2

State of the art

2.1 Introduction to FPGAs

A Field programmable Gate Array (FPGA) is an integrated circuit that is designed to be configured after manufacturing. FPGAs can be used to implement any logic function that an Application Specific Integrated Circuit (ASIC) can perform. For varying requirements, a portion of FPGA can also be partially reconfigured while the rest of an FPGA is still running. Unlike other technologies, which implement hardware directly into silicon, any errors in the final FPGA-based product can be easily corrected by simply reprogramming the FPGA. Any future updates in the final product can also be easily upgraded by simply downloading a new application bitstream. The ease of programming and debugging with FPGAs decreases the overall non-recurring-engineering (NRE) costs and time-to-market of FPGA-based products.

The reconfigurability of FPGAs is due to their reconfigurable components called as logic blocks, which are interconnected by a reconfigurable routing network. There are two main routing interconnect topologies: Tree-based routing network [A.DeHon, 1999] [Marrakchi et al., 2009] and Mesh-based routing network [Betz et al., 1999]. A tree-based FPGA architecture is created by connecting logic blocks into clusters. These clusters are connected recursively to form a hierarchical structure. On contrary, a mesh-based FPGA architecture interconnects logic blocks through a 2-D mesh of routing network. Tree-based interconnect topology occupies less area than the mesh-based interconnect topology [Marrakchi, 2008], however a tree-based FPGA suffers from layout scalability problems. The layout of a mesh-based FPGA is scalable and is thus commonly used by commercial FPGA vendors such as Xilinx [Xilinx, 2010] and Altera [Altera, 2010]. This work focuses only on mesh-based routing

H. Parvez and H. Mehrez, *Application-Specific Mesh-based Heterogeneous FPGA Architectures*,
DOI 10.1007/978-1-4419-7928-5_2, © Springer Science+Business Media, LLC 2011

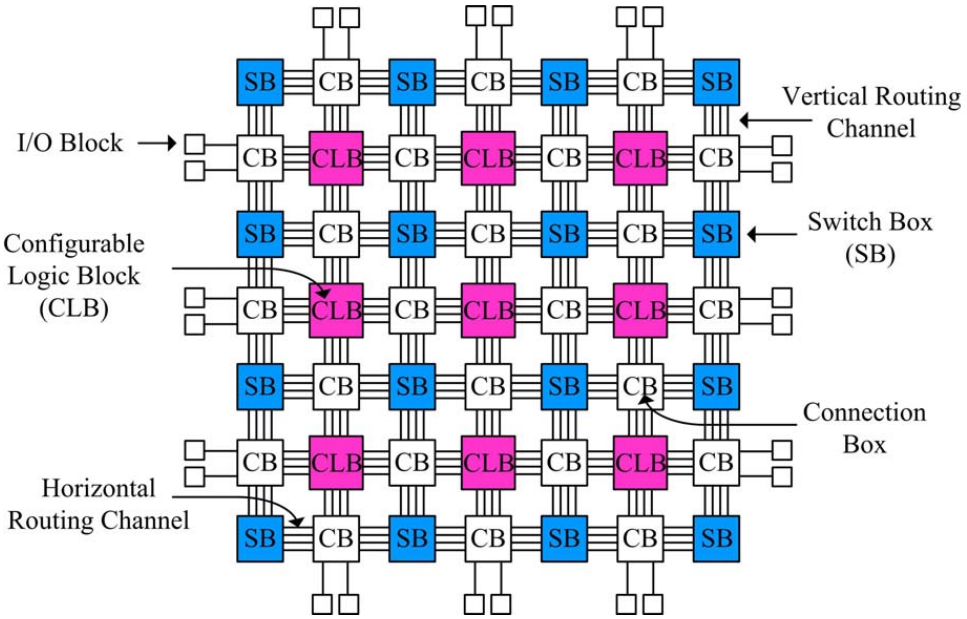


Figure 2.1: Overview of FPGA Architecture [Betz et al., 1999]

topology.

Figure 2.1 shows a traditional mesh-based FPGA architecture. The configurable logic blocks (CLBs) are arranged on a 2D grid and are interconnected by a programmable routing network. The Input/Output (I/O) blocks on the periphery of FPGA chip are also connected to the programmable routing network. The routing network comprises of horizontal and vertical routing channel tracks. Switch boxes connect horizontal and vertical routing tracks of the routing network. Connection boxes connect logic and I/O block pins with adjacent routing tracks. A software flow converts a target hardware circuit into interconnected CLBs and I/O instances, and then maps them on the FPGA. The software flow also generates a bitstream, which is programmed on the FPGA to execute the target hardware circuit. The mesh-based FPGA, and its software flow is described in detail as below.

