Communicating Process Architectures 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Reconnetics: A System for the Dynamic Implementation of Mobile Hardware Processes in FPGAs

Ralph MOSELEY
Computing Laboratory, University of Kent, Canterbury, KENT CT2 7NF
rm27@ukc.ac.uk

Abstract. The capacity to utilise FPGA designs in such a way that they are compositional, mobile, and 'interactive', offers many new possibilities. This holds true for both research and commercial applications. With the system described here, hardware becomes as easy to distribute as software and the mediating links between the two domains allow for manipulation of physical resources in real time. Designs are no longer monolithic 'images' downloaded at one time, but mobile entities that can be communicated over a distance and dynamically installed at run-time many times and at many places. Such twin domain designs can be as complex as a processor, or as small in scale as a few logic gates. The run-time system, *Reconnetics*, provides an environment of high-level control over such elements, which requires little knowledge of the underlying hardware technology.

1 Introduction

Field Programmable Gate Array (FPGA) technology is improving rapidly in terms of gate size, on-chip memory and support infrastructure [1]. Conventional cycles of design for such large devices can prove to be time-consuming and inflexible. It is now possible to interact with reconfigurable devices in such a way that dynamic configuration is fast and unobtrusive, even to an active unit.

The run-time system and associated infrastructure presented here allows the user to control FPGAs locally or remotely. This functionality is further enhanced by the utilisation of special design entities. These are developed using any conventional design tool and further processed by software which generates a program. In this new form the design has both a software and physical hardware component. The software part provides a mediating link to communicating systems and the means to implement the circuit, which is detailed within its code.

This paper details how the system is constructed, its basic model and its functionality. It also introduces the ideas behind its transferable components, here defined as *mobile hardware processes* (MHP).

Firstly, the background of the work is detailed and its relation to the current research in the area. The system itself is then presented in overview, together with a more detailed explanation of the main sub-systems. An exploration of the uses of the system is then given, together with examples. Finally, some further developments and direction are noted.

2 Background

From the early developments of programmable electronics, the goal has always been to capture some of the flexibility of logic and memory present in conventional software based systems. FPGA technology took this a step further by being able to implement complex circuits, logic and memory, in ways that were easily modifiable. A FPGA can be visualised as a 2D-matrix, which contains a great many controllable resources. Using these resources, complex logic circuits can be built up. Normally the user of a system will develop at a high-level, utilising a design environment that has either Hardware Description Language (HDL) or graphic design capability. The languages used can describe the circuits structurally or behaviourally. It has become possible, using languages based on C and occam, to describe the functionality of a circuit algorithmically, that is, there becomes little difference between a program written for a conventional computer and a FPGA. Where the conventional computer program may be compiled to native code, the FPGA-targeted program is synthesised to gates and finally placed on to the surface of the matrix. For a large design this can take many minutes. Normally, a design is a discrete block, not divisible or alterable, fed into the FPGA at some point, such as power-up.

Relatively recent developments allow configuration of these devices to take place while it is active, modifying only specific areas. For example, a circuit may take up half of a device's resources in a roughly rectangular area, on one side of the matrix. This may be active, in the sense that it is powered up and functioning, while the free half of the circuit is configured with a new design.

There is still a large gap between the design capabilities of FPGA hardware and the software which supports it. Designs are, in the main, usually downloaded as one large image. Even though the general view of the engineer is based on components or modules, this is relegated to a 'static' design-time.

Previous work in this area has realised both the difficulties and possible applications of such devices to be configured on-the-fly [2]. The work has primarily focused on developing a means of mapping designs at run-time[3], introducing ideas such as logic caching [4][5] and dynamic synthesis [6]. Compton and Hauck offer an overview of reconfigurable systems to date [7].

The work detailed in this paper builds upon efforts by Xilinx and others, utilising the dynamic reconfiguration capabilities in the Virtex series ICs [8]. The scope of this work is centrally defined by the concept of introducing flexibility to the FPGA design process in the form of mobility. For example, it should be possible to place a design as a reusable compositional unit anywhere on the surface of a FPGA, or transfer it over networks. Going a step further than this, the work attempts to bridge the gap between the physical circuit and any software which engages with it. This may be for information exchange, mobility of the entity itself, or for manipulating the actual form of the hardware object.

To allow access to low-level resources and fast manipulation of the bitstream, Xilinx JBits Java classes were used, as well as other techniques, developed as the system evolved. It is beyond the scope of this paper to explain the usage of JBits or discuss the hardware mechanisms of low-level dynamic reconfiguration [9]. Such information is available elsewhere – particularly through the Xilinx website [10] and its accompanying library of applications papers.

