CHAPTER TWO

Development System Tutorial

The **H.O.T. Works Development System** requires the understanding of the both hardware design and software programming. The following tutorial takes you through the complete process of designing hardware objects and utilizing them in a executable program.

CONTENTS

H.O.T. Works Development System Tutorial

Overview

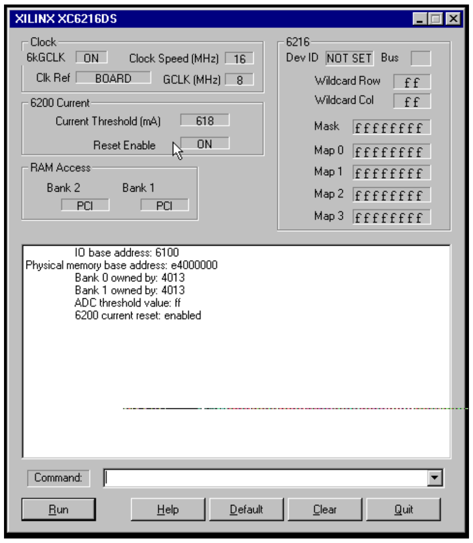
There are various other tutorials and examples included on the H.O.T. Works CD and XACTstep 6000 CD (See Chapter 1). If you are designing in either schematic entry or VHDL and are not not familar with XACTstep 6000 software, we highly recommend you run the XACTstep6000 interactive demo (created by the XC6000 team). This demo is located on your harddisk (if you selected that option) at the following location: You may also run the demo from the XACTstep6000 CD at: `demo/Xactdemo.exe.`

```
../Xact6000/demo/Xactdemo.exe
```

PCITest Program

PCITest Program is you initial doorway to the H.O.T. Works Board. It enables you to probe the C6200RPU, read and write instructions, data and commands to the board and SRAM. The following is an introduction to it's features and use.

⇒ Under Windows, the Hotworks Setup installs a folder on the Start Menu Called **Hotworks**.
In this folder one of the items is the **PCITest Program** OR/
⇒ <u>Run</u> `PCITest.exe` Located in: `C:Hotworks\pci_test` directory.



6kGCLK: ON
Clock Speed (MHz): 16
Clk Ref: BOARD
GCLK (MHz): 8

6200 Current
Current Threshold (mA): 618
Reset Enable: ON

RAM Access
Bank 2: PCI
Bank 1: PCI

6216
Dev ID: NOT SET
Bus:
Wildcard Row: ff
Wildcard Col: ff
Mask: ffffffff
Map 0: ffffffff
Map 1: ffffffff
Map 2: ffffffff
Map 3: ffffffff

```
            IO base address: 6100
Physical memory base address: e4000000
            Bank 0 owned by: 4013
            Bank 1 owned by: 4013
            ADC threshold value: ff
            6200 current reset: enabled
```

Figure 1 PCITest Main Window

The PCITest Program is controlled by commands entered into the command prompt (see Figure 1).

PCITest Commands

Commands to test the H.O.T. Works Board can be entered via the command line of the PCITest window. These core commands interface directly to the XC6200DS functions. The commands can be combined together in Command files allowing the user to create their own command sequence. Currently within PCITest there are a suite of command files allowing core testing of the board functionality of the board.

The following Basic Main Commands can be entered on the command line of PCITest:

ack	ack <mask>	- acknowledge events
comments	comments <on|off>	- switch comment display on/off in command files
command	command <on|off>	- switch command display on/off in command files
event		- gets event status
help	help [topic]	- give help listing on commands
mask	mask <mask>	- sets event mask
obey	obey <file>	- run selected command file
pci		- dumps PCI config header information
pclk	pclk <frequency (MHz)>	- sets on board clock frequency
poll		- waits for interrupt
prg_ref	prg_ref <0|1|2>	- selects reference clock
quit		- exits PCITest
stat	- show status information	
ver	- show version information	

The following Basic 6200 Commands can be entered on the command line of PCITest:

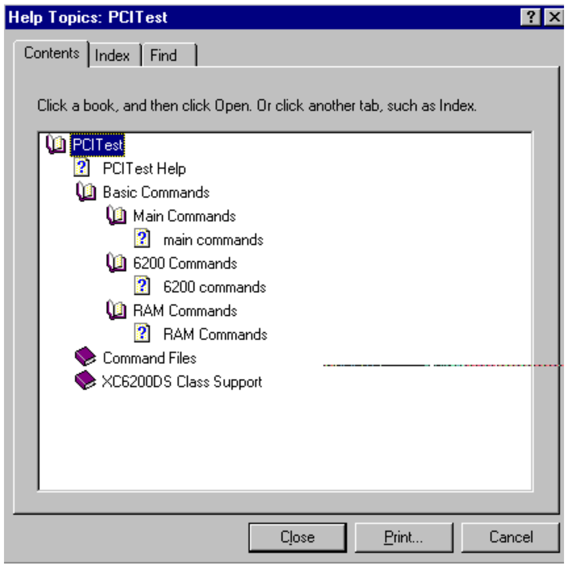
clock	clock <on|off>	- Turn 6200 clock on/off
clear		- Clear 6200
con		- Read con output port
current		- Read 6200 current
download		- Serial download from PROM

getreg	getreg <register>	- Read selected register
lisym	lisym <a|c|s> (a-all symbols c- composite symbols s-sequential symbols)	- List symbols from loaded symbol table
load	load <filename>	- Load CAL file (.CAL)
lsym	lsym <filename>	- Load SYM file (.SYM)
rcol	rcol <col>	- Read column
read	read <addr>	- Read address location from 6200
reset		- Reset 6200
rsten	rsten <on|off>	- Enable/disable 6200 current reset
rsym	rsym <symbol_name>	- Read symbol
setreg	setreg <register> <value>	- Write to selected register where <value> is the 32 bit value to be written and <register> is config - configuration register row_wild - row wildcard register col_wild - column wildcard register mask - mask register map_0 - map register (lower 32 bits) map_1 - map register (second 32 bits) map_2 - map register (third 32 bits) - only on XC6264 device map_3 - map register (top 32 bits) - only on XC6264 device id - device id
step	step <n>	- Step clock
thr	thr <current (mA)>	- Set current threshold
wcol	wcol <col> <data>	- Write to 6200 column where col is the col of address on the 6200 chip design and data is the data to be written
write	write <addr> <data>	- Write to address location on 6200
wsym	wsym <symbol_name> <data>	- Write symbol writes data to the named symbol, the symbol must be present in the already pre-load symbol table list, see lsym. Use of the symbol access allows the user to write and read from a 6200 register without having to manually set the map registers, or specifying the location of the register column within the device.

The following RAM Commands can be entered on the command line of PCITest:

em	em ramaddr [count]	- examine SRAM where ramaddr is the starting address in SRAM and count is the number of locations to be read
fm	fm ramaddr value [count]	- file SRAM where ramaddr is the starting address in SRAM , value is the 32 or 16 bit value to be written and count is the number of locations to be written
bctl	bctl <val>	- set bank control where val is 0 - 4013 owns both banks 1 - 6200 owns bank 1 and 4013 owns bank 2 2 - 6200 owns bank 2 and 4013 owns bank 1 3 - 6200 owns both banks
rspd	rspd <offset> <blocksize>	- PCI read bandwidth
wspd	wspd <offset> <blocksize>	- PCI write bandwidth

Details of all the above can be found in PCITest Help.
⇒ Select the Help Button to explore the Help features.


Help Topics: PCITest

Contents | Index | Find

Click a book, and then click Open. Or click another tab, such as Index.

- PCITest
  - PCITest Help
  - Basic Commands
    - Main Commands
      - main commands
    - 6200 Commands
      - 6200 commands
    - RAM Commands
      - RAM Commands
  - Command Files
  - XC6200DS Class Support

Close | Print... | Cancel

Figure 2 Help Window

Command Files

Command files allow the user to enter their own sequences of commands.

Command files are simple text files made up of a combination of the Basic PCITest Commands and appropriate comments. Each line of the text file must consist either of a PCITest command or a comment, comment lines begin with an exclamation mark !.

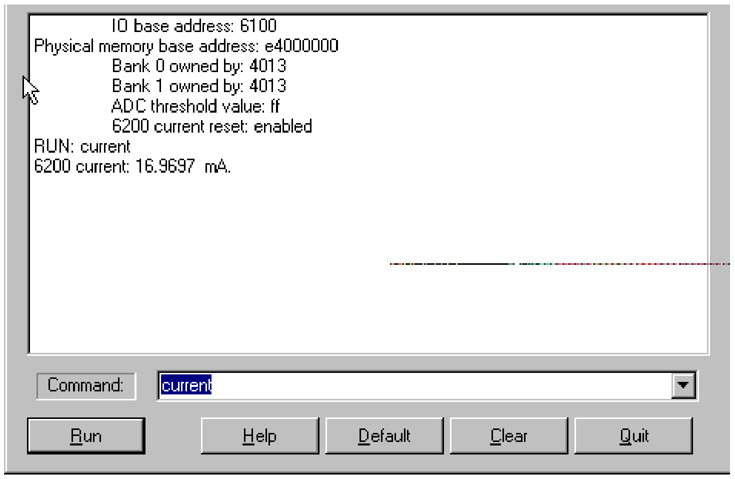
To run the command file simply enter the command obey followed by the command file name.

The following board test command files allow the detection of most hard faults on the PCI board. Each of the tests is fully self-contained.

Running a Command in PCITest

⇒ Running a command file:

There are two ways to enter a command, you may type the command in to the command prompt field OR you can go to the command you wish to run in the Help Window, copy it (Ctrl.C) tnan paste it (Ctrl.V) into the command prompt field. Once the command is entered, press the Run Button on the screen or the Enter enter key on your keyboard.

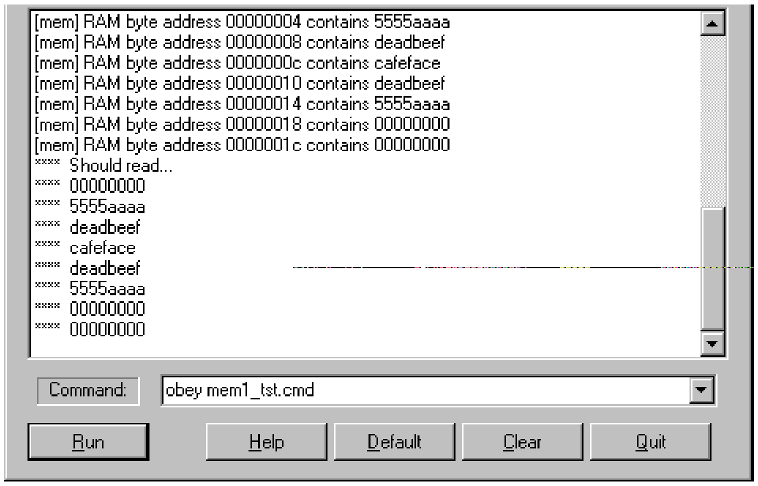


```
            IO base address: 6100
Physical memory base address: e4000000
            Bank 0 owned by: 4013
            Bank 1 owned by: 4013
            ADC threshold value: ff
            6200 current reset: enabled
RUN: current
6200 current: 16.9697  mA.
```

Command: current

*Figure3 Entering a Basic 6200 Command -**Current***

Run a test by entering the command `obey <filename>` at the command prompt, where filename is one of the .cmd files listed below.



```
[mem] RAM byte address 00000004 contains 5555aaaa
[mem] RAM byte address 00000008 contains deadbeef
[mem] RAM byte address 0000000c contains cafeface
[mem] RAM byte address 00000010 contains deadbeef
[mem] RAM byte address 00000014 contains 5555aaaa
[mem] RAM byte address 00000018 contains 00000000
[mem] RAM byte address 0000001c contains 00000000
****  Should read...
****  00000000
****  5555aaaa
****  deadbeef
****  cafeface
****  deadbeef
****  5555aaaa
****  00000000
****  00000000
```

Command: obey mem1_tst.cmd

Figure 4 Entering a Test Command - ***mem1_tst.cmd***

The Test Command Files

cur1_tst.cmd - Demonstrates that current sensing and 6200 current reset works

- Resets 6200, enters continuous clock mode, disables current reset, sets current threshold to 200mA and clears any events from previous tests.
- Shows current with no design loaded. (user checks for expected value).
- Loads the current test design into the 6200.
- Shows current with current test design loaded (user checks for expected value).
- Sets current threshold to 50mA, triggering the E_ADC event. Does not reset 6200.
- Shows current (user checks for expected value).
- Turns on RSTEN bit in ADC Threshold Register, resetting 6200 and lowering current to below threshold.
- Shows current (user checks for expected value).

mem1_tst.cmd- Demonstrates that 32bit SRAM accesses work

- Sets 8 words of memory to zeros.
- Writes a pattern into five locations.
- Reads back pattern (user checks for expected value).

mem2_tst.cmd- Demonstrates that 16bit SRAM accesses to bank 1 work

- Sets 8 words of memory to zeros in 32 bit mode.
- Writes a pattern into eight locations in 32 bit mode.
- Overwrites pattern with a different pattern, but only to bank 1 in 16 bit mode.
- Reads back 8 locations only from bank 1 in 16 bit mode (user checks for expected value).
- Reads back 8 locations in 32 bit mode (user checks for expected value).

mem3_tst.cmd- Demonstrates that 16bit SRAM accesses to bank 2 work

- Sets 8 words of memory to zeros in 32 bit mode.
- Writes a pattern into eight locations in 32 bit mode.
- Overwrites pattern with a different pattern, but only to bank 2 in 16 bit mode.
- Reads back 8 locations only from bank 2 in 16 bit mode (user checks for expected value).
- Reads back 8 locations in 32 bit mode (user checks for expected value).

6k1_tst.cmd - Demonstrates that 6200 accesses in continuous clock mode work

- Resets 6200 and enters continuous clock mode.
- Loads comparator design into 6200.
- Reads 6200 configuration register (user checks for expected value).
- Reads current (user checks for expected value).
- Reads column 42 (user checks for expected value).
- Writes pattern into column 42.
- Reads column 42 (user checks for expected value).

- Writes pattern into column 42.
- Reads column 42 (user checks for expected value).
- Writes pattern into column 42.
- Reads CON pins (user checks for expected value).
- Writes pattern into column 42.
- Reads CON pins (user checks for expected value).
- Writes pattern into column 42.
- Reads CON pins (user checks for expected value).

6k2_tst.cmd - Demonstrates that 6200 accesses in single clock mode work

- Resets 6200 and enters single clock mode.
- Loads comparator design into 6200.
- Reads 6200 configuration register (user checks for expected value).
- Reads current (user checks for expected value).
- Reads column 42 (user checks for expected value).
- Writes pattern into column 42.
- Reads column 42 (user checks for expected value).
- Writes pattern into column 42.
- Reads column 42 (user checks for expected value).
- Writes pattern into column 42.
- Reads CON pins (user checks for expected value).
- Writes pattern into column 42.
- Reads CON pins (user checks for expected value).
- Writes pattern into column 42.
- Reads CON pins (user checks for expected value).

6k4_tst.cmd - Demonstrates that 6200 step clock mode works

- Resets 6200
- Loads clock step design into 6200.
- Reads 6200 configuration register (user checks for expected value).
- Reads current (user checks for expected value).
- Reads IOB's (user checks for expected value).
- Switches off GCLK
- Clears design counter.
- Reads IOB's (user checks for expected value).
- Steps into design '1' clock cycles (minimum step count)
- Reads IOB's (user checks for expected value).
- Clears design counter.
- Reads IOB's (user checks for expected value).
- Steps into design '65535' clock cycles (maximum step count)
- Reads IOB's (user checks for expected value).
- Clears design counter.
- Reads IOB's (user checks for expected value).
- Steps into design 'X' clock cycles .
- Reads IOB's (user checks for expected value).

6k5_tst.cmd - Demonstrates that 6200 to Ram interface works (6216 owns memory banks 1 and 2).

- Resets 6200, enters continuous clock mode, disables current reset, sets current threshold to 200mA.
- Writes a pattern into eight locations of RAM (4013 owns RAM).
- Gives ownership of RAM to 6200.
- Loads 6k_ram design into 6200.
- Sets map registers

- Puts RAM design into write mode
- Writes ram address to RAM design
- Writes data to RAM design
- Puts RAM design into read mode
- Writes ram address to RAM design
- Reads data from RAM through 6200
- Gives ownership of RAM to 4013.
- Checks 6200 RAM modifications through 4013 interface

6k6_tst.cmd - Demonstrates that 6200 to Ram interface works (6216 owns memory banks 1 and xc4013 owns bank 2).

- Resets 6200, enters continuous clock mode, disables current reset, sets current threshold to 200mA.
- Writes a pattern into eight locations of RAM (4013 owns RAM).
- Gives ownership of RAM Bank1 to 6200.
- Loads 6k_ram design into 6200.
- Sets map registers
- Puts RAM design into write mode
- Writes ram address to RAM design
- Writes data to RAM design
- Puts RAM design into read mode
- Writes ram address to RAM design
- Reads data from RAM through 6200
- Gives ownership of RAM to 4013.
- Checks 6200 RAM modifications through 4013 interface

6k7_tst.cmd - Demonstrates that 6200 to Ram interface works (6216 owns memory banks 2 and xc4013 owns bank 1).

- Resets 6200, enters continuous clock mode, disables current reset, sets current threshold to 200mA.
- Writes a pattern into eight locations of RAM (4013 owns RAM).
- Gives ownership of RAM Bank1 to 6200.
- Loads 6k_ram design into 6200.
- Sets map registers
- Puts RAM design into write mode
- Writes ram address to RAM design
- Writes data to RAM design
- Puts RAM design into read mode
- Writes ram address to RAM design
- Reads data from RAM through 6200
- Gives ownership of RAM to 4013.
- Checks 6200 RAM modifications through 4013 interface

6k8_tst.cmd - Demonstrates that 6200 serial download works.

