Pre-Routed FPGA Cores for Rapid System Construction in a Dynamic Reconfigurable System

Timothy F. Oliver and Douglas L. Maskell
Centre for High Performance Embedded Systems (CHiPES)
School of Computer Engineering
Nanyang Technology University, Singapore
tim.oliver@computer.org, asdouglas@ntu.edu.sg

Abstract

This paper presents a method of constructing pre-routed FPGA cores which lays the foundations for a rapid system construction framework for dynamically reconfigurable computing systems. There are two major challenges that need to be considered by this framework. The first is how to manage the wires crossing a core's borders. The second is how to maintain an acceptable level of flexibility for system construction with only a minimum of overhead. The perceived advantages of full independent core development are weighed against the loss in placement flexibility and elimination of the opportunities to optimise a system across cores. Few existing methodologies allow the independent compilation of FPGA cores through every step of the design flow. In this paper we analyse the wire detail of modern FPGA architectures to determine how the interconnect architecture effects the shape of pre-routed cores and the wire bandwidth available to interfaces. We have adapted academic placement and routing algorithms to our architectural model. The design flow has been modified to include a wire policy and interface constraints framework that tightly constrains the use of the wires that cross a core's boundaries. Using this tool set we investigate the effect of pre-routing on overall system optimality. A simple example using the pre-routed approach shows only a 2% increase in total wire use over the pre-placed approach. Place and route times are vastly reduced for systems composed of regular modules. Being able to break a system into independent cores reduces the placement and routing time even for non-regular systems.

1 Introduction

Many FPGA-based systems are built up from cores developed by multiple third parties. Each compilation step that a developer performs before delivery adds value in terms of a cores performance, predictability and readiness for purpose. An example of this is delivering the value of extra effort on placement optimisation using locking constraints in existing FPGA design tools [13]. As the placement and routing time does not scale well with circuit size, breaking a system into modules and performing place and route on each as separate smaller problems reduces compilation time [13], [15]. The amount of optimisation locked in at the component level could then be traded off against the loss of flexibility and overall system optimality. If end users are able to build applications requiring very little expertise in FPGA EDA tools, then the barriers to entry into using FPGA technology are lowered and the potential user base increased. Further to this, better-optimised cores that are easy to integrate should enjoy a higher level of re-use. This is of course dependent on the impact of pre-routing on performance and flexibility not outweighing these benefits.

Run-time routing, while being flexible, currently requires a large amount of computing bandwidth in comparison to the increase in performance provided through circuit specialization. We investigate the perceived advantages of independent core development for a run-time system versus the loss in placement flexibility and the elimination of the ability to optimise across cores. In the following sections we briefly describe an experimental set of FPGA compiler tools that integrate wire policies and interface constraints to allow the independent compilation of FPGA cores. These tools are used to ascertain the effect that pre-routing has on a simple benchmark system.

2 Uniform routing architecture (URA) model

Modern commercial FPGA architectures use fully buffered unidirectional wires [9], [17], [18]. Unidirectional routing fabrics are superior to bi-directional wire fabrics [8]. With regard to inter-circuit interference it is important to note that a unidirectional routing fabric alleviates any possibility of wire contention. Another important point to note is that a fully buffered interconnect allows a simplified timing model. Path delay is more closely related to the number of wires used as opposed to the wire length used [20]. FPGA interconnects constructed from a single

Dagstuhl Seminar Proceedings 06141
Dynamically Reconfigurable Architectures
http://drops.dagstuhl.de/opus/volltexte/2006/741

tile are both efficient and easy to develop [8]. This led us to a key realisation:

> In a single interconnect tile architecture the placement flexibility of a post-routed core is maintained.

Wire patterns can be re-located by an offset of an integer number of tiles. This led to a second key realisation:

> Abutting the borders of two cores co-locates a set of border edge wires that were independently represented in both cores.

So it is possible in such an architecture to maintain the placement flexibility of pre-routed cores and there is a mechanism for inter-communication between cores. Although the interconnect architecture does not affect placement flexibility, the heterogeneous resource map on modern FPGA devices presents a further challenge to circuit placement flexibility [20]. A widely used FPGA modelling tool is VPR [1]. Unfortunately, this model does not capture the regularity that is required for our methods. As a result, we have created a new architectural model that is based around a single interconnect tile.

2.1 Regular Tiled Architecture

In our URA model [12], a two-dimensional tile space is defined. The positive X direction is to the right along the horizontal axis, while the positive Y direction is down along the vertical axis. Every tile has an identical signal interconnect box, and an identical set of wires that connect to the signal interconnect box. There are sets of wires that cross the tile in the Y direction (The Y channel) and sets of wires that cross the tile in the X direction (The X channel). Additionally, there are sets of wires that connect the resource inputs to the interconnect box (The I channel) and sets of wires that connect resource outputs to the interconnect box (The O channel). The last set of wires connects to a set of global signal outputs (The G channel). The conceptual tile layout is shown in Figure 1.