While JBits provided a means of low-level access to resources and to dynamic reconfiguration options, methods had to be developed for manipulating processes as a whole and the individual components contained within them. With the JBits classes providing access to low-level resources and routing, there is still a question of handling various collections of resources which form a process; a very complex, physical object.

Methods were required for copying CLBs[1] and associated wiring within and outside of their original bitstreams. Mechanisms were also added for manipulating connection trees, BRAMs[2] [11], physical pads and the capability to read in hex data files.

Many high-level features of JBits were not required, but can be integrated if so desired; for example, 'cores' can be turned into MHPs for manipulation in real time at a high level.

3 System Overview

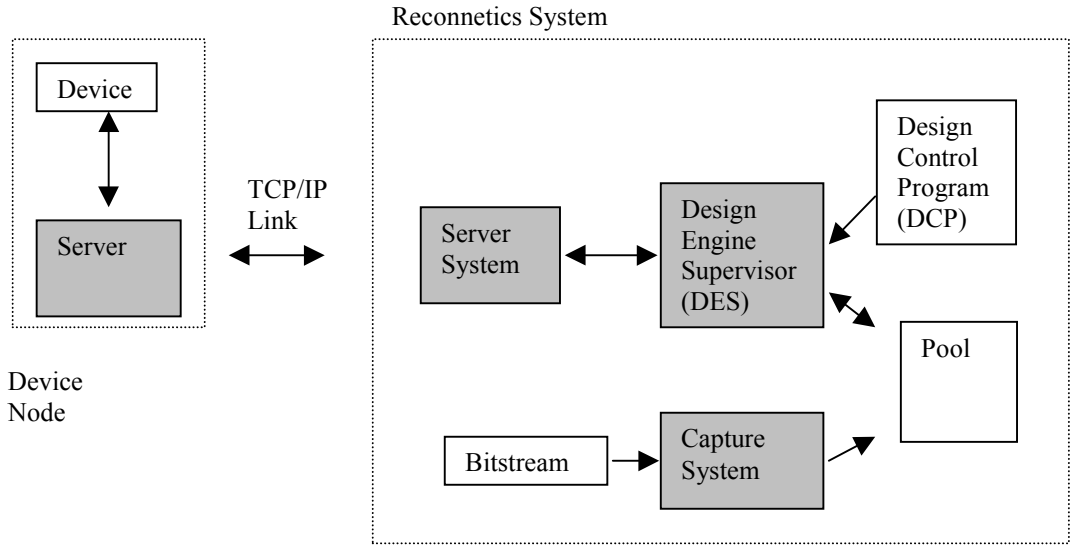

Figure 1. *Reconnetics* overview

The *Reconnetics* system allows dynamic interaction with a reconfigurable device. The user supplies designs, as a synthesised 'bitstream', written with any suitable HDL or graphic tool. The design itself can be as large and as complex as required, although it would make more sense to build from smaller blocks, compositionally. These are captured and placed in an archive for later utilisation by the design engine. The engine is directed by a high-level user program, to load, place and interact with these processes, as well as manage on-board resources.

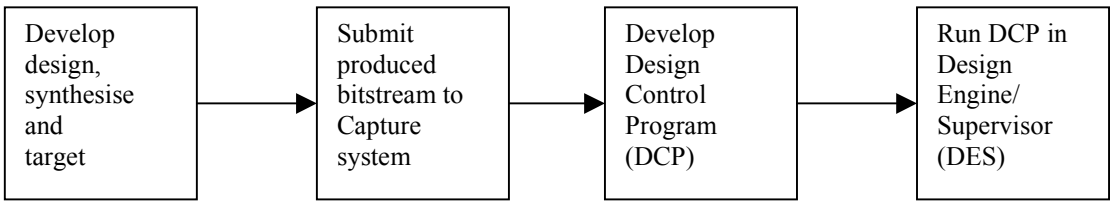

Figure 2. Development cycle

The language used to control the engine is deliberately simple and focused at a high level; all low-level wiring and resource handling is taken care of by the system itself. A user needs only to decide which processes need to be connected together and what kind of interaction is required. It may be that only placement is desired, or diagnostic work. This is possible but a wider scope of control and communication is also offered. For example, processes can be interacted with by *Reconnetics* in such a way that a user can supply input

[1] Configurable Logic Block, a programmable unit containing logic, flip-flops and multiplexors.
[2] Block Random Access Memory, a discrete memory unit, separate to the main programmable matrix area.

directly to the hardware without external I/O connections or other methods. This is true also for collecting output, making the system particularly suitable for development work.