2.1.1 Configurable Logic Block

A configurable logic block (CLB) is a basic component of an FPGA that implements logic functionality of a target application design. A CLB can comprise of a single basic logic element (BLE), or a cluster of locally interconnected BLEs. A simple BLE consists of a Look-Up Table (LUT), and a Flip-Flop. A LUT with k inputs (LUT-k) contains 2^k configuration bits; it can implement any k-input boolean function. Figure 2.2 shows a simple BLE comprising of a 4 input Look-Up Table (LUT-4) and a D-type Flip-Flop. The LUT-4 uses 16 SRAM (static ran-

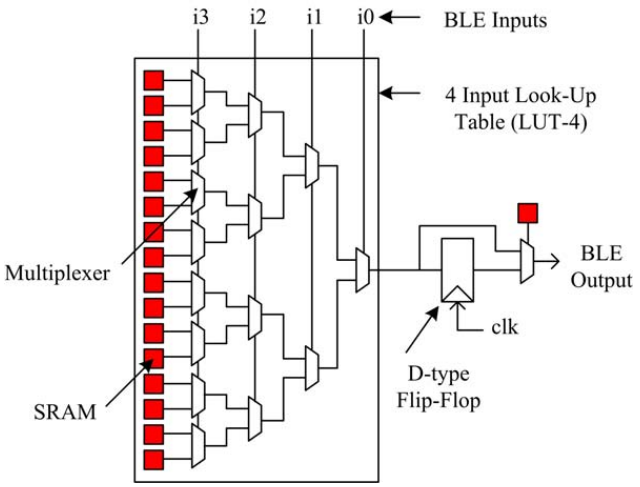


Figure 2.2: Basic Logic Element (BLE)

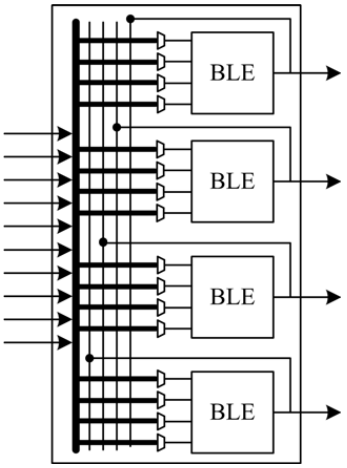


Figure 2.3: A Configurable Logic Block (CLB) having four BLEs

dom access memory) bits to implement any 4 inputs boolean function. The output of LUT-4 is connected to an optional Flip-Flop. A multiplexor selects the BLE output to be either the output of a Flip-Flop or the LUT-4.

A Look-Up Table with more number of inputs reduces the total number of LUTs required to map a hardware circuit. More logic functionality can be mapped in a single LUT. This eventually reduces the intercommunication between LUTs, and thus the speed of hardware circuit improves. However, a LUT with more number of inputs increases its area exponentially. [J.Rose et al., 1990] and [E.Ahmed and J.Rose, 2000] have measured the effect of the

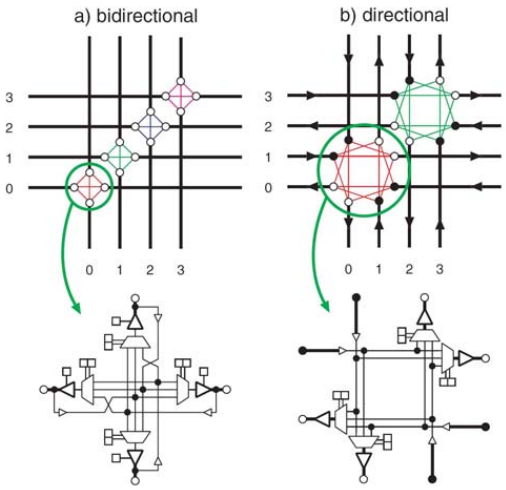


Figure 2.4: Switch Block, length 1 wires [G.Lemieux et al., 2004]

number of LUT inputs on area, speed and routability of FPGAs. They have concluded that 4-input LUTs provide a good tradeoff between speed and density of FPGAs.