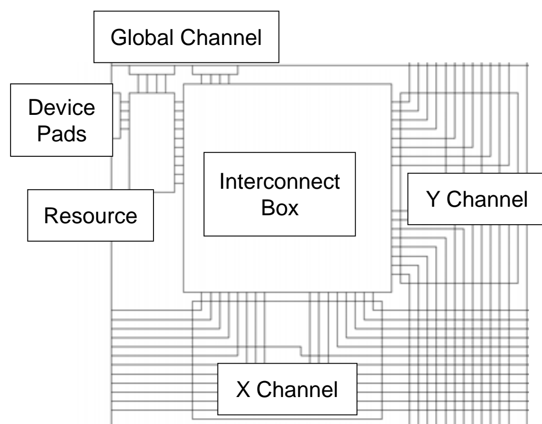


Figure 1. Conceptual Tile Layout

2.2 The Resource Map

Although the interconnect is uniform, each tile may have a unique type of resource. A tile either contains one whole computing resource or a portion of a computing resource. A device will have a width of W tiles in the X dimension and a height of H tiles in the Y dimension. Its tiles are given X, Y coordinates. The top left tile is at 1,1 and the bottom right tile is W, H.

A set of resource types is defined. The number of tiles a resource spans in both the X and Y directions is specified. Each resource must define its configuration fields with a name, bit width and depth. This information is used in the configuration-mapping phase. A resource type can optionally add a number of device IO pads. The device resource map is built up by defining arrays of resource types with an origin coordinate and the number of resources in the X and Y dimension. The resource map generator checks for overlapping arrays.

2.3 Interconnect Box Definition

The interconnect box facilitates the connection between the five channels. Previous works have a separate connection box to connect resource pins to channel wires and a switch box to connect between X and Y channel wires [1]. Connection box flexibility is characterised by the number of tracks to which logic block pins can connect. Switch box flexibility is characterised by the number of choices offered to each incident track by a switch block [22]. In this framework we merge these boxes. This allows us to define all the connection options in the same way. Each input to a channel may be driven by one of several outputs selected from any channel. Note that this is not the physical structure that is being dictated here but the conceptual URA model.

Many previous works have investigated ways to generate good patterns of input to output options [22]. It has been shown that a switch box flexibility of 3 will yield a good interconnect architecture [1]. However on inspecting commercial architectures it was noted that they exhibit a flexibility of up to 8. Certain interconnect patterns inside the box have been shown to improve the performance of an interconnect fabric [22].

A set of MUX patterns is first defined, each made up of a number of turns from the set straight, from left, from right, u-turn, from resource and from global (Figure 2). The directional turns are relative to the signal travelling from an output and turning to an input. This allows the same patterns to be re-used for inputs to all four directions. The architecture generator attempts to find outputs to populate the MUX inputs using a specified pattern.

2.4 The URA Wire Model

The URA wire model is based on X and Y channels built from a number of wire sets [12]. Every channel has one or more groups defined. Each group has one or more members defined. The G, I and O channels have one wire per channel member. A resource's wire members in an I-Channel group may be declared as equivalent. This is true if the wires in a group are connected to the same LUT or logic gate or if a MUX is used inside the resource to select between group members. O-Channel groups may be declared as having equivalent members. A wire set is defined by three values:

W: Wire set size, pins numbered 1 to W
W is greater than 1

IN: The input selection binary vector of length W, Elements numbered 1 to W

OUT: The output selection binary vector of length W, Elements numbered 1 to W

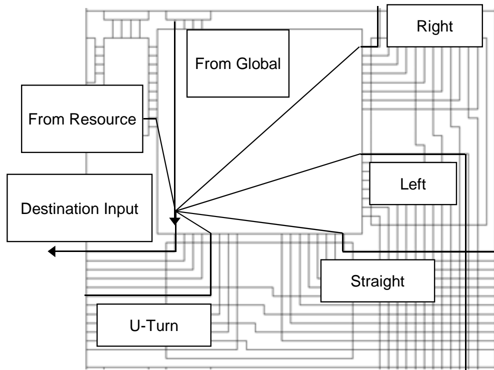


Figure 2. Pattern Turn Definitions

A channel wire set of size W has W wires in each tile. In each tile, one wire in a set will begin and one will end. If not truncated by the edge of the device each wire in the set will span W tiles. A wire has a pin in every tile it passes through, numbered 1 to W.

Wires do not need to have connections at every tile they span. This is commonly referred to as internal connection depopulation [1]. A wire pin may be connected to a sink or source signal from the tiles' interconnect box. The IN vector indicates the pins that have an input to the wire from the tile interconnect box. The OUT vector indicates the pins that have an output from the wire to the tile interconnect box. If the sum of all elements of the IN vector is greater than one (the wire set has more than one input) then each input driver to the wire requires a tri-state control. We assume a fully buffered interconnect. The interconnect box specification adds further detail of what resource or wire outputs can connect to other resource or wire inputs.

Figure 3 shows the detail of one wire in a set where W=3 IN=101 and OUT=101. It also shows a unidirectional wire set of W=2 IN=10 and OUT=01. The input MUXs are shown with 4 inputs selected using 2 configuration bits for each MUX. The W=3 wire has more than one driver so tri-state buffers are required. For a wire set with more than one input, the default state of the input driver on the pin closest or equal to 1 is to drive the wire (At 1,1 for the W=3 example wire). The default state of all other input drivers is to be tri-stated. For a wire set to be valid a signal has to be able to travel its full length from pin 1 to pin W or pin W to pin 1.

