speed of a design or core
  - Gradually tighten the timing constraints
  - Turn up the effort levels on the place and route tools
  - Choose sensible pins
  - Don’t just build the design with slack timing constraints and turn up the physical clock speed
- Conclusion: place and route your design before making any firm conclusions about how fast it will run

# Tips for Taking Your Design to Hardware

- Test bits at a time in HW
  - Tempting to run huge simulation of entire design and then assume it will work in HW
  - Build up unit tests – e.g. memory access, pipeline stages
  - Makes it easier to identify problems
  - Compare the hardware results to the simulation results
- Put interfaces on chunks and test
  - Logic estimator
  - Look at the EDIF if you are unsure
- Use debugging information
  - LEDs, 7-segment, VGA (PAL console), transfer to host
  - Check whether your design reaches certain states
  - Be aware that this affects the design

# Key Points

- An FPGA is made up of a vast number of identical flip-flop/LUT pairs
- Use the manufacturer’s data book to find out about the resources available in your FGPA
- Learn how to use the place and route tools
- Always check the timing report after place and route
- Look for changes in the map report
- LUTs and flip-flops are the only real gauge of the size of a device
- Place and route your design before making any decisions about whether it will or fit or run at the desired speed
- Test bits of your code at a time in hardware

Celoxica™

Software-Compiled System Design

# Handel-C and Hardware

An overview of how DK maps Handel-C source to FPGA

# Technology Mapping

- The gate-level netlist output of DK contains basic gates: OR, XOR, NOT and AND
- The FPGA itself is built up of lookup tables (LUTs) and other components
- Technology mapping is the process of packing gates into these components
- Each LUT has a fixed propagation delay associated with it, regardless of what it is doing
- DK has its own technology mapping, discussed later
  - You can always use the vendor’s technology mapping instead
    - Results are sometimes worse, sometimes better
    - You won’t get useful area and delay debugging info from DK
    - You can’t use the retiming tool in DK without using DK’s technology mapping

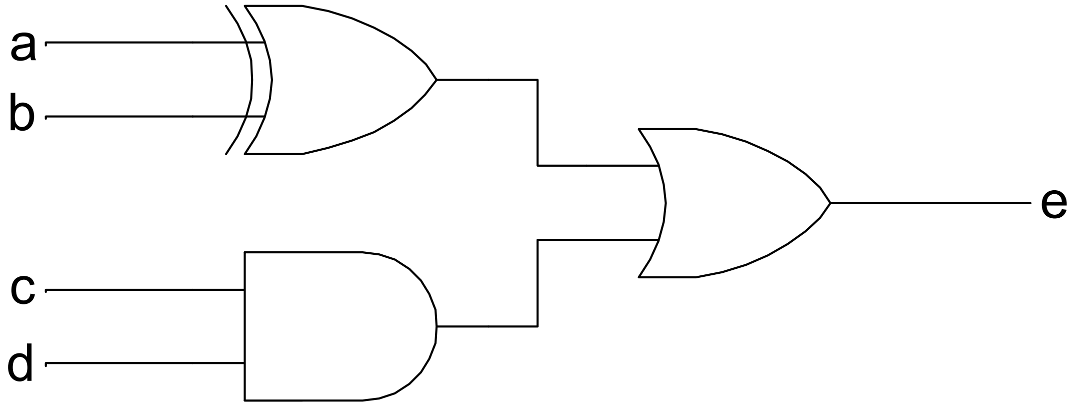
# LUTs

- Both Altera and Xilinx FPGAs have 4-input LUTs (except Stratix II)
- A LUT can be though of as implementing the truth table for a four input, one output circuit
- Any circuit with four inputs and one output can be mapped into a single LUT
- Each LUT's truth table is fixed at start-up, so a LUT only ever performs the same logic function
- If you perform a different logic function with the same four inputs, then another LUT will be used
- A LUT in an Altera or Xilinx FPGA is actually made out of an SRAM block
  - Changing the four input values addresses a different value in the table

# LUTs as Truth Tables

- This code can be implemented in a single LUT

```
unsigned 1 a, b, c, d, e;
e = (a ^ b) | (c & d);
```

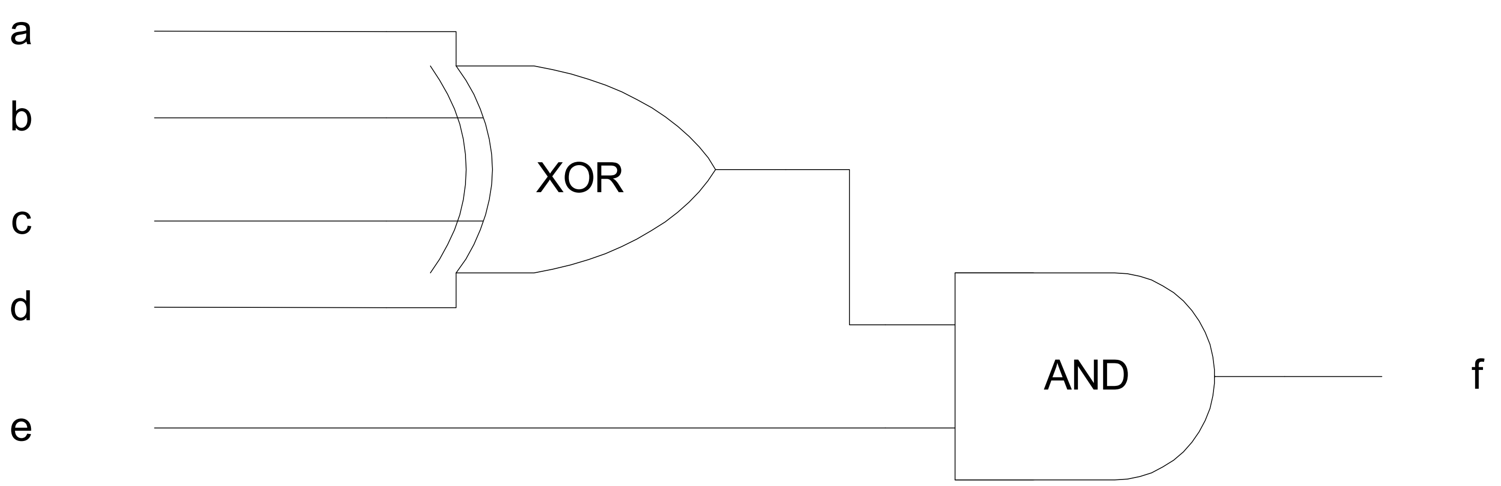


| a | b | c | d | e |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# How Logic Maps to Multiple LUTs

- This example has 5 inputs

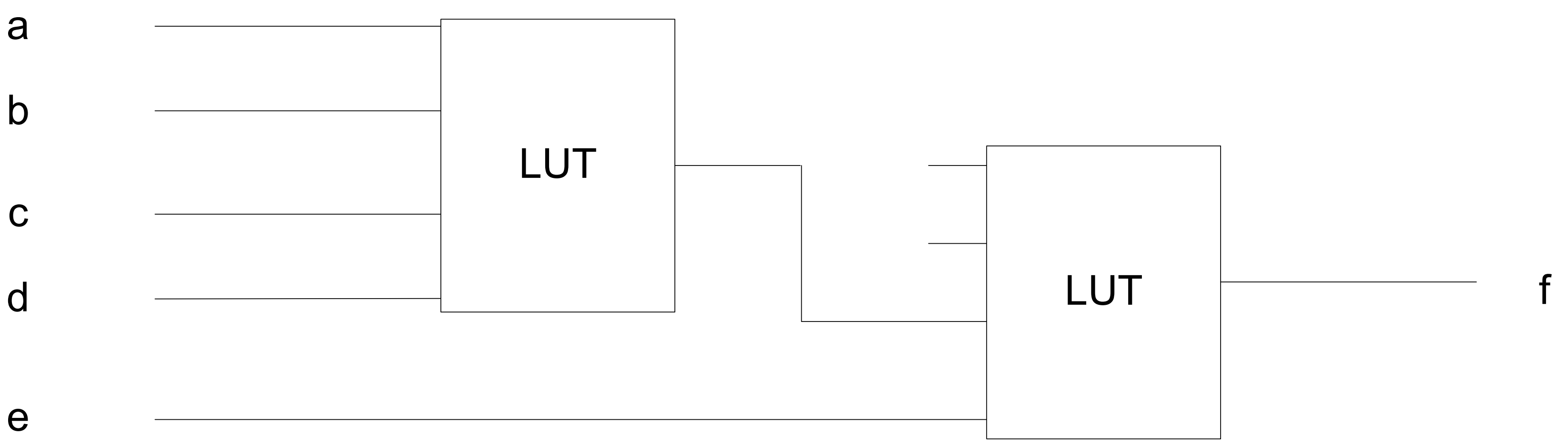
```
unsigned 1 a, b, c, d, e, f;
f = (a ^ b ^ c ^ d) & e;
```



Celoxica Software-Compiled System Design

# How Logic Maps to Multiple LUTs - 2

- The code will therefore map to 2 LUTs



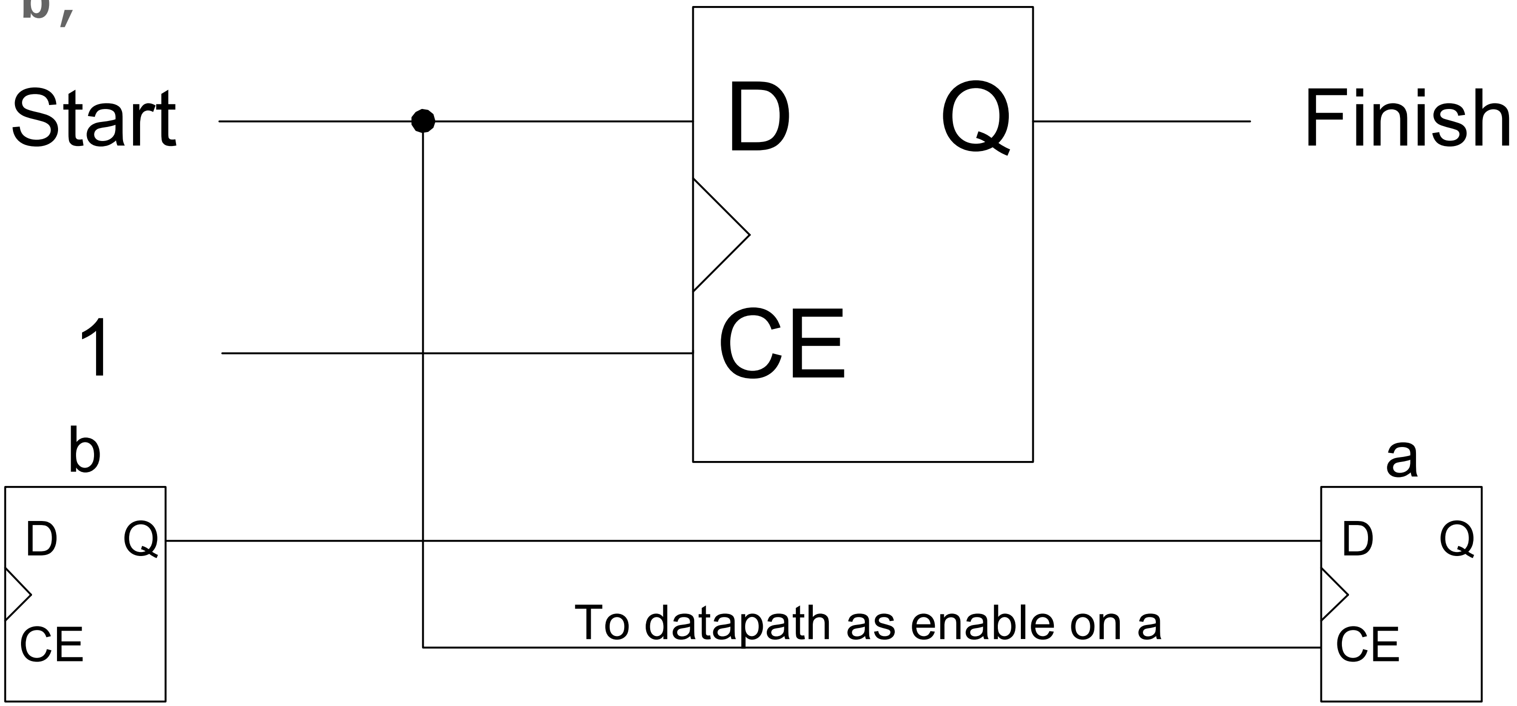
Celoxica Software-Compiled System Design

# How Handel-C Control Constructs Map to Hardware

- The next few slides show examples of how Handel-C statements map to control path blocks
- Each block has an input signal Start and an output signal Finish
- The Start signal comes from the enclosing block
- The main function has its own special Start signal
- The Start signal goes high for one clock cycle when the statement is executed
- Each cloud in the diagrams represents another statement, which takes 0 or more clock cycles to execute

# Assignment

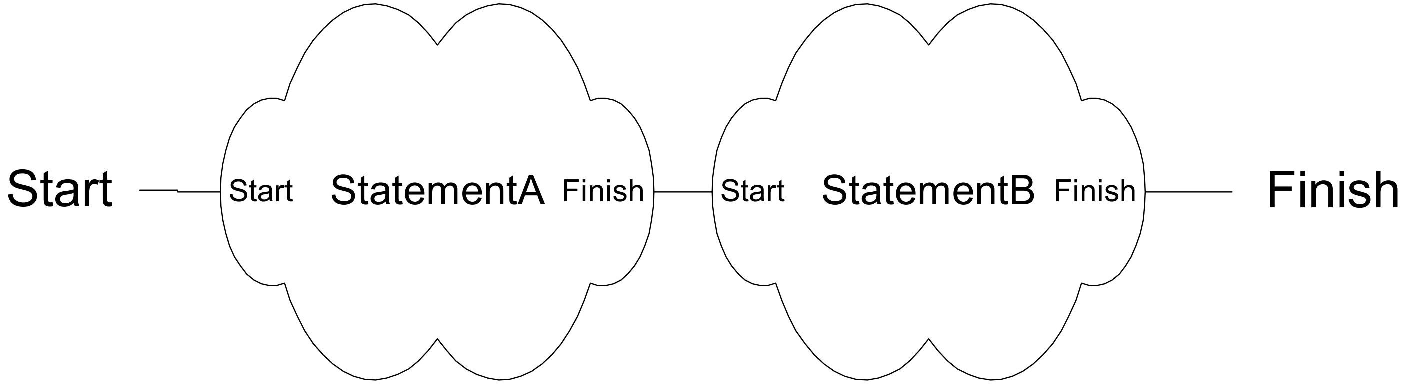
- `a = b;`



- CE is the Clock Enable
  - Makes the flip-flop output the value on its input on the clock edge