The capture system analyses a bitstream, which can be the product of any design method or language (HDL, schematic etc.). Its task is to produce a Java class which can rebuild the circuit at any given point on the surface of an FPGA. This includes status of clocked logic, routing and any other necessary resource usage. The class is primarily produced for use by the run-time system but is sufficiently self-contained to be incorporated into any user written Java program too. After capture, the class is placed in an archive known as the `pool`, which is simply a directory. Together, when instantiated in their appropriate environments, they are known as a MHP.

The generated Java program's principal function is to act as a vehicle for the circuit and to reproduce it at any given point on an FPGA matrix. Another function of the Java program is to act as a mediator between the Design Engine/Supervisor (DES) and the physical circuit. A user can communicate with the process in this way, collecting information or directly manipulating the circuit's form or connections.

DES manages the device itself, along with organisation of the processes. It can run user-written scripts which direct activities, these are known as design control programs (DCP). All low-level resource management is left to DES, making the actual command language very simple, an example of which is given later. The language also includes various diagnostics, if required.

The server system provides the ability to communicate over TCP/IP to device nodes on the current machine which is running *Reconnetics*, or a device at a remote location – identified by an IP address. It is the job of the server system to synchronise between *Reconnetics*' internal representation of a device and the device itself.

4 Capture

This section considers the capture system in detail; if you do not require a low-level perspective, then re-join at section 5[3].

As stated above, the main task of the capture sub-system is to produce Java classes which can reproduce a circuit at some other location. It must capture all essential design data, such as the status of multiplexors, look-up tables (LUTs) and routing. A large number of resource types are available in devices – some of which are not mapped into JBits at this time.

During analysis of a circuit this discrepancy led to incomplete route tracings. This was resolved by building in a certain amount of design knowledge. A first scan by the system makes a list of incomplete nets. Solutions to an incomplete net can be found by recognising specific attributes of the end resource and by being aware of disrupted routes elsewhere, which conform to certain criteria. In this way, the system can apply a patch to fix an incomplete tracing.

This method has not yet been known to fail in its attempt to re-establish a problem design. It is possible that this technique could be applied to correct corrupted bitstreams, or instantiated designs, up to a certain point.

A further optional stage is available, called *pre-capture*, which adds connection points to a process, allowing several modes of communication with other processes, or external I/O. Further details are provided below.

[3] A suitable starter guide is available from Xilinx [12].

4.1 Scanning

Here resource usage is determined and mapped. Three different areas are checked: the CLBs, BRAMs and external I/O. The supplied bitstream is analysed and each resource found to be in use within the design is marked for further examination.

When the entire matrix bitstream has been scanned, an object is built which contains the details of the used resources and their relationships to each other, such as positioning. This object is a sub-matrix of the larger containing matrix within the bitstream. The sub-matrix can be manipulated in terms of its geometry; it can use its default shape, or have new positioning imposed on it, while still maintaining correct connections.

4.2 Code Production

The scanning procedure gives a basic map of the CLB and BRAM usage. This map is then used to analyse, in greater depth, specific resources and their settings. Finally, a program is generated which describes the implementation. There are many types of resource contained within each of the basic areas, whose state has to be captured and translated into program statements, such as multiplexors, switches, LUT arrays and routing, some of which may be to external I/O points.

The Java program produced is made up of several key methods. These allow a control program to build, destroy, communicate and route a process's physical circuit. From the point of view of the average user of the system, this method information is not required. DES and the command language take care of such detail.

The basic 'shape' of the design is kept intact; that is, the map which shows resource usage also holds the basic relational details. At the time of code production such detail is captured by producing offset row and column data between each CLB and a point of origin. The entire design can then be relocated anywhere on the surface of an FPGA, while maintaining its relational and connection constraints. This ensures that issues such as timing and gate propagation are met, as they were when initially compiled. It is possible to override positioning data for CLBs if necessary.

Used BRAMs are analysed and separate methods built for their construction. As before, in the case of the other resources, appropriate statements are generated to implement correct settings. Data content in the BRAM is translated into an array format, held in the method, which is downloaded at build-time. Finally, statements are written for output wiring. This connects the BRAM to its owning process. It is important to note here that it is possible to position the BRAM at any BRAM location and it will still be wired to correctly. This makes it easy for a run-time system to place and route such resources in whatever way is most suitable at the time.