Figure 3 also shows a Y channel with 4 wire sets with W=2 and 4 wires sets of W=3. The W=3 wires can be seen spanning from tile 1,1 to 1,3. The W=2 wires can be seen spanning from tile 1,1 to 1,2 and 1,2 to 1,3. Note that every tile is identical. Wires in a wire set of size W occupy W-1 positions along their length in a staircase fashion. These positions increment in the positive direction perpendicular to the positive direction of the wire. The tile at the minimum end, the wire begins in position 1. In the second tile the wire will move from position 1 to position 2 and so on until the maximum tile, where it has reached position W-1. Thus wires in a set are stacked up on a single tile. A device is laid out by tessellating this tile. The wires connect by abutment.

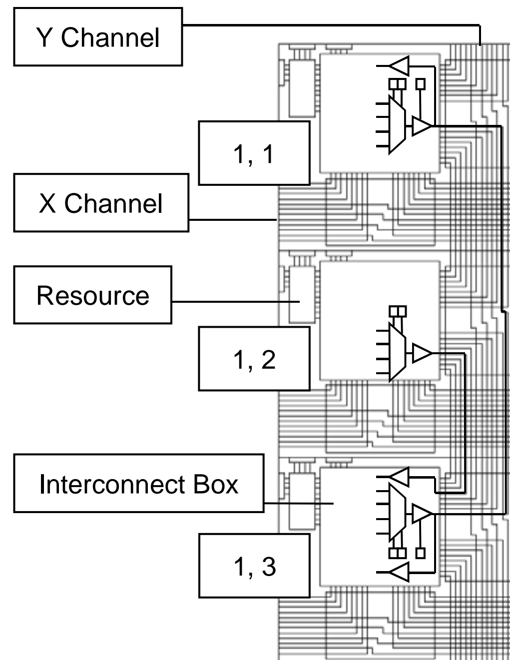


Figure 3. Y-Channel wire set construction using tiles. Overlay showing switch model

3 Rapid System Construction on FPGA

A good component-based methodology for system construction on FPGA will encourage component sharing and re-use. We have extracted some of the basic ideas from the abstract tile methodology described in [7]. An abstract tile represents a core before placement and routing has been performed. Each core has a unique functional identity and a number of signals visible externally. Each signal has an associated direction. The signals are grouped into abstract interfaces. Each abstract interface is assigned to one of four border edges. We refer to an interface assigned to a particular border edge as a port. Abutting the ports of two communicating cores creates a link. Therefore an interface cannot be split across more than one border edge. A link only connects point-to-point between two ports. A tile will only affect other tiles it is linked to through its ports. Initially an abstract tile does not have any dimensions. In the design flow presented here, the function and signals of a core are defined by a net-list of components. Each component is a piece of logic that may be configured on the target FPGA. A core is first shaped. Then its interfaces are assigned to absolute edge positions. A major issue in the independent construction of cores on an FPGA is wire contention [2]. In order to remove any chance of wire contention between cores on the same FPGA we have developed a wire policy framework. This explicitly defines which wires are free for use by a core constructor for interface signals and for internal connections. Once the policy has been applied the interfaces are assigned to specific wires on the border edge. The core is then placed and routed within its borders.

The system constructor must adhere to two rules:

- All cores use the same wire policy
- Core boundaries must not overlap

This provides two valuable properties:

- No interference between cores
- Ports are connected by abutment

Contention avoidance has been ensured at core compile time by adhering to a wire policy. Thus the construction process does not need to consider the detailed allocation of signals to wires or detailed component placement within cores, and so does not have to run any complex placement or routing algorithm. Instead the process only needs to place a core's rectangle to match up connecting ports without overlapping any core already placed. This rapid system constructor is referred to as a "Placer-Connector" as it performs placement and connection simultaneously.

3.1 Wires Crossing Core Borders

Consider a bounding box around all the tiles assigned to a particular core. Signals either travel in a positive (P) or negative (N) direction across this border, along either the vertical (Y) or horizontal (X) axis. Consider two independently constructed cores, using wires for P and N directed signals, one on each side of the border. If one constructor selects a wire without negotiating with the other there is a high probability that the same wire is selected for both the P direction on one side and the N direction on the other. Furthermore it may be that a wire that crosses a border is also used to connect two tiles on one side of the border. When the two cores are abutted destructive contention will occur.

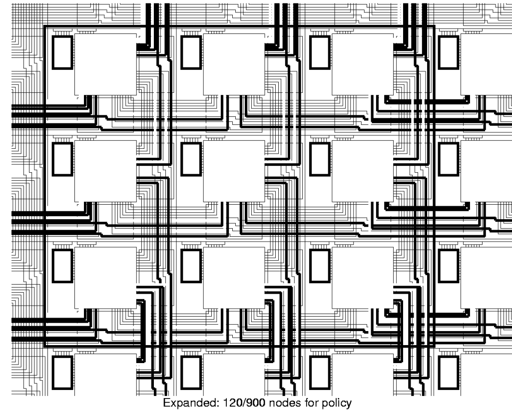


Figure 4. Core border edge wire starvation effect. Contended wires highlighted

Routing exclusion zones have been proposed to avoid this type of contention. These zones have to be as wide as 3 to 6 tiles in commercial architectures. Resources in the exclusion zone cannot be used [4], [6]. If we do not use the wires that cross a border the core area will experience wire starvation. Figure 4 illustrates this effect where tiles in the corners have half their wires excluded. This effect is more pronounced when longer wires are used and for wires with a high degree of internal connection depopulation.