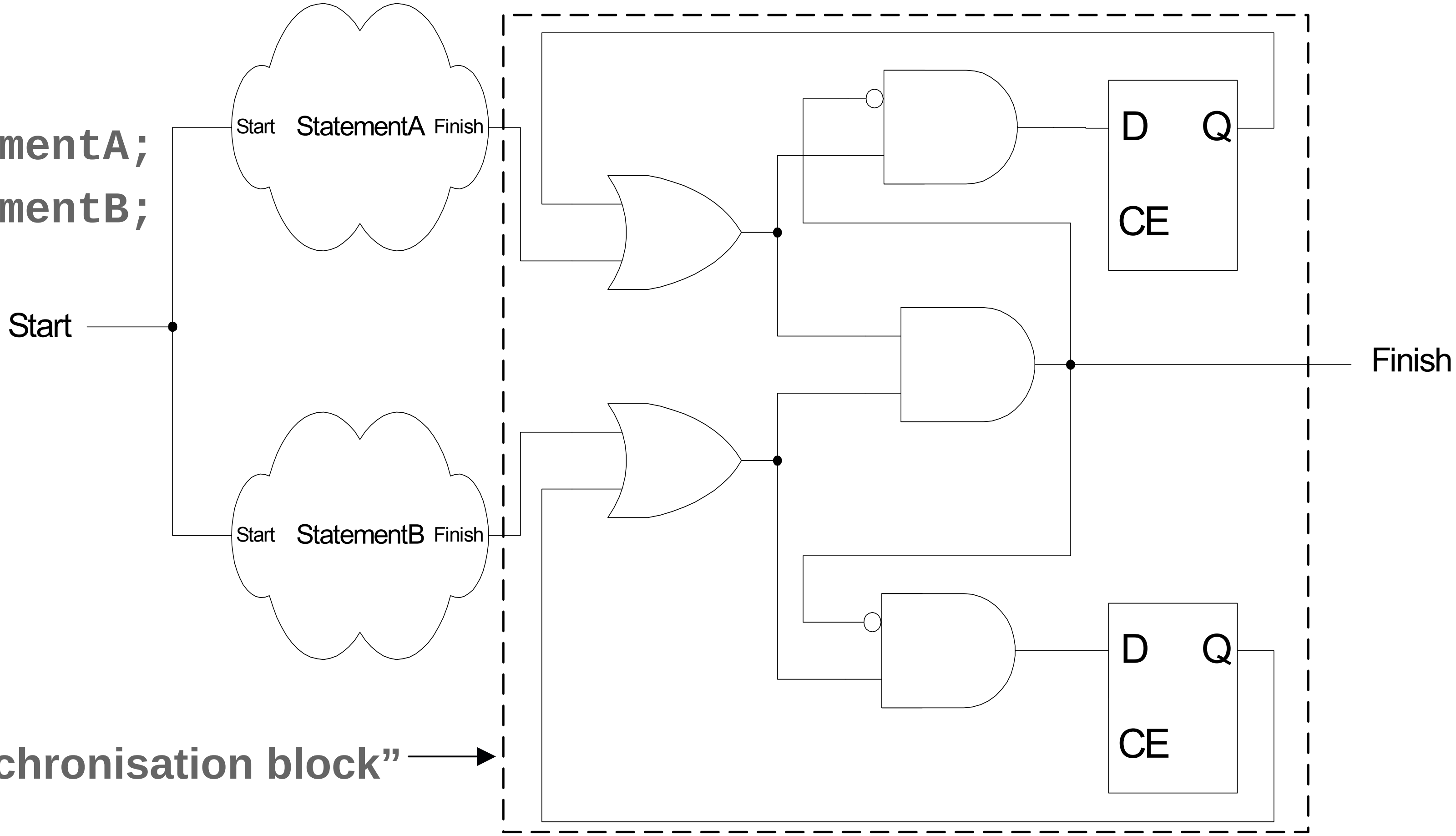
# Sequential execution

```
seq
{
    StatementA;
    StatementB;
}
```



# Parallel execution

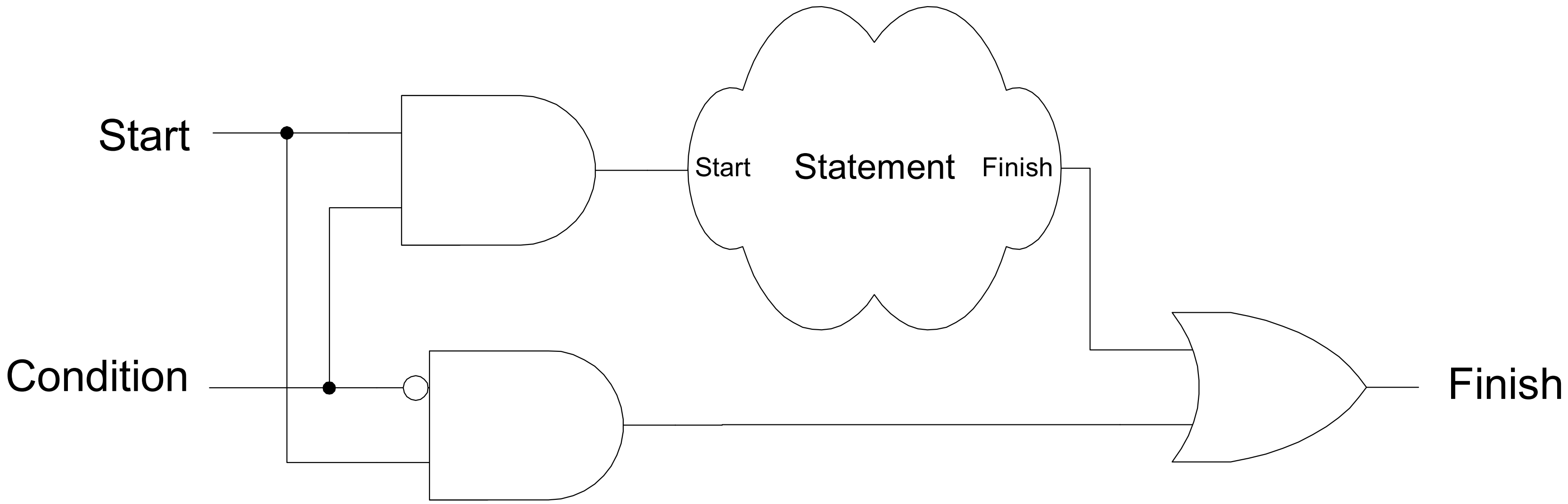
```
par
{
    StatementA;
    StatementB;
}
```



“par synchronisation block”

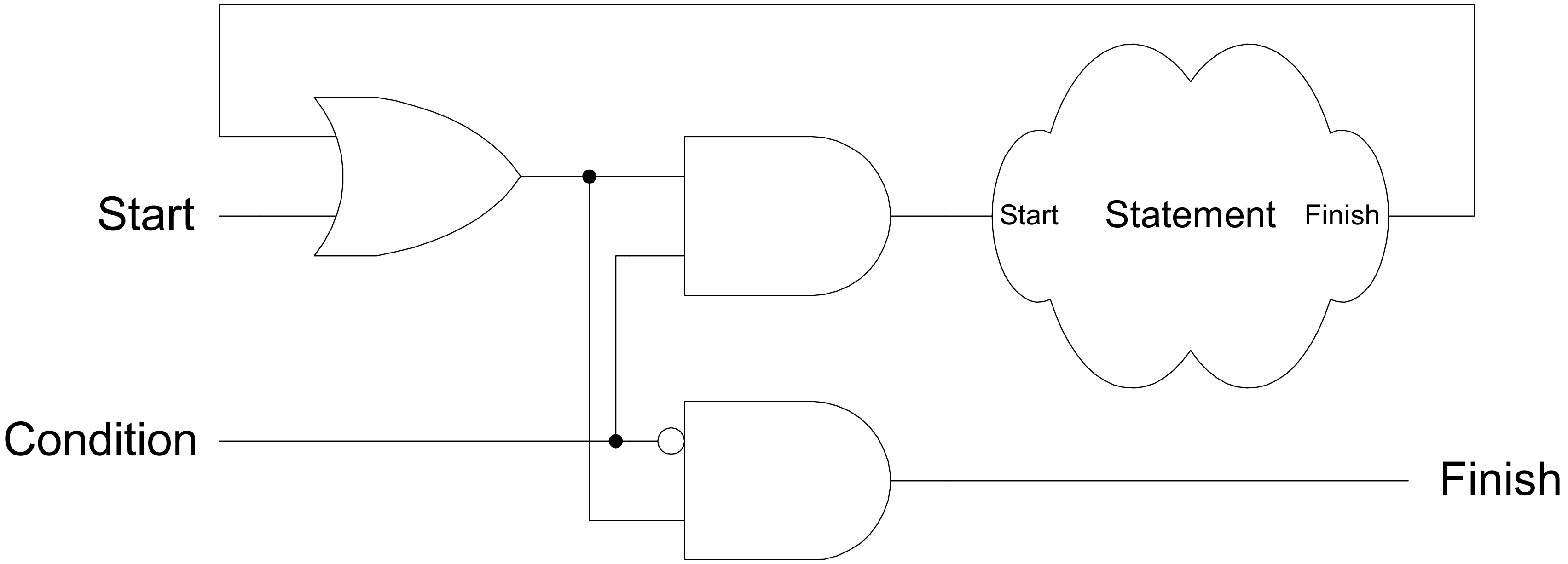
# if statements

```
if (Condition)
    Statement;
```



# while loops

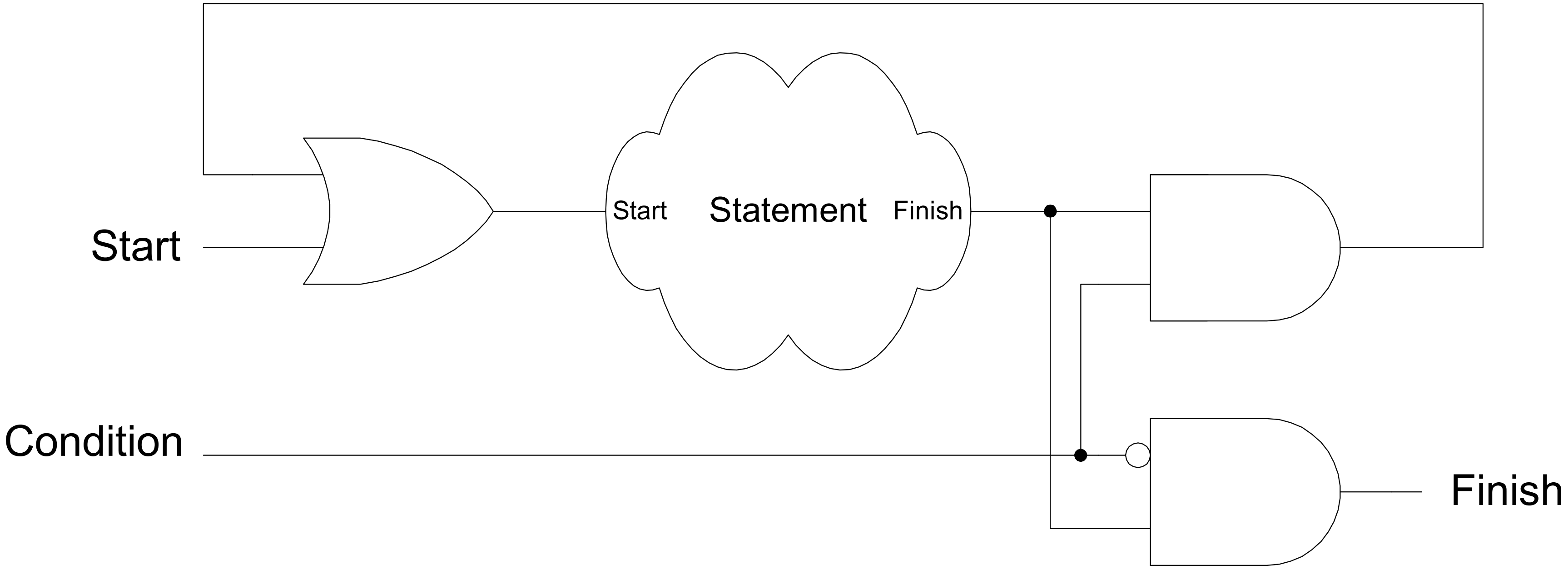
```
while(Condition)
{
    Statement;
}
```



Celoxica
Software-Compiled System Design

# do{} while loops

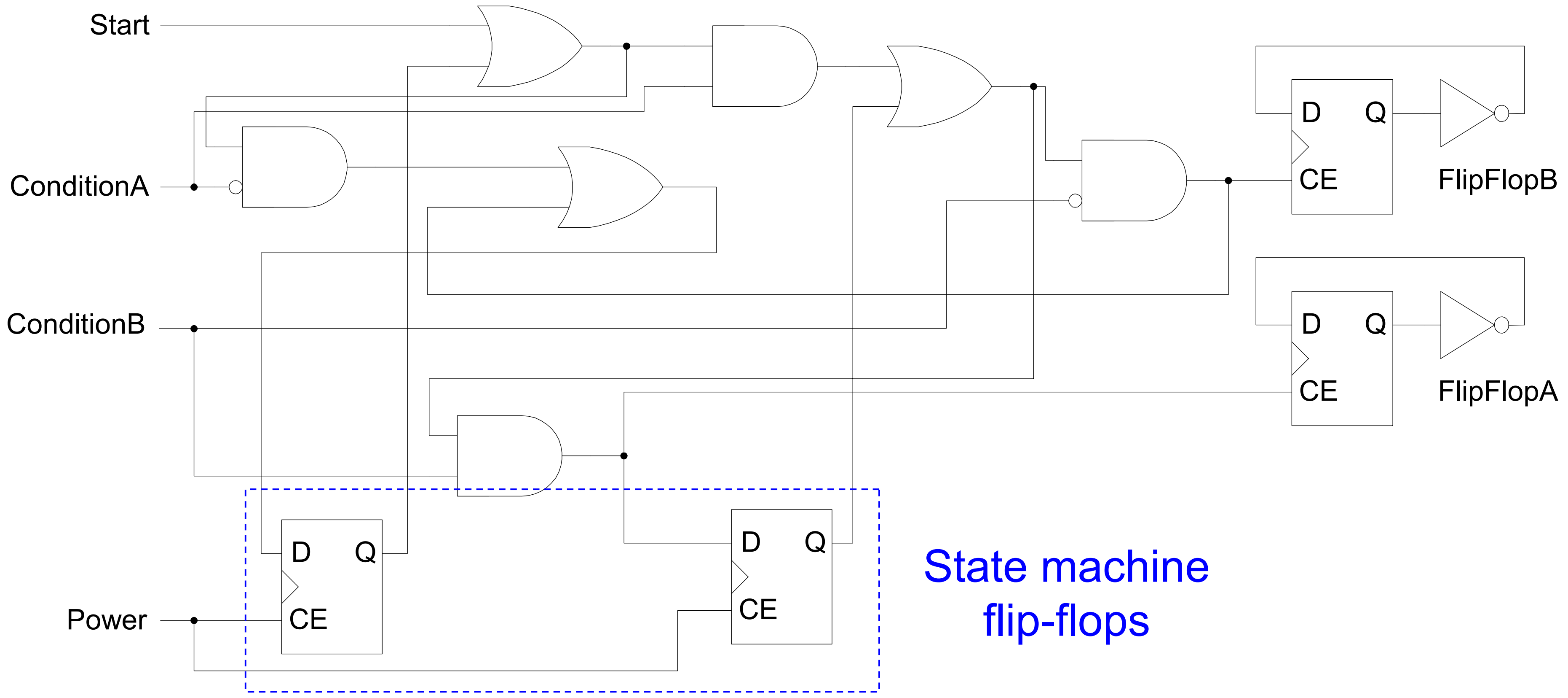
```
do
{
    Statement;
} while(Condition);
```



# Source Code for Simple Circuit

```
void main
{
    static unsigned 1 FlipFlopA = 0;
    static unsigned 1 FlipFlopB = 0;
    signal unsigned 1 ConditionA;
    signal unsigned 1 ConditionB;
    while(1)
    {
        if (ConditionA)
        {
            while(ConditionB)
            {
                FlipFlopA = ~FlipFlopA;
            }
            FlipFlopB = ~FlipFlopB;
        }
        else
            delay;
    }
}
```

