Automatic Identification of Swappable Logic Units in XC6200 Circuitry

Gordon Brebner

Department of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, Scotland

Abstract. A Swappable Logic Unit (SLU) is an FPGA-based logic circuit that can be managed by a virtual hardware operating system. Since the SLU concept is still in its infancy, it is unreasonable to expect circuit design systems to be enhanced to include SLU awareness at this stage. Therefore, this paper describes a tool that can be used as a back end to any XC6200 circuit design system. It analyses XC6200 circuitry, to detect SLUs and then collect information about their external interfaces. This information is needed by an operating system so that it can manage the SLUs when they are in use.

1 Introduction

In [1], the author described the components of a simple operating system able to support two virtual hardware models using physical FPGA resources, specifically the Xilinx XC6200 family [5,7]. The central feature of the virtual hardware models was the **Swappable Logic Unit** (SLU) — an FPGA-based logic circuit capable of performing a function in terms of specified input(s), and then supplying results on specified output(s). An SLU is the virtual hardware analogue of a page or segment in a virtual memory system. In [3], the nature of SLUs was explored in more detail: their hardware environment; the software environment from which they are harnessed; their basic operational properties; and their input/output interfaces. The central theme was the need to make virtual hardware available efficiently in a normal general-purpose system environment with standard compilers, run time systems, operating systems and hardware architectures.

In this paper, the production of SLU circuitry itself is considered. The central theme is akin to that of [3] but, in this case, to investigate how SLU circuitry can be produced using normal general-purpose hardware design tools. The aim in both cases is to avoid the need to construct new, and different, tool sets, and so make it easier to entice designers from either hardware or software backgrounds into a world of hardware/software co-design. In this case, a further consideration is that it is not possible or desirable to import the notions associated with SLUs into conventional tools until the concept is stable and accepted. Therefore, in this paper, one new tool is introduced — a technology-dependent back-end useable after any particular circuit design system.

The tool analyses a piece of circuitry, and identifies one or more SLUs within it. Information about each SLU found is then stored for use by a virtual hardware operating system. The information required is explained in detail in [1] but, in summary, must include:

– a rectangular bounding box; and
– details of input/output interfaces

and, preferably, to allow efficient swapping:

– replicated areas within the SLU (to allow parallel configuration); and
– similarities between the SLU and others (to allow partial reconfiguration).

A further piece of information required by the operating system is an indication of the position dependence of the SLU — ranging from placeable anywhere on the configurable array through to requiring a fixed position on the array.

The detailed operation of such a tool, for the specific case of the XC6200 FPGA family, is discussed in this paper. A main issue thoughout is: how feasible is implementing a fully-automatic back-end approach? Unfortunately, for the XC6200 at least, full automation is not possible in some cases, and so a little human assistance is required. The essential difficulty is one of *reverse engineering*: trying to find structure that existed at an earlier stage of design but has been flattened out in the circuitry being analysed. Given that the need for human guidance is not particularly desirable, a conclusion is that the incorporation of SLU awareness at higher (technology-independent) levels of hardware design will be very useful, and indeed necessary, in the future.

Section 2 of the paper discusses the general problems associated with finding Swappable Logic Units in XC6200 circuitry described in bit stream form. The two sections following are concerned with the detail of finding input and output interfaces for SLUs. Section 3 deals with the more straightforward case of finding perimeter input/output wiring; Section 4 deals with the trickier case of finding mapped input/output registers within the circuitry. Finally, Section 5 contains some conclusions drawn from the work.

2 Finding SLUs in XC6200 circuitry

The tool makes use of the internal graph-based representation of XC6200 circuitry described in [2]. Circuitry consists of **ports**, represented by graph vertices, and wires between ports, represented by graph edges. There are four types of port: compute, input/output, multiplexor and source. The overall representation style is suitable for technologies other than the XC6200 as well, and so the overall algorithms described here are applicable in more generality. However, some of the detail parts of algorithms are technology-specific, and so would require reworking for other technologies.