We propose constraining the use of border-crossing wires and using them to carry signals between abutting cores. Commercially available CAD tools provide a limited facility to assign signals to wires [19]. An improved solution for commercial architectures has been reported [13]. Previous investigations suggested that 50% more routing resource is required for such a scheme of locking signals to wires [15].

We define the number of wires crossing the border between two tiles as the maximum tile edge bandwidth W_{FPGA}. The W_{FPGA} of an interconnect architecture is equal to the sum of Wi-1 for all i wire sets in a channel. This is the theoretical maximum number of signals that can propagate across the tile segment of a core's boundary edge. The estimated available W_{FPGA} in the routing channel of commercial reconfigurable architectures is:

Virtex	84	Spartan-3	138
Virtex-II	168	Virtex-4	168

These figures exclude global, long and tri-state lines. Also note that any one core cannot use the entire W_{FPGA}

bandwidth of a border. It must be divided between cores on both sides of a border, based on the wire constraints framework, which is explained in the next section.

4 Wire Constraints Framework

The wire constraint framework has two layers. The first layer is the wire use policy, which ensures contention free inter-operability of independently constructed cores on the same FPGA device. The second layer is the interface layer that allows designers to develop pre-routed cores with compatible interfaces. The policy layer provides an appropriate wire bandwidth to the interface layer. It is envisaged that policies will be developed and optimised to an FPGA architecture by device experts. The interface definitions are then developed and shared between designers of compatible cores.

4.1 Wire-use Policy Layer

It is possible to use wire constraints with cores constrained to any polyomino. For simplicity we assume that each core is constrained to a rectangular region. Four border edges define a core's boundary namely: Positive X, Negative X, Positive Y, and Negative Y (PX, NX, PY, and NY). The wire policy defines how every wire that crosses these borders may be used. The combination of wire set direction and the border that the policy is being applied to determines whether a wire set is incoming or outgoing. All wires belonging to a reserved set, including those that do not cross a border are considered external.

A wire policy specifies the following:
- The direction of each wire set
- The wires in a set that carry interface signals
- Whether a wire set is reserved

For wires on a border belonging to an:
- Incoming wire set
  - Interface wires are only used for core inputs
  - Non-interface wires are considered external
- Outgoing wire set
  - Interface wires are only used for core outputs
  - Non-interface wires are available for internal use

The policy is applied uniformly to every channel in a given direction. This maintains placement flexibility between abutting cores along the axis parallel to their abutting surfaces in increments of whole tiles. All wires in a set are forced to follow the same direction. Any wire that has both its end points outside of a core's boundary is external to that core. If such a wire were driven within the core the direction set by the policy would be violated. Enforcing the same direction within a wire set maintains placement flexibility along the axis of a wire channel.

As an example of a policy formulation, consider the wire set W=3, IN=100 and OUT=011. This set potentially provides two signal paths across the border. However the first wire in this set crossing the border is also useful for making internal connections. The decision on whether to provide extra bandwidth or to keep this wire for internal use is captured in a wire policy. The policy is also used to share bandwidth between directions across a border. The channel in Figure 3 has a W_{FPGA} of 12. There are two negative going sets of W=2 IN=10 and OUT=01, two positive going sets of W=2 IN=01 and OUT=10 and four bi-directional sets of W=3 IN=101 OUT=101. Now consider how many signals are available in a given direction. The bi-directional sets can provide 8 signals in either direction. The policy layer seeks to fix the direction before interface design time. As cores in a system will probably have the same average number interface signals in each of the two directions, it is sensible to share the bi-directional wires between positive and negative. Then the W_{FPGA} is split into a WP_{FPGA} of 6 and a WN_{FPGA} of 6. As the pins on the W=3 wires are bi-directional they provide routing flexibility through dogleg opportunities. Therefore we allow only one wire in a W=3 set to be a signal carrier. This reduces the W_{FPGA} in each direction to 4.

4.2 The Interface Layer

The policy provides wire bandwidth to interfaces. An abstract interface definition is simply an ordered list of identified signals and their direction. Links are always point-to-point so distribution of data has to be handled within modules. A link requires two compatible mirror image interfaces. An output signal from one side implies an input to the other side and vice versa. Bi-directional interface signals are not allowed. An abstract interface is firstly declared in its original sense. It is either interpreted in its original sense or in the inverse sense depending on the direction of the signals defined in a core's net-list. When an abstract interface is applied to one of the four directions then each signal in the interface is assigned a wire. An assigned interface is specific to a direction and defines the tile offset, wire set and wire index for every signal in the interface. An assigned interface is suitable for export to multiple core developers. An assigned interface is assumed to be in its original sense. The design tools will interpret it in accordance with the applied policy in a consistent way. For any given signal in a compatible interface, the actual wire selected on each side of a border is the same wire when the two border edges are co-located.

4.3 Core Shape and Wire Relationship

It is important to note how wires affect the choice of a cores shape. And in turn how a core's shape affects the availability of wires. An assigned interface has a fixed width and its wires will have a fixed depth reaching in to a core from the border. The interfaces used dictate a minimum core size.