# Circuit Diagram



State machine flip-flops

# Expressions

- Expressions translate to combinational data path
- Example

```
a > b ? x * y : z
```

- DK creates optimised arithmetic using the facilities of the FPGA
  - Counter-intuitive to those familiar with multipliers implemented in logic
  - Xilinx Virtex-II
    - □ MUXCY, XORCY and MULT_AND components
    - □ Dedicated carry in and out for each slice
  - Altera Stratix
    - □ Arithmetic mode on LCELL (LUT) – innards hidden away
    - □ Dedicated carry in and out for each LE

# Key Points

- Mapping is the process of translating generic gates to device specific resources
- Xilinx and Altera (except Stratix-2) devices have 4-input LUTs
- You have seen how DK statements map to logic
- Expressions translate to combinational data path logic
- DK produces device-specific, optimised arithmetic

# Pipelining Your Design

## Assembly line analogy

# Assembly Line Analogy

- Think of the program as a set of tasks that need to be performed on a datum, one after the other
- In an entirely sequential program, the tasks are executed sequentially in time, one task per clock cycle
- Imagine this is one worker in a factory performing each task one after the other, for example to make a car
- In order to increase output, we can do one of two things:
  - Employ more workers to perform exactly the same function
  - Employ more workers and give them each a single task to perform on an assembly line
- In hardware, it is very expensive to have multiple workers: each task that they can do is a piece of logic that will be unused most of the time
- On an assembly line, each of our workers can work continually on each datum as it passes in front of them

# Pipelining Your Design

- Basic principle is to break a complex expression into multiple smaller expressions and run them all simultaneously
- Each stage in the pipeline reads the result of the previous stage while the new value is being written
- Increase clock rate by decreasing logic depth
  - One result per clock cycle
  - Increase throughput
- Results in increased data latency
  - Delayed by pipeline length
  - Usually not a problem
- Results in increased flip-flop usage
  - More flip-flops need to store intermediate stages of calculation
  - FPGAs are flip-flop rich

# Simple Pipeline Example

```
while(1)
{
    Result = (MyRAM[Index]*b)+c;
    Index++;
}
```

- Can be simply pipelined like this

```
while(1)
{
    par
    {
        RAMRegister = MyRAM[Index]; // Stage 1
        Index++;                    // Stage 1
        MultResult = RAMRegister*b; // Stage 2
        Result = MultResult+c;      // Stage 3
    }
}
```

- Now the longest path is the multiply
- The first result appears at the beginning of clock cycle 4

# Assembly Line Analogy Continued

- The speed at which the conveyor belt runs depends on the slowest worker on the assembly line
- If you have one worker who is taking longer to perform his task than the other workers take for theirs, then he is holding back the speed at which you can run the conveyor belt
- Three solutions
  - Add another worker to perform some of his task
  - Have another worker either before him or after him on the assembly line perform some of his task
  - Analyse why he is taking so long and improve his performance
- Maybe the output is too high or all workers have time to spare!
  - Slow down the conveyor belt
  - Remove some workers and divide their tasks up amongst the rest

# Balancing the Pipeline

- You need to balance the logic evenly between pipeline stages
  - The maximum clock rate of the design is limited by the pipeline stage with the deepest logic
- To reduce the logic depth for a stage you can
  - try to make it more efficient
  - move some of the logic to another, shallower stage
  - move some of the logic to a new stage, lengthening the pipeline
- Breaking up a stage can be non-trivial
  - For example, the slowest stage might be an arithmetic operation like a multiply or an addition
  - You may need to write your own pipelined implementation of a common operation
  - Another alternative is to use retiming, which we will discuss later

# Pipeline Latency

- The overhead of pipelining is the delay introduced in getting the result from an input
  - Once you have the first result, you get one result per clock cycle
  - The more stages in the pipeline, the longer the delay
- For many designs, pipeline latency is not a problem
  - Raw throughput is often more important
- Time from input to output = clock period x pipeline depth
  - Either increase the clock rate or decrease the pipeline length to reduce the time from input to output
  - Clearly, the clock rate is related to the maximum logic depth in the pipeline, so any changes will be trade-offs

# Pipelining Is Not Always Possible

- Pipelining a sequential loop can use much more logic
  - e.g. a large FFT or FIR
  - Consider a semi-pipelined approach
- For example where each result depends on the previous
  - e.g. searching a binary tree
- Is it possible to use a different algorithm?
- Data parallelism is still usually possible
  - Make multiple copies of the same process
  - Run them all in parallel

# Key Points

- If you have trouble imagining a pipeline, try thinking about an assembly line
- The principle is to break a complex expression into smaller expressions and run them all in parallel
- Allows you to increase throughput
- Results in an increase in flip-flop usage and data latency
- Try to balance the logic evenly between the stages
- You can't always pipeline an algorithm

Celoxica
Software-Compiled System Design

# Synchronising the Pipeline

Methods for synchronising a pipeline

# Synchronising the Pipeline

- A pipeline starts with either some initialised or undefined data in it
  - Starting a pipeline with some useful initial values is called priming the pipeline
- Most of the time you want to ignore all the results until the first valid input propagates to the end of the pipeline
- Often you also need to stop and start the pipeline from one end or the other
- The solutions to these two problems are often interdependent

# Synchronising the Pipeline - Solutions

- Delay the start of the sink process (the downstream end)
  - Count out n outputs from the pipeline, where n is equal to the pipeline depth
- Label data as valid at the source process (the upstream end)
  - Useful when the availability of an input resource is unpredictable
- Pause with a signal from sink
  - Useful when the availability of an output resource is unpredictable
- Run an identical process at the source as at the sink
  - Start n clock cycles earlier, where n is equal to the pipeline depth
  - e.g. sync generator for display output

# Synchronisation Counting out Data

- Either use a counter or sequential delays to count the invalid data at the end of the pipeline
  - Sequential delays will be mapped to shift registers
    - □ Very shallow logic depth – next event will only depend on 1-bit output of shift register
    - □ Small logic utilisation – in Xilinx FPGA, each 16-bit shift register (1 LUT) deals with 16 clock cycle delay
  - Use counter for long delays
- Only works for source and sink processes that never pause

```
seq(i = 0; i < PipelineDepth; i++)
    delay;
```

- Using the valid token method without inserting invalid data after start-up will create very similar logic and is more versatile

# Synchronisation Counting out Data - 2

```
static unsigned 4 Delay = 0b1111; // initialise with all 1s
unsigned 8 a[4], Output;

while(1)
{
    par
    {
        a[0] = Input;       // stage 1
        a[1] = a[0] + 1;    // stage 2
        a[2] = a[1] << 1;   // stage 3
        a[3] = a[2] ^ a[1]; // stage 4

        Delay = Delay << 1; // shift our delay register for ever
        if (Delay[3] == 0)  // wait until all of the 1s are out of Delay
            Output = a[3];  // read the output
        else
            delay;          // otherwise do nothing
    }
}
```

# Synchronising with Valid Flag

- Mark each piece of data from the source with whether or not it is valid

```
#define VALID 1
#define INVALID 0

typedef struct
{
    unsigned 1 Valid;
    unsigned 8 Value;
} Datum;
```

- Small overhead in terms of space and logic
  - Valid flag propagation will be mapped into SRL16s automatically in Xilinx parts
- No counting data in or out

# Synchronising with Valid Flag - 2

- Initialise the pipeline with invalid flags

```
void main()
{
    static Datum Data[3] = {{INVALID},{INVALID},{INVALID}};

    par
    {
        Source(&(Data[0]));
        PipeStage1(Data[0], &(Data[1]));
        PipeStage2(Data[1], &(Data[2]));
        Sink(Data[2]);
    }
}
```

# Synchronising with Valid Flag - 3

- Mark each datum out of the source

```
macro proc Source(DataOut)
{
    if (ResourceReady())
    {
        par
        {
            DataOut->Valid = VALID;
            ReadResource(&(DataOut->Data));
        }
    }
    else
        DataOut->Valid = INVALID;
}
```

# Synchronising with Valid Flag - 4

- Each stage must propagate the valid flag from the previous stage
- Each stage continually runs – they don’t care whether or not the data is valid

```
macro proc PipeStage1(DataIn, DataOut)
{
    par
    {
        DataOut->Valid = DataIn.Valid;
        DataOut->Data = ~(DataIn.Data);
    }
}
```

# Synchronising with Valid Flag - 5

- Only at the end of the pipeline is the valid flag actually read and acted upon
- No need to count data out as the pipeline is initialised with invalid data

```
macro proc Sink(DataIn)
{
    while(1)
    {
        if (DataIn.Valid == VALID)
        {
            WriteResource(DataIn);
        }
        else
            delay;
    }
}
```

# Synchronising with a Signal

- Downstream end of the pipeline can pause the rest of the pipeline when it is not ready to receive data

```
void main()
{
    signal unsigned 1 Pause;
    static Datum Data[3] = {{INVALID},{INVALID},{INVALID}};

    par
    {
        Source(&(Data[0]), Pause);
        PipeStage1(Data[0], &(Data[1]), Pause);
        PipeStage2(Data[1], &(Data[2]), Pause);
        Sink(Data[2], &Pause);
    }
}
```

# Synchronising with a Signal - 2

- Sink process

```
macro proc Sink(DataIn, Pause)
{
    while(1)
    {
        if (ResourceReady() == 1)
        {
            par
            {
                *Pause = 0;
                if (DataIn.Valid == VALID)
                {
                    WriteResource(DataIn);
                }
                else
                    delay;
            }
        }
        else
            *Pause = 1;
    }
}
```

# Synchronising with a Signal - 3

- Pipeline stage

```
macro proc PipeStage1(DataIn, DataOut, Pause)
{
    while(1)
    {
        if (Pause == 0)
        {
             par
             {
                 DataOut->Valid = DataIn.Valid;
                 DataOut->Data = ~(DataIn.Data);
             }
        }
        else
            delay;
    }
}
```

# Controlling From Either End

- A combination of valid flags and signal synchronisation can be used if either end needs to control the pipeline
- The example in the previous slides pause all the stages of the pipeline
- You can pause only the source itself and buffer up all of the valid data in the pipeline at the sink using a FIFO
  - Reduces the fanout of a pause signal going to all of the stages
  - Makes the pipeline easier to write as each stage doesn’t depend on the pause signal
  - Only put valid data into the FIFO
  - The sink always reads from the FIFO
  - FIFO needs to be at least as deep as the longest pause anticipated
  - Use a FIFO implemented in block RAM as a circular buffer where you track the head and the tail
- If you can’t pause the source, then you may end up dropping data

# Key Points

- You can count out invalid data from the pipeline if it is never paused
- To pause from the source, flag data as either valid or invalid at the sink and propagate this flag to the end
- To pause from the sink, pass a signal back up the pipeline
- Consider pausing only the source process, rather than all of the stages, and buffer up the valid data using a FIFO
- If you can't pause the source process, then you may end up dropping data if the sink is not always ready

# Efficient Handel-C

Efficient use of conditions, loops, rams and channels

# General Notes on Efficiency

- Premature optimisation is the root of all evil
- Make sure your algorithms are efficient at the design stage
- Get the program working before you optimise
  - It will be easier to debug
- Don’t sacrifice readability for efficiency
  - Especially if the design meets the requirements as it is
  - Someone else may have to maintain your code
- Learn to rely on the compiler optimisations
  - Let the compiler do the work

# Efficient Condition Evaluation

- All conditions before a statement add logic depth to the enable of that statement
- Keep the expression as simple as possible
  - Avoid testing RAM, ROM, randomly accessed array output directly in expression
- Test for equality rather than using > or <
- Test the smallest number of bits possible
  - Even in 4-input LUTs, a 4-bit comparison is not necessarily the same as a 1-bit comparison – this depends on the number of control paths joining at that point
  - There is always at least one bit for the control path
- Register your condition the previous clock cycle
- Beware using conditions on signals
  - Logic depth in assignment to signals could be significant

# Efficient Condition Evaluation - 2

- Avoid deeply nested conditions

```
if (x > y)
{
    if (x == 255)
        x = y * 3;
    else
        delay;
}
else
    delay;
```

- Avoid nesting if the conditions are mutually exclusive
  - Use parallel `if`s or `switch`

# Efficient Condition Evaluation - 3

- Use a default on `switch` statement
  - Even when all cases are covered

```
switch (a)
{
    case 0: a++;
    case 1: a--;
    case 2: a+=2;
}
b++;
```

- Here, b++ is dependent on the test on a
- Adding a `default: delay;` statement makes b no longer dependent on a

# Avoid for Loops

- They have a clock cycle overhead per iteration for the increment statement.
- Example

```
for (i=0; i!=0; i++)
{
    MyRAM[i] = i;
}
```

- Is functionally equivalent to:

```
i=0;
do
{
    MyRAM[i] = i;
    i++;
}while(i!=0);
```

# Loop Optimisation

- Often necessary to loop through all values of a variable
  - e.g. An address for a RAM

```
unsigned 17 i;
while(1)
{
    i = 0;
    while(i<0x10000)
    {
        MyRAM[i] = x;
        i++;
    }
}
```

# Loop Optimisation - 2

- Increment in parallel with the body of the loop

```
unsigned 17 i;
while(1)
{
    i = 0;
    while(i<0x10000)
    {
        par
        {
            MyRAM[i] = x;
            i++;
        }
    }
}
```

# Loop Optimisation - 3

- Initialise the counter and count to zero

```
static unsigned 16 i = 0; // only need 16 bits, no multiplexing
while(1)
{
    do // don’t care what Stop is at first iteration of loop
    {
        par
        {
            MyRAM[i] = x;
            i++;
        }
    } while(i>0); // i will be 0 at next iteration
}
```

# Loop Optimisation - 4

- Test for equality

```
static unsigned 16 i = 0; // only need 16 bits
while(1)
{
    do
    {
        par
        {
            MyRAM[i] = x;
            i++;
        }
    } while(i!=0); // test for equality
}
```

# Loop Optimisation - 5

- Register the loop condition

```
static unsigned 16 i = 0;
unsigned 1 Stop; // Register loop end condition
while(1)
{
    do
    {
        par
        {
            MyRAM[i] = x;
            i++;
            Stop = (i==0xffff);
        }
    } while(!Stop) // i++ now depends on 1 bit test
}
```

# Loop Optimisation – 6

- For complicated, multi-cycle body, increment counter in parallel with the final statement of the body
- Example:

```
do
{
    statement0();
    statement1();
    statement2();
    par
    {
        statement3();
        i++;
    }
}while(i!=0);
```

Software-Compiled System Design

# do{} while() Versus while(){}

- The two code examples below are subtly different in their meaning
- Example A:

```
while(cond1);
{
    while(cond2)
    {
        statement();
    }
}
```

- Example B:

```
do
{
    do
    {
        statement();
    } while(cond2);
} while(cond1);
```

- Example A takes 0 or more clock cycles, example B 1 or more
- Compiler warns of combinational cycle in A and tries to fix it
  - Path where cond1 is true and cond2 is false, giving 0 cycle loop
  - Unpredictable result

# do{} while() Versus while(){} - 2

- The two code examples below are subtly different in their meaning
- Example A:

```
if (cond1)
{
    while(cond2)
    {
        statement1();
    }
    statement2();
}
else
    delay;
```

- Example B:

```
if (cond1)
{
    do
    {
        statement1;
    } while(cond2);
    statement2;
}
else
    delay;
```

- Example B is more efficient
  - Use do while if your loops will always be executed at least once
- In example B, there is a path through cond2 to `statement2()`, but not through cond1