Given a circuit description, the first step for the SLU finding tool is to identify unconnected sub-graphs, since each sub-graph corresponds to a potential SLU. From this stage on, each SLU candidate is treated separately so, without loss of generality, only a single SLU is considered in the description that follows. In practice, the expectation would be that a single SLU candidate is presented to the tool; however, finding multiple SLUs is straightforward using a standard algorithm for finding unconnected components of a graph.

Obtaining a minimum bounding box for the SLU is not too hard, given the coordinate information stored in the internal circuit representation. A little care is necessary however, since the exact perimeter is not necessarily marked by the coordinates of the most extreme configured cells. For example, flyover wires, driven by configured cells, may protrude beyond the apparent perimeter. Although contributing nothing to function, inclusion of such wires may be unavoidable because of the nature of the XC6200 technology, and so must be considered as being within the bounding box.

Finding input/output interface information is less straightforward. The work required depends on SLU type. In [3], three different types of SLUs are identified:

- accelerator SLUs, which have register-style interfaces;
- PE (processing element) SLUs, which have perimeter wiring interfaces; and
- system SLUs, which have interfaces that can use any available mechanisms on the host FPGA.

A fully-automatic tool must decide what type of SLU has been found in a piece of circuitry, in order to then obtain appropriate interface information.

In fact, the nature of a putative SLU might be ambiguous. A simple heuristic to help make the right decision is:

1. if there are inputs from (non-clock) global signals, or if there are fixed-position inputs or outputs from or to FPGA pins, it is a system SLU
2. if there is wiring to the perimeter of the circuit, it is a PE SLU
3. if there is at least one input and one output register in the circuit, it is an accelerator SLU
4. otherwise, it is not an SLU candidate

Of course, there is scope for making a wrong decision, and this makes a case for a small amount of checking interaction with the user. However, this can be very high level, for example, just asking the question: 'is this an accelerator, part of a parallel harness or something else?' Alternatively, the tool user can supply a hint initially, in which case the heuristic above is used to check that the circuitry is consistent with the hint.

Once the type of SLU is ascertained, appropriate interface information can be gathered. In the case of system SLUs, no further work is required, since the operating system does not need to be aware of the SLU's specific input and output arrangements — software directly accesses the SLU through the external interface of the FPGA. For PE SLUs, only perimeter interface information is sought; the existence of registers within the SLU is of no relevance. For accelerator SLUs, register interface information is sought; however, perimeter interface information is also sought. In principle, the existence of perimeter interfaces on a putative accelerator SLU is an error. However, depending on location, the tool may be able to convert such interfaces into register interfaces by extension of the circuitry under user control. Sections 3 and 4 discuss the search process for perimeter interfaces, and for register interfaces, in detail.

In general, care is necessary in order to avoid finding spurious interfaces. For example, there may be redundant perimeter wiring, perhaps as a harmless side-effect of an automatic design system. Alternatively, there may be registers present for purely internal use in sequential logic, and these might be wrongly identified as interface components. One solution to this is further interaction with the user, to check whether potential interfaces are intended to be externally visible. A different solution, and one which is to be investigated in future work, is to use typing information from the software side of the system being designed. For each SLU, there will be a description of a corresponding function, giving its parameter and result types. This information can be used to guide the search for SLU interfaces. When external guidance is used, either through interaction with the user, or from software environment information, there is scope for the tool to be proactive in extending the circuitry it is analysing.

Apart from capturing the basic SLU information, analysis of each potential SLU can find replicated patterns within, in order to assist parallel reconfiguration. The search must be done with respect to any actual parallel mechanisms available for the FPGA technology being used. In the case of the XC6200, the mechanism is 'wildcarding', which allows FPGA cell addresses to be wild-carded so that multiple cells can be written with the same configuration data simultaneously. In [6], Luk et al describe an algorithm for making optimal use of the wildcarding mechanism, which has been incorporated into their ConfigDiff tool. This involves enumerating all possible wildcarded addresses, to find which matches a required addressing pattern most closely.