For example, the depth of a core along a wire channel has an effect on the wire bandwidth available. Consider a core of depth D=4 on an architecture with the bi-directional wire set W=7, IN=1001001, OUT=1001001. Without a policy, two wires in this set are able to make connections internal to the core. Two wires in this set pass right through this core. If one of these is driven from its mid-point it will drive signals in both directions. Once a direction is enforced by a policy only one wire is available for internal connection. The two through wires may only be used as inputs. A further four wires may be used as inputs and three wires are available as outputs

In order to illustrate some of the bandwidth issues consider a simple registered adder core. This core will use two instances of a dual integer interface. A dual 8-bit integer interface has two 8-bit buses and two valid flags, requiring 18 wires. All signals in the original abstract interface are declared as outputs. An inverse instance placed on the NX edge creates an input interface. An original instance placed on the PX edge creates an output interface. Assuming that each tile resource in the example architecture in Figure 3 can accommodate one registered full adder. A further resource is required to perform an AND operation on the two input valid signals and register it before output. Thus 9 tiles are required for the 8-bit adder. This would fit neatly into a 3 by 3 box, but would require a W_{FPGA} of 6. Fitting a 16-bit adder into a near square 4 by 5 box with the same port locations requires a W_{FPGA} of 7. A 32-bit adder in a 6 by 6 box requires a W_{FPGA} of 11.

Clearly this is getting impossible. The module area needs to be expanded just to accommodate the required W_{FPGA}. The extra resources in the expanded area would go unused. In this case simply fixing the X dimension to 2 forces the interface edges to lengthen. The 8-bit adder fits in a 2 by 5 box and has a W_{FPGA} within the fictitious 4 wires per tile constraint. Both the 16-bit adder in a 2 by 9 box and 32-bit adder in a 2 by 17 box meet the W_{FPGA} constraint too. Although this example was applied to a very constrained architecture, similar issues will arise in larger architectures with higher logic density and more complex cores. These issues are exacerbated further if there are ports on both X and Y-axis edges.

5 Experimental Tools

In order to investigate the impact of pre-routed cores we have put together a set of core compiler tools. The design flow is as follows:

The Core is described in Verilog HDL
Interface signals are annotated in a HDL wrapper
The core is synthesised to an EDIF net-list
The Core's shape and wire policy selection is defined
The Net-list is mapped to architectural primitives
Pre-assigned interface positions are elaborated
Combined placement and packing is performed
The Interconnect graph is created
The wire policy constraints are extended to the graph
Routing is performed respecting the wire constraints
The circuit is packed into a core ready for construction
The system constructor places and connects the cores

We use Verilog HDL to describe a core's function. The Verilog module is placed in a wrapper module that includes the interface type and port location in each signal name. The Verilog code is compiled into an EDIF net-list which is annotated with a reference to the policy and the module shape. The synthesis target is the Xilinx Virtex architecture, with the EDIF primitives mapped to our basic logic cell (BLC) which contains a 4-input LUT, MUX, FF and Carry logic similar to half of a Virtex CLB slice.

Before placement an area of resource is allocated to the core. Shaping this area is a complex process that has to take into account the different types of resource required (e.g. IO, RAM or special purpose logic) and what is available in the desired location of the core. It is further complicated by the constraints placed on a core's shape due to the interface width and depth. Cores are currently shaped manually. We hope to develop a better understanding of the interrelationship between the factors involved and then develop effective ways to automate the core shaping process.

The next step is to pack the BLCs created in the mapping phase into a minimum number of resource sites while accounting for the connectivity between them. This packing problem is complicated by the fact that some BLCs connect to interface wires. The simulated annealing placement algorithm from VPR [1] has been adapted to perform packing and placement in a single step. Previous work has shown that this is more optimal [3]. We then only need to handle the interface wire placement in the placer.

The PathFinder negotiating based congestion driven algorithm [10] has been modified to take advantage of the uniformity inherent in the URA model and to handle the wire constraint framework. Since the device is constructed from a single uniform interconnect tile we only need to represent the connectivity graph for this single tile. We reduce the memory required by only constructing an interconnect graph for the area covered by the core being routed, as opposed to the whole device. Due to the discrete interconnect structure of modern commercial devices, the delay difference is nearly equal for different length wire sets. For example the Xilinx Virtex-II architecture hex lines have only a slightly longer delay than the delay of the double or direct lines [20]. We therefore assume that a minimum delay trace will use a minimum number of wire lines. Thus we use a router cost function that is based on the number of wire hops used in a trace.

5.1 Run-time constructor

Port compatibility is ensured at core compile time so the construction process only has to place a core's rectangle to correctly match connecting ports without overlapping any

core already placed. This "Placer-Connector" method of rapid system construction is shown in Fig. 5. A set of pre-routed cores are shown in Fig. 5(a). The IF core connects to external signals so has to be placed in a specific location on the FPGA as shown in Fig. 5(b). The PE and IF cores use two instances of a single interface type constrained to port regions indicated by the thin lined boxes. Arrows indicate the interface sense, not the signal direction. Each PE core performs the same function but has a different interface-edge combination. Fig. 5(c) shows the system created from abutted cores. Links are created by the co-location of port wires, as shown by the co-located arrows.

The IF core connects to the FIFO core which uses memory in the centre of the device. The ports between IF and FIFO are mapped onto tunnelling wires. The FIFO and IF are connected using two tunnelling-link cores overlaying the PE cores. The PE cores were constructed for an area of CLBs. The position of the right most PEs is displaced by the memory resource. Wire extension cores are used to connect the PEs across this region.