- Switches 6200 GCLK off.
- Asserts Serial line low.
- Resets 6200.
- switches 6200 GCLK on.
- Waits for approx. 80ms (until serial download is complete).
- Negates Serial line high.
- Reads xc6200 configuration register (value should equal 0x40)
- Reads serially loaded design counter value (should equal zero).
- Clocks counter design.
- Reads serially loaded counter value (value should equal one).

evt1_tst.cmd- Demonstrates that clock step event works

- Resets 6200 and enters single clock mode.
- Sets ADC threshold to 200mA to prevent ADC threshold event.
- Clears any events from previous tests.
- Reads Event Status (user checks for expected value).
- Sends 10 clock pulses to 6200.
- Reads Event Status (user checks for expected value).
- Acknowledges event.
- Reads Event Status (user checks for expected value).

evt2_tst.cmd- Demonstrates that 6200 event works

- Resets 6200.
- Sets ADC threshold to 200mA to prevent ADC threshold event.
- Clears any events from previous tests.
- Reads Event Status (user checks for expected value).
- Loads comparator design.
- Writes a value into column 42 such that CON[1] goes low, signalling an interrupt from 6200.
- Reads Event Status (user checks for expected value).
- Acknowledges event.
- Reads Event Status (user checks for expected value).

evt3_tst.cmd- Demonstrates that current sense event works

- Resets 6200 and enters continuous clock mode.
- Sets ADC threshold to 18mA.
- Clears any events from previous tests.
- Reads Event Status (user checks for expected value).

- Loads comparator design.
- Reads Event Status and current (user checks for expected value).
- Attempts to acknowledge event (but should fail to clear E_ADC bit).
- Reads Event Status and current (user checks for expected value).
- Resets 6200..
- Reads Event Status and current (user checks for expected value).
- Acknowledges E_ADC event.
- Reads Event Status (user checks for expected value).

evt4_tst.cmd - Demonstrates that events work independently

- Resets 6200 and enters continuous clock mode.
- Sets ADC threshold to 200mA.
- Clears any events from previous tests.
- Reads Event Status (user checks for expected value).
- Writes the value 10 to the Step Register (causes E_STEP event)
- Reads Event Status (user checks for expected value).
- Loads comparator design, and writes to column 42 such that 6200 takes CON[1] low to request an interrupt (cause E_6200 event)
- Reads Event Status (user checks for expected value).
- Sets ADC threshold to 18mA (causes E_ADC event).
- Resets 6200
- Reads Event Status (user checks for expected value).
- Acknowledges E_6200 event
- Reads Event Status (user checks for expected value).
- Acknowledges E_ADC event
- Reads Event Status (user checks for expected value).

- Acknowledges E_STEP event
- Reads Event Status (user checks for expected value).

irq1_tst.cmd - Demonstrates that clock step interrupt works

- Resets 6200 and enters single clock mode.
- Sets ADC threshold to 200mA to prevent ADC threshold event.
- Clears any events from previous tests.
- Reads Event Status (user checks for expected value).
- Sends 10 clock pulses to 6200.
- Polls for interrupts.

irq2_tst.cmd - Demonstrates that 6200 interrupt works

- Resets 6200.
- Sets ADC threshold to 200mA to prevent ADC threshold event.
- Clears any events from previous tests.
- Reads Event Status (user checks for expected value).
- Loads comparator design.
- Writes a value into column 42 such that CON[1] goes low, signalling an interrupt from 6200.
- Polls for interrupts.

irq3_tst.cmd - Demonstrates that current sense interrupt works

- Resets 6200 and enters continuous clock mode.
- Sets ADC threshold to 18mA.
- Clears any events from previous tests.
- Reads Event Status (user checks for expected value).
- Loads comparator design.
- Polls for interrupts.

- Reads current.
- Resets 6200..
- Reads current .

irq4_tst.cmd - Demonstrates that interrupts work independently

- Resets 6200 and enters continuous clock mode.
- Sets ADC threshold to 200mA.
- Clears any events from previous tests.
- Reads Event Status (user checks for expected value).
- Writes the value 10 to the Step Register (causes E_STEP event)
- Polls for interrupts
- Loads comparator design, and writes to column 42 such that 6200 takes CON[1] low to request an interrupt (cause E_6200 event)
- Polls for interrupts
- Sets ADC threshold to 18mA (causes E_ADC event).
- Polls for interrupts.
- Resets 6200.

pclk_tst.cmd - Demonstrates that programmable download works.

- Runs the programmable clock functions (programmable frequency set to 1MHz).
- Runs the programmable clock functions (programmable frequency set to 66MHz).
- Runs the programmable clock functions (programmable frequency set to 33MHz).

ref_tst.cmd - Demonstrates that reference clock selection works.

- Runs the programmable reference clock functions (reference clock set to PCI clock).
- Runs the programmable reference clock functions (reference clock set to on-board oscillator).

prf1_tst.cmd - Measures raw read and write bandwidth from and to SRAM

- Performs block writes (length 4096 bytes) to SRAM for 5 seconds, measuring bandwidth.
- Performs block reads (length 4096 bytes) from SRAM for 5 seconds, measuring bandwidth.

XC6200DS Class Functions

The class support allows users to interface to the PCI card from their own C++ code with a few simple function calls. All the source code for the class is contained within the file PCI6200.cpp with its attached header file PCI6200.h. The code uses the hotworks.vxd device driver interface to the board. The following classes provided a C++ interface to the XC6200 PCI board (XC6200DS). The code uses the hotworks.vxd device driver to interface to the PCI board.
There are three basic classes which make up the XC6200DS interface:

1. XC6200DS contains support for the following aspects of the board interface:
    - reading and writing to the XC6200 Control Registers
    - loading designs
    - reading/writing to the chip through the FastMap interface
2. PCIBoard contains support for the following aspects of the board interface:
    - writing to the PCI board IO_space registers
    - clock management
    - current management
3. PCICore contains support for the following aspects of the board interface:
    - device driver memory initialization
    - device driver interface
    - low level read and write to the memory mapped board

XC6200DS Class

Load Cal:

```
int    loadCalFile(char *filename);
```

Column:

```
void setColumn(byte addr, word  data);
word getColumn(byte column);
```

Config Register:

```
void setConfig(byte config);
byte getConfig(void);
```

Bus Width:

```
void setBusWidth(unsigned int  width);
int getBusWidth(void);
```

Row Wildcard Register:

```
void setRowWildcard(byte  row_wildcard);
byte getRowWildcard(void);
```

Column Wildcard Register:

```
void setColumnWildcard(byte  row_wildcard);
byte getColumnWildcard(void);
```

Mask Register:

```
void setMask(word  mask);
word getMask(void);
```

Map Register:

```
void setMap(byte offset, word data);
void setMap(MapRegister &MapReg);
word getMap(byte offset);
void getMap(MapRegister &MapReg);
```

Device Id:

```
void setDeviceId(byte device_id);
byte getDeviceId(void);
byte scanDeviceId(void);
```

PCIBoard Classes

Reset:

```
void   reset(void);
```

Clear:

```
void   clear(void);
```

Clock:

```
void   clockOn(void);
void   clockOff(void);
void   clockStep(unsigned int  steps = 1);
```

Current:

```
void   setAdc(byte reset_enable)
void setAdcThreshold(char adc_thr)
byte getAdc(void)
char getAdcThreshold(void)
char getResetEnable(void)
double  adcToMa(byte ad_value)
```

CON:

word getCon(void);

Bank Control:

void setBankControl(byte state);
void setBankControl(byte bank1,byte bank2);

Clock Frequency:

void sendClockData(word bits, int count);
void sleep(clock_t wait);
void sendProtocolWord(void);
float setClockFreq(float ref_mhz, float desired_mhz);

Events:

void setEventMask(word mask);
word getEventStatus(void);
void acknowledgeEvents(word mask);

PCICore Classes

The following functions interface to the device driver for direct write and read access to the PCI board.

PCI Read/Write:

void write(word addr, word data);
word read(word addr);

- write and read to 6200 chip equivalent to write6200, read6200 except they use the device IO control access to the chip, which is slower than the inline assembly code access.

void write6200(word addr, word data);
word read6200(word addr);

- write and read to 6200 chip using fast assembly code access.

void writeIOSpace(byte addr, word data);
word readIOSpace(byte addr);

- write and read to the IO Space registers

void writeRAM(word addr, word *data, word count);
void readRAM(word addr, word *data, word count);

- write and read to the on board SRAM, data is a pointer to a buffer of length count of data words.

The following functions give user access to the memory mapping of the low level device interface.

Device Driver Interface:

inline byte getIOBase(void)

- returns the address of the I/O space.

inline word getPhysicalBase(void)

- returns the physical offset of the PCI board

inline word getMembase(void)

- returns the memory mapped location of the board, this address is used for all board read and writes.

PCICONFIG_S * dumpPCIConfigHeader();

- returns the entire PCI header structure.

In the Run-Time Tutorial section of this chapter you will find further examples of using PCITest.

WebScope Program

WebScope, is a Java-based tool for real-time design emulation (hardware debugger). WebScope enables one to load and test designs on your host system, over the network or the internet. Source code is provided for the quick understanding of the programming techniques for H..O.T. Works.

The On-line WebScope Document (HTML Format) can be found in:

```
C:\Hotworks\webscope\webscope.html
```

The files which come with WebScope are:

Java Class files: Since WebScope is a Java program, it exists as a group of compiled classes. These are dynamically downloaded by either the Java interpreter in the case of a standalone application, or by the Web browser in the case of an applet. Both of these approaches actually use the same files.

Bitmap Files: The other files shipped with WebScope are the bitmap files (in GIF format). These define many of the graphic elements in WebScope. They may also be used in the on-line documenation.

HTML Files: There are some HTML files with Webscope. These are the applet launcher files and this help file.

CAL Files: These files contain 6200 configuration data which can be downloaded to the PCI/6200 hardware using WebScope. The CAL files shipped with WebScope are typically small demonstration circuits. See Sample Designs, below for more information on these files.

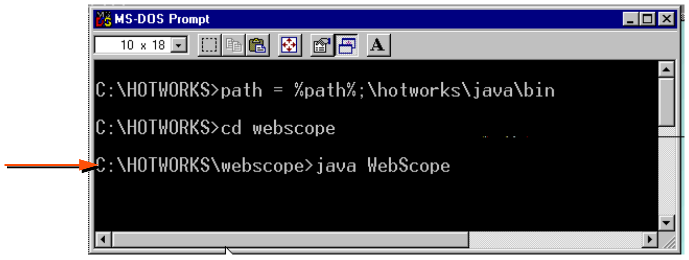
SYM Files: These files contain symbolic information associated with a given CAL file. These symbol files may be loaded into WebScope to support symbolic debugging.

For Remote Access and Applet mode, see the WebScope Doucument `webscope.html`

⇒ Under Windows, the Hotworks Setup installs a folder on the Start Menu Called **Hotworks**. In this folder one of the items is the **Webscope Program** OR/

⇒ Run `webgo.bat` Located in: `C:Hotworks\webscope` directory. OR/

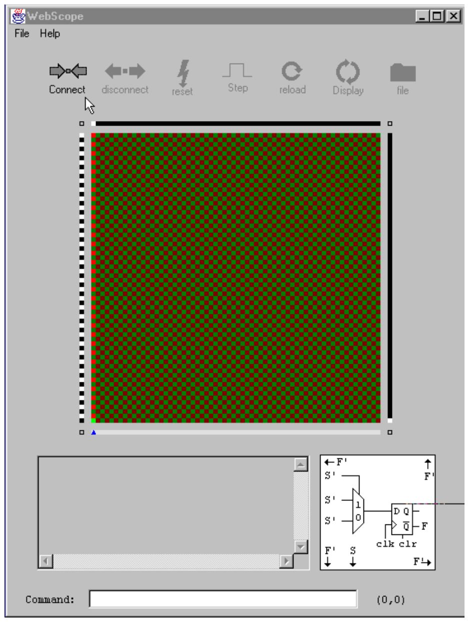
⇒ Run from the DOS prompt the following: (see Figure 5)



```
C:\HOTWORKS>path = %path%;\hotworks\java\bin

C:\HOTWORKS>cd webscope

C:\HOTWORKS\webscope>java WebScope
```

Figure 5

The main window of WebScope will pop up. Use the **connect button** to attach WebScope to the H.O.T. Works Board. The viewer will turn green when connection is sucessful



The User Interface

The user interface of WebScope is broken up into four main regions. The top region of the interface consists of a row of command buttons. The bottom of the display is a simple command line interface based on the Pcitest MS-DOS application. Above this command line is the status portion of the interface. This status portion contains a scrolling text window and a graphical window showing XC6200 cell configuration information.

The primary control provided by WebScope is via the row of buttons across the top of the display. When WebScope is initialized, only the Connect button is enabled. Only after successfully connecting to the H.O.T. Works Board will the other command buttons become active.

The command buttons are described below in more detail below.

Connect

The connect button is the button in the upper left corner of WebScope. It is used to connect to either the H.O.T. Works Board or to the WsServer server. Only after a successful connection can interaction with the H.O.T. Works Board begin. Until a successful connection is estabilshed, no other buttons will function, and no other aspects of the point and click interface will operate.

disconnect

The disconnect button is used to terminate the connection with the target hardware. Once WebScope is disconnected from the H.O.T. Works Board, all functions, except for the connect button, are disabled.   Disconnecting from the target is not a disasterous event. Re-connecting will resume the session where it was left off (assuming someone else hasn't connected to the server and modified the target hardware state).

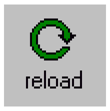
reset

The reset button is used to reset the XC6200RPU to its default state. This is accomplished by toggling the hardware reset line on the XC6200 device. After a successful reset, the graphical display should show all cells set to '1' (green).**Warning**: A reset will cause any configuration data which has been loaded into the device to be lost. While you don't want to accidentally hit the reset button, it is recommended that the system be reset before a new CAL file is loaded.

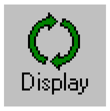
Step

The step button is used to advance the state of the system. Currently, this means that:

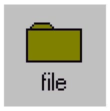
Each entry in the symbol table is accessed in sequence (data read from a column for read-only symbol table entries and written for writable entries). A single clock pulse is sent to the design over the global clock line. This is done primarily for historical reasons. Stepping the design and its relationship to clocking is a somewhat complicated issue. It is recommended that you read the section below on Clocking below.



The reload button is used to re-read all of the data from the target and re-display the result. In practice, you should never really have to use the reload function, but sometimes it is good just to verify data, or to re-draw the screen in the event that the graphics have somehow become damaged. There is one place where the reload button is useful. When developing external software to interface to the H.O.T. Works Board, this software will almost certainly modify the state of the H.O.T. Works hardware. Since WebScope has no way to know that these modifications have taken place, they are not reflected in the WebScope display. Clicking the reload button, however, will display the most recent state of the hardware.



The display button is used to toggle WebScope's primary display. Currently, there are three displays. The first is the graphical display which is the array of XC6200 cells. This is the default display when WebScope is initialized. The second display is the symbolic display. This display gives a table of symbol values which represent columns of cells in the XC6200. The third display is the waveform display. The wavefom display produces strip chart style traces of the variables in the symbol table.



The file button is used to load files into the WebScope. When this button is clicked, a dialog box requesting a file is popped up. The type of file to be downloaded is selected by the buttons in the display. Currently, CAL design files and SYM symbol files can be downloaded. The CAL file will modify the state of the hardware, while the SYM file is used to load symbol table information into WebScope. The loading of symbolic information does not change the state of the hardware.

The Graphical Display

When WebScope is initialized, the default display is the graphical display. This display consists of a grid of 64 x 64 elements, representing the state of the cells in the XC6200. A green cell represents a cell with a logic '1' output; a red cell represents a logic '0'. When WebScope is initialized, a checkerboard pattern of red and green cells is presented. This simply indicates that the system has not connected to the H.O.T. Works Board and is uninitialized. The cell array is surrounded on its four sides by bars. The bar to the left of the cells represents the Map register. This register determines which cells are used during state accesses. Black cells indicate bits masked off by the Map Register, while white cells indicate enabled bits. Grey cells indicate bits that are emabled by the Map Register, but will not appear on the external bus, due to the current bus width setting.

On the top of the cell array is the Column Wildcard register. This register determines which columns of the array are accessed during configuration. The Column Wildcard register is currently displayed assuming an address value of zero. Clicking on any part of the Column Wildcard register causes the value of this register to be displayed in the Status window in the bottom portion of the display. Unlike the Map register, this register exists in a 6 bit encoded form, and thus cannot be modified on a bit per bit basis.

Similarly, the Row Wildcard register determines which rows of the array are accessed during configuration. The Row Wildcard register is currently displayed assuming an address value of zero. Clicking on any part of the Column Wildcard register causes the value of this register to be displayed in the Status window in the bottom portion of the display. Unlike the Map register, this register exists in a 6 bit encoded form, and thus cannot be modified on a bit per bit basis.

Below the cell array is a bar used to point to a given column in the array. Clicking on this bar selects a column of cells in the array. Cells selected by the Map register are highlighted to help indicate their involvement in data accesses.

Finally, clicking on cells in the array return their configuration value. This is displayed in the status panel at the lower right portion of the display. Note that this cell configuration panel has two modes. One displays textual information, the other displays a graphical representation of the circuit in the selected cell. Click on the status panel to toggle from one type of display to the other.

The Symbolic Display

The second display in WebScope is the symbolic display. This display treats groups of cells in a column as a symbolic variable and permits software-like symbolic access to the XC6200. These symbols are loaded into WebScope by clicking the file button and entering the file name into the dialog box.

The symbol file format is an ASCII file containing five to six fields per line, seperated by whitespace. Each line represents a variable implemented as a group of XC6200 cells in a column. Lines beginning with a hash ('#') are treated as comments and ignored.

The first field in a symbol entry is the symbol name. This value is a string used to identify the variable. The second field is the column in the XC6200 which contains the register. This column must be a value between 0 and 63.

The next two entries define the Map register associated with the variable. Since the map register is 64 bits long, the value is split into two 32 bit quantities, Map High, the upper 32 bits of the register, and Map Low, the lower 32 bits. Note that values of '0' enable a row, while values of '1' disable.

The next entry is the read/write indicator. Rather than passively reading values from the hardware and displaying them, WebScope permits values to be read or written. A value of "R" indicates a variable which is to be read; a "W" indicates a variable which is to be written. Following the read/write field is an optional value field. This field is not necessary for read-only variables, but will be accepted nonetheless. It is, however, mandatory for writable variables. This gives the value to be written to the variable in the hardware on each step.

When the symbol table data is loaded and displayed, these values will be in read-only text fields if they are read-only variables, and in editable text fields if they are writable. This permits the value of writable variables to be easily modified at run-time.