A CLB can contain a cluster of BLEs connected through a local routing network. Figure 2.3 shows a cluster of 4 BLEs; each BLE contains a LUT-4 and a Flip-Flop. The BLE output is accessible to other BLEs of the same cluster through a local routing network. The number of output pins of a cluster are equal to the total number of BLEs in a cluster (with each BLE having a single output). Whereas the number of input pins of a cluster can be less than or equal to the sum of input pins required by all the BLEs in the cluster. Modern FPGAs contain typically 4 to 10 BLEs in a single cluster.

2.1.2 Routing Network

The routing network of an FPGA occupies 80-90% of FPGA chip area, whereas the logic area occupies only 10-20% area [Betz et al., 1999]. The flexibility of an FPGA is mainly dependent on its programmable routing network. A mesh-based FPGA routing network consists of horizontal and vertical routing tracks which are interconnected through switch boxes (SB). Logic blocks are connected to the routing network through connection boxes (CB). The flexibility of a connection block (Fc) is the number of routing tracks of adjacent channel which are connected to the pin of a block. The connectivity of input pins of logic blocks with the adjacent routing channel is called as Fc(in); the connectivity of output pins of the logic blocks with the adjacent routing channel is called as Fc(out). An Fc(in) equal to 1.0 means that all the tracks of adjacent routing channel are connected to the input pin of the logic block. An Fc(in) equal to 0.5 means that only 50% tracks of the adjacent routing channel are connected to the input pin.

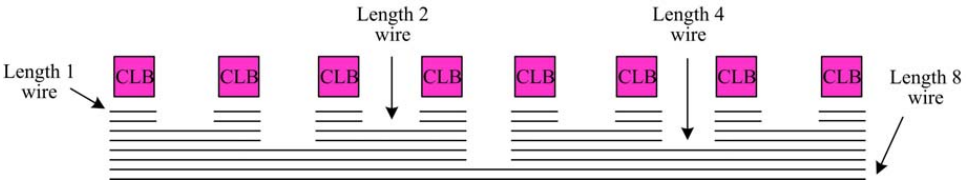


Figure 2.5: Channel segment distribution

The flexibility of switch box (Fs) is the total number of tracks with which every track entering in the switch box connects to. The routing tracks connected through a switch box can be bidirectional or unidirectional (also called as directional) tracks. Figure 2.4 shows a bidirectional and a unidirectional switch box having Fs equal to 3. The input tracks (or wires) in both these switch boxes connects to 3 other tracks of the same switch box. The only limitation of unidirectional switch box is that their routing channel width must be in multiples of 2. Multi-length wires are created to reduce area and delay. Figure 2.5 shows an example of different length wires. Longer wire segments span multiple blocks and require fewer switches, thereby reducing routing area and delay. However, they also decrease routing flexibility, which reduces the probability to route a hardware circuit successfully. Modern commercial FPGAs commonly use a combination of long and short wires to balance flexibility, area and delay of the routing network .

Generally, the output pins of a block can connect to any routing track through pass transistors. Each pass transistor forms a tristate output that can be independently turned on or off. However, single-driver wiring technique can also be used to connect output pins of a block to the adjacent routing tracks. For single-driver wiring, tristate elements cannot be used; the output of block needs to be connected to the neighboring routing network through multiplexors in the switch box. Modern commercial FPGA architectures have moved towards using single-driver, directional routing tracks. [G.Lemieux et al., 2004] show that if single-driver directional wiring is used instead of bidirectional wiring, 25% improvement in area, 9% in delay and 32% in area-delay can be achieved. All these advantages are achieved without making any major changes in the FPGA CAD flow.

2.1.3 Software Flow

One of the major research aspects of FPGAs is the development of software flow required to map hardware applications on an FPGA. The effectiveness and quality of an FPGA is largely dependent on the software flow provided with an FPGA. The software flow takes an application design description in a Hardware Description Language (HDL) and converts it to a stream of bits that is eventually programmed on the FPGA. Figure 2.6 shows a complete software flow for programming an application circuit on a mesh-based FPGA. A brief description of various modules of software flow is described below.