Here, a slightly more abstract approach, consistent with keeping SLU information technology-independent, is adopted. The search is guided by the organisation of the XC6200 address space, but seeks patterns in the cell structure rather than patterns in the required FPGA addressing structure. It looks for increasingly large patterns replicated decreasingly often. That is, the ideal is initially to find that one cell is replicated everywhere; if unlucky, the search finds that one group of cells occurs once, i.e., there is no replication. Note, however, that the methods of Luk et al are a useful contribution to this work, but at the stage when the operating system must instantiate SLUs on the XC6200 array.

Potentially, some assistance in finding replicated patterns has been ignored. In a circuit that has significant structural replication, this fact is likely to be reflected in the original technology-independent design, for example, in a VHDL description. This is a drawback of the back-end approach, in contrast to infiltrating the SLU concept into the higher-level tools. It may also be the case that placement and routing manages to disguise replication so that the back-end tool fails to spot it in the circuit configuration data it is analysing. Note, however, that spotting replication is not *essential* to make virtual hardware work, so here there is an example of a trade-off: improved performance versus generality of design tools; such a trade-off is not uncommon, of course.

A further way of improving performance is by finding similarities between different SLUs. The purpose of this is to assist the use of partial reconfiguration. Clearly, the analysis of a single SLU, as described so far, is not capable of doing this, because it has no family context. However, the tool is able to find one class of information that can lead to partial reconfiguration: information about circuit state, when sequential logic is included. This can be exploited if the same circuitry is being used, but with different internal states.

The simplest way for identifing family similarities, in general, is for the tool to be supplied with at least two examples of family members; on the basis of these examples, it can identify internal bounding boxes of rectangular blocks that differ. This requires some connivance with the designer. A more general alternative is, for each SLU under investigation, to compare it automatically with an existing collection of SLUs, in order to place it in a family context. This seems more promising, being an automatic approach, as long as the collection is not too large. In [6], Luk et al describe their algorithm for calculating incremental configurations, which is at the heart of the ConfigDiff tool mentioned previously. This compares two configurations, and finds the incremental reconfiguration needed to move from one to another. The approach used is directly applicable here for finding SLU family resemblances.

As for finding replicated patterns, so finding similarities from the low-level circuit descriptions is likely to be losing potential benefits from higher level descriptions. These may make clear structural commonality between SLUs. However, as for replication, spotting similarities is not essential to making virtual hardware work, but involves the trade-off between performance and generality.

Finally, the tool can be used to find one further piece of information that is necessary for the operating system. This is to determine the degree of position dependence of the SLU. Essentially, this involves analysing the circuitry to find use of asymmetric features of the XC6200 technology. For example, use of length-4 flyovers or magic wires means that the SLU can only be placed at positions on a four-cell pitch, or use of input/output blocks means that the SLU can only be placed at the FPGA perimeter. Techniques for increasing freedom of placement have been suggested by Burns et al [4]. These include the use of standard two-dimensional geometrical transformations, where irregularities and asymmetries in the XC6200 architecture do not invalidate the circuitry. This offers the possibility of extra proactive action by the tool described here.

3 Finding perimeter input/output interfaces

Finding perimeter input/output interfaces is more straightforward, and has distinctly less ambiguous results, than finding register input/output interfaces. A main aspect of the work is already done when finding the minimal bounding box for the SLU. The search for interfaces then involves an exploration of the perimeter of the box. A complication arises when the circuitry is not a pure rectangle in terms of its external interfaces — this may arise from the arrangement of the configured cells or, as mentioned in Section 2, from wiring that overhangs the extreme configured cells. Since SLUs are inherently rectangular (to aid run time placement by the operating system), an optional proactive facility for the tool is in extending perimeter wires so that all interfaces are on the perimeter of the bounding box.

Given a perimeter that is uniformly rectangular, searching for interfaces involves examining the ports located on the perimeter, to check for wires entering from, or departing to, the area beyond the perimeter. By definition, these are the potential perimeter input/output interfaces. As discussed in Section 2, there is the possibility of the tool finding 'bogus' interfaces. However, the fact is that the tool fails safe: all possible interfaces are identified and, if nothing else, finding unexpected potential interfaces gives the user a warning that the circuitry is not completely self-contained.