In all the above produced code, no routing data is generated that leads to external IOB positions, however, details are stored and checked, for uniqueness and integrity. Only end point IOB information is used as default connection data, stored in the generated program.

4.3 Design Considerations and Constraints

A process is extracted from a configuration bitstream by analysing resource usage and I/O requirements. Nets which lead to I/O points are viewed as the process-to-be's main interface points to the outside world. If such points are named in an accompanying User Constraints File (UCF), in the usual format, then the name persists through from the design/programming stage to run-time.

There are few alterations necessary when supplying a design to the capture system; BRAMs will be automatically captured and available for run-time loading. Bi-directional wires are simply split into two uni-directional wires.

While processes can easily be used on devices they were originally compiled for, there is also a degree of device independence. Re-use capability is determined by three main parameters: the actual device type, clock speed and clock routing. Using this system the device type (currently within the Xilinx Virtex series) can be changed quite easily. A design originally intended for a small device (such as a XCV50), can easily be transferred to a much larger device and vice versa, resources allowing. The clock speed can be maintained at its old frequency, or changed within the DCP. The clock routing for a specific bitstream can also be altered, with the supplied tools.

A process owns its output routing but not any wires connected to external inputs. When the process is unloaded, its internal logic and routing is released, as well as any output links that it owns. Routing to a process's input port persists after its destruction and must be unrouted (if required) by the owning unit.

4.4 Input / Output Connection Points

It is possible to pre-process a bitstream, so that any I/O is passed through a special connection point. This pre-processing stage is particularly useful for designs which already exist (for example, as a legacy design) and there is a need to turn it into a fully self-contained process. It is also possible (though not as simple), to manually add a connection point, using the system itself at run-time. In both cases a special system object known as `procio` is used. This unit added in is actually very small, being only a quarter of a CLB for each wire.

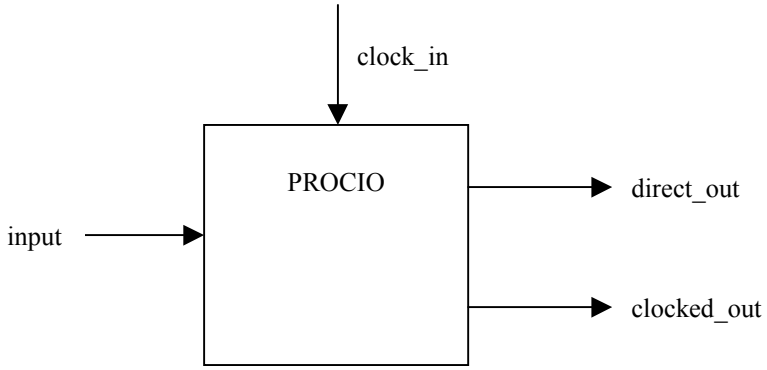

Figure 3. PROCIO unit showing main connections

There are three functions for the I/O points: to decouple, to simplify routing to other processes, and finally to act as a controllable gateway to the world. The idea here is to allow a degree of control and buffering at I/O points. When this is done it is possible to control the connection point with DCP commands, such as `inputON` and `inputOFF`. The functionality of the `procio` object allows I/O signals to be disconnected, clocked, or fed straight through.

4.5 Recapture

Recapture is the name given to the ability to capture a process at run-time. Many of the techniques used in the pre-run-time capture stage are reused within recapture, although obviously initial design details are not available (such as the data contained in the UCF,

giving port labels). Information for capture at run-time is gathered from two sources; the instantiated circuit and the encapsulating Java program, via communication with its `port()` method. These two units that make up an MHP provide current resource details, structure, and port I/O information. Together, the information gathered allows a new entity to be produced. This contains any changes made to it since its initial instantiation, captured in real time, while maintaining persistence of initial design semantics and high-level user details, such as port names.

This generated program is then dynamically compiled and its separate files organised into the appropriate directories: `.java` to the `design` directory and `.class` to the `pool` directory. This all occurs at run-time, automatically, without the user intervening in any way. The new MHP can then be used in the same user program, if so desired.

5 Design Engine / Supervisor

When DES runs a user-supplied program, it initialises a device model, which is a representation of the external physical device. The processes, initially in the form of the Java classes, are loaded into memory via a Java class loader from the archive `pool`. Processes can then be directed to instantiate onto the FPGA, route to I/O and interact.