# Optimising RAM Usage

- Avoid using a RAM in more than one place
  - Use both ports of a block RAM
    - They have dedicated address and data ports
- More block RAM optimisation later
- Register Address and Data
  - Pipeline memory accesses to maintain throughput
  - Use one register only for address, one for input data and one for output data
- Choose appropriate RAM type
  - Distributed, block, external
  - Tri-matrix in Altera, can leave Quartus II to decide which RAM to use

# Prialt-Signal Trick With Channels

- There is no language construct to check channels for readiness
- You can get around this using a neat trick

```
//channel down which to send data
chan unsigned 8 MyChannel;
//data to send
unsigned 8 Data;
//signal indicating readiness of the channel
static signal unsigned 1 MyChannelNotReady = 0;
```

# Prialt-Signal Trick With Channels - 2

```
while(1)
{
    par
    {
          prialt
          {
                    //try to send data
                    case MyChannel ! Data: break;
                    //otherwise indicate that the channel is not ready
                    default: MyChannelNotReady = 1; break;
          }

          if(MyChannelNotReady)
          {
                    DoSomething();
          }
          else
          {
                    DoSomethingElse();
          }
    }
}
```

# Key Points

- Optimisation starts at the design stage
- Keep expressions as simple as possible
- Use == or != instead of > or <
- Register conditions
- Use a `default: delay;` in a `switch` statement
- Increment in parallel with last statement in body of a loop
- Count to zero for powers of two
- Use `do{} while{}` if you know the loop will be executed at least once
- Register address and data in and out of RAM
- Use the `prialt-signal` trick to test a channel for readiness

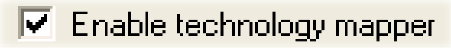
Celoxica
Software-Compiled System Design

# Debugging Area and Delay

Logic estimator, timing constraints, timing analysis

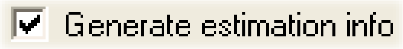
# DK Logic Estimator

- **Enable technology mapping in the** Synthesis **tab of the project settings**
  - **Without tech mapping enabled, the information is not as detailed**



☑ Enable technology mapper

- **Enable estimation information in the** Linker **tab**



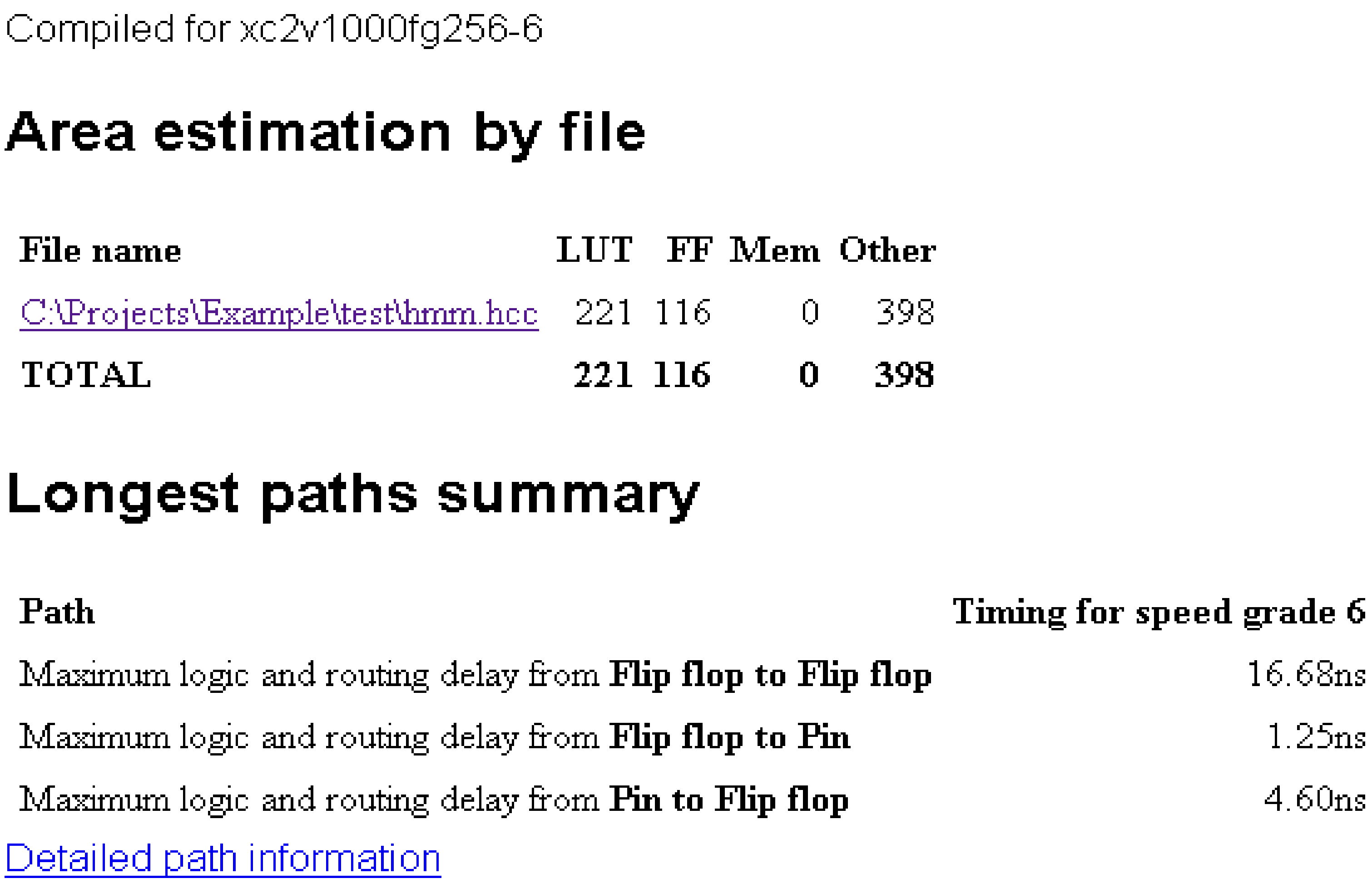
☑ Generate estimation info

- **Choose a chip in the** Chip **tab**
  - **DK knows the component delays for each supported device**
  - **Otherwise it will only report logic levels, which can be misleading**
- **When you build for** EDIF**, some HTML files will be produced in the *EDIF* directory**
  - **Double click on *Summary.html***
- **Shows device utilisation and logic depth in the context of the source code**

# DK Logic Estimator - 2

- Device utilisation
  - Flip-flop count for user registers and state machine registers
  - LUTs for logic, distributed storage, shift registers, multiplexers
  - Automatically mapped ALUs
  - “Other” for arithmetic multiplexers and other device specific resources (CARRY_SUM, CARRY for Altera, MUXCY, XORCY, MultAND for Xilinx)
- Logic depth
  - Uses vendor information to provide accurate component delays
  - Routing delay based on fanout only
- This is NOT placed or routed, so is not 100% accurate
  - Routing can account for both logic delay and utilisation
  - Does not always identify longest path
  - Less accurate as the design gets closer to filling up the chip
  - LUTs may be removed in optimisation
  - LUTs used as route through may be added

# Area and Delay Estimation Summary



Compiled for xc2v1000fg256-6

## Area estimation by file

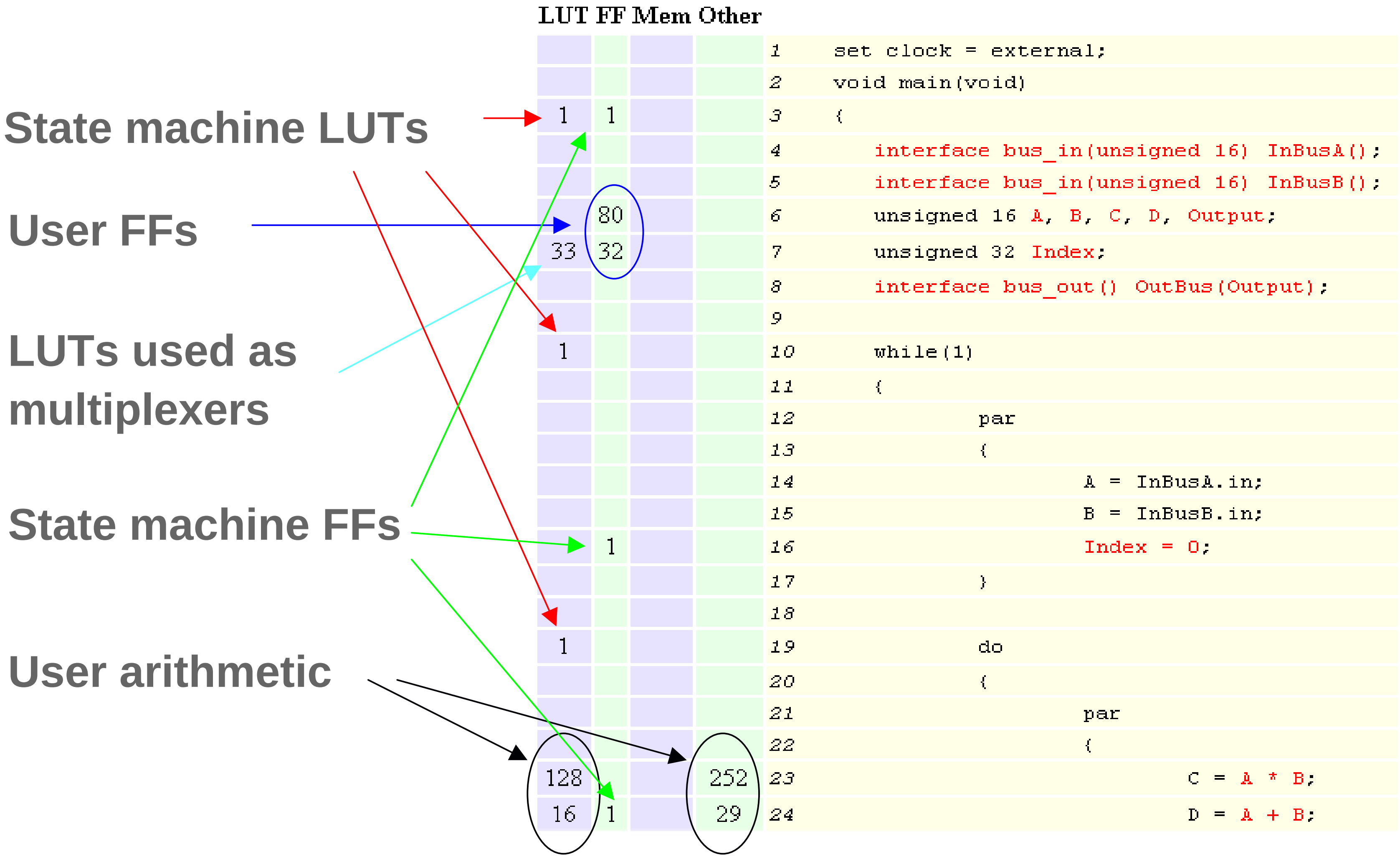
| File name | LUT | FF | Mem | Other |
|---|---|---|---|---|
| C:\Projects\Example\test\hmm.hcc | 221 | 116 | 0 | 398 |
| **TOTAL** | **221** | **116** | **0** | **398** |

## Longest paths summary

| Path | Timing for speed grade 6 |
|---|---|
| Maximum logic and routing delay from **Flip flop to Flip flop** | 16.68ns |
| Maximum logic and routing delay from **Flip flop to Pin** | 1.25ns |
| Maximum logic and routing delay from **Pin to Flip flop** | 4.60ns |

Detailed path information

# Area Estimation



State machine LUTs

User FFs

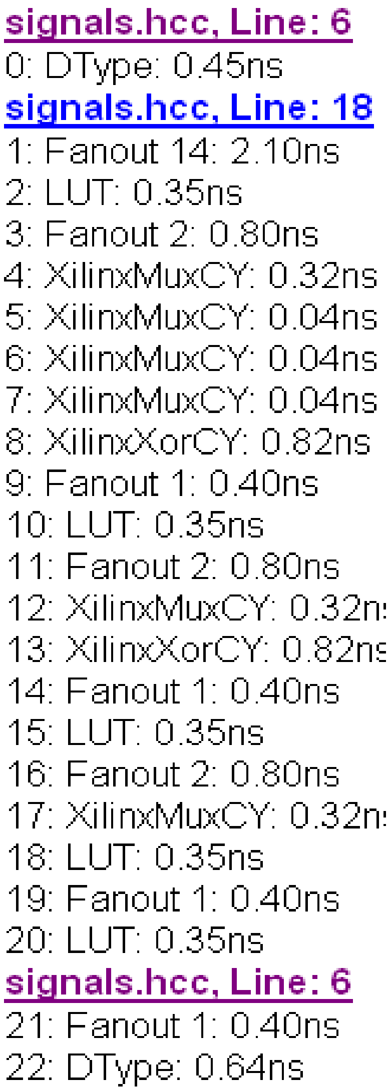
LUTs used as multiplexers

State machine FFs

User arithmetic

| LUT | FF | Mem | Other | |
|---|---|---|---|---|
| | | | | set clock = external; |
| | | | | void main(void) |
| 1 | 1 | | | { |
| | | | | interface bus_in(unsigned 16) InBusA(); |
| | | | | interface bus_in(unsigned 16) InBusB(); |
| | 80 | | | unsigned 16 A, B, C, D, Output; |
| 33 | 32 | | | unsigned 32 Index; |
| | | | | interface bus_out() OutBus(Output); |
| | | | | |
| 1 | | | | while(1) |
| | | | | { |
| | | | | par |
| | | | | { |
| | | | | A = InBusA.in; |
| | | | | B = InBusB.in; |
| | 1 | | | Index = 0; |
| | | | | } |
| | | | | |
| 1 | | | | do |
| | | | | { |
| | | | | par |
| | | | | { |
| 128 | | | 252 | C = A * B; |
| 16 | 1 | | 29 | D = A + B; |

# Longest Paths



```
signals.hcc, Line: 6
0: DType: 0.45ns
signals.hcc, Line: 18
1: Fanout 14: 2.10ns
2: LUT: 0.35ns
3: Fanout 2: 0.80ns
4: XilinxMuxCY: 0.32ns
5: XilinxMuxCY: 0.04ns
6: XilinxMuxCY: 0.04ns
7: XilinxMuxCY: 0.04ns
8: XilinxXorCY: 0.82ns
9: Fanout 1: 0.40ns
10: LUT: 0.35ns
11: Fanout 2: 0.80ns
12: XilinxMuxCY: 0.32n
13: XilinxXorCY: 0.82ns
14: Fanout 1: 0.40ns
15: LUT: 0.35ns
16: Fanout 2: 0.80ns
17: XilinxMuxCY: 0.32n
18: LUT: 0.35ns
19: Fanout 1: 0.40ns
20: LUT: 0.35ns
signals.hcc, Line: 6
21: Fanout 1: 0.40ns
22: DType: 0.64ns
```

- Flip flop to Flip flop: 11.65ns
- Clicking on the hyperlinks takes you to the line in the source code
- The path is from source to sink, top to bottom
- Components are marked by name
  - From the MuxCYs and XorXYs, you can tell that this example is for a multiplier
  - DType means a D-type flip-flop
- Routing delay estimates are marked as “Fanout”
  - Fanout means the number of places that the output is connected to
  - The number afterwards in the size of the fanout
  - In general, the larger the fanout, the greater the routing delay
- The last number is the delay in nanoseconds