For SLU purposes, the presence of input/output pads on the perimeter should not be an issue here, since pads cannot present on non-system SLUs. However, there are two related issues worth mentioning. First, some design tools might only produce bit stream data for circuitry that is complete with input/output pads. Thus, an optional role for this tool would be in stripping away perimeter pads, to give circuitry suitable for inclusion in a parallel harness. Second, a possible option, not particularly relevant in this context but perhaps of use elsewhere, would be automatically to add perimeter input/output pads to a circuit once its perimeter input/output interfaces had been identified.

A more interesting proactive role for the tool is to add additional perimeter registers, so that circuitry with perimeter input/output interfaces can be used as an accelerator SLU. Figure 1 shows an example of how this can be done. In Figure 1*(a)*, a circuit with three input and five output wires at its right-

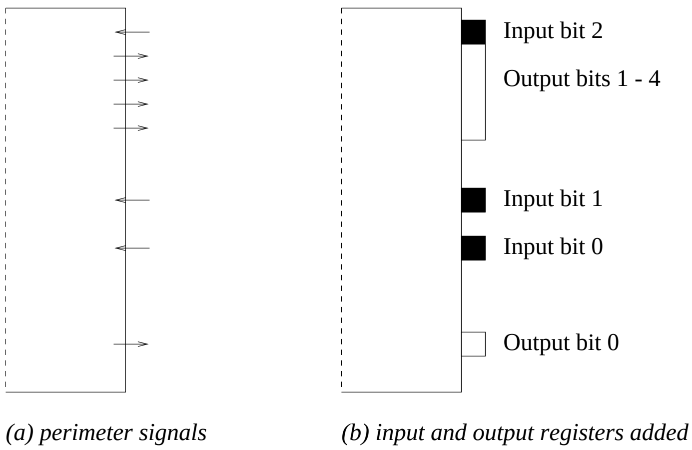

Figure 1. Adding input/output registers at right-hand side

hand edge is shown. Figure 1*(b)* shows extra cells added in an extra right-hand column, to give a three-bit input register and a five-bit output register. The mechanism for implementing the extra registers with interleaved bit positions hinges on the XC6200 mapped register interface [5].

As the example indicates, adding registers at the right-hand edge, or the left-hand edge, is relatively easy, since bit-mapped registers may be placed in single columns. The only issue is to ensure that the total number of input, or of output, bits is less than or equal to the bus width used for register access. Given that this can be no more than 32 bits wide, it would be necessary to use more than one register if there were 56 input bits, for example. Also, of course, an extra column has to be appended to the original circuit. In some cases, it would be feasible to avoid this, if unconfigured registers were available in all the correct positions in the rightmost column of the original circuit.

In contrast, adding registers at the top or bottom edges is not so feasible, because bit-mapped registers cannot be placed in single rows. An additional one or two bit register would be needed for each column that contains an input/output perimeter wire — this is not particularly desirable. In fact, responsibility for avoiding this issue lies with the original circuit designer. It is necessary for XC6200 SLU designers to ensure that registers (or perimeter wires that might be converted) are placed vertically, to use the memory-mapped addressing efficiently.

4 Finding mapped register input/output interfaces

Finding register input/output interfaces is distinctly harder than finding perimeter interfaces, largely because there are no geometrical hints. The situtation is complicated by the fact that registers may be present for internal reasons, unrelated to input and output. In fact, this rules out a totally automatic approach in general, because *any* register can be accessed using the bus-mapped interface, and so there is a need for extra information to identify those registers that are explicitly intended for external access.

As mentioned in the last paragraph of the previous section, the nature of registers relates to the address mapping mechanism of the XC6200. Registers must have a vertical orientation to allow reading and writing with single operations. A subsidiary heuristic, for possible use in finding registers, is that reads and writes can operate in 8, 16 and 32 bit modes, and so these are a natural size to use for registers. Unfortunately, the address mapping mechanism of the XC6200 makes it very straightforward to use register sizes of any number of bits up to 32, so this heuristic is rather weak. The more serious complication introduced by the sophistication of the mechanism is that register bits may be placed in arbitrary column positions, i.e., they need not be contiguous. Not only does this hamper searching for a single register, but it opens up the possibility of two or more registers being interleaved in the same column.