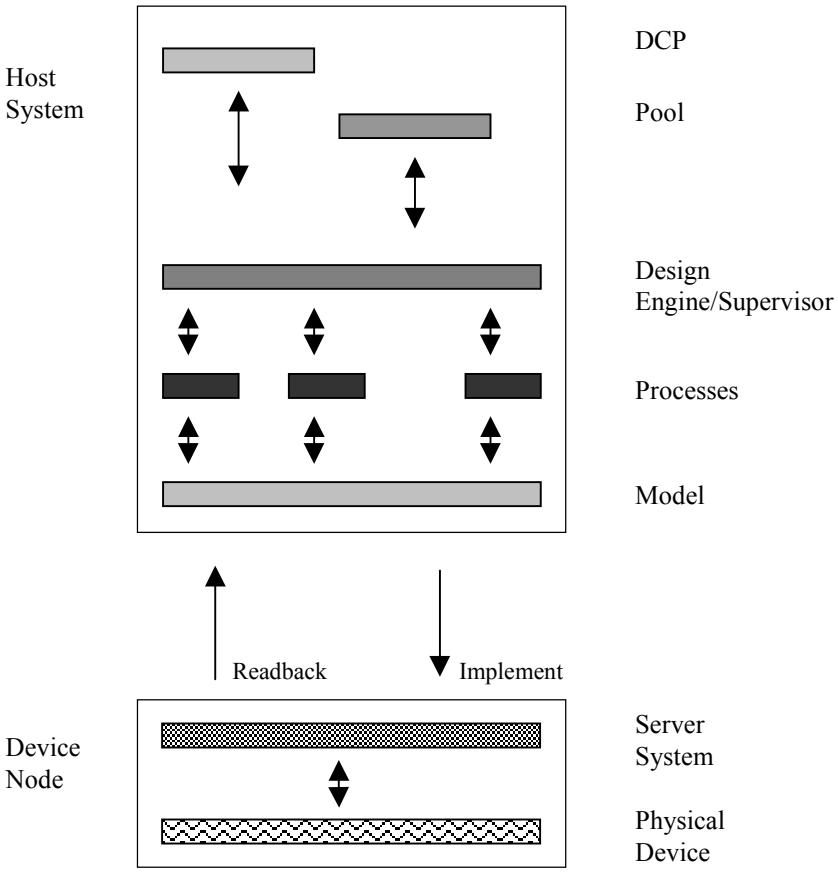

Figure 4. Run-time visualisation

This is visualised in figure 4. The readback and implementation loop, which in reality takes place over a local or network TCP/IP connection, occurs only when necessary, such as an event instigated on either side. Any exchange of data is limited to packets rather than whole bitstreams, to enable fast update or information retrieval.

One of the main tasks of DES is to run the DCP script written by the user. There are several areas that this control language deals with:

- *Initialising device and internal representation model.* This includes commands which configure the system to a connected device (which may be local or remote), so the model, for example, conforms to the correct type.
- *Location or port definitions.* Locations can be assigned names which can be used for referencing within the program. Some locations, such as a process's ports, have already defined names associated with them.
- *Device control.* This includes such commands as resetting the device, clock stepping, and forcing a 'readback'.
- *Process control.* A number of commands are available for the placing and routing of processes. They may be automatically placed, or organised within a user-specified pattern. Various ports (or locations) can be defined on a process, then observed or linked to the I/O of other processes or external device pads. Processes can be unloaded and the resources associated with them can be freed for further use. A process's I/O can be controlled and switched on and off for isolation. A process may also be 're-captured', for storage and transport to another location.
- *Interconnect.* This set of commands allows defined locations to be routed or unrouted. Locations can be device pads, IOBs or process I/O.
- *Interaction.* Using defined locations it is possible to collect data from active processes, which can then be acted on.
- *Program sequence and control.* A range of repetition and control structures exist.
- *Diagnostics.* Although most low-level activity is shielded from the user, it is still possible to use diagnostic functions, if required, to check circuits. A good example of this is the `tree` command, which enables a user to view a hierarchical tracing of connections from a defined point.

6 Communication

Interaction is possible between the processes themselves and between the processes and the system. Inter-process communication is, of course, wire-based. Such wire connections can be controlled within the user-written DCP.

The system itself can observe and interact with the instantiated design, by the user defining locations, usually registers, in the circuit. These usually are clocked registers, which may be, for example, buffered I/O points. As well as collecting data from such points, it is also possible to set values at inputs.

Processes can be placed (using `load()`), interacted with and destroyed (using `unload()`). If a design as a whole is to function properly during its manipulation (in particular, its destruction), then certain protocols need to be adopted.