An example file, `counter.sym` is reproduced below:

```
#
#  Symbol file for WebScope
#
# Name   Col      Map_high      Map_low    R/W     Value
#-----   ---      --------     --------    ---     -----
clock      0    0xffffffff   0xfffffffe      W        0
clock      0    0xffffffff   0xfffffffe      W        1
count     10    0xffffffff   0xffffff00      R
```

Note that in the example above, the same variable, clock is referenced twice, each time as a writable variable. This technique is used to produce a software-driven clock pulse on each step. Because the symbol table entries are read/written in the order in which they are listed, the cell at location (0,0) will be set to '0', then to '1'. This may be used to clock a circuit in a controlled manner. See the section below on Clocking for more information on clocking issues.

The Waveform Display

The waveform display draws strip chart diagrams for the variables in the symbol table. This permits the history of the variable over time to be viewed. This is not only useful for spotting transient irregularities in data, but is useful for digital signal processing (DSP) applications as well. This display may also be favored by hardware engineers and others who are more comfortable with the digital waveform displays found in circuit simulators. As with the Symbolic Display, the Waveform Display is driven by the values in the symbol table.

Clocking

How a design is clocked is particularly important in a debugging environment such as WebScope. When designing a circuit, the first approach is typically to use a single global clock. While this is fine in many cases, the reading and writing of state data by WebScope necessarily causes clocking of the XC6200. This will result in extra clocks being sent to the circuit. This is usually undesirable.

If you have already designed a sychronous circuit using the standard global clock circuitry (an IPAD connected to a BUFGP with an attribute of GLOBAL=GCLK), it is fairly easy to modify the circuit to be WebScope-friendly.

The circuit diagram below shows the standard global clock circuit with some modifications. First, an RPFD writable flip-flop is inserted before the BUFGP. The old global clock signal is used to clock this flip-flop and its output (Q) is used to drive the global clock. This RPFD is RLOCed to some cell location, in this case (0,0) by the attribute RLOC=X0Y0. In addition, now that the BUFGP is no longer driven by an IPAD, the IPAD must be manually attached to a pin. Usually, the global clock defaults to pin 84, the global clock input on the hardware. In this case it must be manually set with an attribute of LOC=P84.

Now the RPFD at cell (0,0) is ready to be used as a software-controlled clock. Writing a '0', then a '1' to this register will produce a single clock pulse, irrespective of the system's global clock. As an added benefit, the software controlled clock is easily converted back to a single globally clocked circuit. Removing the RPFD and connecting the clock from the IPAD to the GPBUF restores the original circuit. And don't worry about the LOC=P84 attribute on the IPAD. This is the default location of the global clock input. There is no need to remove or modify this attribute.

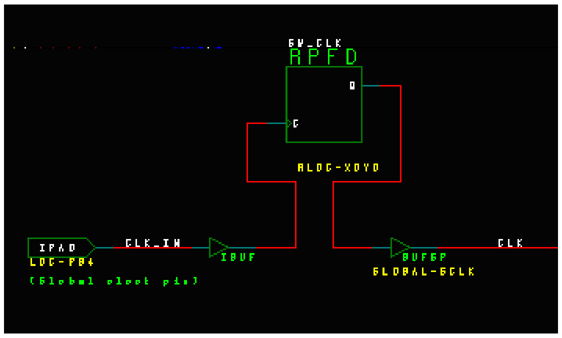


Figure 6

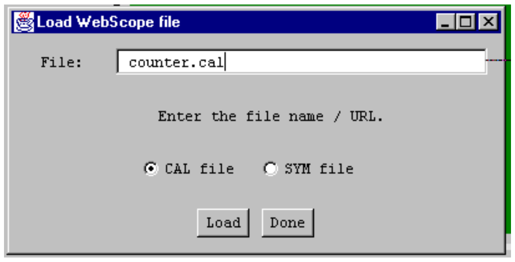
In the section above on The User Interface, under The Symbolic Display section, a method of using a SYM file to clock such a circuit is outlined. In fact, this exact symbol file used as an example in this section may be used to clock this design on each step.

Sample Designs

This section supplies some simple, debugged designs to allow you to experiment with WebScope, without actually having to create a custom design first. All of these designs were input using the ViewDraw schematic editor from the WorkView Office package. EDIF files created with WorkView Office were then imported into Xilinx's XACT/6000 tool. This tool generated the CAL files used by WebScope. The SYM files were entered by hand using a text editor. Only the CAL files and corresponding SYM files are provided.

A Counter

The first design is an 8-bit counter. This design is in the file counter.cal. Its corresponding symbol file is under counter.sym. This design is a synchronous counter implemented using 8 toggle flip-flops and 7 AND gates. A cell at location (0,0) also contains a writable register (RPFD) which is used as a software controlled clock. The SYM symbol file contains variables which show the state of the counter as well as control the clock.



counter.cal

Enter the file name / URL.

[✓ CAL file] SYM file

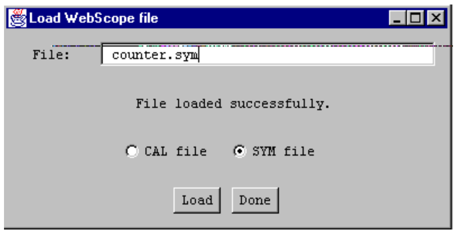
Figure 7 Loading a CAL File

After you have brought up WebScope and connected to the target, reset the hardware and use the file dialog to download the `counter.cal` CAL file (See *Figure 7)* Part of counter.cal below.

```
# Camelot Version 0.3.1
#  Build number 36
#  Build Date Sep  6 1996
# Block COUNTER
#
# Bus Width 8
# Clock Speed: gclk/16
# Inversion Compensation: on
# Prologue: on
# End State: none
# Padding Cycles: 5
# control device config
0xc000  0x0
#
```

```
# Begin prologue
#
# mux WEST at 0x0 0x0, byte 0x0
0x4000  0x70
...
```

Once the CAL file is downloaded, use the display button to switch to the symbol display. Again, use the file button to bring up the file dialog box. But this time, select the SYM file option in the dialog box and enter the name of the SYM file, counter.sym (See Figure 8).


Load WebScope file

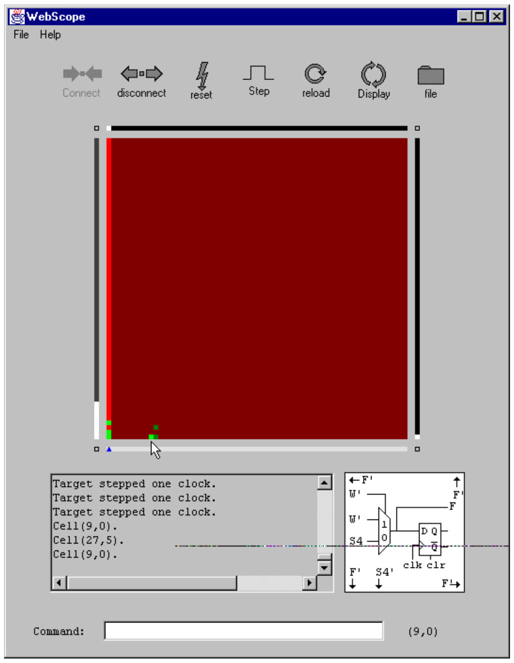
File: counter.sym

File loaded successfully.

CAL file [✓ SYM file]

Figure 8 Loading SYM File

After this file is loaded, the count and clock variables should be visible in the symbol table.



```
Target stepped one clock.
Target stepped one clock.
Target stepped one clock.
Cell(9,0).
Cell(27,5).
Cell(9,0).
```

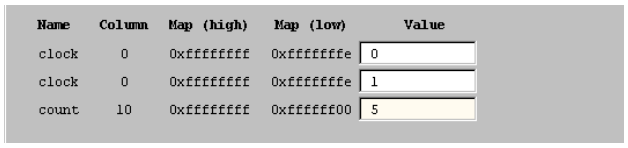
Figure 9

If the step button is pressed, the software controlled clock is cycled by the clock entries in the symbol table, and the count variable should be updated by one.

⇒ Click Step Button five (5) times to watch the design count.

If the display button is used to switch back to the graphical display, you will notice some cells at and near column 10 set to a state of '1' (green). Successive steps will cause the values in column 10 to count in binary, with row 0 being the least significant bit. Note that some neighboring cells will also change state when the design is stepped. These cells contain combinational logic (AND gates) used to control the counter. If you watch them closely, you will probably be able to determine their function.

Let’s take a look at the information being displayed in the various display modes. (Figure 9 - Graphic Mode).



Name	Column	Map (high)	Map (low)	Value
clock	0	0xffffffff	0xfffffffe	0
clock	0	0xffffffff	0xfffffffe	1
count	10	0xffffffff	0xffffff00	5

Figure 10 Symbolic Display Mode

⇒ Click the Graphical Display Button to move to the Symbolic Display Mode (see Figure 10).

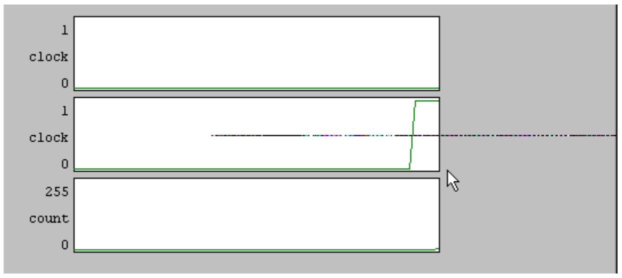
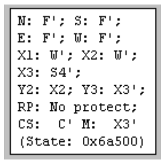


Figure 11 Waveform Display Mode

⇒ Click the Graphical Display Button to move to the Waveform Display Mode (see Figure 11).

You should have stepped the design five times for the following figures to correspond to the WebScope screen. If you are not sure. Then Press Reset Button and Reload the counter CAL & SYM files.

⇒ Using the Graphical Display mode of Webscope, move the cursor to cell (9,0) and click the mouse button to select that cell. You can see the cursor's cell coordinates displayed at the lower right side of the WebScope screen (See Figure 9). The cell 9,0's functional information is contained in the lower right hand window (See Figure 12). If you click the mouse button on this window, it will flip to the graphic mode (See Figure 13).



```
N: F'; S: F';
E: F'; W: F';
X1: W'; X2: W';
X3: S4';
Y2: X2; Y3: X3';
RP: No protect;
CS:  C' M:  X3'
(State: 0x6a500)
```

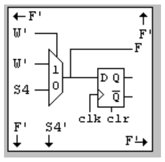


Figure 12 *Figure 13*

Figure 12 & Figure 13 show two views of the information contained in the selected cell. Below is the legend for these two views.

N: - north **S:** - south **E:** - East **W:** - West
X1: - Input **X2**: - Input **X3**: - Input **F:** - Output
Y2: - Mux Input **Y3**: - Mux Input
RP: - Register Protect (0 - Register input from logic 1- Register input from register)
Only the microporicessor port can change the register’s value.
CS: - Combinatorial Select (0 - Input from Register 1 - Input from logic)
M: - Magic Line **(State: ______)** - Bits configuring the cell.

Function Unit Internals

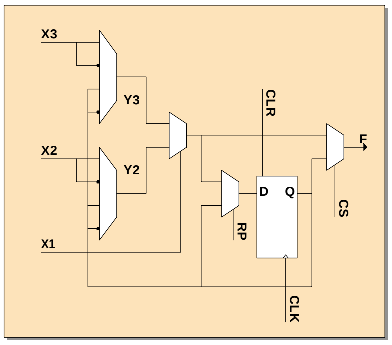


Figure 14

So we can see that there are three inputs X1, X2, X3 and that they have a value of W’ , W’ and S4’. To find out what function this represents, we look up Function Derivation (Table 1 - Figure 11 in Chapter 3). From this table we can identify the assignments for the cell multiplexer which compute the various logic gate functions.

Cell 9,0 values: (The ‘ means an inverted signal.)

X1 = W’, X2 = W’ , X3 = S4’
Y2 = X2, Y3 = X3’, RP = No Protect
CS = C’ , M= X3’, State = 0x6a500

The function of this cell 9,0 is $\bar{A}.B$ (Figure 15).

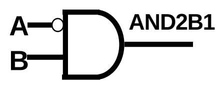


Figure 15

Let's move to the cell above.

⇒ Using the Graphical Display mode of Webscope, move the cursor to cell (9,1) and click the mouse button to select that cell.). The cell 9,1's information is contained in the lower right hand window.

Cell 9,1 values: (The ' means an inverted signal.)

X1 = W', X2 = W' , X3 = N'
Y2 = X2, Y3 = X3, RP = No Protect
CS = C', M= X3', State = 0xa710

After looking up in the Function Table, the function of this cell 9,0 is $\bar{A}$+B *(see* Figure 16).

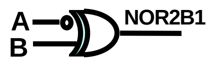


Figure 16

Let's move to the cell below and to the right.

⇒ Using the Graphical Display mode of Webscope, move the cursor to cell (10,0) and click the mouse button to select that cell.). The cell 10,0's information is contained in the lower right hand window.

Cell 10,0 values: (The' means an inverted signal.)

X1 = S4', X2 = S4' , X3 = S4'
Y2 = Q', Y3 = Q, RP = No Protect
CS = Q' , M= X3', State = 0xa1b5900

After looking up in the Function Table, the function of this cell 10,0 is INV_FAST (Figure17).

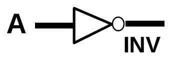


Figure 17

Cell 10,1 is the same as cell 10,2.   Let us now look at the information in the graphic cell information wind and see how these functions are hooked up or routed. Below are the four cell information window arranged as they are on WebScopes graphic display window (See Figure 18).

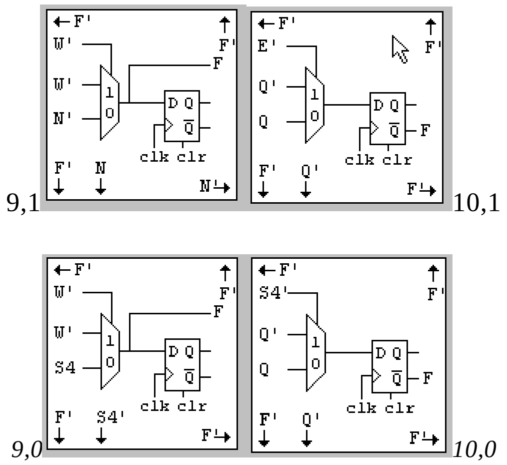


Figure 18

These diagrams represent the routing lines carrying the data in and out of the central logic cell. The central logic cell architecture is show above in figure 14. The X1, X2, X3,F, etc. lines from the central logic cell are than routed to other cell or I/O pins in the RPU. The routing lines which connect the various cells are identified by their type and direction of travel. A more detailed explanation of the routing resources of the XC6200RPU can be found in Chapter 3.

The XC6200 RPU has various sets of routing lines within and around the logic cells. Some of these lines go to the next neighbor cell or four cell skips or fly across half the RPU. Each cell has it's own routing resources. The diagram below shows the arrangement of the cell's routing resources. (See Figure 19).

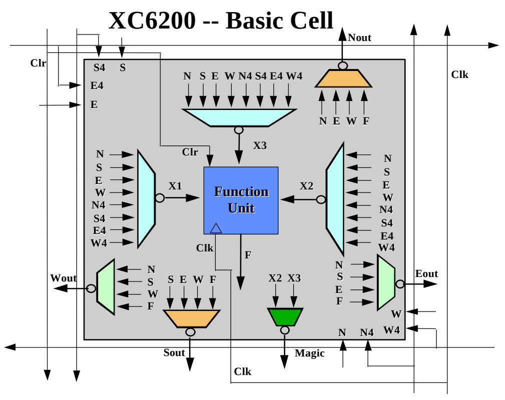


Figure 19

Linear Cellular Automata

The next design is a linear cellular automata (LCA) demonstration. This uses a linear feedback shift register to produce pseudorandom values based only on nearest neighbor information. This technique has uses in random number generation, cryptography and simulation.

The design is in the file lca.cal. Its corresponding symbol files are under lca_init.sym and lca.sym. The design consists of a loadable register with a large shift register across the columns at each bit. The loadable register is 32 bits wide, but begins at row 8, rather than at the bottom of the array.

The loadable register is controlled by a bit in row 0 of column 1. When this bit is set low, data written to column 3 is latched into the linear feedback shift register at column 5. When the enable bit at row 0, column 1 is set to 1, random number generation can begin. The design is clocked by the cell ar row 0, column 0.

Again, after you have brought up WebScope and connected to the target, reset the hardware, use the file dialog to download the lca.cal CAL file. This is a fairly large file and may take some time to download, particularly over a network connection.

Once the CAL file is downloaded, use the display button to switch to the symbol display. Again, use the file button to bring up the file dialog box. But this time load the SYM file, lca_init.sym.

After this file is loaded, the load_reg, enable, clock and random variables should be visible in the symbol table. Click the step button to load the register. This initializes the feedback shift register.

Now that the system has been initialized, load he second symbol file, lca.sym. Again, use the file button to bring up the file dialog box, but this time enter the name of the second SYM file, lca.sym.

Return to the graphical display screen using the display button. Clicking the step button should produce the characteristic "triangle" pattern of the LCA. Notice the trick of using two SYM files in this example. The first is used to initialize registers. Once the data is initialized, a second SYM file is used for execution.

Troubleshooting

Q: When I run WebScope as an applet, it refuses to execute. I get error messages along the lines of "Java security violation". What should I do?

A: It is likely that you are running with an old version of Java. You should download and install a new one. Testing has been performed with Netscape 3.0, Microsoft Internet Explorer 2.0 and the Sun Java Development Kit (JDK) 1.0.2 for the PC under Windows 95. Any one of these should work.

Q: When attempting to load WebScope as an applet, a message similar to "your browser does not understand the applet tag" is returned by the Web browser. The applet never executes.

A: Your Web browser either does not have Java installed, or it does not have it installed correctly. If you have the most recent version of your browser, try re-installing it. If not, download a more recent version and install it. The previous question gives pointers to the locations of the popular Web browsers.

Q: WebScope comes up fine, but I am unable to connect to the target hardware.

A: There are several possibilities: First, be sure that you are attempting to connect to the correct machine and port number. There are applet parameters in the WebScopeA.html applet launcher file and command line parameters in the standalone version. See the section in this document on Running WebScope for more information. If you are attempting to connect over a network, be sure that the target machine is running WsServer. See the section on Remote Access for more information. Finally, be sure someone else isn't connected to your hardware. WsServer only permits a single connection. Other attempts at connection are refused.