Logic synthesis :

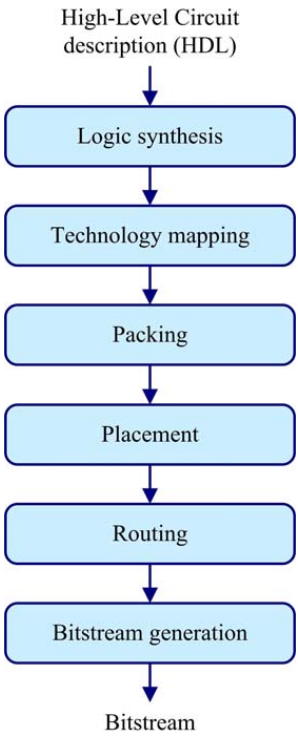


Figure 2.6: FPGA software configuration flow

Logic synthesis [Brayton and McMullen, 1982] [Brayton et al., 1990] transforms an HDL description (VHDL or Verilog) into a set of boolean gates and Flip-Flops. The synthesis tools transform the register-transfer-language (RTL) description of a design into a hierarchical boolean network. Various technology-independent techniques are applied to optimize the boolean network. The typical cost function of technology-independent optimizations is the total literal count of the factored representation of the logic function. The literal count correlates very well with the circuit area. Further details of logic synthesis are beyond the scope of this work.

Technology mapping :

After logic synthesis, technology-dependent optimizations are performed. These optimizations transform the technology-independent boolean network into a network of gates in the given technology library. The technology mapping for FPGAs transforms the given boolean network to the available set of blocks on an FPGA. For a traditional FPGA architecture, the boolean network is transformed into Look-Up Tables and Flip-Flops. Technology mapping algorithms optimize a given boolean network for a set of different objective functions including depth, area and power. The FlowMap algorithm [J.Cong and Y.Ding, 1994a] is a

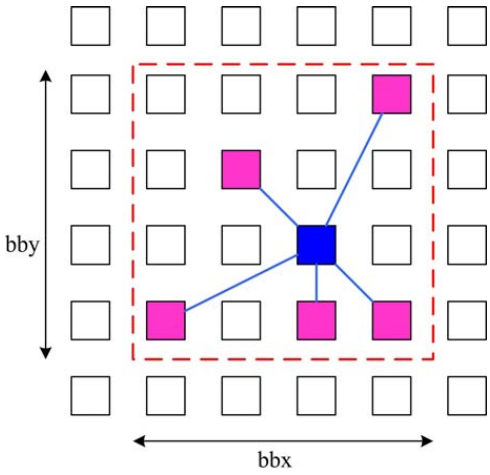


Figure 2.7: Bounding box of a hypothetical 6-terminal net [Betz et al., 1999]

widely used academic tool for FPGA technology mapping. FlowMap is able to find a depth-optimal solution in polynomial time. Later versions of FlowMap are further improved to optimize area and runtime of a boolean network while maintaining the same depth [J.Cong and Y.Ding, 1994b] [J.Cong and Y.Hwang, 1995] [J.Cong and Y.Ding, 2000]. The final output of FPGA technology mapping is a network of I/Os, LUTs and Flip-Flops.

Packing :

A mesh-based FPGA consists of an array of Configurable Logic Blocks (CLBs). Each CLB consists of a cluster of Basic Logic Elements (BLEs). A BLE consists of a Look-Up Table and a Flip-Flop. The packing phase, also called as clustering phase groups a Look-Up Table and a Flip-Flop in a BLE, and groups different BLEs into a cluster of BLEs. These BLEs or clusters of BLEs can then be directly mapped on the CLBs of an FPGA. The main optimization goal is to cluster the Look-Up Tables, Flip-Flops and BLEs in such a way that inter-cluster communication is minimized. Less inter-cluster communication ensures less routing resource utilization in an FPGA. The final output of packing is a network of I/Os and CLBs.

Placement :

The placement algorithm determines the position of CLB and I/O instances in a packed netlist on the respective CLB and I/O blocks on the FPGA architecture. The main goal of placement algorithm is to place connected blocks near each other so that minimum routing resources are required to route their connections. The placement algorithm can also serve to fulfill other architectural or optimization requirements, such as balancing the wire density across FPGA.