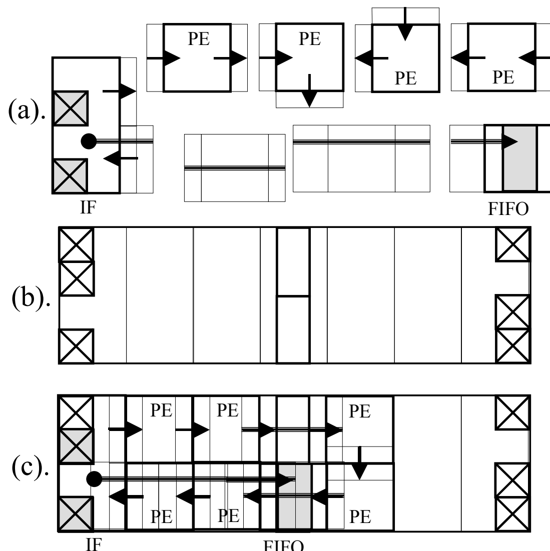


Figure 5. (a). Pre-routed cores (b). Target FPGA device (c). Cores placed and connected on device.

6 Experimental Results

In order to judge the impact of pre-routing cores we compare three methods of system construction. First every core in the system is synthesised into a net-list. Two directives are used to build a system. The "Instance" directive places cores and the "Link" directive specifies which ports are to be connected. Each system is constructed using the following three methods:

- Normal: Merge all the cores, then Place and Route the system
- Pre-placed: Place each core, merge all cores, then route the system
- Pre-routed: Place and route each core then merge the system

The performance of each method is measured and compared in terms of construction time and the number of wire hops used across the system. For systems with multiple instances of the same core we expect to see a reduction in construction time for the pre-routed core method. For systems with entirely unique instances there should still be a reduction in construction time for the pre-routed core method. We expect this as the placement and routing problems are being broken into smaller sub-problems. The anticipated counter effect of this is an increased number of wires required to connect the system.

6.1 A Simple Benchmark

The benchmark system used is a simplified version of an FPGA accelerator for the Smith-Waterman algorithm used for pair-wise alignment of DNA sequences with a linear gap penalty [11]. The system is constructed from a number of identical processing elements (PE). A PE has one input interface and one output interface. They are connected up into a linear array that is terminated at either end by a host interface. A subject sequence is first loaded into the array. Then any number of sequences may be passed through. As they exit the array they will be annotated with the alignment score. This system has two abstract core tiles; the host interface (HI) and the PE. The size of each of these cores is shown in Table 1.

Table 1. Core parameters

Core	Blocks	Nets	Sinks
PSA HI	85	63	86
PSA PE	205	221	382

In order to be able to create the system from pre-routed tiles several versions of the PE are required. Each version has a different interface-edge combination. The cores required for a simple case are shown in Figure 6. With 4 PE interface-edge combinations the linear array is able to extend to the right, loop around and return to the host interface. In order to extend in more directions, more interface-edge combinations are required.

The target FPGA architecture has a wire channel composed of 16 W=2, IN=01, OUT=10 unidirectional wire sets and 16 W=3, IN=101, OUT=101 bidirectional wires sets. Each tile in the FPGA either has 4 pads or 4 logic blocks. The layout of a system of four PEs and the HI is shown as a schematic in Figure 6(b) and as pre-routed cores mapped to the FPGA architecture shown in Figure 7.

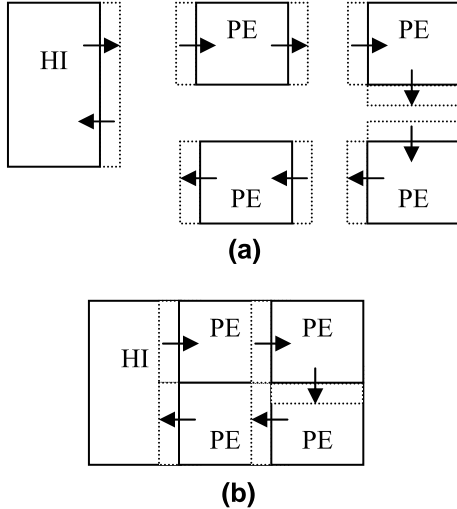


Figure 6. (a) Pre-routed cores with different edge combinations (b) 4 PE system construction

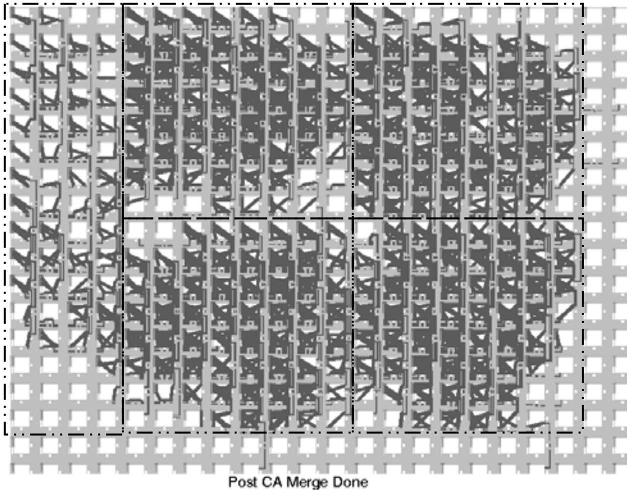


Figure 7. A system of four PEs constructed from pre-routed cores on an FPGA model

Systems of 2, 4, 6, 8 and 10 PEs where defined and then constructed using each of the three methods. The 2 PE system uses 3 unique module instances. The rest use 5 unique module instances. A comparison of each construction method is shown in the graphs of Figure 8.