Q: After a reset, CAL files do not load. The graphical display is all green (logic '1'), and the map reister is disabled. Settings to the map register disappear after a reload.

A: The XC6200 device is not enabled. Enabling occurs when a silicon revision (or "ID") number is written to the configuration register at address 0xC03F. This code is usually the last entry of the CAL file. It may be manually edited. If your silicon and your software are of different revisions, however, it is recommended that you upgrade to the lastest versions. Using the command line interface command of setreg config n may also work, where n is the silicon revision number.

The Correlator Program

Image correlation is performed by passing a template over an image and determining, at each pixel position, if a match has been found. The design described uses a 32x16 pixel match image and a 32x16 pixel mask image to construct the template. The match image is a small image to be located. The mask image allows masking out of regions in the match image. Only unmasked pixels will be correlated. If the number of matching pixels, at any point in the image, exceed a threshold a Hit bit is set to true.

The match and mask images are hardwired into the logic. See Figure 20. The ability to dynamically reconfigure gates means that the image template can be rapidly reconfigured an unlimited number of times. Partial reconfiguration and fast reconfiguration times make XC6200 ideal for this application and with the flexibility and ease with which reconfiguration data can be generated using XACT6000, development time is minimised.

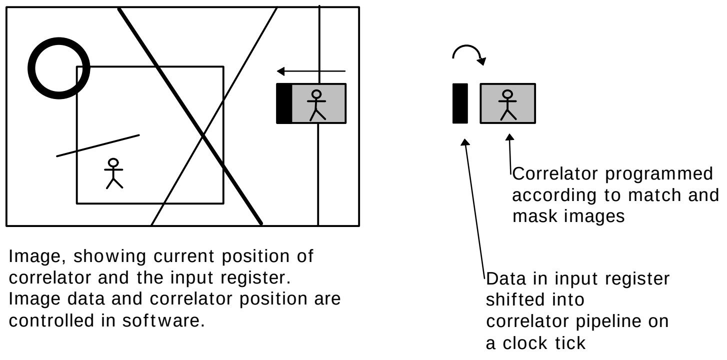

Image, showing current position of correlator and the input register. Image data and correlator position are controlled in software.
Correlator programmed according to match and mask images
Data in input register shifted into correlator pipeline on a clock tick

Figure 20

The XC6200 RPU on the H.O.T. Works Board is programmed with a design to perform correlation on binary images. This note describes design flow, showing how, using the codesign approach, software and hardware tasks can be tackled simultaneously. Details are given of correlator development, modelling correlator functionality in software, logic simulation, and emulation (actual design running in an RPU). The standard XC6200DS libraries, linking users design elements to actual array resources, are presented. The logic architecture for an array of custom adders is described. ASIC designers will be familiar with this static ASIC design process; altering design for regularity and reconfiguration will be new. Finally, performance data is given for both image correlation and reconfiguration.

Design Flow

Design flow uses a codesign approach allowing software and hardware to be implemented in parallel.

In software:

- Write C++ program to read in the image, mask and match templates and call a correlator. Create C++ classes for *Correlator*, *CorrelatorSoft*, *CorrelatorHard*, *CorrelatorSim* and *CorrelatorBoard*.
- Model correlator design in software using class *CorrelatorSoft*

In hardware:

- Capture the design as a schematic, in this case ViewDraw was used.

Then in software:

- Simulate and debug by defining a correlator of type *CorrelatorSim* and using a simulator, ViewSim was used.

Then in hardware:

- Generate the **Edif** for the design.
- Use **XACTstepSeries6000** to optimally place and route the design. Generate the cal file, the symbol table and the RAL file.

Then in software:

- Redefine the correlator in the C++ program as type *CorrelatorBoard*
- Run the program using the XC6200 hardware accelerated correlator.

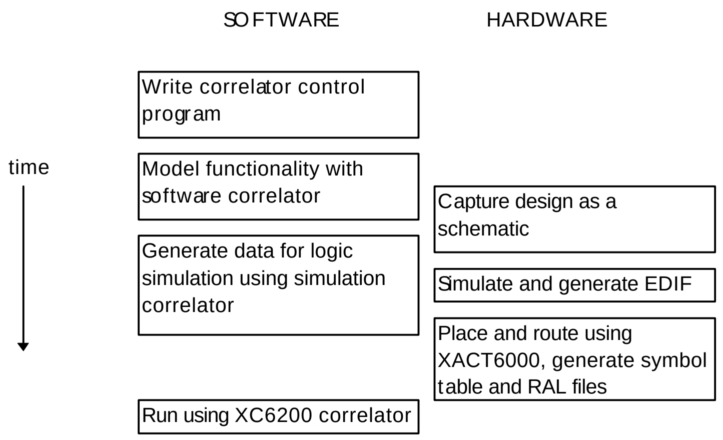

SOFTWARE
HARDWARE
time
Write correlator control program
Model functionality with software correlator
Generate data for logic simulation using simulation correlator
Run using XC6200 correlator
Capture design as a schematic
Simulate and generate EDIF
Place and route using XACT6000, generate symbol table and RAL files

Figure 21

Software Design -- The Control Program

The C++ program controls both setup and operation. Setup includes the following steps:

- Initialisation of correlator design in one of software, simulation or hardware modes
- Reading in the match and mask data from files
- Reconfiguring correlator according to the match and mask images
- Reading in the image to match against

Correlator operation:

- Alternately, writing the image data to the input register in 16 pixel columns and checking for Hit
- Displaying crosshairs on the Hit location if Hit = 1.

The Correlator Classes

A correlator base class, *Correlator*, is defined as part of the C++ program. On it are defined arrays for the mask image and the match image; the dimensions of the search image and the threshold value. From this base class is derived the class *CorrelatorHard*. *CorrelatorHard* contains methods common to both simulation and utilisation of the ASIC design e.g. a list of the blocks with reconfigurable gates (see section on Logic Design) and a function *build_correlator()* which generates the hierarchical path names for each of these reconfigurable gates. This design follows a recursive structural pattern through the hierarchy making the generation of these strings an elegant process. See Appendix B.

CorrelatorBoard and *CorrelatorSim* in turn derive from *CorrelatorHard*. *CorrelatorSim* performs simulation of your design by generating simulation files from image data. ViewSim was used in this instance. *CorrelatorBoard* uses the *AccessRegister* and *Pci6200* classes to interface to the Development System.

The AccessRegister Class

The *AccessRegister* classes *AccessRegisterSim* and *AccessRegisterBoard*, allow registers in the logic design to be treated as variables in the controlling C++ program. These classes are generic and can be used for variables in any design. *AccessRegisterSim* writes a command file for ViewSim. This file is imported by ViewSim and shows the operation of the correlator via simulation of the ViewLogic design. *AccessRegisterBoard* utilises the symbol table data from *XACT6000* to evaluate the Map Register, column select and bus width for each variable in the design e.g. for this design there are only two variables, the 16bit input register and the 1bit hit register. The 16bit Map Register selects the rows from which data will be passed on to the bus. Column select selects the column to or from which these bits are transferred. The image data is then written to the hardware correlator. See Appendix B and refer to the documentation on the *AccessRegister* classes for elaboration.

The Pci6200 Class

Below the *AccessRegister* classes in the software hierarchy, lies the *Pci6200* function library. These functions perform the low level interface to the XC6200 and H.O.t. Works Board. Included in the functionality of this low level library: downloading a design onto the array and reprogramming it, via the *FastMap* interface; setting the Map Register and control of Global Clock and Clear. For further information see the documentation for the *Pci6200* class.

The RAL Class

The address data pairs for reconfiguring individual function units are provided by a Ral Software Library function.This library extracts information from a special Ral file generated by XACT6000 for this design. Provided with the following parameters: hierarchical instance name; hierarchical net names and the XC6000 library function name for the new reconfiguration, a function call returns the address/data pairs required to reconfigure a specific function unit in the array.

Logic Description

The design comprises the following:

- 16bit input register
- 32x16bit correlator block programmed with mask and match
- Threshold block
- 1bit hit register
- CBUF clock: generates a clock pulse on each write to the input register

Figure 22 is a block diagram illustrating the structure of the correlator logic. The image data is written in columns of 16 pixels to the input register via the control store interface. See App Note XAPP063 for details. On each write to this register, the CBUF generates a clock pulse clocking the design and shifting the columns of data into the pipeline. All pixels in the pipeline are summed in an array of custom adders.

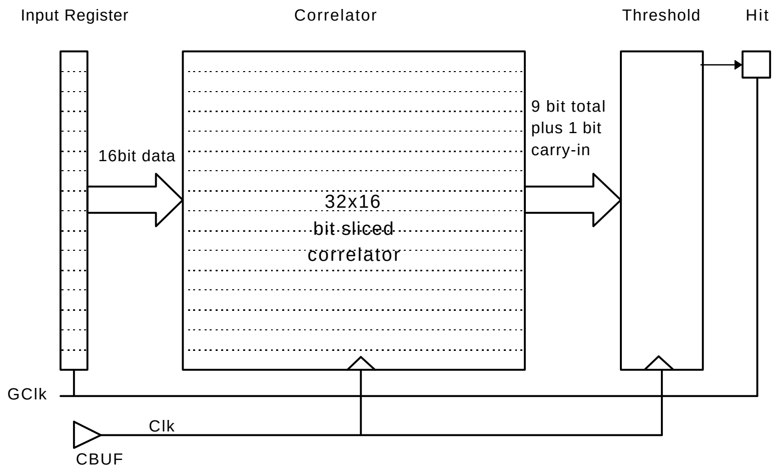


Figure 22

Correlator Logic Details

The operation of the correlator is as follows. On each clock step, the 16bit column of data in the input register is shifted into the 32 bit long pipeline as the next column is written by the control program to the register. Each pixel is logically compared with the match and mask data (hardwired as gates) for its current position, and the total number of matches evaluated.

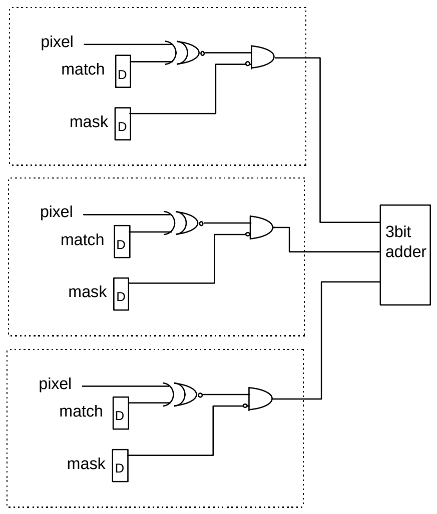


Figure 23

Figure 23 shows a 3 pixel correlator. The gates inside each dotted box perform selection on the pixels depending on the match and mask data. All of these gates can be eliminated from the design if it is possible to reconfigure the gates within the three bit adder, minimizing the adder logic according to the match and mask data for each pixel.

Figure 24 is the base configuration for the lowest level block of the design showing a reconfigurable 2 stage adder. The adder inputs come from a section of the pipeline. Operation is as follows. Pixel data is clocked through the shift register. Summing of the number of matched pixels takes place over two clock cycles. The output from ABXOR is delayed for one clock cycle and then operates as in a standard adder design when the three pixels being correlated lie in the registers. The adder sums the number of matched pixels, the sum has a one clock cycle delay, the carry a delay of two. When placed and routed using XACT6000, this block occupies 6 function units, the design in Figure 23 would require 15.

Figure 25 shows this three pixel adder. As discussed, correlation, or matching against different pixel patterns, is achieved by making each gate of this adder reconfigurable, basing the new gate on the mask and match data. For example, if the pixel on the input were masked and the pixel in the first flip-flop matched with 1, the gate ABXOR would be configured to be a buffer. If the pixel in the first flip-flop matched against 0, ABXOR would be configured to an inverter. Similarly, gates and their net connections are evaluated for CMUX and SUMXOR. See Figure 28 for all possible reconfigurations of mask and match combinations.

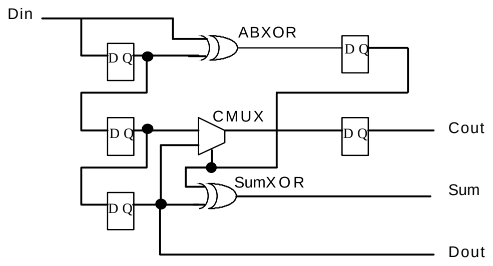


Figure 24

The correlator is constructed hierarchically from these low level blocks. Two blocks are chained together to make a 6bit shift register. See Figure . The sum and carry bits from these two blocks are then summed to give a matched pixel total for the 6bits. As a space saving trick here, an extra pixel is taken as a carry in, making a seven pixel correlator block. The carry in pixel must be delayed by one clock cycle to synchronize timing with the sum output from the 3 pixel correlator to which it is being added. The gate which takes the carry in is reconfigurable, another ABXOR but with only one of the inputs being a pixel value the other is assumed always to have a mask value of 0 and a match of 1.

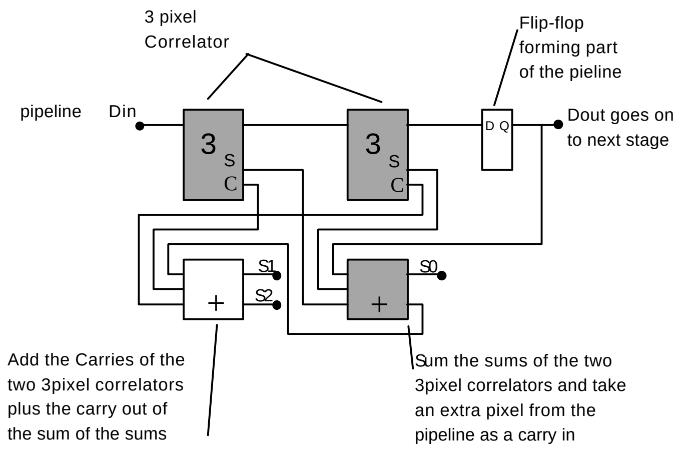

3 pixel Correlator
Flip-flop forming part of the pieline
pipeline
Din
3
S
C
3
S
C
D Q
Dout goes on to next stage
S1
S2
S0
+
+
Add the Carries of the two 3pixel correlators plus the carry out of the sum of the sums
Sum the sums of the two 3pixel correlators and take an extra pixel from the pipeline as a carry in

Figure 25

Two of these 7 pixel correlators are combined in parallel, their totals summed, plus an extra pixel on the carry in, delayed by two, to make a 15 pixel correlator, see. Two 15 pixel correlators are combined in series with an extra carry in delayed by three clock cycles, to make a 31 bit correlator; and so on, until a 511 pixel correlator is constructed, leaving one pixel untested.
The sum from the 511 pixel correlator plus the 512th pixel are fed into the Threshold block.

Threshold Block Logic

The threshold value determines the quality of the match of the image with the template. Logically, the threshold block compares the number of matches with a preset threshold, *t*, and if it is greater, sets the *Hit* flag. For a perfect match the threshold value would be 32x16=512. In practice, the threshold block operates by adding the total number of matched pixels plus the extra carry in pixel to a constant value and generating an overflow when the threshold is exceeded. This means programming the threshold block with a constant value, 512-*t*, where *t* is the threshold value. This is done by performing a single threshold register write before correlation of the image begins. The overflow or carry appears in the single bit *Hit* register.

Figure 26 shows how the reconfigurable gates map to the pixels in the mask and match template images.

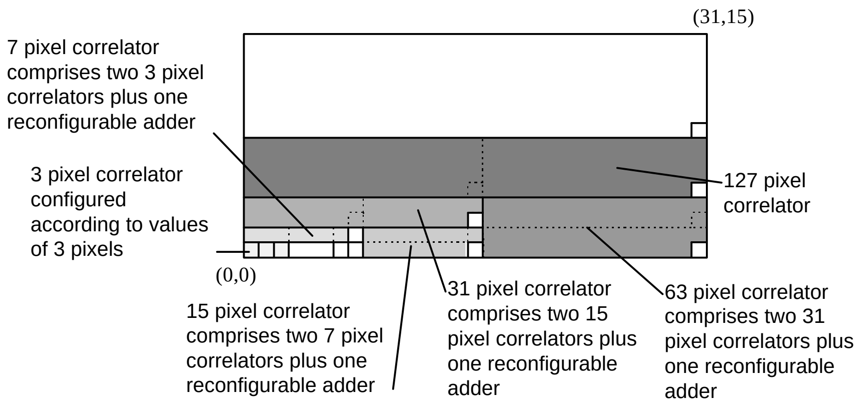

(31,15)
7 pixel correlator comprises two 3 pixel correlators plus one reconfigurable adder
3 pixel correlator configured according to values of 3 pixels
127 pixel correlator
(0,0)
15 pixel correlator comprises two 7 pixel correlators plus one reconfigurable adder
31 pixel correlator comprises two 15 pixel correlators plus one reconfigurable adder
63 pixel correlator comprises two 31 pixel correlators plus one reconfigurable adder

Figure 26

XACT6000 Instantiation

Knowing the regular structure of the XC6200 (ref. Datasheet), it is easy to exploit this in structured logic design. XACT6000 performs automatic placement and routing. It can be forced to place and route according to user preference by attaching constraints during the schematic capture phase. This feature is exploited fully in the development of this design. The advantages XACT6000 offers, of maintaining the hierarchy in the physical instantiation of a design, are shown clearly in Figure 27.