Three major types of commonly used placement algorithms include (i) min-cut (partitioning) based placement algorithm [A.Dunlop and B.Kernighan, 1985] [D.Huang and A.Kahng,

1997] (ii) analytical placement algorithm [G.Sigl et al., 1991] [C.J.Alpert et al., 1997], and (iii) simulated annealing based placement algorithm [S.Kirkpatrick et al., 1983] [C.Sechen and A.Sangiovanni-Vincentelli, 1985]. (i) The partitioning based placement approach is generally suitable for hierarchical FPGA architectures. The partitioner is recursively applied to distribute netlist instances between clusters. The aim is to reduce external communication and merge highly connected instances in the same cluster. (ii) Analytical placement algorithms commonly utilize a quadratic wire length objective function. Although, a quadratic objective is only an indirect measure of the wire length; its main advantage is that it can be minimized very efficiently and is thus suitable for handling large problems. A quadratic function does not give the best possible wire length; it is often followed by some local iterative improvement techniques. (iii) The simulated annealing placement algorithm uses the annealing concept for molten metal which is cooled down gradually to produce high quality metal objects. The simulated annealing algorithm is very effective at finding an acceptably good solution in a limited amount of time. This work concentrates on simulated annealing based placement algorithms.

The simulated annealing placement algorithm is good at approximating an acceptable placement solution for a netlist to be placed on an FPGA. A wire length cost function is used to measure the quality of placement. Netlist instances are initially placed randomly on the FPGA. Different instance moves are made to gradually improve the quality of placement. The “temperature” parameter of the algorithm cools down (decreases) systematically. At each temperature step, move operations are performed “Iteration Count” number of times. “Iteration Count” is proportional to the number of instances in a netlist. The main objective function of placer is to achieve a placement having minimum sum of half-perimeters of the bounding boxes of all the nets. Figure 2.7 shows a bounding box of a hypothetical 6-terminal net. An instance is randomly moved from one position to another; the change in the cost function is computed incrementally. If the cost decreases (improves), the move is always accepted. If the cost increases, the move can still be accepted. The probability of accepting a move that increases the cost is high during the initial phase of the algorithm. But this probability decreases gradually, until in the final phase only those moves are accepted which decrease the cost.

Routing :

Once the instances of a netlist are placed on FPGA, connections between different instances are routed using the available routing resources. The FPGA routing problem consists of routing the signals (or nets) in such a way that no more than one signal use the same routing resource. PathFinder [L.McMurchie and C.Ebeling, 1995] routing algorithm is commonly used for FPGAs. In order to perform routing on an FPGA architecture, the routing architecture is initially modeled as a directed graph where different nodes are connected through edges. Each routing wire of the architecture is represented by a node, and connection between two wires is represented by an edge. Figure 2.8 represents a small portion of routing architecture in the form of a directed graph. When a netlist is routed on the FPGA routing graph, each net (i.e connection of a driver instance with its receiver instances) is routed using a congestion driven Dijkstra’s "Shortest Path" algorithm [T.Cormen et al., 1990]. Once all nets in a netlist

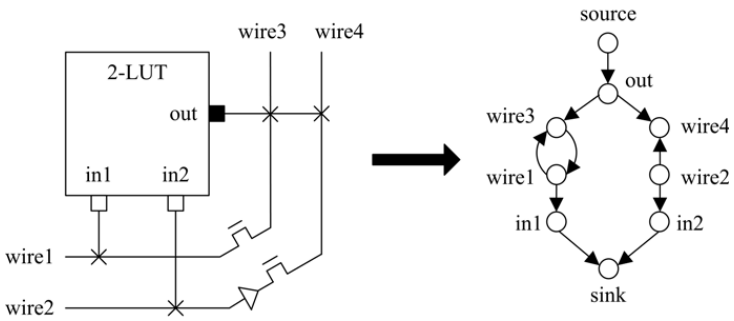


Figure 2.8: Modeling FPGA architecture as a directed graph [Betz et al., 1999]

are routed, one routing iteration is said to be completed. At the end of an iteration, there can be conflicts between different nets sharing the same nodes. The congestion parameters of the nodes are updated, and routing iterations are repeated until routing converges to a feasible solution (i.e all conflicts are resolved) or routing fails (i.e. maximum iteration count has reached, and few routing conflicts remain unresolved).

Bitstream generation :

Once a netlist is placed and routed on an FPGA, bitstream information is generated for the netlist. This bitstream is programmed on the FPGA using a bitstream loader. The bitstream of a netlist contains information as to which SRAM bit of an FPGA be programmed to 0 or to 1. The bitstream generator reads the technology mapping, packing and placement information to program the SRAM bits of Look-Up Tables. The routing information of a netlist is used to correctly program the SRAM bits of connection boxes and switch boxes.