Of course, before a complete register can be identified, it is necessary to locate individual flip-flops that can form part of registers. The XC6200 architecture allows a D-type flip-flop to be configured in any cell of the array. In order to discuss ways of finding register bits, it is necessary to understand exactly how the function unit of a cell is configured. A (slightly simplified) schematic is shown in Figure 2. The two configurable multiplexors labelled 'Y2 mux' and

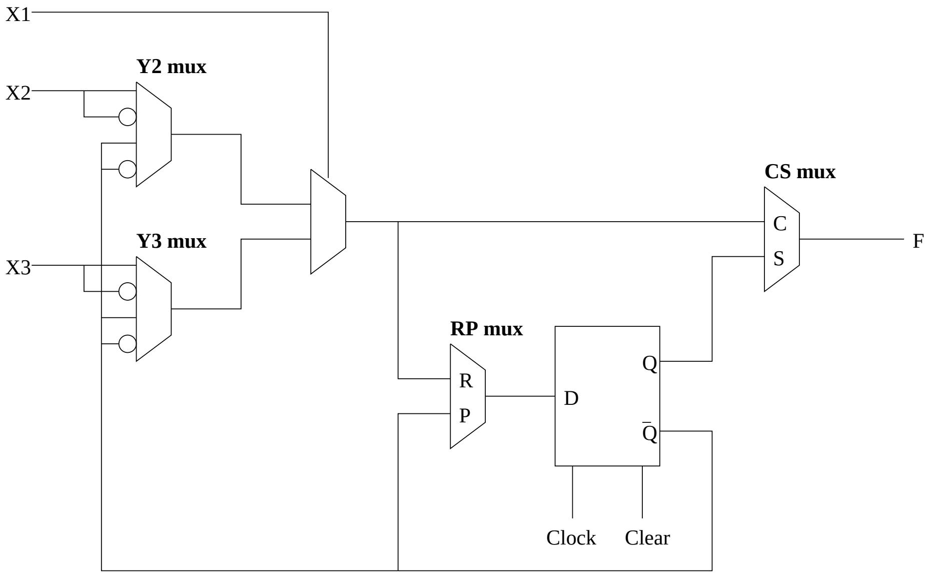

Figure 2. Schematic of XC6200 function unit

'Y3 mux' are concerned with the function computed in the cell, in terms of the input values X1, X2 and X3. The multiplexor labelled 'CS mux' determines whether the logic is combinational or sequential, as can be seen from the figure. Finally, the multiplexor labelled 'RP mux' determines whether the stored bit value is protected or not, that is, whether it can be over-written by the function result.

One preliminary check to determine whether a flip-flop is configured is to confirm that its clock input is connected up. If not, it can be assumed that the flip-flop is not in use. A further check, to rule out flip-flops that contain constant values, is to confirm that the circuit configuration data does not include writing an initial value in the flip-flop. Apart from these two checks, the search heuristics depend on whether an input flip-flop or an output flip-flop is being sought; it is assumed that no flip-flop is designed for both input and output use.

For an input flip-flop, the following are all possible features that are plausible aids to detection, and can be ascertained from the stored circuit representation:

- X1, X2 and X3 are not connected;
- Y1 and Y2 have default settings;
- CS mux is set to S;
- RP mux is set to P.

The first two make sense in that computation of a function seems pointless if the cell is being used to accommodate an input flip-flop. However, they are not totally reliable, in that default or harmless wiring might lead to X1, X2 or X3 being connected, and that default Y2 and Y3 settings might in fact be intentional settings. Moreover, the setting of the CS mux to S seems an essential at first sight. However, looking at Figure 2 more closely reveals that the flip-flop output may act as an input to combinational use of the cell. Therefore, a more apt heuristic is to check that the CS mux is set to S *or* the Y2 or Y3 mux selects the flip-flop output as an input. Note that the second condition cannot arise from the default settings for the Y2 or Y3 mux, so must result from a conscious design decision.