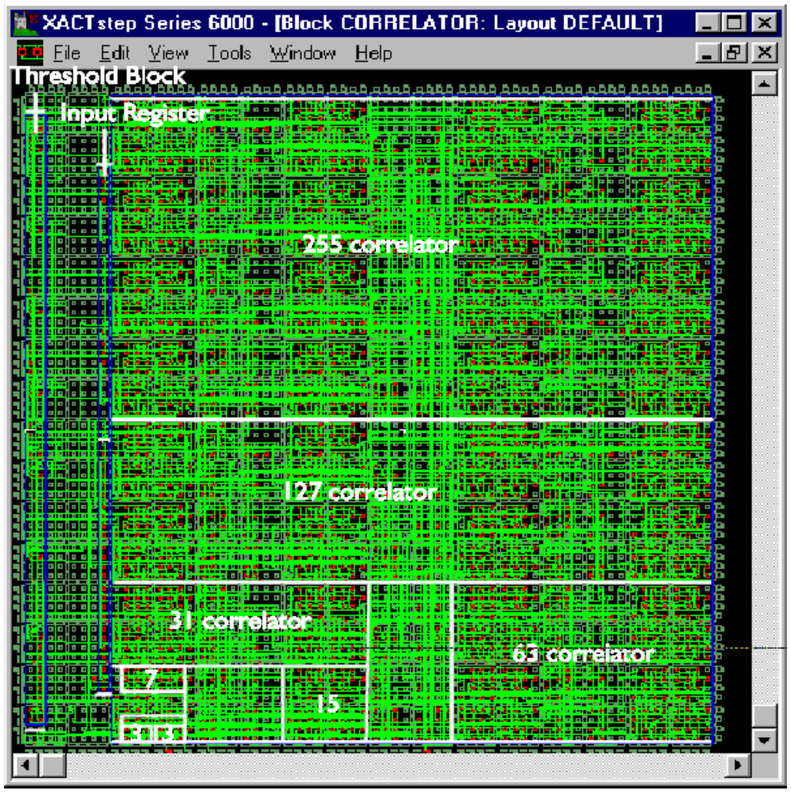

XACTstep Series 6000 - [Block CORRELATOR: Layout DEFAULT]
File Edit View Tools Window Help
Threshold Block
Input Register
255 correlator
127 correlator
31 correlator
63 correlator
7
15
3
13

Figure 27

This figures shows the XACT6000 layout and clearly maps the structure of the logic, Figures 22, 24, 25 and 26 and the hierarchical breakdown of the image in the same way show logic mapping directly into hardware.

Placement Constraints

RLOC constraints are attached to the instances in the low level blocks forcing them to be placed in rectangles which can then be tiled together.

Transform Constraints

The way that blocks in this design connect together can be specified by the use of transforms. In the layout of the 15 pixel correlator (Figure25) for example, it is sensible to place two 7 pixel correlators back to back with the adders in between them. The REF0 constraint is attached to one of the blocks which forcing a reflection in the Y axis of the default layout for the block.

Flatten Constraint

The FLATTEN constraint forces all gates within an instance to be placed as gates rather than as a single block i.e. the hierarchical structure of the block is removed. This is particularly useful when mapping gates and flip-flops into single cells or when space is at a premium and placing blocks would cause redundancy.

Routing Constraints

Routing is deferred to a higher level with the attachment of the RTDEFER constraint to a schematic sheet. Optimal routing for this design results when deferring low level routing to the 15 pixel correlator block level. This block is routed limiting routing resources to local and length four routing. Subsequent levels in the hierarchy have routing deferred to the top level. All resources are made available for top level routing.

Performance Measures -- Speed of Operation

For a test image of dimension 512x512 pixels, assuming all preprocessing is complete and the image is sitting in primary memory, the total time taken for correlation depends on the frequency the circuit clock. This is dependent on the amount of time taken to perform reads and writes using XC6200 because the CBUF clock pulse to the design is generated on every write to the data input register. After the first 31 writes in each row necessary to fill the correlator, each write is alternated, by reads from the Hit register. Writing to and reading from a register require 2 global clock cycles each. So the maximum Clock speed for this design on the demonstration board is approximately one quarter Gclk: 8.25Mhz. Calculating time taken to correlate a 512x512 image gives a value of approximately 30ms for the whole image. Of course, some applications will abort further correlation once a template has been located and will not require to correlate the whole image. To improve speed, *Hit* could be wired to an interrupt signal.

Speed of Initialization

In performing reconfiguration, time taken is dependent on the method of reconfiguration. In the most simple method, also the most time consuming, each address/data pair is written individually from the control program. It is also possible to perform batch reconfiguration.

Total time taken to change the mask and match image for this design is given by the following equation:

$$T = G \times 2Nclk \times {}^{1}\!/\!{}_{Fclk}$$

where *T* is the total reconfiguration time, *G* is the number of gates, *Nclk* is the number of Clock cycles taken for a write of one address/data pair and *Fclk* is the frequency of the clock.

$$T = 512 \times 2 \times 5 \times {}^{1}\!/_{33 \times 10^{6}} = 0.155ms$$

Performance with multiple templates
Given that correlator reconfiguration with a 32x16 mask and match template requires less than 0.2ms and correlation over a 512x512 image takes approximately 30ms, reconfiguration time between templates can be regarded as insignificant.

Summary
XC6200 proves ideal as a hardware accelerator for correlation. Given the problems of image correlation: large amounts of data, intensive computation and the desire, in most applications, to frequently change mask and match images two advantages of XC6200 are clear. It has both the flexibility of rapid partial reconfiguration for the correlation of a different image template, times of less than 0.2ms calculated; and the capability of being fully reconfigured to perform different functions.

Durations of 30ms are obtainable for full correlation of images of dimension 512x512 pixels.
The ease with which the design process proceeds has been shown using the codesign approach where software and hardware are developed in parallel. Interfacing to the design is completely automated by use of the *AccessRegister* and *Pci6200* classes and reconfiguration supported via the RAL Library.

Gate reconfigurations depending on match and mask values.

Gate	Condition	Reconfiguration	Net connections
ABXOR	Match A = Match B	XOR2	A, B
	Match A != Match B	XNOR2	A, B
	Match A = 1, B Masked	BUF	A
	Match A = 0, B Masked	INV	A
	Match B = 1, A Masked	BUF	B
	Match B = 0, A Masked	INV	B
	A, B Masked	GND	
RCMUX	Match C, Match D = 1	M2_1	C, D, S
	Match C = 0, Match D = 1	M2_1B1	Cinv, D, S
	Match C = 1, Match D = 0	M2_1B1B	C, Dinv, S
	Match C, Match D = 0	M2_1B2	Cinv, Dinv, S
	Match C = 1, D Masked	AND2B1	C, S
	Match D = 1, C Masked	AND	S0, D1
	Match C = 0, D Masked	AND2B2	Cinv, Sinv
	Match D = 0, C Masked	AND2B1	S, Dinv
	C, D Masked	GND	
SUMXOR	Match D = 1	XOR2	D, S
	Match D = 0	XNOR2	D, S
	D Masked	BUF	S

Figure 28

Running the Correlator Program

The Correlator files are located in `C:\Hotworks\Correlator` directory.

⇒ Under Windows, the Hotworks Setup installs a folder on the Start Menu Called **Hotworks**. In this folder one of the items is the **Correlator Program** OR/
⇒ Run Click on `correlator.exe` - located in `C:\Hotworks\Correlator\Code` directory. The program takes a few seconds to start up.

Loading an image

⇒ From the Command menu or using the icons: Load Image. From the images directory, choose an `board_bw.ppm` image.

Note: No mask is the default.

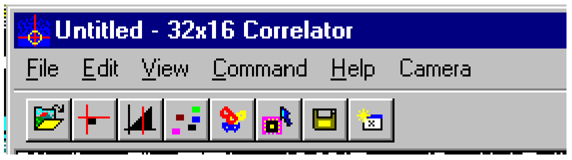

Untitled - 32x16 Correlator
File Edit View Command Help Camera

Figure 28 Correlator Controls

Choosing a match template

⇒ Use RIGHT MOUSE BUTTON to click on the Xilinx logo on one of the two chips, the match image appears selected within the red rectangle OR/

⇒ Choose match0 from the "choose match" dialog (spotty icon on the icon bar).

Setting the threshold

⇒ Set the correlation threshold to 390. (Command menu or toolbar icon). You may need to play with this threshold value, but around 400 is usually OK. It depends on the pixels you have selected for your match. In a real world app, you'd take an average over a large number of templates.

Selecting the type of correlator

⇒ Select your choice of software or hardware correlator. (Command menu or toolbar icon). Press space bar or click crosshairs icon to correlate. Time taken displayed in status bar.

Loading an image

⇒ Load microserf.ppm.

Choose match1 from the "choose match" dialog. Match1 is the pixel pattern for the word "spots". Set the threshold to 420.

If you have a QuickCam attached to your machine you can correlate against sequences of images. Remember this correlator is not able to deal with changes in obect scale and orientation. QuickCam options disabled if no QuickCam present.

N.B.Software correlator is at least 10 TIMES slower. E.g. correlation of whole board image takes ~40s.

KNOWN BUGS AND PROBLEMS

If correlator app is covered with another window, it will not refresh image display when uncovered. This is to speed up display of crosshairs. To redisplay, click on the refresh icon.

Run-Time Programming Tutorial

use on the H.O.T. Works Board. The design example you will work through adds 4 bit numbers to a 4 bit constant. After installation, the tutorial and related files can be found in

`C:\Hotworks\pci_tutorial`

Run-Time Programming Tutorial

Directory Location: CHotworks/pci_tutorial
VHDL Files; ...\pci_tutorials\velab
Cal Files: ... \pci_tutorial\cpp
Sch Files: ... \pci_tutorial\vlogic
Edif Files: ... \pci_tutorial\xact6000

This tutorial takes a simple design from the design entry point (using VHDL Code for design entry method) to running an executable program utilizing the H.O.T. Works Board and the XC6200 RPU. This tutorial is courtesy of Xlinx Inc. The following software tools will be used in this tutorial:

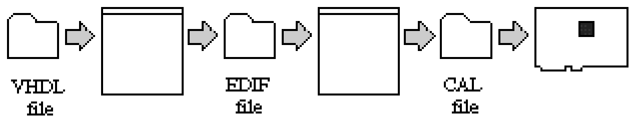
1. Velab (supplied with H.O.T. Works DS)
2. XACTstep 6000 (supplied with H.O.T. Works DS)
3. Class Libraries (supplied with H.O.T. Works DS)
4. PCITest (supplied with H.O.T. Works DS)
5. A C/C++ Compiler (not included in H.O.T. Works DS)

If you have no knowledge of the XC6200 RPU structure, VHDL or XACTstep 6000, we recommend that you view the interactive demo supplied on the XACTstep 6000 CD first. This presentation gives a good introduction to the XC6200 family of RPUs, design methodologies and through explanation of it's features.

⇒ Means you must perform an action.

Design Flow

Design |Conversion|Design | Map | Design | Execution



Velab **XACT6000** **Run-Time Program**

The design example you will work through adds 4 bit numbers to a 4 bit constant.

Tutorial 1

Proceed as follows:

1. Create the design in VHDL
2. Place and Route your design in XACT6000.
3. Use PciTest to check your design
4. Design Applications software for the XC6200
5. Details about topics learned in this tutorial

1. Create the design in VHDL

⇒ Open the VHDL file **`adder4.vhd`** in directory:
`C:\Hotworks\Pci_tutorials\velab`

You will see that the top level design comprises: two 4 bit input registers, one for the constant value and one for the input data; a 4 bit adder; and a 5 bit output register for the sum.

```
--
-- A 4 BIT INPUT REG
--

LIBRARY PRIMS;
USE PRIMS.ALL;

ENTITY REG4 IS
      PORT (CLK : IN STD_LOGIC;
             AOUT  : OUT STD_LOGIC_VECTOR (0 TO 3) );

      ATTRIBUTE BBOX OF REG4  : ENTITY IS "TALL";
      ATTRIBUTE RTMAX OF REG4 : ENTITY IS "X1Y1";
END REG4;

ARCHITECTURE STRUCT OF REG4 IS

      COMPONENT RPFD
            PORT ( C : IN STD_LOGIC;
                         Q : OUT STD_LOGIC);
      END COMPONENT;

BEGIN
```

Notice that some components have constraints attached to them. For example:

ATTRIBUTE BBOX OF REG4 : ENTITY IS "TALL";

Attributes are attached to your design to constrain **XACT6000**. For example, you can specify: placement position; routing resources used and reconfigurability by using such constraints.

The BBOX constraint instructs **XACT6000** to place the block REG4 such that the shape of the bounding box of the block is tall rather than square, which is the default. Other options are WIDE and SQUARE.

ATTRIBUTE RTMAX OF REG4 : ENTITY IS "X1Y1";

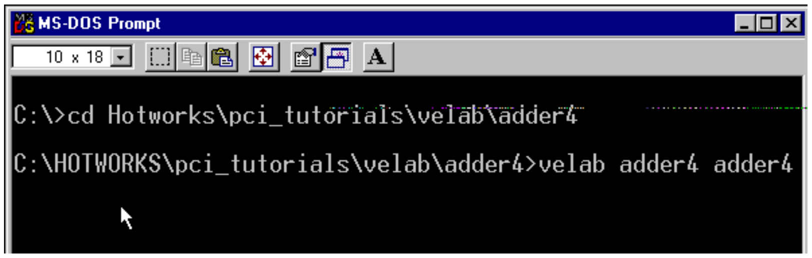
RTMAX is a routing constraint. The value of X1Y1 tells the router the maximum length of routing to use in both the x and y directions. In this case, local routing for both.

ATTRIBUTE RLOC OF A1 : LABEL IS "X0Y1";

An RLOC constraint instructs **XACT6000** to place the instance in the given location relative to the origin which is the bottom left corner. In this case, block A1 will be placed in position 0,1 within the layout for the 4bit adder.

ATTRIBUTE RTDEFER OF ADD1 : ENTITY IS "";

The RTDEFER constraint tells the router to defer routing this block until the level above. This is useful because an unrouted block can be placed in a wider choice of positions. Routing reduces the choices of positions because it introduces alignment considerations e.g. if a block contains length 4 wires then it can only be placed at a length 4 boundary.c



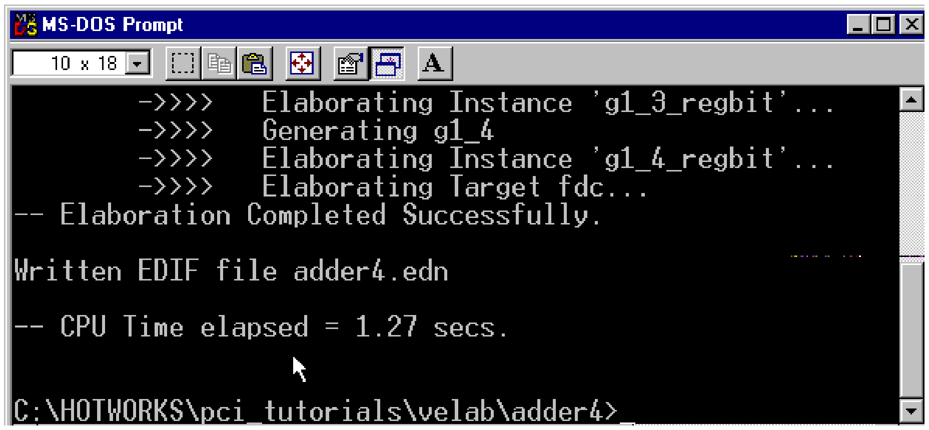
```
C:\>cd Hotworks\pci_tutorials\velab\adder4

C:\HOTWORKS\pci_tutorials\velab\adder4>velab adder4 adder4
```

Figure 29

⇒ At the DOS promptin the `c:\Hotworks\Pci_tutorials\Velab\adder4` directory, run Velab on adder4.vhd file as shown.

```
> velab adder4 adder4
```

This instructs **Velab** to elaborate the adder4 entity, using the file adder4.vhd. For the correct setup of **Velab** and the velab.ini file consult the Release_notes.txt in the Velab directory.
A lot of status information will be given concluding with some messages about elaboration having been successful, adder4.edn written and the amount of time taken. If not, please check you have Velab set up as described in the Release_notes.txt and that the VHDL file adder4.vhd is unchanged.



```
        ->>>>   Elaborating Instance 'g1_3_regbit'...
        ->>>>   Generating g1_4
        ->>>>   Elaborating Instance 'g1_4_regbit'...
        ->>>>   Elaborating Target fdc...
-- Elaboration Completed Successfully.

Written EDIF file adder4.edn

-- CPU Time elapsed = 1.27 secs.


C:\HOTWORKS\pci_tutorials\velab\adder4>_
```

Figure 30

You should now have**`adder4.edn`** `(the adder4 EDIF Format file)` in your

`c:\Hotworks\Pci_tutorials\velab\adder4\` directory.

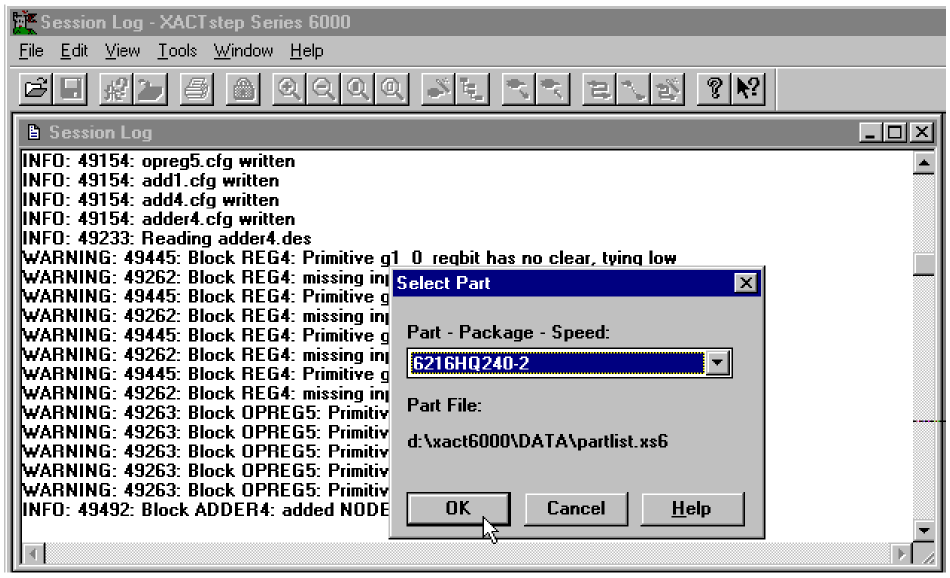
2. Place and Route your design using XACT6000

You now have your design as an EDIF (.edn) file. If, for some reason, you were unable to complete part1, rename the file adder4bk.edn to adder4.edn. You will notice there is another file already in this directory; adder4.cmd. This is for testing your design later on.

⇒ Run XACT6000.

⇒ Open the **`adder4.edn`** EDIF file you created in step 1 in
`C:\Hotworks\Pci_tutorials\XACT6000\adder4\adder4.edn.`

⇒ Choose the part that corresponds to the part you have on your board. Current H.O.T. Works Boards have the **6216HQ240-2 part** in them so choose this option when prompted by XACT6000 to Select a Part. You will see some warnings in the session log about the input registers having no inputs and the output register having no output. This is because they will be read and written by your program through the FastMap® interface, not via pins.