2.2 Research trends in FPGAs

FPGA-based products are very effective for low to medium volume production, they are easy to program and debug, and have less NRE cost and less time-to-market. All these major advantages of an FPGA come through their reconfigurability. However the very same reconfigurability is the major cause of its disadvantages. The flexibility of FPGAs is mainly due to its reprogrammable routing network which takes 80 to 90% of the entire FPGA area [Betz et al., 1999]. The logic area is only 10 to 20% of the FPGA. Due to this reason FPGAs are much larger, slower and more power consuming than ASICs [I.Kuon and J.Rose, 2007]; thus they are unsuitable for high volume production, high performance or low power consumption.

Reconfigurable hardwares and FPGA architectures have many active research domains. A major aspect of research in reconfigurable hardwares revolves around decreasing the drawbacks of FPGAs, with or without compromising upon its major benefits. Following are few of the major tradeoff solutions that have been proposed in solving the area, speed, power and/or volume production problems of FPGAs.

- **Hard-Blocks:** Logic density of an FPGA is improved by incorporating dedicated hard-blocks in an FPGA. Hard-Blocks, or in other words small ASICs, in FPGAs increase their speed and reduce their overall area and power consumption. Hard-blocks can include multipliers, adders, memories, floating-point units etc. In this regard, [Beauchamp et al., 2006] have introduced embedded floating-point units in FPGAs, [C.H.Ho et al., 2006] have developed virtual embedded block methodology to model arbitrary embedded blocks on existing commercial FPGAs. [Hartenstein, 2001] has presented a brief survey of a decade of R&D on coarse grain reconfigurable hardware and their related compilation techniques. Figure 2.10 shows a commercial FPGA architecture that uses embedded hard-blocks.

- **Application Specific FPGAs:** The type of logic blocks and the routing network in an FPGA can be optimized to gain area and performance advantages for a given application domain (controlpath-oriented applications, datapath-oriented applications, etc). These types of FPGAs may include different variety of desired hard-blocks, appropriate amount of flexibility required for the given application domain or bus-based interconnects rather than bit-based interconnects. [Marshall et al., 1999] have presented a reconfigurable arithmetic array for multimedia applications, [Verma and Akoglu, 2007] have presented a coarse grained reconfigurable architecture for variable block size motion estimation and, [Ye and Rose, 2006] have used bus-based connections to improve density of FPGAs for datapath circuits. Figure 2.14 shows a reconfigurable arithmetic array for multimedia applications.

- **FPGA to Structured-ASIC:** The ease of designing and prototyping with FPGAs can be exploited to quickly design a hardware application on an FPGA. Later, improvements in area, speed, power and volume production can be achieved by migrating the application design from FPGA to other technologies such as Structured-ASICs. In this regard, Alter provides a facility to migrate its Stratix IV based application design to HardCopy IV [HardCopy, IV]. The eASIC Nextreme [eASIC, 2010] uses an FPGA-like design flow to map an application design on SRAM programmable LUTs, which are later interconnected through mask programming of few upper routing layers. cASIC [Compton and Hauck, 2007] explores the design space between ASIC and FPGA; configurable ASIC cores are designed to execute a given set of application designs at exclusive times. Tierlogic [TIERLOGIC, 2010] is a recently launched FPGA vendor that offers 3D SRAM-based TierFPGA devices for prototyping and early production. The same design solution can be frozen to a TierASIC device with one low-NRE custom mask for error-free transition to an ASIC implementation. The SRAM layer is placed on an upper 3D layer of TierFPGA. Once the TierFPGA design is frozen, the bitstream information is used to create a single custom mask metal layer that will replace the SRAM programming layer.

- **FPGA with processors:** A considerable amount of FPGA area can be saved by implementing the control path portion of a circuit on a microprocessor, and only the compute intensive datapath portion of a circuit is implemented on FPGAs. An FPGA is connected to a microprocessor in different ways (i) A soft processor is implemented on

FPGA reconfigurable resources (like shown in [NIOS, II], [MicroBlaze, 2010] and [Peter Yiannacouras and Rose, 2007]), (ii) a processor is incorporated in an FPGA as a dedicated hard-block (like AVR Processor integrated in Atmel FPSLIC [ATMEL, 2010] or PowerPC processors embedded in Xilinx Virtex-4 [Xilinx, 2010]), or (iii) an FPGA is attached with the pipeline of a processor to execute customized hardware instructions (like [Callahan et al., 2000] and [Jones et al., 2005]). Figure 2.13 illustrates a VLIW processor that supports application-specific hardware instructions.