A problem could occur, for example, when some communication may be taking place between processes and the system wants to extract one of them. This situation necessitates the use of communication signals to determine the state of a process at a given time. A protocol in this way could be used to interact with the run-time system to make its safe extraction possible. This is also true during the re-capture of a process when it must first enter a 'safe' state that must be signalled, via defined points.

The least interfering and non-invasive way to achieve safe extraction is to isolate the process. This involves switching off any inputs to the process and possibly generating a

stand-by signal to the connected units. Input isolation is achieved through the `inputOFF()` command. It would be a simple matter to provide the correct hand-shaking protocol and routing for a stand-by signal in processes, if this is required.

7 Using the System

There are several ways that the system can be used. At the most simplistic level it can be used to build designs, controlling placing and routing to a specific pattern. This may include no dynamic interaction. The bitstream produced can then be used, as is normally the case, to configure devices. A further simple use of the system would be as a distributor of updates to hardware.

The main use of the system is for dynamic design, which envisages a design cycle extending into run-time. Instantiated processes can be interacted with, verified and data downloaded from. The device can be amended, as required, on specific events taking place. A device may also *request* certain processes to be instantiated; for example, on a fault condition arising.

Program	Description	Tests
Pulse Counter	Two processes (pulse generator and pulse counter) plus link	Basic functioning of system
Microcontroller 1	Larger more complex process (third party design). Program runs on hardware and interacts with DES	Handling and placement of larger processes. Interaction
Microcontroller 2	As above but utilising multiple processes placed dynamically at staggered points in time	Installation of multiple complex processes, dynamically
Flash	Process to link to various external pads	External IOB linking
Datalink	Setup various processes experiment with communication protocols	Inter-process communication
Generator	One process communicates with DES to request loading of new processes	FPGA initiated control
Reconfigurable Computer	On-going computer design, based on reconfigurable components	Large application. Tests data sharing, swapping various units, sharing of data with host system

Table 1. Test programs

Table 1 shows a suite of test programs devised to experiment with various possibilities. More information on these is available at the *Reconnetics* website [13].

8 Performance

Table 2 shows averaged performance figures in milliseconds of various activities with the system. They are intended to give a rough idea of system capabilities rather than precise information. The system, as currently implemented, is pure Java with JBits API extensions. There are identifiable areas for optimisation and various techniques could be used to achieve speed-up. If speed is the critical factor, rather than adherence to the platform independence of Java, then certain functionality can be sped up using the Java Native Interface (JNI) to a pre-compiled language. All data was gathered on 1GHz computers and a Celoxica RC1000-PP board incorporating an XCV1000 Xilinx FPGA.

RC1000-PP Board (XCV1000 device)				
	Local		Over LAN 100Mbps	
	Process 1	Process 2	Process 1	Process 2
CLB size	2	35	2	35
Auto place	2000	8300	2600	8930
User place	1890	8000	2200	8812
Destroy	140	420	250	600
Recapture	4000	12000	5580	13200
Recapture and deploy	7500	15000	8570	16500
Single wire link	1000	1200	1580	1780
Variable Read	800	1200	1500	1320

Table 2. Performance

9 Example 1

Example 1, below, shows a DCP which loads a microcontroller (labelled `kprocessor` in the archive), with a defined output port named `REGOUT` and a width of 8. This microcontroller is available from Xilinx in Electronic Design Interchange Format (EDIF) [14]. The hardware design could have equally been developed by the user with a structural, behavioural or algorithmic language, inside an electronic design environment.

We will now step through the program. Initialisation concerns selecting the correct internal representation model, initial bitstream (or canvas) and choice of connection to the physical device over a network or locally. This is done with the `device()` statement. In

this case the device is a XCV300, with an appropriate blank bitstream and an IP address of 192.168.0.168. Once connected, the device is reset and the clock frequency selected.

Two processes are then defined using the `process()` statement. This takes the form of an identifying name and the name of the archived process class in the pool. The first process is then loaded up using an automatic placing method. The Java program builds the circuit and stays resident to act as mediator between the run-time system and its physical side. A variable is then defined for use in the program; this is associated with an output register on the process and its bit size defined. Names set at design-time are still available and persist through to run-time programming. In this case the `var` statement consists of the variable name `reg0`, on `processor1`, with a port named `REGOUT` and the width being 8 bits.