Session Log - XACTstep Series 6000
File Edit View Tools Window Help
Session Log
Select Part
OK
Cancel
Help

```
INFO: 49154: opreg5.cfg written
INFO: 49154: add1.cfg written
INFO: 49154: add4.cfg written
INFO: 49154: adder4.cfg written
INFO: 49233: Reading adder4.des
WARNING: 49445: Block REG4: Primitive g1_0_regbit has no clear, tying low
WARNING: 49262: Block REG4: missing in
WARNING: 49445: Block REG4: Primitive g
WARNING: 49262: Block REG4: missing in
WARNING: 49445: Block REG4: Primitive g
WARNING: 49262: Block REG4: missing in
WARNING: 49445: Block REG4: Primitive g
WARNING: 49262: Block REG4: missing in
WARNING: 49263: Block OPREG5: Primitiv
WARNING: 49263: Block OPREG5: Primitiv
WARNING: 49263: Block OPREG5: Primitiv
WARNING: 49263: Block OPREG5: Primitiv
WARNING: 49263: Block OPREG5: Primitiv
INFO: 49492: Block ADDER4: added NODE
```

Part - Package - Speed: 6216HQ240-2

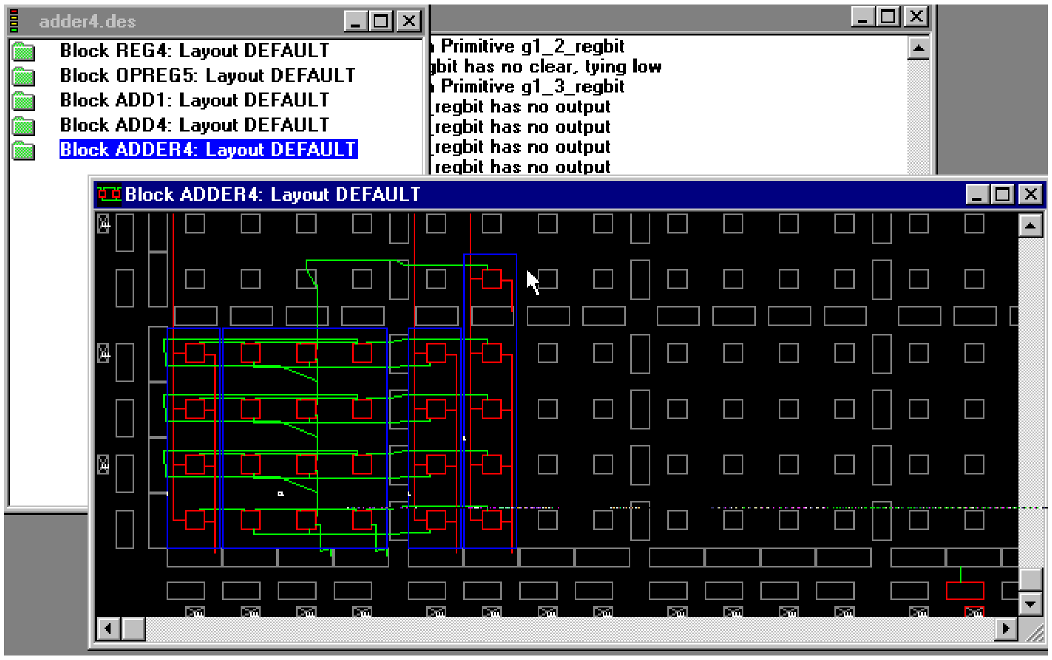
Part File:

d:\xact6000\DATA\partlist.xs6

Figure 31

⇒ The Design Wizard will then offer to Place and Route the whole design for you. Say Yes. If you have previously opened the design, XACT6000 will ask what you want to do with the old blocks, choose "For All Blocks to Overwrite With New". Automatic Place and Route will then run.

⇒ Open the **top level layout** for the design by double clicking on block Adder4 in the Design Browser. You will see the blocks representing part of your design. Your registers should appear in the layout at the positions spcified in the RLOC constraints contained in the VHDL source.



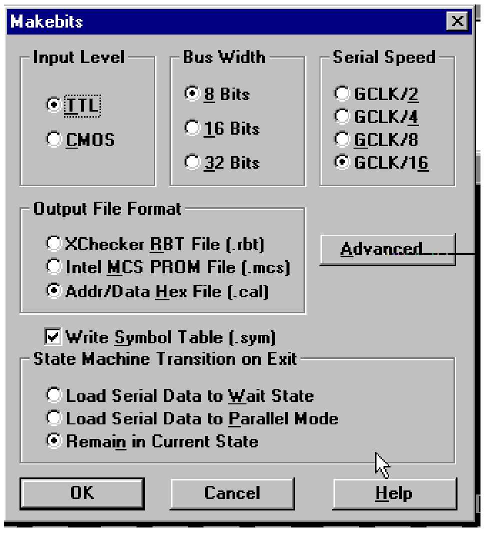
Block REG4: Layout DEFAULT
Block OPREG5: Layout DEFAULT
Block ADD1: Layout DEFAULT
Block ADD4: Layout DEFAULT
Block ADDER4: Layout DEFAULT

```
Primitive g1_2_regbit
bit has no clear, tying low
Primitive g1_3_regbit
regbit has no output
regbit has no output
regbit has no output
regbit has no output
```

Figure 32

⇒ Select a **block in the top level layout** by clicking on it. Then click the right mouse button. This displays a useful menu. Selecting Edit layout will open the layout for that block.

This design should place and route without any problems. Check that the top level layout has a green icon indicating full placement and that there are no unroutes as indicated in the status bar and select the Makebits option from the File menu. Keeping the default settings (TTL, 8bits, GCLK/16, Addr/Data Hex File, Write Symbol Table, Remain in Current State), write out your CAL file, calling it adder4.cal.


Makebits

Input Level: [✓ TTL] CMOS

Bus Width: [✓ 8 Bits] 16 Bits 32 Bits

Serial Speed: GCLK/2 GCLK/4 GCLK/8 [✓ GCLK/16]

Output File Format: XChecker RBT File (.rbt) Intel MCS PROM File (.mcs) [✓ Addr/Data Hex File (.cal)]

[✓ Write Symbol Table (.sym)]

State Machine Transition on Exit: Load Serial Data to Wait State Load Serial Data to Parallel Mode [✓ Remain in Current State]

Figure 33

You now have your design placed and routed using **XACT6000**. You have used XACT6000 to generate a CAL file called adder4.cal and a symbol table, SYM file called adder4.sym, using the Makebits command. The CAL file contains the configuration data necessary to program your XC6200 device. The SYM file contains the addresses on the chip of all the symbols in your design.

Don't worry about the contents of these files at this stage. You are ready to test your design on the board.

3. Use PciTest to check your design

Before you write any code you can test your design using **PciTest**.

PciTest allows you to test your design by loading the CAL file onto the board and controlling it from the command line. You can type individual commands on the command line or you can run a command file. The command file adder4.cmd contains the following commands.

```
!Reset the board
reset
!Load design (CAL File named adder4.cal)
load adder4.cal
!Set map register for reg a
setreg map_1 ffffffff
setreg map_0 fffffff0
!Write 1010 to rega
wcol 0 a
!Map register for reg b same as for reg a so don't need to reset
!Write 0010 to regb
wcol 4 2
!Set map register for sum
setreg map_1 ffffffff
setreg map_0 ffffffe0
!Read sum
rcol 5
```

Register A lies in column 0 and occupies rows 0..3. Register B lies in column 4, occupying rows 0..3 and Register SUM lies in column 5 occupying rows 0..4 . This is seen clearly in the **XACT6000** layout editor for the top level design.

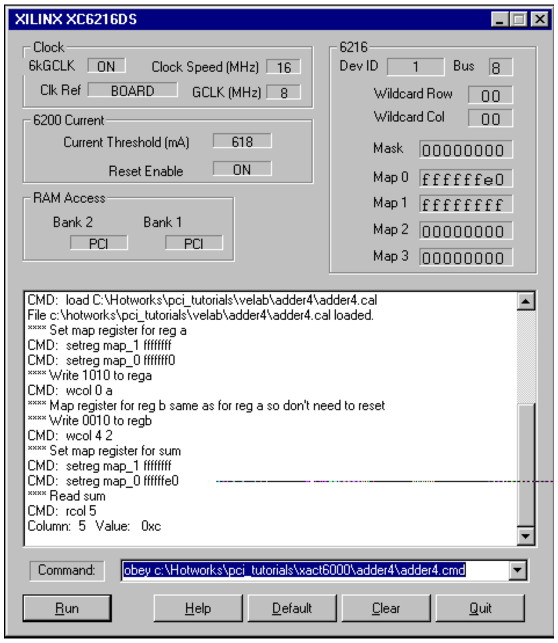
The Map Register

You read from and write to registers by selecting the register column and then connecting only the rows in which your register lies to the bus. The rows connected to the bus are determined by the 64 bit Map Register on the **XC6200**. To select a row you set the corresponding bit in the Map Register to 0.

```
      setreg map_1 ffffffff
      setreg map_0 fffffff0
```

sets the map register to allow the bits in rows 0 to 3 onto the lowest 4 bits of the bus.

⇒ **Run** PciTest.



```
CMD:  load C:\Hotworks\pci_tutorials\velab\adder4\adder4.cal
File c:\hotworks\pci_tutorials\velab\adder4\adder4.cal loaded.
**** Set map register for reg a
CMD:  setreg map_1 ffffffff
CMD:  setreg map_0 fffffff0
**** Write 1010 to rega
CMD:  wcol 0 a
**** Map register for reg b same as for reg a so don't need to reset
**** Write 0010 to regb
CMD:  wcol 4 2
**** Set map register for sum
CMD:  setreg map_1 ffffffff
CMD:  setreg map_0 ffffffe0
**** Read sum
CMD:  rcol 5
Column:  5   Value:   0xc
```

```
obey c:\Hotworks\pci_tutorials\xact6000\adder4\adder4.cmd
```

Figure 34

⇒ Run the command file **adder4.cmd** N.B. assumes you have a standard H.O.T. Works installation.

```
obey c:\Hotworks\pci_tutorials\xact6000\adder4\adder4.cmd
```

at the PciTest command prompt. You will notice that hex values are read back when using the rcol command. If the register read is only 5bits wide, as in this example, any bits above that should be ignored. You should see a result, perhaps 0x4C, equivalent to 0100 1100. Take the lower 5 bits to get the answer to the addition.

You have shown that your design works for one data example. You may wish to perform more thorough testing at this stage by editing your command file to input further data sets or by typing commands at the command line, before going on to the next stage.

4. Design applications software for the XC6200

The H.O.T. Works Development System provides a C++ class library for controlling your design on the board. XACT6000 is supplied with RALLib, a C++ library which supports symbol table access and reconfiguration. Tutorial 3 focuses on reconfiguration.

To load a CAL file and control the design on the XC6200, you need to write a C++ program which calls functions from the**XC6200** PCI board interface classes. There are 3 classes XC6200DS, PCIBoard and PCICore.

About the XC6200 PCI Board Interface

XC6200DS is the top level class and provides a complete interface to the XC6200 Development System. This class contains all the 6200 device access functions, including reading and writing of control registers and loading and accessing the user design.

PCIBoard is a subclass of the XC6200DS class. This class contains the board IO space register access functions.

The supporting low level class PCICore handles the actual reading and writing to the PCIBoard as a memory mapped device. This class interfaces to the device driver.

⇒ Copy your **`adder4.cal`** and **`adder4.sym`** files from the

`c:\HotworksPci_tutorials\XACT6000\adder4\` directory to the
`c:\Hotworks\Pci_tutorials\CPP\adder4\` directory.

The files `adder4bk.cal` and `adder4bk.sym` are provided as a backup in case you were unable to complete part 3. Rename them adder4.cal and adder4.sym.

⇒ Open the files **`adder4.cpp`** and **`adder4.h`** in directory

`c:\Hotworks\Pci_tutorials\CPP\adder4\.`

Accessing registers in your design

You write a C++ program which calls functions from the **RALSymbolTable** class to extract map register values and column position for each of the symbols in your design. You use **XC6200DS** library to control your board. Functions are provided to load your design and to read and write data. When reading back data you should be aware that only the lower n bits on the bus are defined, where n is the number of 0s in the Map Register. See XC6200 data sheet.

The Data Bus

The bus is 32 bits wide but can be set to 8 or 16 bits wide if you need the bus for other things while you are reconfiguring. When reading back a 4 bit value from a register on the **XC6200** you need to mask off the upper 28 bits since they are undefined. For a 4 bit number, this is done by ANDing with 1111_2 or 0xF or by use of the Mask Register. See XC6200 data sheet.

At the top of adder4.h you will find the class Register defined as follows.

```
class Register
{
public:
Register    ( const char *name,  //Register instance name
       RALSymbolTable *symTab,   //XACT6000 sym table
       XC6200DS *xc6200ds );     // Board object

void write( word value );        // Write to a register
word read();                           // Read from a register

private:
   XC6200DS *_xc62;              // Pointer to the board
   MapRegister _mapReg;          // Set for each register
   word       _busMask;          // Value to mask out undefined
                                 // bits on the bus i.e.
             // a 5bit value on a 16bit wide
                   // bus is ANDed with 1F
      word _column;              // column of 6200 where register
                                 // lies
};
```

For the registers A, B and SUM of the adder4 design the fields are initialized for each register in the constructor, see file adder4.cpp. Also in adder4.cpp you will find two functions, read and write, on the class Register which use **XC6200DS** functions to read from and write to the **XC6200**.

Initializing the XC6200 from your program

The class HardwareResources is also defined in adder4.h.

```
class HardwareResources
{
public:
       Register _rega, // Three registers in the design
       _regb,
       _regsum;

       HardwareResources(  const char *aname,
                           const char *bname,
`                          const char *sumname,
                           RALSymbolTable *symTab );

BOOL initialiseBoard();

private:
XC6200DS _xc6200ds;        // Board object
RALSymbolTable *_symTab;   // XACT6000 symbol table };
```

The following code in adder4.cpp initialises the **XC6200**, sets the clock frequency and bus width, and loads the CAL file for the 4 bit adder. You need the following include files.

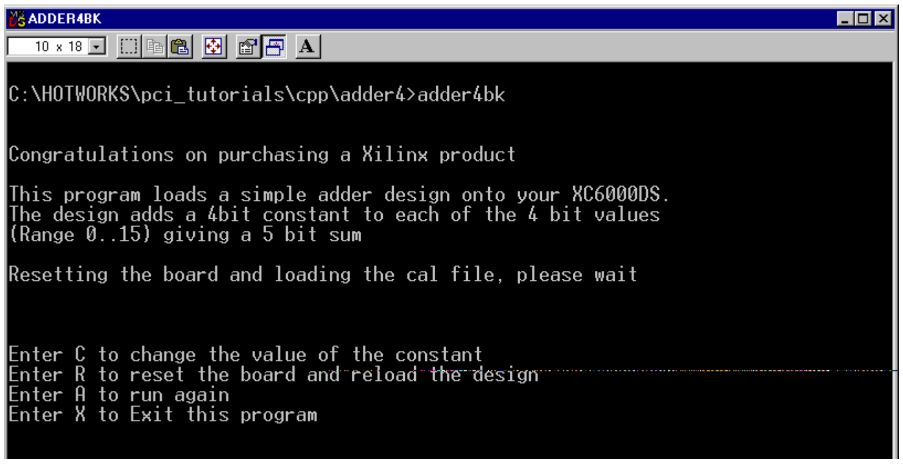
```
#include "xc6200ds.h" //Board interface
#include "ralstab.h" //Symbol Table classes

BOOL HardwareResources::initialiseBoard()
//Initialise the design and load the CAL file
{
       // Reset the board
       if ( _xc6200ds.checkBoard())
       {
             return FALSE; //Board initialisation failed
       }

       _xc6200ds.reset();

       // Set the clock frequency XC6200 clock initialised
       // to 66/2 MHz
             float ref_freq;
             float des_freq;
             float freq;

             ref_freq = (float) 16.0;
             des_freq = (float) 66.0;
       _xc6200ds.setRef(0);
// make sure we are taking
// the clock from the board
             freq = _xc6200ds.setClockFreq(ref_freq, des_freq);

             if ( freq != des_freq )
             {
              return FALSE; // Cannot set clock to this freq
             }
             //Set the bus width, one of 8, 16 or 32
      _xc6200ds.setBusWidth( 32 );

             // Load the adder4 design onto the XC6200
      int result = _xc6200ds.loadCalFile("adder4.cal");

      return TRUE;
} //initialiseBoard
```

The clock frequency you use depends on the critical path of your design. Look at the Timing Analyzer section in the **XACT6000** online help or in XACTDEMO to find out more. The function set_clock_freq() sets the board clock to the desired value. The clock to the **XC6200** is set to half of this value. More about clocking designs in Tutorial 2.

N.B. The CAL file sets the bus width. When you generated the CAL file using the Makebits command you chose a bus width. This should be 32 unless you plan to use some wires of the bus for other things while you are reconfiguring your chip.

⇒ **Compile your program** including the RALLib files and the PCI board interface class files in your project.

⇒ **Run** the program to check that your adder works properly. An executable adder4bk.exe is provided.