# Timing Constraints

- Timing constraints instruct the vendor's tools to place and route the design such that its logic depth is shallow enough to clock at the target rate
  - Doesn't affect component delays
- We recommend always using timing constraints
- Constraints are passed on to the place and route tools
  - Xilinx - *.ncf* file created by DK automatically picked up by P&R tools
    - □ Remember to copy *.ncf* file if you move the EDIF file
  - Altera - run the TCL script generated by DK in Altera's Quartus-II tool
    - □ Easy to forget
- Learn how to use place and route tools
  - Increase effort levels if constraints are not met
  - Specifying higher clock speeds can make the tools try harder
    - □ Useful for finding fastest possible clock rate for a core

# Timing Constraints - 2

- `rate` specification

  ```
  set clock = external with {rate = 150}; // 150 MHz Clock
  ```

  - Will constrain all paths in that clock domain to that clock rate
    - □ Flip-flops to flip-flops
    - □ Other components e.g. block RAM, multipliers
    - □ NOT to or from PADs
  - Example output

    ```
    NET ClockInput PERIOD = 6.6667 ns;
    ```

  - Does not create a hardware clock generator of that frequency
    - □ Just constrains the paths
    - □ You still need to assign an input clock, either internal or external

# Timing Constraints - 3

- `intime` and `outtime` specifications

```
interface bus_in(unsigned 1 I) I() with {intime = 5};
```

  - `intime` = Maximum delay in ns from input pads to flip-flops
    - Makes sure you read the data while it is stable on inputs
  - `outtime` = Maximum delay in ns from flip-flops to output pads
    - Ensures that data output meets setup time requirements

# Timing Analysis

- Xilinx and Altera have their own timing analysis reports and tools
- They can report timing information before or after placement and routing are complete
  - Post place and route is by far the more useful
- They can report failing paths, selected paths or all paths against timing constraints
  - You either investigate a specific path that interests you
  - Otherwise, you can just let it tell you about any problems
- The components in the path will match the components in the DK Logic Estimator output

# Relating EDIF Back to Handel-C

- In order to understand the timing analysis report, you need to relate the EDIF back to the Handel-C source
- Nets in DK take predictable names based on the Handel-C source
- Nets connected to the output of a variable derive their name from that variable

  ```
  unsigned 8 MyVar;
  ```

  - Creates nets named `MyVar_0` to `MyVar_7`
  - If net names clash, then DK reverts to a more complex scheme
- Nets connected to pins take their names from the interfaces

  ```
  interface bus_out() myBus(unsigned 8 out = x);
  ```

  - Create nets named `PADOUT_myBus_out_0` to `PADOUT_myBus_out_7`
- Some special nets are named
  - ClockInput, Reset, GND, VCC
- Other nets follow complex scheme described in the manual

# Timing Analysis Example

```
set clock = external with {rate = 350}; // external clock at 350 MHz
void main()
{
    unsigned 8 Input;
    unsigned 8 Output;
    interface bus_out() OutputBus(Output);
    interface bus_in(unsigned 8) InputBus();

    while(1)
    {
        par
        {
            Input = InputBus.in;
            Output = Input+1;
        }
    }
}
```

Software-Compiled System Design

# Timing Analysis Example - 2

- Built for Xilinx Virtex-II 6000, ff1152 package, speed grade 6, retiming off
- Timing information displayed after place and route

```
Requested    | Actual      | Logic
             |             | Levels
--------------------------------
2.857ns      | 4.744ns     | 2
1 constraint not met.
```

  - The place and route tools were unable to meet the timing constraints
- Timing analysis using Xilinx Timing Analyser
  - Analysed against timing constraints
  - Timing Summary: `Timing errors: 7`
  - Only reports 3 failing paths by default
  - You can look up the components in the data book

# Timing Analysis Example - 3

```
Data Path: B8_TimingAn_5_main_DTYPE0 to B17_TimingAn_6_main_DTYPE0
   Delay type         Delay(ns)  Logical Resource(s)
   ----------------------------  -------------------
   Tiockiq               0.443   B8_TimingAn_5_main_DTYPE0
   net (fanout=1)        1.118   Input_6
   Topxb                 0.781   Input_6_rt
                                 B64_TimingAn_15_main_MUXCY
   net (fanout=1)        0.798   W83_TimingAn_15_main
   Tilo                  0.347   B66_TimingAn_15_main_LUT4
   net (fanout=1)        0.907   W85_TimingAn_15_main
   Tioock                0.307   B17_TimingAn_6_main_DTYPE0
   ----------------------------  ---------------------------
   Total                 4.701ns (1.878ns logic, 2.823ns route)
                                 (39.9% logic, 60.1% route)
```

# Resolving Timing Problems

- Long component delay
  - Extend the pipeline
- Large fanout
  - Add a layer of registers
  - Xilinx ISE performs “logic replication” during map stage that will duplicate function generators with large fanout
- Large multiplexers
  - Register inputs
  - Run multiple copies of the operation
    - Use an array of functions
  - Example of serialisation using shift registers versus bit selection

# Large Multiplexer Resolution

```
unsigned 256 Input;               // large input
unsigned 8 Output;                // serialised output
while(1)
{
    par
    {
        Input = NewInput;         // read in the new input
        Output = Input[255:248];  // write out the top word
    }
    seq (i = 0; i < 248; i+=8)    // write out each of the words
    {
        Output = Input[i+7:i];    // using bit selection
    }
} // This multiplexes 16 inputs into Output
```

# Large Multiplexer Resolution - 2

```
unsigned 256 Input;               // large input
unsigned 8 Output;                // serialised output
static unsigned (log2ceil(32)) DelayCount = 0; // Delay counter
while(1)
{
    par
    {
        Output = Input[7:0];  // Always output bottom 7 bits
        DelayCount++;         // Count
        if (DelayCount == 0)  // Once every 32 clock cycles
            Input = NewInput; // Read in a new input
        else
            Input = Input>>8; // Otherwise shift right by 8
    }
} // This multiplexes 2 inputs into Input
```

# Large Multiplexer Resolution - 3

- Xilinx place and route report outputs
- Using bit selection

```
----------------------------------
Requested  | Actual     | Logic
           |            | Levels
----------------------------------
5.000ns    | 6.897ns    | 12
----------------------------------
```

- Using shifts

```
----------------------------------
5.000ns    | 4.837ns    | 3
----------------------------------
```

# Key Points

- Enable technology mapping and logic estimation in the Linker tab
- Choose a chip and package to get accurate results
- You need to place and route to get completely accurate results
- Always use timing constraints
- Use rate to specify constraints for all the paths in the clock domain
- Use `intime` and `outtime` to specify constraints for interfaces
- DK produces predictable names for nets in the EDIF output
- Reduce multiplexers by reducing the number of different places from which you assign into a variable

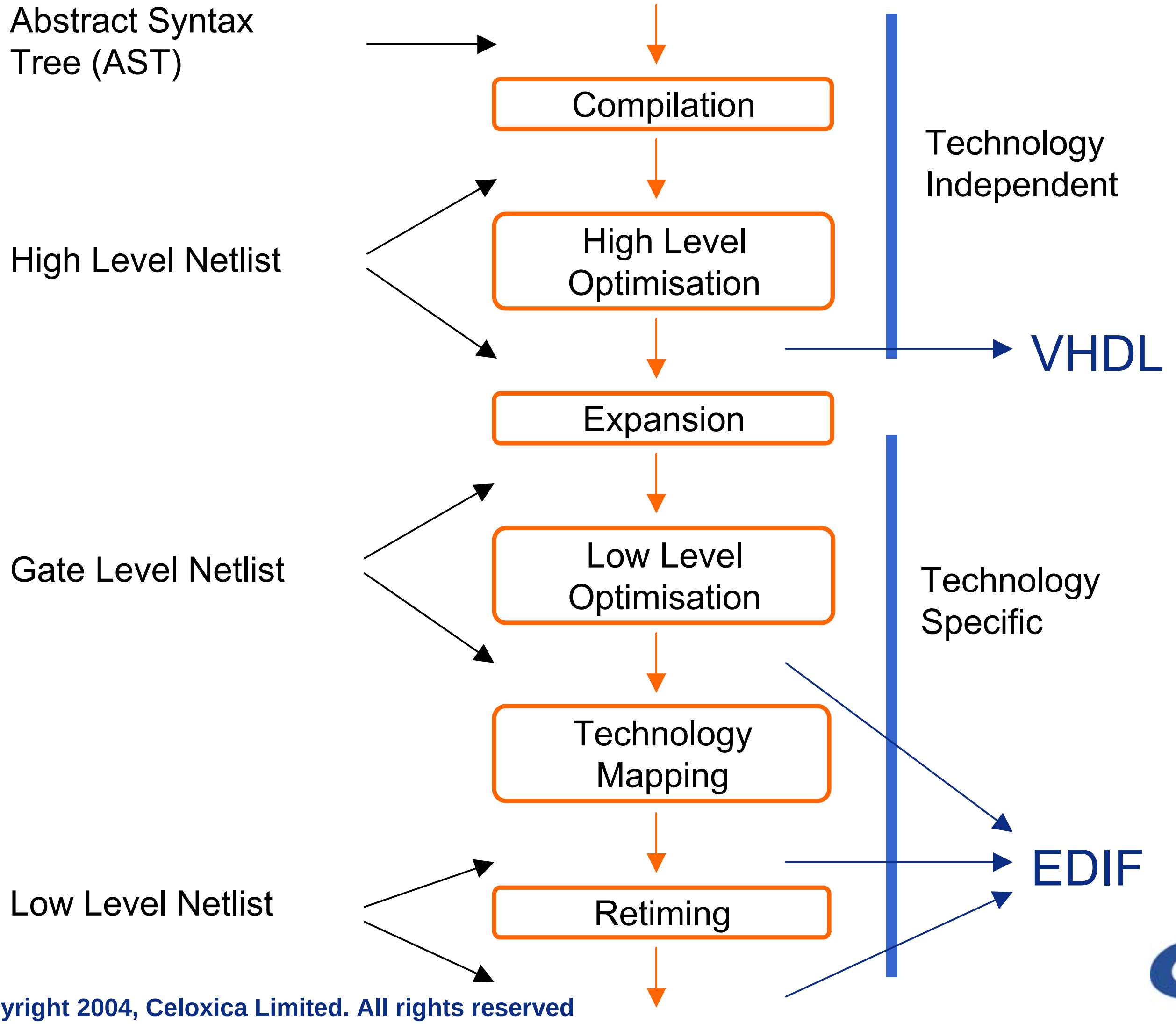
Celoxica™

Software-Compiled System Design

# DK Optimisations

Description of optimisations, their affect on your code and how to make the most of them

# DK Compilation Process



Abstract Syntax Tree (AST)

Compilation

High Level Netlist

High Level Optimisation

Technology Independent

VHDL

Expansion

Gate Level Netlist

Low Level Optimisation

Technology Specific

Technology Mapping

EDIF

Low Level Netlist

Retiming

Celoxica Software-Compiled System Design

# Optimisations

- High Level Optimisations – performed on high level netlist
  - Dead code elimination
  - Common Sub-Expression Elimination (CSE)
  - Rewriting
- Low Level Optimisations – performed on gate level netlist
  - Dead code elimination
  - Common Sub-Expression Elimination (CSE)
  - Rewriting
  - Conditional Rewriting (CR)

# Dead Code Elimination

- Any code that does not have an influence on the output of the design is removed
- Example

```
void main()
{
    unsigned 8 a;
    unsigned 8 b;
    interface bus_out() O(unsigned 8 O = a);
    while(1)
    {
            a++;
            b++;
    }
}
```

- b has no effect on the output of the design, so the flip flops and the adder are optimised away
- The line `b++;` will be replaced with a clock cycle delay

# Dead Code Elimination - How Does This Affect Your Design?

- You don’t need to worry if parts of your code are not used but remain in the code – they will be optimised away
- You can build a design that has redundant code in it that will then automatically optimised away by the compiler
  - This is often easier than writing code that makes sure that only relevant parts are included
- It can take a while for the compiler to build all of the logic that is then optimised away again
  - If you have parts of code that you know are not used, then try to either comment them out or use pre-processor or select statements to determine whether or not to build them
  - This could reduce compilation time for large pieces of extraneous logic

# Common Sub-Expression Elimination (CSE)

- One of the most important optimisations
- Identifies any logic that performs the same function from the same set of inputs and with the same type of output
- Removes the copies and wires the inputs and outputs to the first copy
- Example

```
if (a == 0)
    b++;
else
    delay;
c = (a == 0);
```

- Only one copy of the comparison of a with 0 will be built and wired to where the result is needed
- Can’t identify all possible different ways of writing the same thing

# CSE – How Does This Affect Your Design?

- You don’t have to try to explicitly re-use a single piece of code that performs the same function as elsewhere
- You may think that 3 copies of a++ will be built, but CSE will see that it is being asked to do the same thing on the same register 3 times
- Only one copy of a++ will be built and the enable OR-ed between three control states

```
while(1)
{
    a++;
    while (a != 10);
    {
        par
        {
            a++;
            b = ~b;
        }
    }
    a++;
    delay;
}
```

# CSE – How Does This Affect Your Design? - 2

- This idea scales to a whole macro called with the same set of inputs
- This does not break the rule that a macro creates a discrete piece of logic for each call
  - Any local variables within the macro are not identified as being the same registers or wires between calls to the macro
- In the macro on the next slide, one copy of `a = ~a;` is built
- But there are two different registers for the local variable b
  - So there will be two different versions of `b = ~b;`
- If you want to share the local variables, then you should use a function

Celoxica
Software-Compiled System Design

# CSE – How Does This Affect Your Design? - 3

```
set clock = external;

macro proc Thingy(a)
{
    unsigned 1 b;

    b = a;
    b = ~b;
    a = b;
    a = ~a;
}
```

```
void main()
{
    static unsigned 1 a = 0;
    interface port_out()
        b_out(a);

    while(1)
    {
        Thingy(a);
        Thingy(a);
    }
}
```

- Only `a = ~a;` is the same between calls and is optimised

# Rewriting and Conditional Rewriting

- Rewriting
  - Removes gates that always have the same output value
  - Reduces the number of inputs to gates where possible
  - Example

```
if (a || ~a) // if (1)
```

- Conditional Rewriting
  - Applies automatically generated test patterns to the netlist
  - Discovers if a gate (or input) is ever used, and removes it if not
  - Example

```
if (a && (b || a))
```

    - □ Removes b from the expression as it has no effect on the outcome

- How do these affect your design?
  - You are not required to simplify your Boolean equations to the simplest form possible

# Shift Register Inference

- Recognises flip-flops that are used as shift registers
  - Maps the flip-flops into SR16 LUTs in Xilinx FPGAs
  - Each flip-flop can only be read into the next flip-flop in the chain
  - Each flip flop can only be written to by the previous flip-flop in the chain
- Altera's Quartus-II will recognise a shift register built of flip-flops and map it to a shift register constructed from a RAM block (option)
- DK breaks off last bit as a flip-flop to improve timing
- Examples
  - Produces 1 SRL16 and 1 Flop-Flop

```
static unsigned 16 a = 0;
while(1)
{
    a = a >> 1;
}
```

  - Produces 1 SRL16 and 1 flip-flop

```
delay;delay;delay;
```

# Optimising Block RAM Usage

- By default, DK inverts the clock to internal RAM on the FPGA
  - This enables it to maintain the single cycle access paradigm
  - Address is set in the first half on the main clock cycle, data is read on the second half, with the inverted clock transition in the middle
  - Effectively halves the maximum clock speed at which you can clock the RAMs
- DK3 introduces a new optimisation to remove the inverted clock
  - Only works on some RAM resources – see DK Help
- Enable memory pipelining transformations **option must be checked in the** Synthesis **tab**
  - Enabled by default for Altera and Xilinx