As expected Figure 8(a) shows that routing each unique module once vastly reduces the number of router iterations required as compared to routing the system as a whole. For the 2 and 4 PE systems we see a reduction in router iterations even when every module is unique. This suggests that being able to break up the routing problem into isolated modules is a valid technique for improving router run-times.

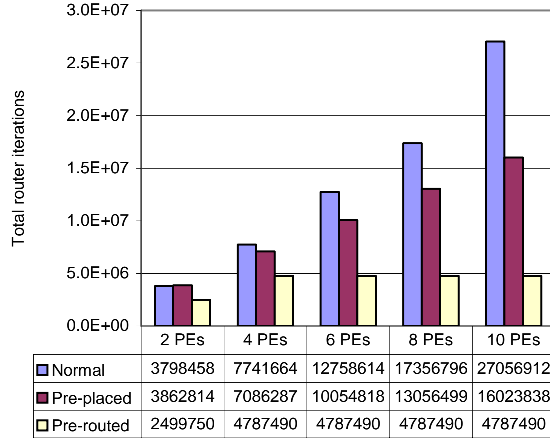
The basic simulated annealing placement algorithm we used does not scale well with the system size. In Figure 8(b) the total wire length of the normal approach is increasing faster than the pre-placed approach. Comparing pre-placed and pre-routed wire usage in Figure 8(b) shows the overhead of pre-routing to be around 2%. The nets in this system have a natural locality so the pre-placed approach works well.

Figure 8(c) compares the longest path across each system generated by each of the different approaches. The benchmark system, as in most practical systems, does not have any nets that lengthen with the number of modules added. Thus we see the longest path is predictable for any number of pre-routed cores.

While the linear array benchmark presented here is admittedly relatively simple and highly conducive to using pre-routed cores, it is intended to provide a simple illustration of the 'placer-connector" concept. Where communications cannot be established by abutting two interfaces, we suggest the use of tunnelling-link cores which use a reserved set of wires as defined by the framework. Alternatively, it is feasible to use a fast run-time router, which has access to these reserved wire sets to connect modules. Our framework has been designed with this in mind, but its implementation has been left as "work in progress".

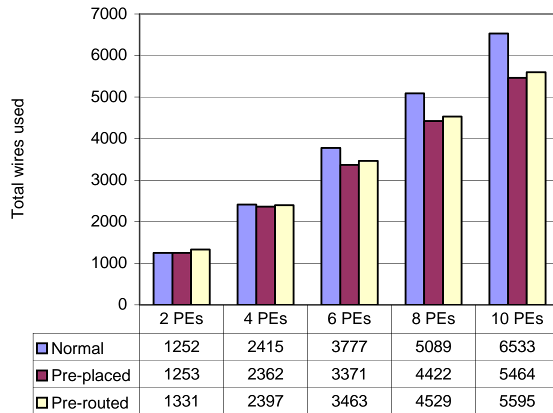
7 Conclusion

Modern FPGA wire fabrics are homogeneous allowing pre-routed patterns to be relocated. We have presented a wire constraint framework that effectively utilises the full bandwidth of a device's interconnect. This has been built into a method of creating pre-routed FPGA cores for rapid system construction. Pre-routing doesn't affect placement flexibility directly. However, pre-routing requires that a core's interfaces be locked to a certain edge. This reduces placement flexibility between connected cores. In order to regain lost placement flexibility we use several versions of the same function core each with a different interface-edge combination. Pre-routing small systems causes an increase in the number of wires used in connecting the system. This may result in an increase in wire delay and a reduction in system performance. The performance of a pre-routed system starts to increase over the normal and the pre-placed versions as the size of the system increases. We define rapid construction as the ability to build a system by just placing pre-routed cores without having to perform complex place and route steps. We see that being able to break a system into cores and independently optimise them accelerates the core construction process as well. The improvement in system performance and reduction in both CAD run-time and system construction time make core pre-routing an interesting avenue for further research.



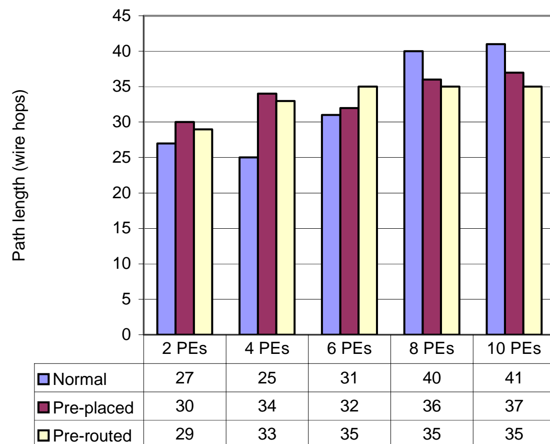
	2 PEs	4 PEs	6 PEs	8 PEs	10 PEs
Normal	3798458	7741664	12758614	17356796	27056912
Pre-placed	3862814	7086287	10054818	13056499	16023838
Pre-routed	2499750	4787490	4787490	4787490	4787490

(a).