The next step involves loading a program into the BRAM, which is connected to the processor using the `readHex()` command. In the example, a simple counting program was fed to the processor. This increments the register, which is then printed for confirmation. Note here how it is the process's Java program which is collecting the value from the physical circuit and passing it back to the run-time system.

When the loop is terminated, the second process, defined earlier, is then loaded. A `wait()` is used to keep the hardware processes running until the `reg0` variable reaches the value 200. Finally, both processes are unloaded. Unloading first releases used resources on the FPGA matrix and then used resources within the host, such as the running process object.

```
// initialise device
device(xcv300, null300GCLK1, 192.168.0.168);
reset();
frequency(10); // set a clock speed of 10MHz

//get the processes
process(processor1, kprocessor);
process(processor2, kprocessor);

//load a process
load(processor1);

//define test location
var(reg0, processor1, REGOUT, 8);

// load program into BRAM
readHex(0,0,16,"test.hex");

do {
     message("reg0:");
     print(reg0);
} while (reg0 < 10);

// load the next process
load(processor2);

wait(reg0,200);

unload(processor1);
unload(processor2);
```

Example 1. Simple DCP

10. Example 2

This example shows the two different kinds of communication, between processes and between process and the run-time system.

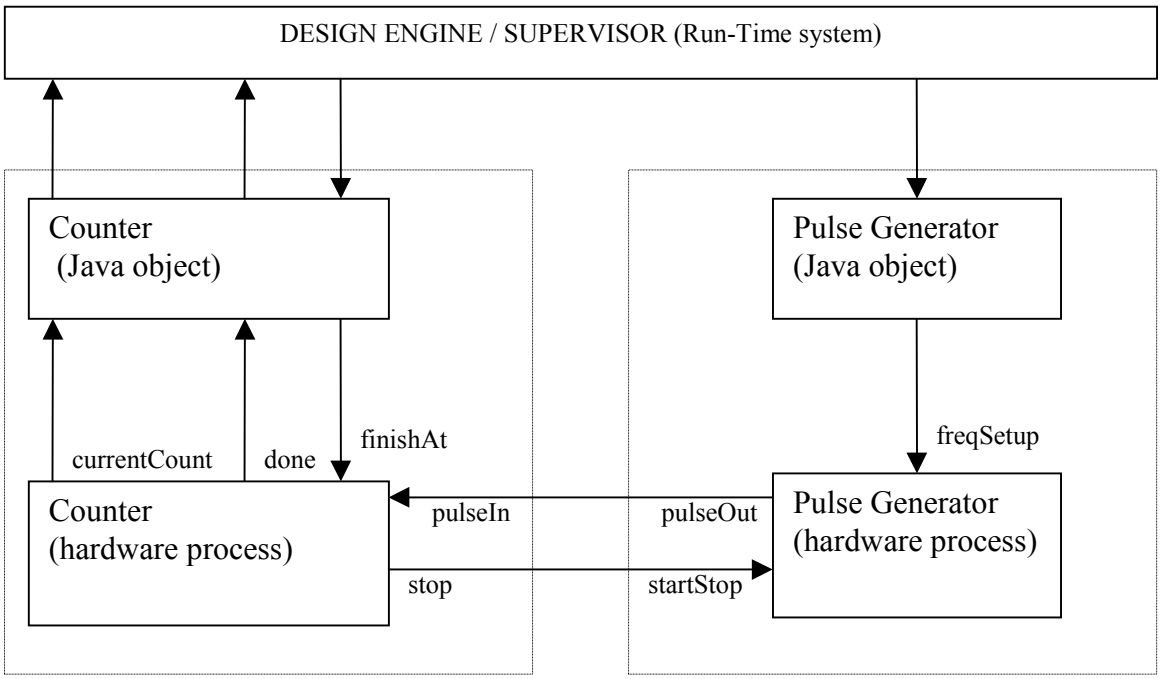

Figure 4. Communicating processes

Two processes are used; a 16-bit counter and a pulse generator. Figure 4 clearly shows how the Java program acts as a mediator between the run-time system and its hardware circuit, once instantiated. The `counter` counts pulses on its `pulseIn` port. The current value appears on its `currentCount` port. When the value (programmed via `finishAt`) has been reached, a stop signal is generated. The pulse generator process can be programmed with a frequency rate and stopped through its `startStop` port.