If this new CS/Y2/Y3 condition holds, in addition to the RP mux being set to P, it is reasonable to conclude that the flip-flop is either used for input, or holds a constant value. The latter case arises when the value is zero, and there is no explicit initialisation in the configuration data; it is the only possible cause of erroneous input flip-flop detection. A final point to note is that the RP mux being set to P is not the default setting, and so this fact on its own is usually enough to identify an input flip-flop.

For an output flip-flop, the situation is distinctly less clear cut. The following are possible features that might be of some assistance:

- X1, X2 and X3 are not not connected;
- F output is not connected;
- CS mux is set to C;
- RP mux is set to R.

Unfortunately, none of them yield definitive heuristics. The first one is weak anyway, and may be true just through default settings. The second is the most tempting, but suffers on two grounds. First, an output flip-flop might legitimately be located within a function unit that generates an output signal, sequentially or not; this also makes checking the setting of the CS mux not much help. Second, the default setting for a cell's nearest-neighbour connections selects the function unit output, so it appears that the F output is connected. The RP mux being set to R is a necessary feature, but does not help in differentiating output flip-flops from internal register flip-flops.

As this discussion may suggest, identification of output flip-flops is the weakest feature of the automatic tool. It is possible to find all of the candidates, but the set also includes all internal register flip-flops, and so user interaction is needed. In essence, this problem is inevitable, because the XC6200 bus-mapped interface makes it possible to read all configured registers. Unfortunately, there is no direct way for a circuit designer to flag registers in a way that can be back-deduced from the configuration data. One possible convention might be to mark output flip-flops specially by including initial values for non-protected flip-flops.

5 Conclusions

The tool described in this paper demonstrates that it is feasible for hardware designers to produce SLU circuitry, without the need for awareness of the existence of the SLU concept, or of the virtual hardware model within which the SLU fits. The only constraint on the design process is that the interface to the circuitry must obey certain rules, of types described in [3]. However, circuit designers are well used to interface constraints, so nothing new and unnatural is being imposed. Thus, hardware components of a hardware/software system can be produced in a familiar manner, just as software will be able to use them via function calls (see [3]), also in a familiar manner. Overall, this indicates that hardware/software co-design need not involve a special-case approach. Designers can focus on the interesting questions, like how best to partition systems between hardware and software, rather than the mechanisms for doing so.

As indicated in previous sections, the process of SLU extraction for the XC6200 cannot be entirely automated, and human assistance is needed to help with ambiguous cases. However, this allows the introduction of specialist SLU knowledge at a stage after the hardware design is complete. To illustrate the concept, the tool was tried on a large, complex circuit designed by Tom Kean, which made full use of all available XC6200 features; the test was 'blind', i.e., with no prior knowledge of the nature of the circuit. The tool correctly found the eight input registers in this circuit, and identified a collection of candidate registers that included the eight output registers.

6 References

1. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200", Proc. 6th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1142, 1996, pp.327–336.
2. Brebner, "CHASTE: a Hardware-Software Co-design Testbed for the Xilinx XC6200", Proc. 4th Reconfigurable Architecture Workshop, ITpress Verlag, 1997, pp.16–23.
3. Brebner, "The Swappable Logic Unit: a Paradigm for Virtual Hardware", Proc. 5th Annual IEEE Symposium on Custom Computing Machines, IEEE Computer Society Press 1997.
4. Burns, Donlin, Hogg, Singh and Watt, "A Dynamic Reconfiguration Run Time System", Proc. 5th Annual IEEE Symposium on Custom Computing Machines, IEEE Computer Society Press 1997.
5. Kean, Churcher and Wilkie: "XC6200 Fastmap Processor Interface," Proc. 5th International Workshop on Field Programmable Logic and Applications, Springer LNCS 975, 1995, pp.36–43.
6. Luk, Shirazi and Cheung, "Compilation Tools for Run-Time Reconfigurable Designs", Proc. 5th Annual IEEE Symposium on Custom Computing Machines, IEEE Computer Society Press 1997.
7. Xilinx Inc., XC6200 advance product specification, Xilinx 1996.