	2 PEs	4 PEs	6 PEs	8 PEs	10 PEs
Normal	1252	2415	3777	5089	6533
Pre-placed	1253	2362	3371	4422	5464
Pre-routed	1331	2397	3463	4529	5595

(b).



	2 PEs	4 PEs	6 PEs	8 PEs	10 PEs
Normal	27	25	31	40	41
Pre-placed	30	34	32	36	37
Pre-routed	29	33	35	35	35

(c).

Figure 8. Comparison of system on FPGA construction techniques (a). Router iterations (b). Total wires used and (c). Longest path

7.1 Future work

The aim of our work is to reduce the overheads of the system constructor to the point that it is possible to use it at run-time in a resource constrained embedded system. Issues such as a run-time router for connecting those blocks that cannot be connected by abutment and the distribution of global signals at run-time still need to be investigated. It is recommended that the number of interface types be kept to a minimum in order to reduce the number of interface-edge combinations required to build a system. There is a large amount of wire bandwidth in commercial FPGA devices. We are investigating ways to use this more effectively. One use is to reserve wire sets and use them to connect non-neighbouring cores, IO pads or special resources across pre-routed regions. The methodology presented here is not path timing driven. It is, however, possible to modify the tool set used to be timing driven.

8 References

[1] V. Betz, J. Rose, and A. Marquardt, “Architecture and CAD for Deep-Submicron FPGAs”, *Kluwer Academic Publishers*, February 1999.

[2] S. M. Charlwood and S. F. Quigley, "The Impact of Routing Architecture on Reconfiguration Overheads", *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, USA, 2003, pp. 102-110.

[3] G. Chen, J. Cong, “Simultaneous Timing Driven Clustering and Placement for FPGAs”, *Proc. Of 14th International Conference on Field Programmable Logic and Application,* Belgium, 2004, pp. 158-167.

[4] M. Dyer, C. Plessl , M. Platzner, “Partially Reconfigurable Cores for Xilinx Virtex”, *Proc. of 12th International Conference on Field-Programmable Logic and Applications*, 2002, pp. 292-301.

[5] K. Eguro, S. Hauck, A. Sharma, “Architecture-adaptive range limit windowing for simulated annealing FPGA placement”, *Proc. of 42nd Design Automation Conference,* USA, 2005, pp. 439-444.

[6] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," *Proc. of 39th Design Automation Conference*, USA, 2002, pp. 343-348.

[7] G. Lee and G. Milne, “Building Run-Time Reconfigurable Systems From Tiles”, *Proc. Of 13th International Conference on Field Programmable Logic and Application, Portugal*, 2003, pp. 252-261.

[8] G. Lemieux, E. Lee, M. Tom and A. Yu, “Directional and Single-Driver Wires in FPGA Interconnect”, *Proc. of the IEEE International Conference on Field-Programmable Technology*, Australia, 2004, pp. 41-48.

[9] D. Lewis, et. al. “The Stratix-II logic and routing architecture”, *In ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, USA, 2005, pp. 14-20.

[10] L. McMurchie and C. Ebeling, “PathFinder: a negotiation-based performance-driven router for FPGAs” *In ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, USA, 1995, pp. 111-117.

[11] T. Oliver, B. Schmidt, D. Maskell, “Reconfigurable Architectures for Bio-sequence Database Scanning on FPGAs”, *IEEE TCASII* Vol. 52, No. 12, 2005, pp. 851-855.

[12] T. Oliver, D. Maskell, “An FPGA Model for Developing Dynamic Circuit Computing”, *IEEE Field-Programmable Technology*, Singapore, 2005.

[13] P. Sedcole, B. Blodget, J. Anderson, P. Lysaght, T. Becker, “Modular Partial Reconfiguration in Virtex FPGAs”, *International Conference on Field Programmable Logic and Applications*, 2005, pp. 211-216.

[14] S. Singh, “Death of the RLOC?” Proc. Of *8th IEEE Symposium on Field-Programmable Custom Computing Machines,* USA, 2000, pp. 145-152.

[15] R. Tessier, “Fast Place and Route Approaches for FPGAs”, *PhD Thesis, Massachusetts Institute of Technology,* 1999.

[16] “Virtex™ 2.5 V Field Programmable Gate Arrays Product Specification” DS003 *Xilinx Inc,* Version 2.5, April 2001.

[17] “Virtex-II™ Platform FPGA User Guide UG002, *Xilinx Inc,* Version 1.9, December 2002.

[18] “Virtex-4 Family Overview”, DS112 *Xilinx Inc,* Version 1.3 March, 2005.

[19] “Application Notes 290. Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations”, *Xilinx Inc,* Version 1.2, September, 2004.

[20] M. Wang, A. Ranjan, S. Raje, "Multi-Million Gate FPGA Physical Design Challenges", *ICCAD*, 2004, pp. 891-898.

[21] S. Williams, “Icarus Verilog”, Available online at http://www.icarus.com/eda/verilog/, Last accessed January 2006.

[22] S. Wilton, “Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories,” *Ph.D. Dissertation, University of Toronto*, 1997.