Using a DCP, the system is initialised and processes loaded with manual place this time:

```
device(xcv1000, null1000GCLK3, localhost);
reset();
frequency(3.5); // set a clock speed of 3.5 MHz

process(counter, counter16);
process(generator, pulsegen);

loadAt(counter, 10,10);
loadAt(generator, 20,10);

//define test variables
var(currentCount, counter, COUNTOUT, 16);
var(finishAt, counter, COUNTIN, 16);
var(stop, counter, STOPOUT, 1);
var(pulseIn, counter, PULSEIN, 1);
var(done, counter, FINISHED, 1);

var(freqSetup, generator, FSETUP 8);
var(pulseOut, generator, PULSEOUT, 1);
var(startStop, generator, ONOFF, 1);
```

Links are established:

```
link(stop, startStop);
link(pulseOut, pulseIn);
```

The frequency and counter are programmed using the `inputVal()` command:

```
inputVal(freqSetup, 10);
inputVal(finishAt, 1000);
```

The process counts the pulses, taking `done` high when it has received the amount of specified pulses. During the counting time, the run-time system simply waits:

```
wait(done, 1);
```

To show the transport of a process from one location to another we can capture it:

```
recapture(counter, new_counter16);
```

then release resources:

```
unlink(stop);
unlink(pulseOut);
unload(counter);
```

Finally, re-deploying somewhere else:

```
process(counter2, new_counter16);

load(counter2); // using auto-place
```

Variables can be then defined again and links re-established. The counter can be told the required amount of pulses and another `wait()` statement executed. When finished, the program can end with resources being released, through `unlink()` and `unload()`.

11. Further Development

Besides on-going improvements to the system, there are two main enhancements that research has brought to light. They are worth mentioning, although it is beyond the scope of this paper to detail them fully here.

The first concerns the ability of such run-time systems to be embedded into devices themselves, allowing self-reconfiguration.

The second idea follows from this, in allowing the processes themselves some degree of autonomy, providing access to configuration features. These have been named *intelligent Mobile Hardware Processes* (iMHP).

Such work as described here implies a merging of the software and hardware domains, creating entities that are composites of both worlds and realising the benefits of each.

12. Acknowledgements

Thanks to Xilinx for developing the Virtex hardware and most all to the JBits team who have been very helpful in my endeavours. Thanks also to my supervisor, David Wood, and the University of Kent Computing Laboratory for supporting this project.

References

[1] Xilinx. *Virtex 2.5V Field Programmable Gate Arrays Data Sheet*, Xilinx Inc. 2001.

[2] N. McKay and S. Singh, *Dynamic Specialisation of XC6200 FPGAs by Partial Evaluation*, Field Programmable Logic and Applications. Tallinn, Estonia. Springer-Verlag, 1998.

[3] M. Wirthlin and B. Hutchings. *Sequencing Run-Time Reconfigured Hardware with Software*, Dept. of Electrical and Computer Engineering, Brigham Young University, Utah, 1996.

[4] P. Lysaght and J. Dunlop. *Dynamic Reconfiguration of FPGAs*, Dept. of Electronic and Electrical Engineering, University of Strathclyde, 1995.

[5] K. Compton and S. Hauck. *Configuration Caching Techniques for FPGA*, University of Washington, Seattle, 2000.

[6] N. McKay and S. Singh. *Debugging Techniques for Dynamically Reconfigurable Hardware*, Proceedings: IEEE Symposium on FPGAs for Custom Computing Machines: April 21-23,1999, Napa Valley, California, edited by K.L. Pocek and J.M. Arnold (IEEE Computing Society,1999).

[7] K. Compton and S. Hauck. *Reconfigurable Computing: A Survey of Systems and Software*, Dept. Of Electrical Engineering, University of Washington, Seattle, 2000.

[8] Xilinx. *Technical Overview*, Xilinx Application Notes XAPP097, Xilinx Inc. 2000.

[9] Xilinx. *Dynamic Reconfiguration*, Xilinx Application Notes, XAPP093, Xilinx Inc. 1998.

[10] Xilinx. Xilinx Home Page. Available at:
`http://www.xilinx.com`

[11] Xilinx. *Using the Virtex Block SelectRAM+ Features*, Xilinx Application Notes XAPP130, Xilinx Inc. 2000.

[12] Xilinx. *Xilinx FPGAs: A Technical Overview for the First-Time User*, Xilinx Application Notes, XAPP097, Xilinx Inc. 1998.

[13] R. Moseley. ***Reconnetics*** Home Page, 2000. Available at:
`http://www.ralph-moseley.co.uk/reconnetics`

[14] Xilinx. *8-Bit Microcontroller for Virtex Devices*, Xilinx Application Notes XAPP213, Xilinx Inc. 2000.