# Optimising Block RAM Usage - 2

- In order for it to work
  - The memory must always be read into a single uninitialised register, and nowhere else
  - Nothing else must write to this register
- There are examples of how to write such code in the DK Help
- Tip - you can improve performance further, if necessary, by registering the output of the RAM again
  - The clock to output time for a flip-flop is significantly less than that for a block RAM
- If you are using RAM resources where this optimisation is not supported, you can write your own interface to the RAM
  - Use CoreGen (Xilinx) or MegaFunction Wizard (Altera) to create a resource
  - Write macros for pipelined read and pipelined write

# Using the Optimiser

- Learn to rely on the optimiser
  - Registers that are not used
    - e.g.in array
  - Bits of a register that are never read from/written to
    - e.g. input width to 2-d array smaller than output width
  - Sections of code that are never called
  - Shift register inference
  - Logic that has no effect on the output
- Write some small examples yourself and look at the logic estimator report or the EDIF
- Example of pipelined adder tree
  - Some registers not used at all
  - Inputs smaller than outputs – bits at each stage optimised away
  - Constants propagated through tree will be optimised away
  - Redundant adds removed

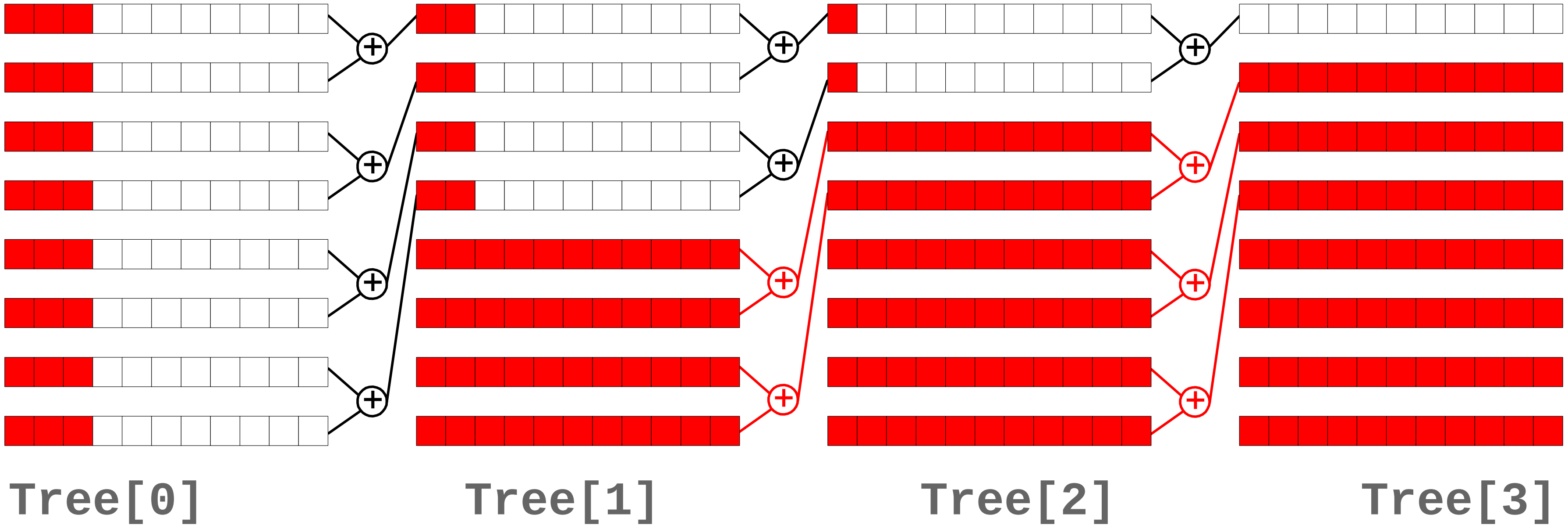
# Using the Optimiser - Example

- Consider the adder tree we saw earlier in the course
- The inputs are 8 bits and the output is 11 bits to hold the maximum value

```
static unsigned 11 Tree[4][8];

while(1)
{
    par(i=0;i<3;i++)
    {
        par (j=0;j<7;j+=2)
        {
            Tree[i+1][j/2] = Tree[i][j] + Tree[i][j+1];
        }
    }
}
```

# Using the Optimiser - Example

- The operations and storage in red are optimised away
- It is much easier to let the compiler do this than to try to hand-optimise it yourself



Celoxica
Software-Compiled System Design

# Key Points

- Dead code elimination will optimise away any code that has no influence on any outputs
- Common sub-expression elimination will remove copies of the same piece of logic with the same inputs and outputs
- Rewriting and conditional rewriting mean that you don’t have to minimise your conditions to the simplest form
- DK infers SRL16s from shift registers for Xilinx parts
- DK will remove the inverted clock on a block RAM when it is read into only one register
- Learn to rely on the optimiser – saves design effort

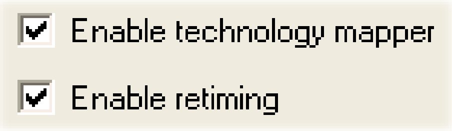
Celoxica
Software-Compiled System Design

# Retiming and ALU Mapping

How retiming works and how to program for it, ALU mapping

# Retiming

- Powerful new option to improve performance
  - Enable it in the Project Settings → Synthesis tab
  - Enabled by default on Xilinx, disabled on Altera
  - Technology mapping needs to be enabled

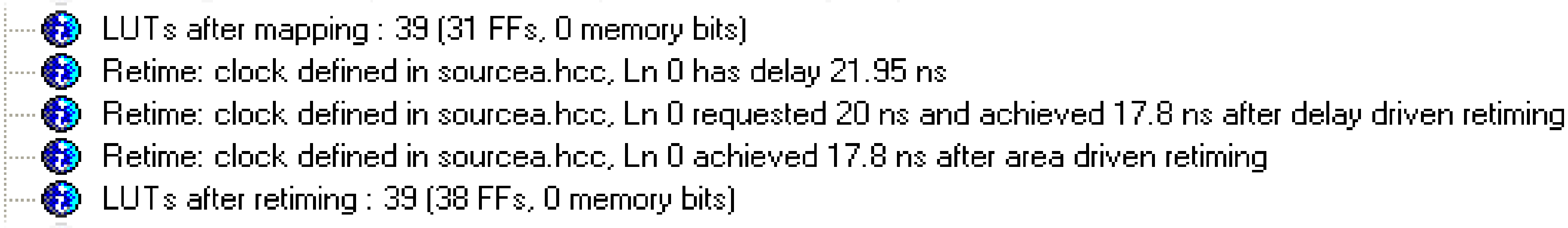


☑ Enable technology mapper

☑ Enable retiming

- Moves registers through logic to minimise logic depth
  - You don’t have to worry so much about getting it right yourself
- Need to specify clock rate on the clock specification
  - DK will minimise logic usage for target clock rate

```
set clock = external with {rate = 50};
```



```
LUTs after mapping : 39 (31 FFs, 0 memory bits)
Retime: clock defined in sourcea.hcc, Ln 0 has delay 21.95 ns
Retime: clock defined in sourcea.hcc, Ln 0 requested 20 ns and achieved 17.8 ns after delay driven retiming
Retime: clock defined in sourcea.hcc, Ln 0 achieved 17.8 ns after area driven retiming
LUTs after retiming : 39 (38 FFs, 0 memory bits)
```

# Assembly Line Analogy Revisited

- The retiming tool knows how long each atomic task takes
- It can automatically determine how long each task in your program will take and move the tasks around the *whole* assembly line
- You can add or remove workers or vary the conveyor belt speed easily: the retiming tool will rebalance the tasks amongst the workers to meet the conveyor belt speed
- Imagine you have several assembly lines in a factory: the retiming tool can move workers around the whole factory to balance all of the assembly lines for the given conveyor belt speed

# How the Retiming Tool Works

- It first works out the boundaries to which it can retime
- It moves all the flip-flops as far as possible in one direction
- It then moves flip-flops back and adds flip-flops to create a circuit that meets the timing requirements
  - This circuit then has a very large number of flip-flops
- Finally, it calculates the minimum number of flip-flops that can be used to implement the circuit
- There is more detail in the white paper at http://www.celoxica.com

# Interpreting Retiming Results

- Flip-flop count
  - The flip-flop count after retiming may well be much higher than beforehand
  - As the speed of the design goes up, so does the flip-flop count
  - Registers have been moved from the inputs to an arithmetic operator to the body of the logic
  - FPGAs are flip-flop rich, so this is generally not a problem
  - Specify an accurate clock speed so that DK will minimise flip-flops correctly once it has achieved the target clock rate – may require tweaking
- Maximum clock speed
  - DK maximum clock speed result may not match place and route tool report
  - DK has accurate information about the component delays and a model of the routing delays
  - Only after placement and routing can you get an accurate figure for the routing delay
  - Generally the DK report is accurate but can become quite inaccurate as the device utilisation nears the limits of the chip

# Limitations of Retiming

- There are boundaries through which DK cannot move flip-flops
  - Clock domains
  - ALU resources
  - RAM resources
  - Manually instantiated components
  - Pads (pins)
- Use the `retime = 0` specification to prevent DK from moving a specific register
  - Important when using flop-flops to resolve metastability on interfaces

```
unsigned 8 x with {retime = 0}; //don’t move this register
```

- Retiming cannot create or destroy layers of registers, and does not change the functionality of the circuit
- Initialised variables in your Handel-C can restrict the retiming moves available

# Programming for Retiming

- Take an example of a multiplier that is holding up your whole pipeline
- You could implement your own pipelined multiplier
  - You'd have to know how a multiplier works
  - You'd have to break it down into stages
  - Tweaking the multiplier by hand would be a difficult programming task
  - Writing the multiplier with a programmable pipeline length would be an even more complicated task
- With retiming, you don't have to understand how the multiplier works
  - You can let the retiming tool do all the hard work
  - A task that could take several days (or more) now only takes several minutes of experimentation

# Programming for Retiming

- Quick and easy method for creating fast, efficient designs
  - Automatic pipelining
  - For a small task, using retiming can achieve results that are just as efficient as doing it by hand
  - For a whole project, it can achieve results that are far superior to those that you can achieve by hand in any useful timescale
- You can change the pipeline length without having to fiddle with balancing the logic yourself
- Method
  - Write a combinational expression
  - Copy the result through a number of registers in parallel
  - DK will balance the logic between the registers

# Programming for Retiming - 3

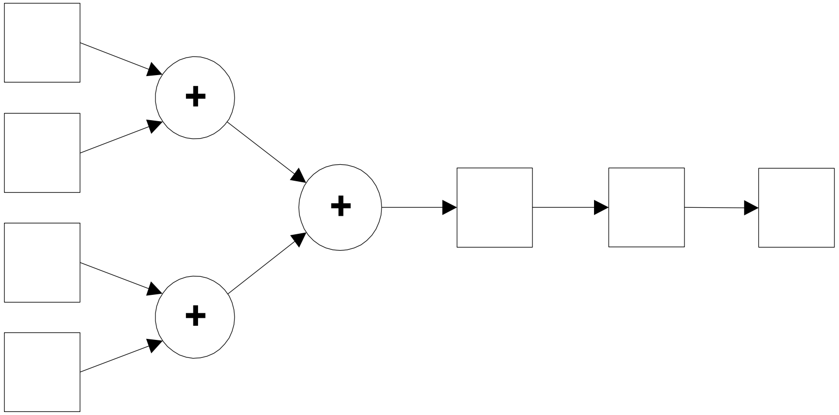
- Example: Adder tree

```
unsigned 8 Inputs[4];
unsigned 11 Results[3];
while(1)
{
    par
    {
        Results[0] = (0@Inputs[3]) + (0@Inputs[2]) +
                     (0@Inputs[1]) + (0@Inputs[0]);
        Results[1] = Results[0];
        Results[2] = Results[1];
    }
}
```

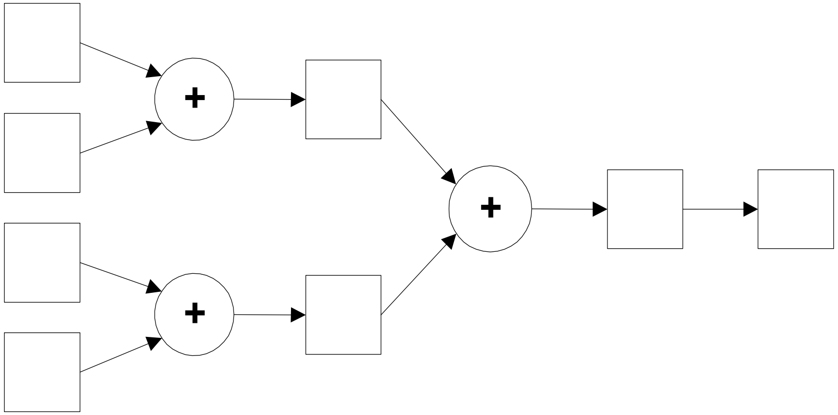
- What happens in simulation?
  - Simulation does not change
  - You will see the calculated value after the first clock cycle, as per the code written
  - The result will be propagated through the registers

# Programming for Retiming - 3

- Original Design
  - Result copied through registers



- Retimed Design
  - Registers moved through design
  - Extra register added
  - Note that DK retimes at the bit level
  - Can achieve better results than by hand



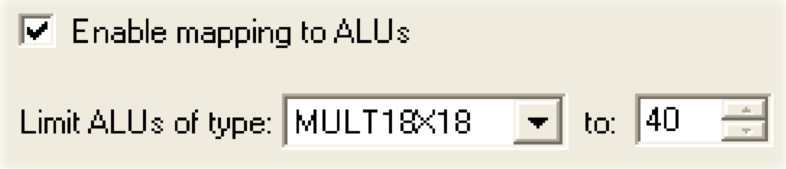
Celoxica
Software-Compiled System Design

# Automatic Pipelining Macro

```
macro proc Pipeline (NumStages, Input, Output)
{
    unsigned Stages[NumStages];

    while(1)
    {
        par
        {
            par (i = 0; i < NumStages-1; i++)
            {
                Stages[i] = Stages[i+1];
            }

            Stages[NumStages-1] = Input;
            *Output = Stages[0];
        }
    }
}
Pipeline (5, a+b+c+d, &e); // call the macro
```

# ALU Mapping

- Xilinx and Altera have embedded ALUs in their devices
  - 18-bit multiplier blocks on Xilinx Virtex-II, Virtex-II Pro, Virtex-II Pro X, Spartan-3
  - 9/18/36-bit DSP (multipliers and adders) blocks on Altera Stratix, Stratix GX, which can be used as a MAC
- DK can automatically map any multiplies in your code to ALUs
- Enable this feature in the Project Settings → Synthesis tab
  - It is disabled by default



☑ Enable mapping to ALUs

Limit ALUs of type: MULT18X18 to: 40

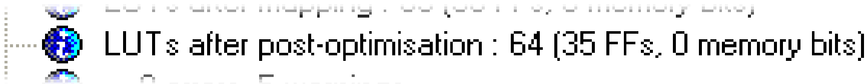
- You must specify a chip in the project settings so that DK knows how many are available
  - Otherwise option will be greyed out

# ALU Mapping - 2

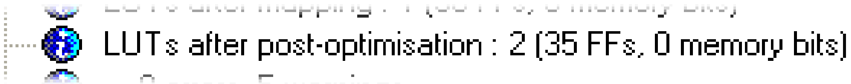
- You can still explicitly instantiate ALUs
- Watch out for automatic and manual instantiations requiring too many multipliers
- DK defaults to being able to use all of the multipliers available
    - You can limit the maximum number that it will use automatically by entering a lower number in the box
- Logic Estimator will tell you how many you are using
    - Place and Route tools will tell you how many in total, including manually instantiated ones
- Can clash with number of Block SelectRAMs used in Xilinx devices
    - Each ALU is situated next to a Block SelectRAM
    - SelectRAM memory can only be used up to 18 bits wide when the multiplier is used