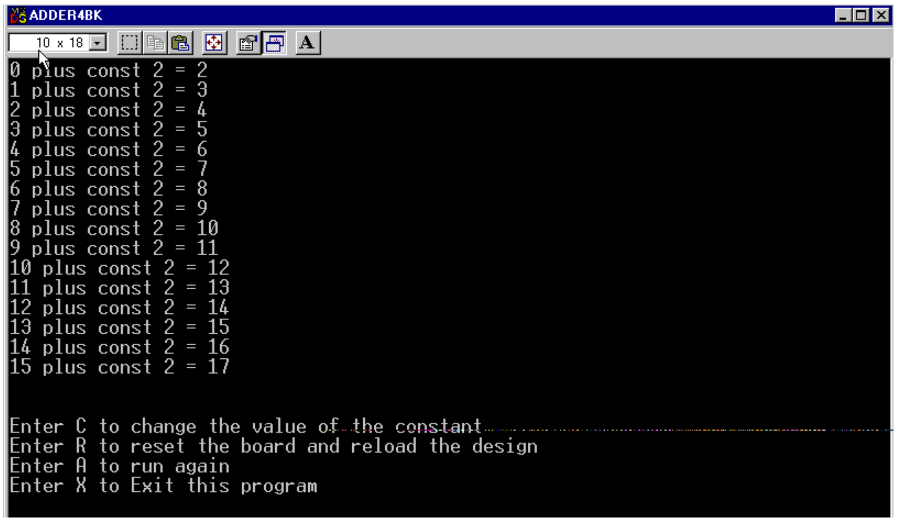


```
C:\HOTWORKS\pci_tutorials\cpp\adder4>adder4bk


Congratulations on purchasing a Xilinx product

This program loads a simple adder design onto your XC6000DS.
The design adds a 4bit constant to each of the 4 bit values
(Range 0..15) giving a 5 bit sum

Resetting the board and loading the cal file, please wait



Enter C to change the value of the constant
Enter R to reset the board and reload the design
Enter A to run again
Enter X to Exit this program
```

Figure35

Debugging design under a C++ debugger

To allow C++ and software to be used as a test bench for hardware, any registers other than those used for wireless I/O must be clocked by the software. Tutorial 2 shows how this can be implemented.



```
0 plus const 2 = 2
1 plus const 2 = 3
2 plus const 2 = 4
3 plus const 2 = 5
4 plus const 2 = 6
5 plus const 2 = 7
6 plus const 2 = 8
7 plus const 2 = 9
8 plus const 2 = 10
9 plus const 2 = 11
10 plus const 2 = 12
11 plus const 2 = 13
12 plus const 2 = 14
13 plus const 2 = 15
14 plus const 2 = 16
15 plus const 2 = 17


Enter C to change the value of the constant
Enter R to reset the board and reload the design
Enter A to run again
Enter X to Exit this program
```

Figure36

5. Details about topics learned in this tutorial

In this tutorial you learned about the following:

Topic	For more information
Attaching constraints to your VHDL design	XACT6000 help "Constraints"
Generating EDIF (.edn) files using Velab Velab,	Release_notes.txt
Using XACT6000 Design Wizard to place and route your design	XACT6000 help "Tools Menu / DesignWizard"
Viewing Layouts of individual blocks	XACT6000 help "Tools Menu / Edit layout"
Using Makebits to generate CAL and SYM files	XACT6000 help "Design Flow / Makebits"
Using PciTest to test your design	PciTest Help
Initializing the XC6200 on your board with your design	XC6200 datasheet and XC6200DS library documentation
Selecting a column and setting the Map Register	XC6200 datasheet and XC6200DS library documentation
Writing data from your control program to a register in your design	XC6200 datasheet and XC6200DS library documentation
Reading data back from the XC6200 and masking unwanted bits	XC6200 datasheet and XC6200DS library documentation

Tutorial 2

Proceed as follows:

1. Clocking a pipelined design: VHDL
2. Place and Route with XACT6000
3. Test with PciTest
4. Run the control software
5. Details about topics learned in this tutorial

1. Clocking a pipelined design: VHDL

Apart from the GCLK and GCLR global routing there are two other global routing resources on the **XC6200**: G1 and G2. See the XC6200 datasheet for more information on global signals. Global routing connects to every function unit on the **XC6200** and also to external pins. The **XC6200** datasheet has pinout details for the different packages. For the purposes of clocking, you can elect to drive G1 or G2 with your own clock.

Global Clock

The global clock, GCLK, must be connected to all input and output registers.

You may want to clock your design in single step mode, or give 50 clock pulses and then read back a result. To do this you turn GCLK off and call the clock_step function.

However, reading a value from a register will cause GCLK to step 3 times in single step mode and writing a value to a register will cause GCLK to step twice. This makes debugging difficult if the entire design is clocked by GCLK. It is impossible to get a “snapshot” of many registers on the same clock pulse. For this reason it is often better to clock your design using a separate clock altogether and only use GCLK to synchronize input/output registers.

CBUF Clock

The example we are going to look at in this tutorial uses CBUF clocking to drive global routing G1 for a pipelined design.

A CBUF_OUT is attached to a register using the RSIGNAL constraint. A clock pulse is generated upon each access to the register. This is obtained from a CBUF_OUT that detects the access indicated by the RSIGNAL constraint. There are two parameters to the RSIGNAL constraint. The first parameter can be either REGWORD or REGDATA. REGWORD generates a clock pulse on a column access; REGDATA generates a clock pulse on a row access. The second parameter is the name of the register.

⇒ Open **`adder4p.vhd`** in directory `C:\Hotworks \Pci_tutorial\velab\adder4p`. Look for the definition of the instance REG4CLK.

Notice how the RSIGNAL attribute works.

```
ATTRIBUTE RSIGNAL OF CB : LABEL IS "REGWORD:REG0";
```

The CBUF_OUT, CB in this design, is attached to bit 0 in the register with the above attribute. Every time the REGWORD signal for that column goes low, the CBUF_OUT output will go low. Since a high pulse is required, this clock signal is put through an inverter and then into a BUFGP to drive a global signal.

```
ATTRIBUTE GLOBAL OF BGSHCLK : LABEL IS "G1";
.
.
.
-- SET UP THE CBUF TO BE GLOBAL CBUF CLK
   CB : CBUF_OUT PORT MAP (O => SHCLKBAR);
-- INVERT THE CLOCK
   SHCLKINV : INV PORT MAP (I => SHCLKBAR,
                        O => SHCLKTEMP);
```

```
-- DRIVE A GLOBAL SIGNAL WITH THE CBUF CLK
   BGSHCLK : BUFGP PORT MAP (I => SHCLKTEMP,
                       O => SHCLK);
```

Either take care to place all registers you access in different columns from those with CBUF_OUTs attached and use only the REGWORD parameter, or attach two CBUF_OUTs to the register you wish to use to generate the clock; one constrained with REGWORD and the other, REGDATA. Then NOR them together to obtain a high clock pulse when both row and column signals go low.

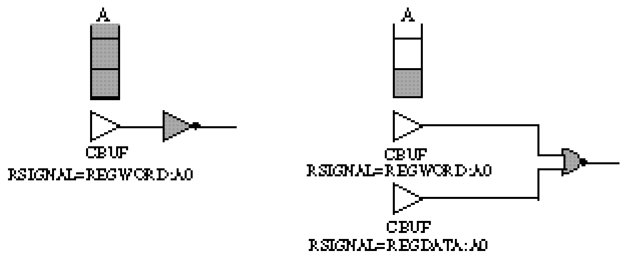


Figure 37

Clearing registers in your design

Connecting registers with a CLR input to GCLR allows them to be synchronously cleared. One way of doing this in VHDL is as follows. Again the BUFGP is used to drive the global signal.

```
A_ZERO : GND PORT MAP(GROUND => CLRTMP);

CLRBUF : BUFGP PORT MAP (I => CLRTMP, O => GCLR );
```

⇒ **Make** the EDIF for adder4p using velab

```
> velab adder4p adder4p
```

2. Place and Route your design using XACT6000

You have elaborated your VHDL design and it is now in EDIF format.

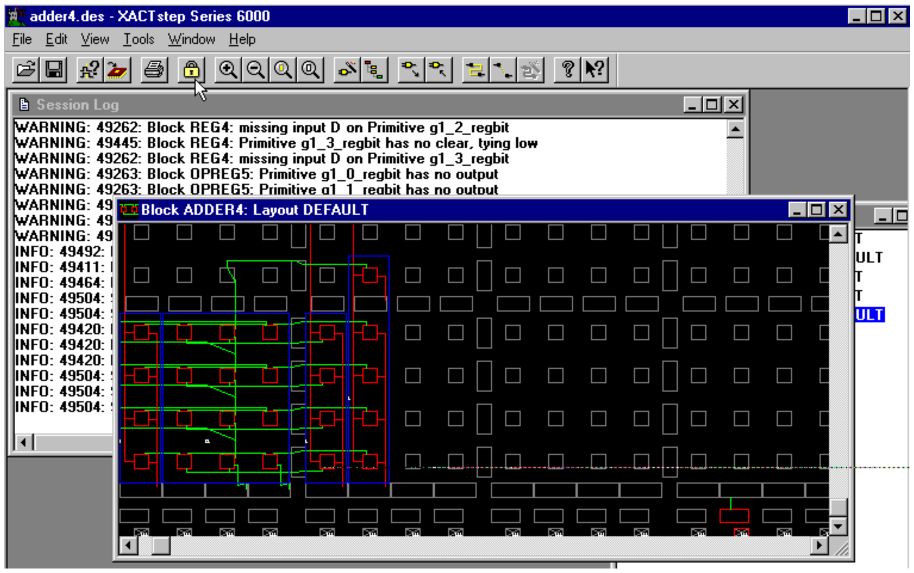
⇒ Copy the file `adder4p.edn` from the `velab\adder4p` directory to the `XACT6000\adder4p` directory.

There is an `adde4pbk.edn` file provided for you if you were unable to complete part 1. Rename it to adder4p.edn.

⇒ Open the file using **XACT6000**.

⇒ Select part 6216HQ240-2 and allow the Design Wizard to place and route the whole design.

⇒ Open the top level layout. See Figure 10


adder4.des - XACTstep Series 6000
File Edit View Tools Window Help
Session Log
Block ADDER4: Layout DEFAULT

```
WARNING: 49262: Block REG4: missing input D on Primitive g1_2_regbit
WARNING: 49445: Block REG4: Primitive g1_3_regbit has no clear, tying low
WARNING: 49262: Block REG4: missing input D on Primitive g1_3_regbit
WARNING: 49263: Block OPREG5: Primitive g1_0_regbit has no output
WARNING: 49263: Block OPREG5: Primitive g1_1_regbit has no output
WARNING: 49
WARNING: 49
WARNING: 49
INFO: 49492:
INFO: 49411:
INFO: 49464:
INFO: 49504:
INFO: 49504:
INFO: 49420:
INFO: 49420:
INFO: 49420:
INFO: 49504:
INFO: 49504:
INFO: 49504:
```

Figure 38

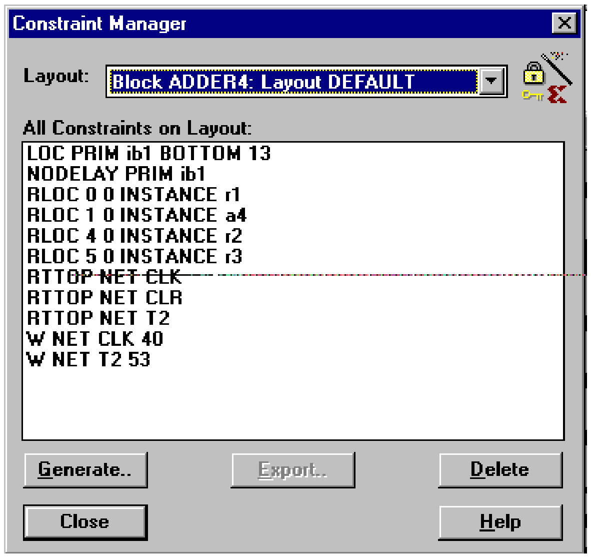
Selecting the part 6216HQ240-2 assumes that you have the XC6216 part on your board. If you have the XC6264 part on your board, you should choose 6264HQ240-2 instead. N.B. You can also use the PART constraint in your VHDL or schematic design.

⇒ When you have placed and routed the design using the Design Wizard you will see routing going from the inverter in position (5,4) to the IOB at the bottom in column 17. G1 is driven from the IOB at bottom 17 so the CBUF clock connects to this IOB.

Global routing connects to all Function Units so it is only shown in the Layout Editor at the 4x4 boundaries (Clk routing) and the 16x16 boundaries (Clr routing). The **XC6216** data sheet shows the pins that have a dedicated signal and how they map onto the pads on the **XC6216**. This is different for all packages.

Notice also that the FDCs are incorporated into the add4 block. This was facilitated by use of the FLATTEN constraint which flattened the hierarchy allowing the FDCs to be mapped in to the same function unit as the gate driving the input.

⇒ From the Edit menu choose Constraint Manager. This shows all the attributes you attached to the symbols in your current layout. You can use the Constraint Manager to generate all the constraints necessary to reproduce the current layout and use them to back annotate to your original VHDL or schematic design. See figure 11.



LOC PRIM ib1 BOTTOM 13
NODELAY PRIM ib1
RLOC 0 0 INSTANCE r1
RLOC 1 0 INSTANCE a4
RLOC 4 0 INSTANCE r2
RLOC 5 0 INSTANCE r3
RTTOP NET CLK
RTTOP NET CLR
RTTOP NET T2
W NET CLK 40
W NET T2 53

Figure 39

⇒ Click on the Generate button. You will see many new constraints have been generated. They all have the keyword "Editor" before them. Clicking on export will allow you to save them as a text file. Use the information in this text file to constrain your design and reproduce the current layout.

⇒ Select the input register REG4CLK in column 4 of the top level layout. Open the layout using a right mouse button click and choosing Edit Layout from the menu. The top instance should be an inverter. Examine the constraints for this block by choosing Constraints Manager from the Edit menu or clicking on the padlock icon. Notice the RSIGNAL constraints discussed in part 1.

⇒ Use Makebits to generate the CAL and SYM files for the whole design.

3. Test with PciTest

⇒ Use **PciTest** to check that your design is working.

⇒ With your text editor, open the command file **`adder4p.cmd`** in directory

`c:\Hotworks\Pci_tutorial\XACT6000\adder4p.`

```
!Reset the board
reset
!Load design
load adder4p.cal
!Set map register for reg a

setreg map_3 ffffffff
setreg map_2 ffffffff
setreg map_1 ffffffff
setreg map_0 fffffff0

!Write 1010 to rega
wcol 0 a

!Map register for reg b same as for reg a so don't need to reset
!Write to regb and tick clock
wcol 5 1

!Set map register for sum
setreg map_1 ffffffff
setreg map_0 ffffffe0
!Read sum
rcol 6
!Set map reg for reg b
setreg map_0 fffffff0
!Write  to regb and tick clock
wcol 5 2
!Set map register for sum
setreg map_0 ffffffe0
!Read sum
rcol 6
!Set map reg for reg b
setreg map_0 fffffff0
!Write  to regb and tick clock
wcol 5 3
```

```
!Set map register for sum
setreg map_0 ffffffe0
!Read sum
rcol 6
!Set map reg for reg b
setreg map_0 fffffff0
!Write  to regb and tick clock
wcol 5 4
!Set map register for sum
setreg map_0 ffffffe0
!Read sum
rcol 6
```

You will see that the test for this design involves alternately writing to the input register and reading from the output register. Writing to the input register clocks the data along the pipeline.

The command file writes a value, the constant, to the register in column 0 and then writes the data to the input register. The pipeline has a delay of one so when reading the output from the output register it should be the sum of the constant plus the previous value.

⇒ Run **PciTest** and check that the design is working properly by running the test command file.

cmd:`obey c:\xc6000ds\xact6000\adder4p\adder4p.cmd`

You should see the following results on reading column 6.

*x*a, *x*b, *x*c, ... where *x* represents the upper 3 bits which are undefined.

4. Run the control software

⇒ Copy the CAL and SYM files from the `XACT6000\adder4p` directory to the `CPP\adder4p` directory. Rename the backup files provided if you need them.

Little change has been made to the code for the design in Tutorial 1. The code now reads the CAL and SYM files for the adder4p design.

⇒ Compile `adder4p.cpp` and run the program or click on `adder4p.exe` provided.
Notice the delay of one clock cycle in the sum.

5. Details about topics learned in this tutorial

In this tutorial you learned about the following:

Topic	For more information
CBUF_OUTs	XACT6000 help “Libraries”
REGWORD REGDATA and FLATTEN constraints	XACT6000 help “Constraints”
XACT6000 Constraint Manager	XACT6000 help “Constraints Manager”

Tutorial 3

If you have completed Tutorial 1 and Tutorial 2 you know how to elaborate a design using **Velab**, constrain its placement and routing for **XACT6000**, use **PciTest** for testing, and use the **XC6200DS** and **RALSymbolTable** classes in your code. Tutorial 3 will teach you about gate reconfiguration.

The **XC6200** allows rapid reconfiguration of individual gates. Given that it takes around 300ns to reconfigure each gate (if your board clock is set to 66MHz), hardwiring data, instead of storing it in registers, can be done repeatedly and allows maximum space efficiency. This tutorial shows you how to reconfigure gates following the usual design flow.

1. VHDL
2. XACT6000
3. Control software: Using RALLib

The design adder4rc performs the same function as adder4 in tutorial 1 but instead of having a 4bit register for the constant value, the constant is “hardwired” into the gates. So we can replace the first XOR in the standard 2 bit adder with either an inverter or a buffer depending on the value of the bit.

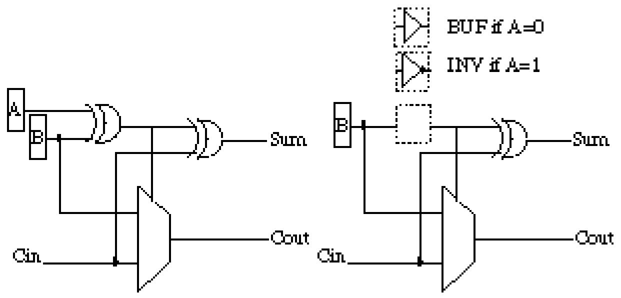


Figure 40

1. VHDL

We design our logic as before choosing a base gate for the reconfigurable gate. This base gate must always be a gate with the maximum number of possible inputs. For example, if a gate had possible configurations, XOR2, AND2, INV and BUF, the base configuration chosen should be the XOR2 or AND2 gate. This is to ensure that all the nets are available for reconfiguration.

So when reconfiguring from a two input gate to a one input gate, all that is necessary is to give information about which net to connect. In this simple design, there are two options for reconfiguration, both with one input from the same net.

⇒ Open the file **`adder4rc.vhd`** in directory

```
c:\Xc6200ds\pciv102\pci_tutorials\velab\adder4rc\.
```

Notice that now the adder only has one input register. If you look at the 1 bit adder block you will see that it now comprises an inverter labelled RCXOR, a MUX and an XOR gate. The gate labelled RCXOR has replaced both the original XOR and the input register, and will be one of either an inverter or a buffer in the final design depending on the bits of the constant.

The line

```
ATTRIBUTE RECONFIG OF RCXOR : LABEL IS "";
```

tells **XACT6000** to treat this gate as reconfigurable. This means that no magic routing will be allowed within the cell and that reconfiguration data will be written for this instance to the RAL file.