- **Time-multiplexed signals:** Instead of using a dedicated routing track for routing a single signal, a routing wire is time-multiplexed and used by different signals at different times [Kapre et al., 2006] [Essen et al., 2009]. In this way, considerable amount of routing resources can be reduced to achieve area gains. Time multiplexing is handled by adding special hardware circuitry. These extra resources make time-multiplexing less attractive for commercial FPGA architectures where generally single-bit routing wires are used. However, these extra resources can be amortized across word-wide routing resources in coarse-grained reconfigurable arrays.

- **Time-multiplexed FPGAs:** The capacity or logic density of FPGAs is increased by executing different portions of a circuit on an FPGA in a time multiplexing mode [Trimberger et al., 1997] [Miyamoto and Ohmi, 2008]. An application design is divided into different sub-circuits, and each sub-circuit runs as a individual context of FPGA. The state information of each sub-circuit is saved in context registers before a new context runs on FPGA. Tabula [Tabula, 2010] is a recently launched FPGA vendor that provides time-multiplexed FPGAs.

The following section discusses different case-studies employing one of the above techniques.

2.2.1 Versatile Packing, Placement and Routing, VPR

Versatile Packing, Placement and Routing for FPGAs (commonly known as VPR) [V.Betz and J.Rose, 1997] [Betz et al., 1999] [A.Marquart et al., 1999] is the most widely used academic mesh-based FPGA exploration environment. It allows to explore mesh-based FPGA architectures by employing an empirical approach. Benchmark circuits are technology mapped, placed and routed on a desired FPGA architecture. Later, area and delay of FPGAs are measured to determine best architectural parameters. Different CAD tools in VPR are highly optimized to ensure high quality results; as poor CAD tools may lead to inaccurate architectural conclusions. Area and delay models are sufficiently accurate to compare the effect of different architectural changes.

The GILES project (Good Instant Layout of Erasable Semiconductors) [Padalia et al., 2003] [Kuon et al., 2005] generates the physical layout of an FPGA from the architecture specifications given as input to VPR. Developing a new FPGA is a challenging and time-consuming task. [Padalia et al., 2003] report that the creation of new FPGA requires 50 to 200 person years;

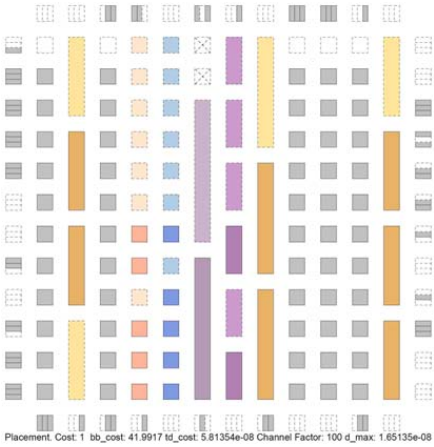


Figure 2.9: A Heterogeneous FPGA in VPR 5.0 [Luu et al., 2009]

thus increasing the overall time-to-market. It is thus an interesting option to significantly reduce the time-to-market of FPGAs at the expense of limited area penalty. GILES automatically generates a transistor-level schematic of an FPGA tile from a high-level architectural specification. The output of GILES is a mask-level layout of a single tile that can be replicated to form an FPGA array.

The latest version of VPR known as VPR 5.0 [Luu et al., 2009] supports hard-blocks (such as multiplier and memory blocks) and single-driver routing wires. Hard-blocks are restricted to be in one grid width column, and that column can be composed of same type of blocks. The height of a block must be an integral number of grid units. In case a block height is indivisible with the height of FPGA, some grid locations are left empty. Figure 2.9 illustrates a heterogeneous FPGA with 8 different kinds of blocks. VPR 5.0 also provides optimized electrical models for a wide range of architectures for different process technologies.

2.2.2 Madeo, a framework for exploring reconfigurable architectures

Madeo [Lagadec, 2000] is a design suite for the exploration of reconfigurable architectures. It includes a modeling environment that supports multi-grained, heterogeneous architectures with irregular topologies. Madeo framework initially allows to model an FPGA architecture. The architecture characteristics are represented as a common abstract model. Once the architecture is defined, the CAD tools of Madeo can be used to map a target netlist on the architecture. Madeo embeds placement and routing algorithms (the same as used by VPR [V.Betz and J.Rose, 1997]), a bitstream generator, a netlist simulator, and a physical layout generator. Madeo supports architectural prospection and very fast FPGA prototyping. Several FPGAs, including some commercial architectures (such as Xilinx Virtex family) and prospective ones (such as STMicro LPPGA) have been modeled using Madeo. The physical layout is produced as VHDL description.