# ALU Mapping - 3

- You will notice a reduced LUT count in the output

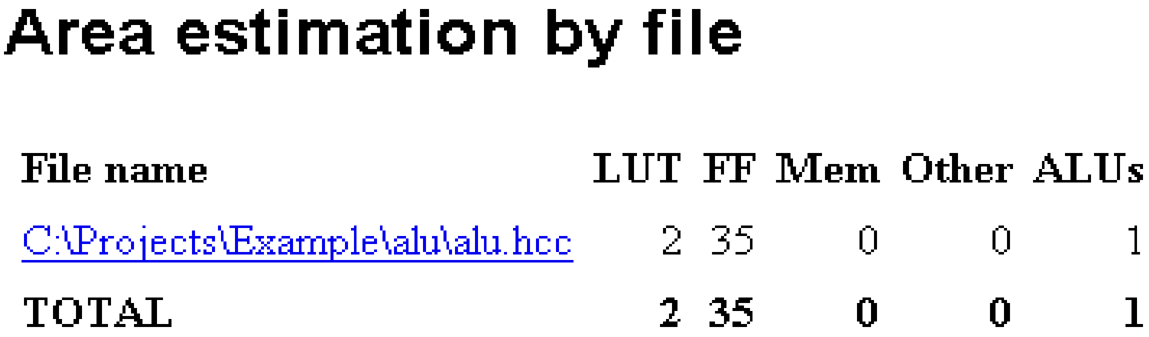


LUTs after post-optimisation : 64 (35 FFs, 0 memory bits)



LUTs after post-optimisation : 2 (35 FFs, 0 memory bits)

- You can see how many ALUs are actually used in the logic estimator HTML output



## Area estimation by file

| File name | LUT | FF | Mem | Other | ALUs |
|---|---|---|---|---|---|
| C:\Projects\Example\alu\alu.hcc | 2 | 35 | 0 | 0 | 1 |
| **TOTAL** | **2** | **35** | **0** | **0** | **1** |

# ALU Mapping Limitations

- Currently does not automatically chain them to create a wider multiplier
  - Can be done manually
- Currently only maps multipliers
  - Doesn’t map to MAC in Altera parts
- Current limitations with ALU mapping and retiming both enabled
  - Won’t retime through the multiplier
  - Will instantiate multiplier even if it won’t meet the clock rate

# Key Points

- Must specify chip type, package and speed grade for retiming and ALU mapping to work
- Must specify a clock rate for retiming to work
- Retiming moves registers through combinational logic to minimise logic depth
- Without a clock rate specified, retiming won’t do antything
- The retimed design may many more flip-flops
- Specify correct clock rate to minimise flip-flop utilisation
- Retiming does not change functionality of circuit
- Simulation remains the same
- Program for retiming to automatically create balanced pipelines
- Retiming usually produces better results than hand-balanced pipelines
- ALU mapping maps multipliers in Handel-C source to embedded multipliers

# Clocks and Interfaces

Interfacing to other code, clock interfaces and multiple clock domains

# Interfacing to External Logic

- Handel-C can be either the top-level or a component instantiated from elsewhere
- Handel-C as top-level – custom interface type
  - Used for interfacing to
    - □ VHDL/Verilog code
    - □ EDIF black-boxes
    - □ FPGA intrinsic components e.g. OBUFs, MULT18X18
- Handel-C as component – port interface type
  - Component in
    - □ VHDL/Verilog design
    - □ Another Handel-C project
- Mixture
  - Your project can be instantiated by external code using custom interface sorts, but still instantiate some other components itself

# Handel-C as Top-Level

```
interface ComponentName (InputPorts) InstanceName(OutputPorts);
```

- `ComponentName` is the name of the type of block to be instantiated
- `InstanceName` is the name of the particular instance
  - You can have any number of instances of the same component
  - Each declaration must have identical port lists
- `InputPorts` and `OutputPorts` can contain any number of ports
- Each item in `InputPorts` and `OutputPorts` connects to the named port on the external block
  - Use `busformat` specification to make DK produce a port list with the correct bracketing and bit order for the buses on the component
- Can connect to an external block from two clock domains
  - Interface must be defined in an *.hcc* file with no clock
  - Useful for things like FIFOs, DCMs/PLLs

# Xilinx Multiplier Interface Example

- Example in PDK

```
macro proc xilinxmult(dest, a, b)
{
    interface MULT18X18(int 36 P with {busformat="B<I>"})
        Multiplier(int 18 A=(int)a with {busformat="B<I>"},
            int 18 B=(int)b with {busformat="B<I>"});

    dest = Multiplier.P;
}
```

- Each call to this macro will create a new instance of a MULT18x18

# Handel-C as Component

```
interface port_in(InputPort) InterfaceName();
```

- Ports into the Handel-C block
- Only one input port allowed per interface
- The name of the input port must be unique in the design

```
interface port_out() InterfaceName(OutputPort);
```

- Ports out of the Handel-C block
- Only one output port allowed per interface
- The name of the output port must be unique in the design

- The top-level design will refer to the port names of the Handel-C component
- The name of the component is the name of the project

# Interfacing to External Logic Example

- Two projects, one which instantiates the other as a component
  - Just to show how to perform interfaces
  - Not to be encouraged as a modularisation technique!
  - ProjectB has one 1-bit output `Out` and two 1-bit inputs `In` and `Clock`
  - ProjectB uses `Clock` as its clock
- Code in Handel-C project ProjectA

```
interface ProjectB(unsigned 1 Out) MyProjectB(unsigned 1 In,
  unsigned 1 Clock = __clock);
```

- Code in Handel-C project ProjectB

```
interface port_in(unsigned 1 Clock) ClockIn();
interface port_in(unsigned 1 In) DataIn();
unsigned 1 DataOutput;
interface port_out() DataOut(unsigned 1 Out = DataOutput);
set clock = internal ClockIn.Clock;
```

Software-Compiled System Design

# Clock Specification

- One clock specification required per main function

```
set clock = external “A12” with {rate = 200};
set clock = internal Expression with {rate = 200};
```

  - Expression can be a signal from another clock domain or an input from another interface

- Handel-C also provides clock dividers

```
set clock = external_divide “A12” 2;
set clock = internal_divide Expression 4;
```

  - Division can be any integer value
  - Simple divider in logic
  - Consider using a DCM/DLL/PLL for fast clocks

# Clocks and Buffers

- Use `__clock` to pass clock signal to an interface
- Use `clockport = 1` specification to tell DK that this is a clock port
  - Otherwise it will complain about combinational loops
- Use the correct clock buffers
  - Place and route tools might complain if the wrong buffers are attached
  - Using the wrong buffers can seriously affect the performance of your design
  - IBUFG is used for global input clock buffers in Xilinx parts
- You can tell DK to insert a specific type of buffer
  - Specify `buffer = “none”` to have no buffer inserted

```
interface bus_in(unsigned 1 Clock) ClockInput() with {buffer =
  “IBUFG"};
```

# Multiple Clock Domains

- Designs with multiple asynchronous clocks are harder to design and debug than single clock designs
  - True in any language
  - Avoid multiple clocks if possible
- When to use them
  - Fast interface clocked by peripheral
    - e.g. Video input chip
- When not to use them
  - Performing tasks that are much slower
  - Talking to a peripheral which is much slower
  - Both can be done using delays
- How to communicate between them
  - You can’t just use a variable
    - Resolving data changes is complicated
    - Metastability can become an issue with asynchronous clocks

# Communicating between Clock Domains

- Channels
  - Simplest solution
  - Use where transfer rate and predictability not a problem
  - For infrequent data or control
  - 4-phase handshaking
  - Time taken for transfer depends on clock relationship
  - Worst case 4 clock cycles in slower domain
- Asynchronous FIFOs
  - For fast, reliable data transfer
  - Good general purpose solution
  - Works regardless of clock relationship
  - Xilinx CoreGen and Altera Megafunction versions
  - Not necessarily the most efficient solution

# Key Points

- Use custom interfaces to instantiate external components from Handel-C
- Use ports to allow another entity to instantiate your project as a block
- A custom interface can be accessed from two clock domains
- Use `__clock` to pass the clock to an interface
- Use the correct clock buffer type on an interface
- Only use multiple clock domains when strictly necessary
- Use a channel for infrequent control communication between clock domains
- Use an asynchronous FIFO for fast transfer of large amounts of data

Celoxica
Software-Compiled System Design

# Advanced DK

Calling ANSI-C from Handel-C, simulation stimuli

# Making Calls to ANSI-C

- DK allows you to talk to C/C++ code within your Handel-C project
- Useful for
  - Simulating peripherals
  - Converting code from ANSI-C
  - Debugging
- Simulation only
  - `#ifdef` out the code for a hardware build
- Allows you to use wealth of common library functions
  - e.g. stdlib, stdio, math
- Allows you to write simulation test-benches

# Making Calls to ANSI-C - 2

- You can
  - Include C/C++ source files in your project
  - Link to existing C/C++ library files
- C types in Handel-C are automatically converted to ANSI-C types
  - `char, short, long`
  - You have to cast everything else
- A call to a C/C++ function takes no clock cycles

# Making Calls to ANSI-C – Example

- C Source

```
void Modify(int *a)
{
    (*a)++;
}
```

- Note that there is no error handling in the example on the next slide
  - Check for EOF, I/O errors etc
- Close any files before ending simulation
  - If you end a simulation without closing an output text file or at least flushing it, then your data may not have been written to file

```
int fflush(FILE *stream);
```

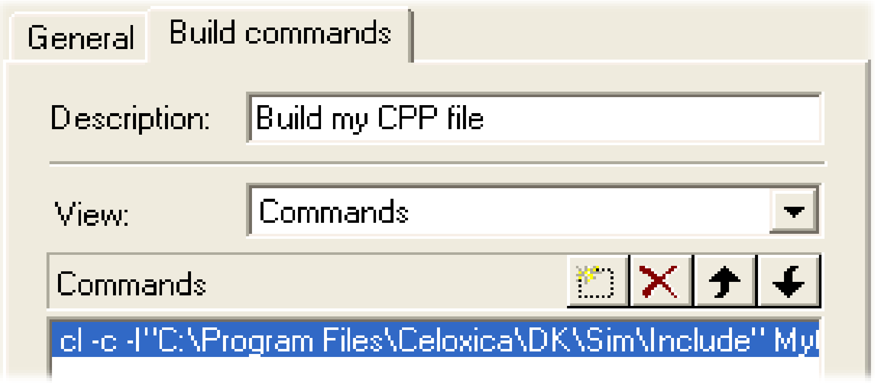
  - Don't call flush too often otherwise file I/O becomes inefficient

# Making Calls to ANSI-C – Example Handel-C Source

```
set clock = external;

extern "C"
{
#define __cdecl
#include <stdio.h>
    // Function prototype
    void Modify(int 32 *a);
}

FILE *OutputFile;
```

```
void main()
{
    static int 32 a = 0;
    OutputFile =
        fopen("OutputFile.bin", "wb");
    do
    {
        Modify(&a);
        a++;
        fputc(a, OutputFile);
    }while(a!=16);
    fclose(OutputFile);
}
```

# Making Calls to ANSI-C – How to Build



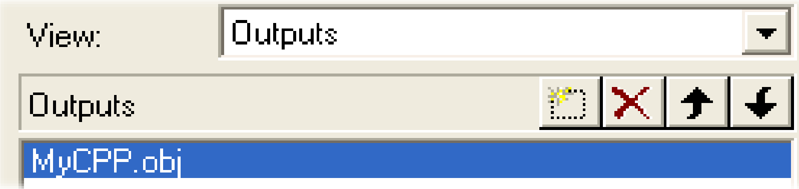
Description: Build my CPP file

View: Commands

Commands

cl -c -I"C:\Program Files\Celoxica\DK\Sim\Include" My

- **Enter a build command**
  - **Go to** Project Settings
  - **Select the C/C++ source file**
  - **Go to the** Build Commands **tab**
  - **Enter a description**
  - **Select** Commands
  - **Enter a command line**
  - ***cl -c -I"C:\Program Files\Celoxica\DK\Sim\Include" MyCPP.c -Fo“MyCPP.obj“***
- **Specify the Output file**
  - **Select** Outputs
  - **e.g.** ***MyCPP.obj***



View: Outputs

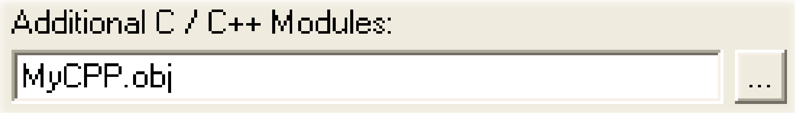
Outputs

MyCPP.obj

Celoxica
Software-Compiled System Design

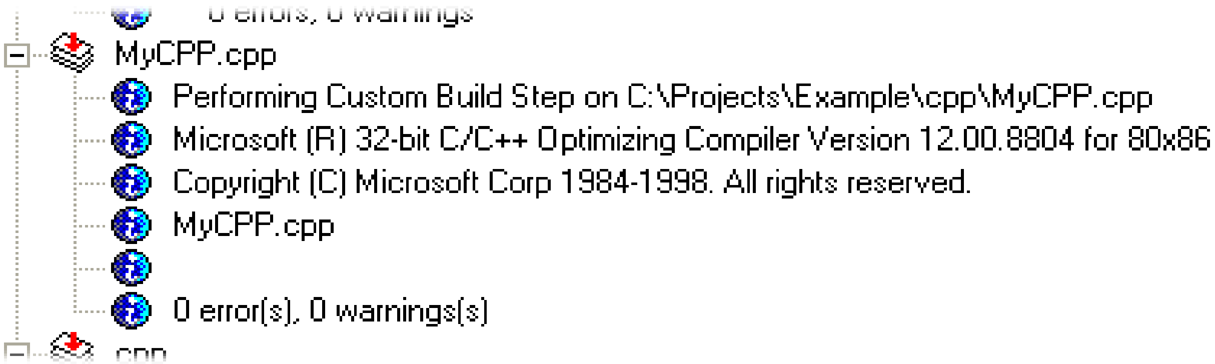
# Making Calls to ANSI-C – How to Build - 2

- **Link to the output**
  - **Go to the** Linker **tab for the project**
  - **Enter the output filename in** Additional C / C++ Modules



Additional C / C++ Modules: MyCPP.obj

- **When you build, DK will display the text output from the C/C++ compiler in the** Output **window**



```
0 errors, 0 warnings
MyCPP.cpp
    Performing Custom Build Step on C:\Projects\Example\cpp\MyCPP.cpp
    Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86
    Copyright (C) Microsoft Corp 1984-1998. All rights reserved.
    MyCPP.cpp

    0 error(s), 0 warnings(s)
cpp
```

# Wide Number Libraries

- Allow you to operate on Handel-C words of any width from C/C++ code
- C *numlib* library
  - Include *DK\Sim\Include\numlib.h*
  - Link to *DK\Sim\Lib\numlib.lib* for Visual C++
  - Link to *DK\Sim\Lib\numlibgcc.lib* for gcc
- C++ *hcnum* library
  - Include *DK\Sim\Include\hcnum.h*

# Wide Number Libraries - Example

- Handel-C

```
extern "C++" int 20 Multiply(int 20 *ArgA, int 20 *ArgB);
```

- C++

```
#include "hcnum.h"

using namespace HCNum;

Int<20> Multiply(Int<20> *ArgA, Int<20> *ArgB)
{
  return (*ArgA) * (*ArgB);
}
```