⇒ Use Velab to write the Edif for the `adder4rc.`

```
>velab adder4rc adder4rc
```

2. XACT6000

⇒ Copy your `adder4rc.edn` file from the `c:\Hotworks\pci_tutorials\velab\adder4rc` directory to the `c:\Hotworks\pci_tutorials\xact6000\adder4rc` directory. `add4rcbk.edn` is provided as a backup in case you were unable to complete the first section. Change its name to adder4rc.edn.

⇒ Open **`adder4rc.edn`** using XACT6000 and allow the Design Wizard to Place and Route it.

You will notice the amount of space and routing resources has been reduced when comparing this version with the ordinary adder in tutorial 1 although functionality is the same.

⇒ Choose **Makebits** from the File menu.

⇒ Select the option Write RAL file, leave the rest as default and click on OK.

This writes out the CAL, RAL and SYM files for the design. N.B. RAL will only be written when the Symbol Table write is enabled.

RAL files

The RAL file holds all the data to perform all possible reconfigurations for the gates which were given a RECONFIG constraint in the VHDL. Extracting the data from the RAL file for your chosen gate reconfiguration is done using classes in **RALLib**.

The CAL file contains the base design. The SYM file contains the information necessary to access the logic symbols in the design.

3. Control software: Using RALLib

The RAL class library uses the RAL and SYM files to return reconfiguration data for any gate with the RECONFIG constraint attached. All the possible reconfigurations of the base gate are stored in the RAL file and all that is necessary is to call the reconfigure function with the full instance name, the nets to be connected and the function required. The address data pairs returned will reconfigure the function unit to the chosen function by sending data to the board via the XC6200DS class.

⇒ Open the files **`adder4rc.cpp`** and **`adder4rc.h`**.

In the main function at the bottom of the `adder4rc.cpp` file you will see the following line:

```
RALConfigTable ralTab("adder4rc");
```

The RALConfigTable constructor loads both the Symbol Table file and the RAL file into the object.

Notice how the HardwareResources constructor sets up two registers as before, and the hierarchical instance names to the four reconfigurable gates before calling the ReconfigGate constructor for each.

The function reconfigure on HardwareResources selects the new gate function on the value of each bit of the constant. The RALConfigTable function getConfig returns the configuration data required to reconfigure a gate with a given function on the given nets. This data is used directly to configure a function unit on the XC6200. You will find this code in adder4rc.cpp function reconfigure.

N.B. The RAL data is only 8bits wide so it is very important to set the bus width to 8 before writing the configuration data to the function unit, and to set it back to 32 after reconfiguration.

```
bool success;
CPair *config; // Pointer to configuration data
int count; // Number of address data pairs
       // returned
_xc6200ds.setBusWidth( 8 ); // RAL is 8 bits

for ( int j = 0; j < 4; j++ )
{ // Get configuration for each recongfigurable gate
       success = _ralTab->getConfig(

              _rega[j]->_full_instance_name,
              _rega[j]->_function,
       _rega[j]->_net0,
       config, count );
     // Gate in table: reconfigure XC6200 on the board
       if ( success == true
          { for ( int i = 0; i < count; i++ )
              {
                     // Two addr data pairs reconfigure the
              // function of a FU
       _xc6200ds.write6200(
                     config->address,
                     config->data );
              config++;
              }
          }
       }
_xc6200ds.setBusWidth( 32 );
```

⇒ Copy the files `adder4rc.cal, adder4rc.sym` and `adder4rc.ral` into the `cpp\adder4rc` directory.

⇒ Compile `adder4.cpp` with the RALlib files and the XC6200DS classes to run this code. An executable `add4rcbk.exe` is provided for you.

4. Details about topics learned in this tutorial

In this tutorial you learned the following.

Topic	**For more information**
Designing Circuits for reconfiguration	Xilinx Application Note “ A 32x16 pixel correlator for the XC6200”
RECONFIG constraint	XACT6000 help / “ Constraints”
CAL, RAL and SYM files	XACT6000 help / “ Makebits”
RALLib reconfiguration functions	RALLib documentation

Tutorial 4.

There are two banks of RAM on your H.O.T. Works board. Either the Pci bus or the XC6200 can control both banks, or they can control one each. This tutorial will show you how to use the RAM. Using the same design as in previous tutorials, the values to be added to the constant will be written from the control program to Bank 1 of the RAM. XC6200 will then be given control of the RAM, read the values in RAM Bank 1 and write results to RAM Bank 2. The control program will read back the final results from RAM Bank 2.

1. Controlling the RAM: VHDL

The SRAM Macro

You are provided with an SRAM macro which is used to interface to the RAM. The macro takes an 18 bit address and will write 32 bit values to the RAM, or read from it, according to the 4 read/write bits. See Figure 13.

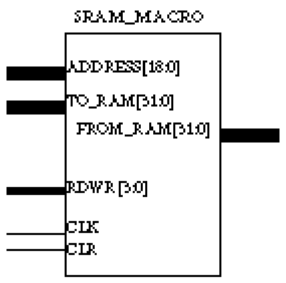


Figure 41

Inside this macro are all the pin connections and read/write control necessary to connect your addresses and data with the bus pins on the XC6216 device. The RDWR signals gate the data bus as follows, 0 allows a RAM write, 1 reads data from the RAM.

Read/Write bit	**Data**	**Bus RAM**
RDWR0	FROM_RAM[7:0]	Bank 1
RDWR1	FROM_RAM[15:8]	Bank 1
RDWR2	FROM_RAM[23:16]	Bank 2
RDWR3	FROM_RAM[31:24]	Bank 2

So it is possible to read data from one bank of RAM and simultaneously write to the other bank of RAM. Data bus bits 0..15 connect to RAM bank 1 and bits 16..31 connect to RAM bank 2.

The 4 banks of RAM on your board are of one of two types: 128k x 8 bit or 512k x 8 bit. Care must be taken when designing that addresses do not exceed 17 bits for the 128k x 8 RAM and 19 bits for the 512k x 8 bit RAM. The RAM access is rated at 15ns and 20ns respectively. This access time must be considered when designing.

⇒ Open the file `adder4rm.vhd` in the velab directory. This design takes the reconfigurable adder of tutorial 3 and feeds it with data from RAM 2 adding it to the constant value it has been reconfigured to, and write the sum to RAM 1.

You will notice that the input bits to the adder are bit 16..19 of the data out. These are the 4 least significant bits in RAM 2.

Generate the Edif for your design using the following command at the DOS prompt.

```
>velab adder4rm sram_mac adder4rm
```

2. Place and Route your design using XACT6000

⇒ Copy the Edif into directory XACT6000. Place and route the design using XACT6000.

⇒ Using the makebits command, write out the CAL, RAL and SYM files checking that you have the bus width set to 32.

N.B. Setting the bus width to 32 gives a 32 bit CAL file. RAL is always written in 8 bit mode. So when using RAL to reconfigure, take care always to have the control program set the bus width to 8 and then set it back to 32 after reconfiguration.

3. Using PciTest to check your design

From PCITest, it is possible to examine the RAM contents and write new values to chosen locations. To test the design, you will write values to the first 16 memory locations in Bank 1, run the adder which will add each one to the constant and write the result to the same location in Bank 2, and then examine the results in Bank 2.

Bank control and RAM reading and writing

The first thing to notice is how PCITest controls RAM bank ownership. The command bctl is used with a parameter. This parameter sets 2 bits in the control register, one for each bank of RAM. 0 indicates that the PCI bus controls the RAM and a 1 indicates that the 6200 controls the RAM.

bctl parameter	**Bank 1**	**Bank 2**
0	PCI	PCI
1	XC6216	PCI
2	PCI	XC6216
3	XC6216	XC6216

Try using the bctl command at the PCITest command line. PciTest displays bank control setting in the RAM Access area.

It is very important that the bank control matches what the RDWR signals are set to on the XC6216 RAM macro and good practise to change them in unison. To explain, never give the PCI bus control of the RAM while the XC6216 is set to write to it. This will cause bus contention. All RDWR bits must be set to 1 before attempting to examine any value via the PCI interface.

Also important is that data should be set up before changing addresses.

Notes on PCI addressing versus XC6216 addressing

When writing a set of data to the RAM the PCITest command fm is used.

```
Command: fm 0 12345678 10
```

This instructs PCITest to fill memory with the given value, in ths case starting at location 0, 12345678 is written 16 times. N.B. All parameters are HEX values.

Examining the memory is achieved in the following manner

```
Command: em 0 10
```

which examines the locaton at address 0 and the subsequent 15 32 bit values. Care must be taken when addressing single locations since the addresses via the PCI interface increment byte wise. E.g. the second 32 bit value is accessed as follows.

```
Command: em 4 1
```

Experiment with using PCITest to read from and write to RAM using the fm and em commands to familiarise yourself with the addressing.
When accessing the RAM from the XC6216, each bank is addressed at the same position but connected to different bits in the data bus. So an address increment of 1 addresses the next 32 bit value.

Test the reconfigurable adder design using the RAM

⇒ Open the command file `adder4rm.cmd.`

The board is initialised by resetting and limiting the current threshold to 200mA.
The board clock is then set to a value of 16MHz which sets GCLK to the XC6216 device to 8 Mhz.

The commands

```
bctl 0

!DATA FROM RAM2 ADDED TO CONST OP TO RAM1
!Adding 4LSBs in RAM 2 to const
!Write to RAM
fm 0 00018888 1
fm 4 00028888 1
fm 8 00038888 1
fm c 00048888 1
fm 10 00058888 1
fm 14 00068888 1
fm 18 00078888 1
fm 1c 00088888 1
fm 20 00098888 1
fm 24 000a8888 1
fm 28 000b8888 1
fm 2c 000c8888 1
fm 30 000d8888 1
fm 34 000e8888 1
fm 38 000f8888 1
fm 3c 00008888 1
```

give the PCI bus control of both RAM banks and fill the RAM with known values. The fm commands fill RAM2 is with values 0..f and RAM1 with 8888.

PCITest can also read XACT6000 SYM files. It will make a list of all the registers in your design and you can read and write to them on the XC6200 using the sym commands. The Map Register vlalues are set automatically.

```
lsym c:\Hotworks\pci_tutorials\xact6000\adder4rm\adde
r4rm
.
.
.
wsym RDWR_REG c
rsym RDWR_REG
```

The previous commands load the symbol table, write to the symbol RDWR_REG and read back from RDWR_REG. The lisym command lists all the register symbols in the design.

Clocking is by means of a CBUF_OUT (see tutorial 2). Writing to the register CLKREG in column 1 generates a clock pulse which increments the address counter accessing the next value to be added to the constant and simultaneously accessing the next location for storing the sum.

This example clocks 8 times.

Notice that before examining the results with the em command, care has been taken to ensure that the RDWR bits are all set to 1 to allow a read to take place via the PCI interface.

⇒ Run the command file using the following command, and check that the results you expect are being written to the 8 LSBs of RAM1. Remember that the adder has not been reconfigured so the data in RAM 1 is added to the fixed 4 bit constant value of 1111.

4. Designing applications software for the XC6200

⇒ Copy the files `adder4rm.cal`, adder4rm.ral and adder4rm.sym. Rename the backup files if you were unable to complete part 3.

In the files **`adder4rm.cpp`** and **`adder4rm.h`** there are two new functions on the HardwareResources class for reading and writing to the RAM. There is also a function for running the XC6200 RAM adder. Notice how these functions handle bank control and the RDWR registers.

⇒ Compile the file adder4rm.cpp linking in the XC6200DS class functions and RALLib.

⇒ Run your executable or the backup one provided to check that the design is working properly. You may want to use PCITest to examine the RAM afterwards to see the effect your program has had.

6. Details about topics learned in this tutorial

In this tutorial you learned about the following:

Topic	**For more information**
About the SRAM macro	SRAM Macro VHDL comments
RDWR controls	SRAM Macro VHDL comments
PCITest bctl command	PCITest help file
PCITest em and fm commands	PCITest help file
PCITest symbol table commands	PCITest help file
Control of RAM from program	XC6200DS class library documentation

Macro Liabraries

There are 30 macros in the included library files, all of them captured in structural VHDL and parametrised by word width. Some of them are also parametrised by pin separation. Placement constraints are specified using the attribute facility in VHDL, and all designs involve only nearest neighbor routing.

A PDF formatted document file can be found in:

```
C:\Hotworks\docs\Velab\paramlib.pdf
```

For each library, we include a short description of its function, its parameters and their default values, and the pin definitions, in addition to the VHDL code. Examples indicating the key attributes for each macro are provided; the attributes include its aspect ratio, the number of 6200 cells used, and the location of the input and output pins. Various circuit diagrams generated from Synopsys and XACTstep6000 tools are also included. A separate document containing only the VHDL entity definitions and the essential information for using the library designs, has also been compiled. See: `C:\Hotworks\docs\Velab\macros`

Footprint and delay characterization of these macros is provided, valid for the given layouts on 6216XXXX-2 parts. Delay characteristics do not take into account the routing to and from the modules. Extending these macros for other layouts and parts is a matter of recoding the attributes in the library VHDL files. This is left to the user.

Velab Examples Multiadd `C:\Hotworks\Velab\multiadd`

This example uses hardware or software to calculate ax + by, where a, x, b and y are 11-bit numbers, and the result is a 23-bit number.

⇒ To use :

1) Compile multadd.cpp and pci6200.cpp to get an executable. (The executable is included in case a compiler is not available).

2) Run Velab with arguments "multadd add_uc mult_uc multadd" to get multadd.edn.

3) Run XACT6000 on multadd.edn to get multadd.cal, which multadd.exe will use. (No user intervention is required - just let the design wizard go ahead, and use the default settings for the Makebits dialog).

4) Run multadd.exe. The -s command line option means that software will be used for the calculation, otherwise the XC6000DS will be used.

5) The program will generate a file "multadd.out" containing the details of the calculations made. To check the results from a hardware-based run, use the command-line option -s to

generate a results file, rename this to, say, multaddsw.out, and then run multadd.exe again without the -s option.

Then compare multaddsw.out and multadd.out..

Velab Examples Multiadd `C:\Hotworks\Velab\parldsr`

This example uses hardware or software to shift random 32 bit numbers into a 48 bit shift register.

⇒ To use:

1) Compile parldsr.cpp and pci6200.cpp to get an executable. (The executable is included in case a compiler is not available).

2) Run Velab with arguments "parldsr parldsr" to get parldsr.edn.

3) Run XACT6000 on parldsr.edn to get parldsr.cal, which parldsr.exe will use. (No user intervention is required - just let the design wizard go ahead, and eventually use the default settings for the Makebits dialog).

4) Run parldsr.exe. The -s command line option means that software will be used for the calculation, otherwise the XC6000DS will be used.

5) The program will generate a file "parldsr.out" containing the details of the calculations made. To check the results from a hardware-based run, use the command-line option -s to generate a results file, rename this to, say, parldsrsw.out, and then run parldsr.exe again without the -s option.

Then compare parldsrsw.out and parldsr.out.

Velab Examples popcnt `C:\Hotworks\Velab\popcnt`

This example uses hardware or software to calculate the number of '1' bits in an 8-bit number. The file lut.vhd was produced by the Synopsys Design Compiler from the file lut.pla, which defines the behavior of the circuit.

⇒ To use

1) Compile popcnt.cpp and pci6200.cpp to get an executable. (The executable is included in case a compiler is not available).

2) Run Velab with arguments "popcnt lut popcnt" to get popcnt.edn.

3) Run XACT6000 on popcnt.edn to get popcnt.cal, which popcnt.exe will use. (No user intervention is required - just let the design wizard go ahead, and eventually use the default settings for the Makebits dialog).

4) Run popcnt.exe. The -s command line option means that software will be used for the calculation, otherwise the XC6000DS will be used.

5) The program will generate a file "popcnt.out" containing the details of the calculations made. To check the results from a hardware-based run, use the command-line option -s to generate a results file, rename this to, say, popcntsw.out, and then run popcnt.exe again without the -s option.

Then compare popcntsw.out and popcnt.out.

Some Design Tips

Below are some general design guidelines, workaround and good advice for working with the PCI/6200 board, the XACT6000 software and the XC6200 device. While not a definitive list, it will hopefully help you to avoid some common pitfalls.

Problem: I/O pins do not operate.

Solution: A silicon revision number, must be written to address 0xC030 to enable the outputs. This should be done automatically by the software. If it is not, it is possible that your XACT6000 software or your CAL file is not in sync for the version of silicon you are attempting to use. You can manually write the proper silicon version (currently "1") to address 0xC030 to work around this. Upgrading either silicon or software is the preferred solution.

Problem: External signals appear inverted which shouldn't be.

Solution: This is caused by inversion compensation. In the XC6200 signals may be logically inverted. This permits faster and smaller circuits to be built than otherwise possible. Since these circuits are typically invisible to the outside world, this is not a problem. Occasionally, however, circuits sent to output pads are not properly inverted. The preferred solution is to add a buffer (BUF) element just before the OPAD. This will provide the circuit optimizer with the opportunity to provide either an inverted or non-inverted version of the signal.

Problem: Register accesses not working.

Solution: The Map Register may not be set correctly. The bits that correspond to the rows of the column to be accessed must be set to '0'. This register consists of eight bytes at location 0xC010. This can be set by software.

Problem: Synchronous circuits not working.

Solution: The clock may not be enabled. Bit six of address 0xC000 contains the clock enable. A value of '1' enables the clock. Note that this only effects the configured circuit. The clock to the control circuitry remains active so that data accesses over the external bus may continue.

Problem: Circuit does not function properly.

Solution: Check that the correct CAL file was loaded. Also, try generating 8 bit CAL files. They will load somewhat more slowly, but have tended to be more reliable.

Problem: Circuit does not route completely.

Solution: Often manually moving cells will solve routing problems. Using the RLOC attribute to manually "lock down" parts of the circuit will also help. It is possible, of course, that the design is simply too large for the device and will never fit.

Problem: Overlapping bounding boxes.

Solution: This is especially a problem when putting LOCed IO pins in a circuit symbol. Put these at the top level of the schematic to avoid this problem altogether. A second alternative is to Unplace the design, then do a Tools.Place from the XACT6000 main menu. Select the Allow Overlap box. This should stop the errors.

Problem: Unable to place parts of the circuit.

Solution: RLOCing circuits too close to other RLOCed circuits, or to LOCed IO pins can cause this sort of conflict. Try spacing things out more.