# Input and Output Channels

- Special channels for reading from and writing to files
- Ignored in hardware build

```
chanin unsigned 8 a with {infile = “in.txt”};
unsigned 8 b with;
chanout unsigned 8 c;
a ? b;
c ! b;
```

- By default, chanout writes to the output pane in DK
  - Specify an `outputfile` to output to file
  - Specify a base to output in a particular base

```
chanout unsigned 8 c with {outfile = “out.txt”, base = 16};
```

  - Output file *out.txt*

```
0x10
0x14 …
```

# Input and Output Channels

- Specify an `infile` for a `chanin`
  - Either use absolute path or place in the project directory (not *DEBUG*)
  - Each value is on a new line, no line termination characters
  - Prefix with `0b` for binary, `0x` for hex
  - Example:

    ```
    12
    0b11
    0x13
    ```
- You can use `with {infile = “in.txt”}` on an interface to attach an input file to that interface
  - This will read an input from the input file every clock cycle

  ```
  interface bus_clock_in(unsigned 8) Data() with {infile = “in.txt”};
  ```
- Similarly, you can attach an output file using `with {outfile = “out.txt”}`
  - This will output a value every clock cycle

# Other Methods for Simulation I/O

- Co-simulation
  - Co-simulation Manager in PDK to co-simulate with other simulators
  - HDL
  - SystemC
  - Instruction Set Simulators ISS
  - Matlab Simulink
  - Multiple DKs
  - And others
- PALSim
  - A simulation platform for PAL resources
  - Memory, VGA output, LEDs, 7-Segment displays
- DSMSim
  - DSM provides abstraction of communications between processor and FPGA
  - DSMSim simulates the data streams
  - You can watch and record the data between the DK simulator and a C/C++ host program

# Key Points

- You can call ANSI C/C++ functions from Handel-C in simulation
- You can link to existing C/C++ libraries
- A call to a C/C++ function takes no clock cycles
- Don’t forget to link to the created *.obj* files
- The wide number libraries allow you to deal with any width of word in C/C++
- `chanin` and `chanout` allow you to input and output text files in simulation
- You can also attach text files to interfaces for I/O in simulation
- PDK provides the Co-Simulation Manager, PALSim and DSMSim for simulation I/O

Celoxica
Software-Compiled System Design

# Appendix

Bonus material for if we get through the rest of the course early

# ANSI-C to Handel-C

- Is your algorithm appropriate for hardware?
  - Not sufficient to just stick in some par statements e.g. FFT
- Approaches
  - Call C from Handel-C and convert function at a time
  - Use co-simulation API
- Frequency of calculation
  - How you implement in hardware depends on the required throughput
    - Size is often traded for speed
    - e.g. FFT – wasteful writing fully pipelined FFT to process 44 KHz audio
  - Consider using a processor (soft or hard) for infrequent calculations
- Minimise differences between the Handel-C and ANSI-C
  - Reduces potential for error
  - Use bit fields for status flags: no need to change in Handel-C
  - No need to change bit manipulation methods

# ANSI-C to Handel-C - Changes to Make

- Make as many of the changes as possible to the ANSI-C source before starting to move to Handel-C
  - You can use the updated ANSI-C to verify your Handel-C design
- Remove dynamic memory allocation
  - Problem common with embedded programming
- Remove casts from one type to another
- Remove any run-time recursion
- Change for loops to while loops
- Convert from floating point to fixed point

# Arithmetic Alternatives

- Distributed Arithmetic
  - Resources on Xilinx website
- Look-up tables
  - E.g. Sine, cosine
- DSP blocks
  - Altera multiply accumulate
  - Xilinx multiplier
- Alternative representations
  - CORDIC

# Design for Reuse

- Make libraries that can deal with
  - Any width – either using width(), calculated widths or inferred widths
  - Signed and unsigned
    - Compile time switch on type
    - Cast to unsigned if just manipulating
  - Any type
    - Use macros that don't care about type
    - Use void pointers as function arguments
- Macros are more versatile than functions for such purposes
- Things to think about
  - Configurable pipeline depth
  - Pipeline synchronisation method
  - Use in different clock domains
  - Multiple instances in same file
- Advice
  - Keep scope of variables local
    - Within function
    - Within file using static

# Why Does the HW not Behave the Same as the Simulation?

- Simulator has inconsistencies with HW for undefined behaviour
- Incorrect pointer arithmetic in your code will corrupt the memory in simulation
- Array or RAM accesses going out of bounds
  - Will corrupt the memory in simulation
  - Will give undefined behaviour in hardware
- Parallel writes to variable
- Read from unassigned signal
- Read from un-initialised variable or pointer
- In any APIs you are using
  - For example, DSM communication

# Structuring Your Project

- Having all your code in one file is OK for small projects, but quickly becomes cumbersome
  - Slows down compilation
  - No modularisation of code
  - Not easy to reuse parts
- Use multiple source files
  - Group related functionality together
  - Only need to recompile each source file when it changes
- Use libraries
  - Re-use the library in other projects
  - Set your main project to be dependent on the library in case you make changes to the library
  - You can provide libraries to others, for example customers, without supplying the source code for the implementation

# Definitions and Declarations

- A declaration advertises the existence of a feature
  - e.g. a variable or function with a given name and type
- A definition specifies the implementation
  - Allocates storage space and optionally initialisation
- Example definitions
  - `int Variable;`
  - `int Function(int Arg) { return Arg*Arg};`
- Example declarations
  - `extern int Variable;`
  - `int Function(int);`
- The `extern` keyword tells the compiler that a named variable exists but is defined elsewhere

# Header and Source Files

- Your header file (.hch) should contain any
  - Declarations of global variables
  - Declarations of global functions
  - Declarations of macro procedures
  - Declarations of macro expressions
  - Declarations of type synonyms (`typedef`)
- Include this from your main project
- Your implementation source file (.hcc) should contain definitions of the above
- It is considered bad practice to use global variables as they prevent effective module abstraction
  - Pass arguments to functions, even if this is more verbose
  - Use static to limit scope within a file
  - Unavoidable in communications between clock domains

# Header File Design

- A header file should be self-reliant
  - It should include any other header files that it requires
- Prevent multiple definitions
  - Sometimes a header file may be included multiple times in a source file through other header files
  - Use a pre-processor #define around the contents of the header file to check if it has already been included

```
#ifndef MY_PROJECT_HEADER_FILE_VER_001
#define MY_PROJECT_HEADER_FILE_VER_001
int a;
… all declarations …
#endif
```

# RAM of Structures

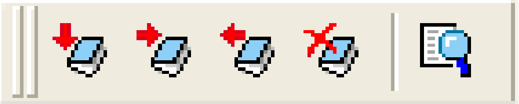
- Only if no channels, semaphores, other RAMs in the structure
- Builds a separate RAM for each element of the structure
- Allows modification of a single element within the structure without having to do an inefficient read, modify, write within a clock cycle
- If you don’t need to modify a single element at a time, consider concatenating the elements of your structure together to form a word and storing that in a RAM instead

# RAM of Structures - 2

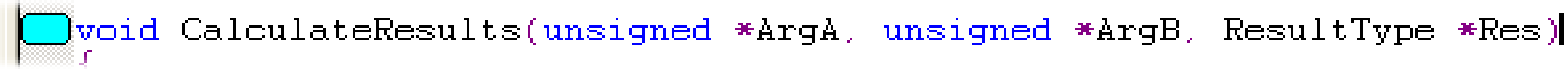
```
struct Pixel
{
    unsigned 8 Red, Green, Blue;
};

ram struct Pixel Buffer[16] with {block = “BlockRAM”}; // 3 block RAMs

par
{
   Buffer[Counter].Red = InputRed;
   Buffer[Counter].Green = InputGreen;
   Buffer[Counter].Blue = InputBlue;
}

ram unsigned 24 Buffer[16] with {block = “BlockRAM”}; // 1 block RAM

Buffer[Counter] = InputRed@InputGreen@InputBlue;
```

# Bookmarks

- Bookmarks are useful for marking lines of code that are of interest
- Bookmark buttons on the Edit toolbar
  - Toggle Bookmark – adds/removes bookmark at current cursor location in the edit window
  - Next and Previous Bookmark – jumps to bookmark
  - Delete Bookmark – deletes all bookmarks
  - Find – find text strings within the file



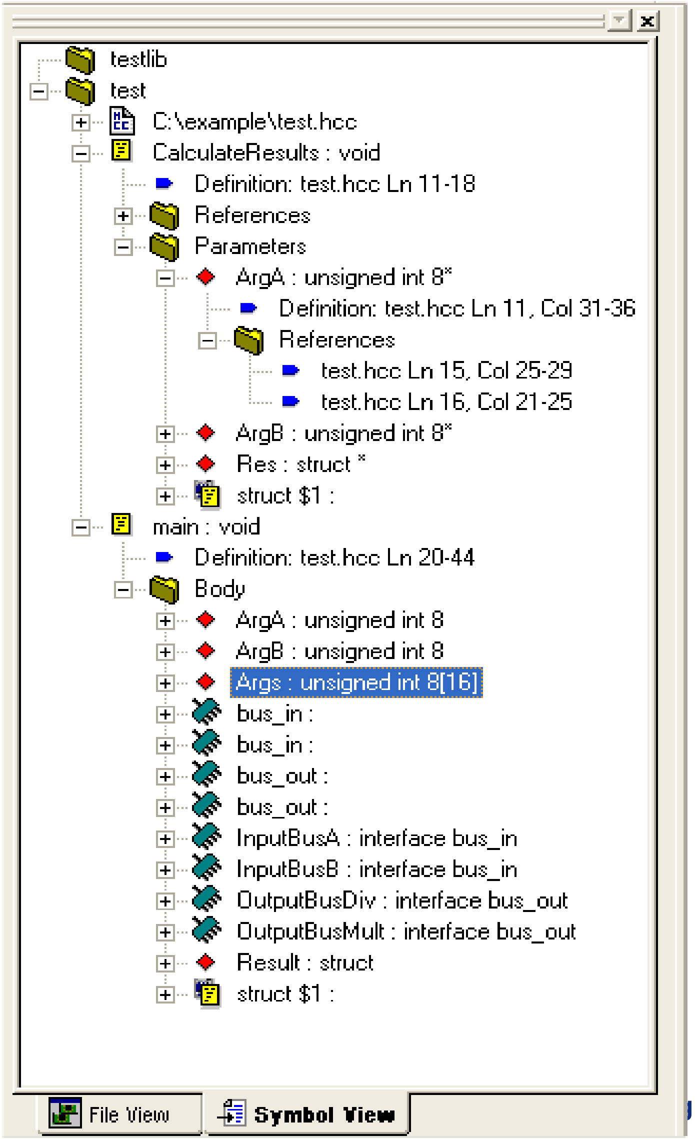
- Click “Toggle Bookmark” on the Edit toolbar to place a bookmark on a line



```
void CalculateResults(unsigned *ArgA, unsigned *ArgB, ResultType *Res)
{
```

- The Find command can add bookmarks for all instances of the search string within the file

# Workspace View – Symbol View

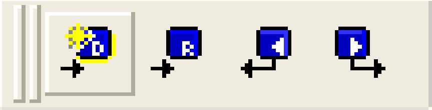


```
testlib
test
  C:\example\test.hcc
  CalculateResults : void
    Definition: test.hcc Ln 11-18
    References
    Parameters
      ArgA : unsigned int 8*
        Definition: test.hcc Ln 11, Col 31-36
        References
          test.hcc Ln 15, Col 25-29
          test.hcc Ln 16, Col 21-25
      ArgB : unsigned int 8*
      Res : struct *
      struct $1 :
  main : void
    Definition: test.hcc Ln 20-44
    Body
      ArgA : unsigned int 8
      ArgB : unsigned int 8
      Args : unsigned int 8[16]
      bus_in :
      bus_in :
      bus_out :
      bus_out :
      InputBusA : interface bus_in
      InputBusB : interface bus_in
      OutputBusDiv : interface bus_out
      OutputBusMult : interface bus_out
      Result : struct
      struct $1 :
```

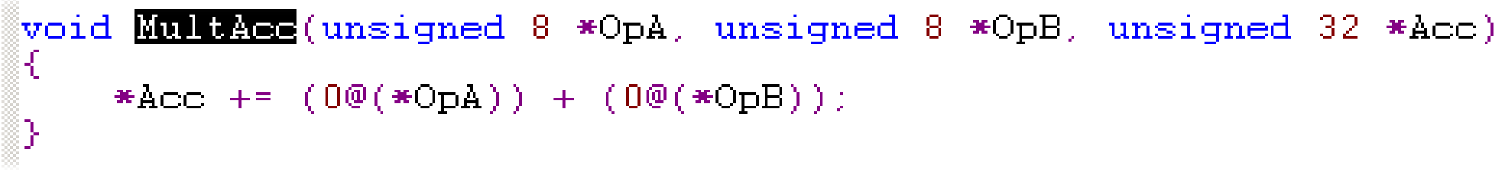
- Available after successful compilation
- Shows
  - Variables, interfaces and functions by project
  - Where in your code they are defined
  - Where they are referenced
- Useful for
  - Navigating large projects
  - Seeing argument types for functions

# Browse Info Toolbar

- Available after successful compilation
- Jump between definitions and references
  - Go to definition
  - Go to reference
  - Switch between references/definitions using “Previous” and “Next”

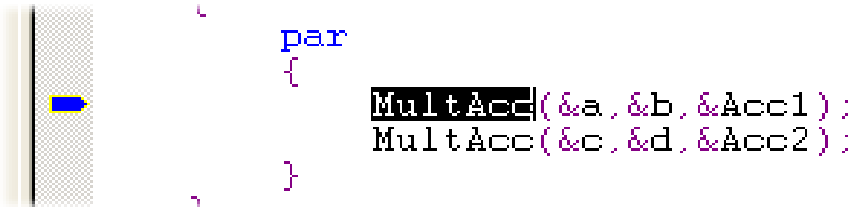


- Example
  - Highlight the function name, e.g. “MultAcc”



```
void MultAcc(unsigned 8 *OpA, unsigned 8 *OpB, unsigned 32 *Acc)
{
    *Acc += (0@(*OpA)) + (0@(*OpB));
}
```

  - Click “Go to Reference” to jump to the first reference



```
        par
        {
            MultAcc(&a,&b,&Acc1);
            MultAcc(&c,&d,&Acc2);
        }
```

# Creating Process Macros

- Client server stuff
  - PDK example
- Continually runs
- Send data to it using macros or channels
  - Example of registered RAM access

# Synchronisation using identical processes

- Use an identical process at the source end as at the sink to work out when to produce data
- Useful when the source process depends on the status of the sink process
  - e.g. graphics generation depends on X and Y coordinates of the video output sync generation process
- Only useful if you do not control the sink process
  - If you have control of the sink process, you can usually just delay this process, which will be more efficient

# Common Warnings/Errors Explained

- Design contains an unbreakable combinational cycle
- Breaking combinational cycle (continue statement) - may alter timing
- Example with combinational cycles – get class to fix it
- Width mismatch
- (Potential) Infinite Recursion
- External tool not found
- Error while compiling simulation output
- Undefined external symbol – don't forget to add the libraries to the linker settings

# New Errors and Warnings

- Indirect self reference
    - Example take width(signal) from signal
- Retiming and ALU mapping

# Last Session

- Work through example design process
- A bit of a C Algorithm
- Design constraints
  - Data rate
  - Data width
- Some image processing
  - Lot of convolution to show error consideration