2.2.3 Altera Architecture

Altera's Stratix IV [Stratix, IV] is a mesh-based FPGA architecture family fabricated in 40-nm process technology. Figure 2.10 shows the global architectural layout of Statix IV. The logic structure consists of LABs (Logic Array Blocks), memory blocks and digital signal processing (DSP) blocks. LABS are distributed symmetrically in rows and columns and are used to implement general purpose logic. The DSP blocks are used to implement full-precision multipliers of different granularities. The memory blocks and DSP blocks are placed in columns at equal distance with one another. Input and Output (I/Os) are located along the periphery of the device.

Logic array blocks (LABs) and adaptive logic modules (ALMs) are the basic building blocks of the Stratix VI device. They can be used to configure logic functions, arithmetic functions, and register functions. Each LAB consists of ten ALMs, carry chains, arithmetic chains, LAB control signals, local interconnect, and register chain connection lines. The internal LAB structure is shown in Figure 2.11. The local interconnect connects the ALMs belonging to the same LAB. The direct link allows a LAB to drive into the local interconnect of its left or right neighboring LAB. The register chain connects the output of ALM register to the adjacent ALM register in the LAB. A memory LAB (MLAB) is a derivative of LAB which can be either used just like a simple LAB, or as a static random access memory (SRAM). Each ALM in an MLAB can be configured as a 64x1, or 32x2 blocks, resulting in a configuration of 64x10 or 32x20 simple dual-port SRAM block. MLAB and LAB blocks always coexist as pairs in Stratix IV families.

The DSP blocks in Stratix IV are optimized to support signal processing applications such as Finite Impulse Response (FIR), Infinite Impulse Response (IIR), Fast Fourier Transform functions (FFT) and encoders etc. Stratix IV device has two to seven columns of DSP blocks that can implement multiplication, multiply-add, multiply-accumulate (MAC) and dynamic arithmetic or logical shift functions. The DSP block supports 9x9, 12x12, 18x18 and 36x36 multiplication operations. The Statix IV devices contain three different sizes of embedded SRAMs. The memory sizes include 640-bit memory logic array blocks (MLABs), 9-Kbit M9K blocks, and 144-Kbit M144K blocks. The MLABs have been optimized to implement filter delay lines, small FIFO buffers, and shift registers. M9K blocks can be used for general purpose memory applications, and M144K are generally meant to store code for a processor, packet buffering or video frame buffering.

2.2.4 Altera HardCopy

Altera gives provision to migrate FPGA-based applications to Structured-ASIC. Their Structured-ASIC is called as HardCopy [HardCopy, IV]. The main theme is to design, test and even initially ship a design using an FPGA. Later, the application circuit that is mapped on the FPGA can be seamlessly migrated to HardCopy for high volume production. Their latest HardCopy-IV devices offer pin-to-pin compatibility with the Stratix IV prototype, making

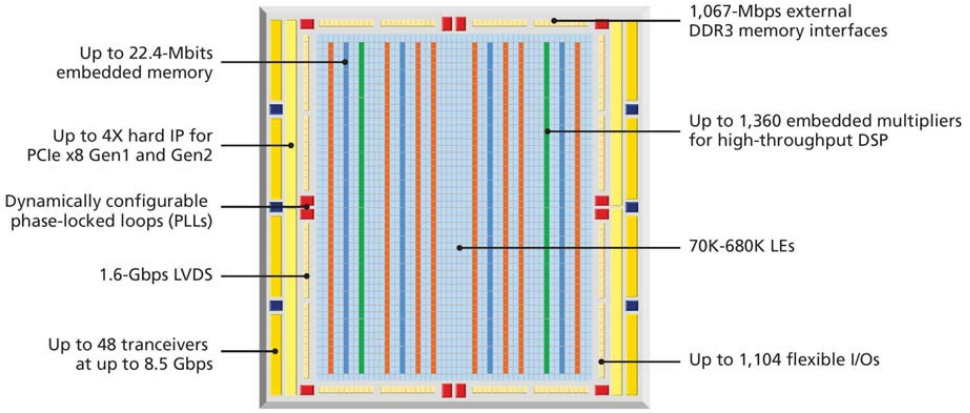


Figure 2.10: Stratix IV architectural elements

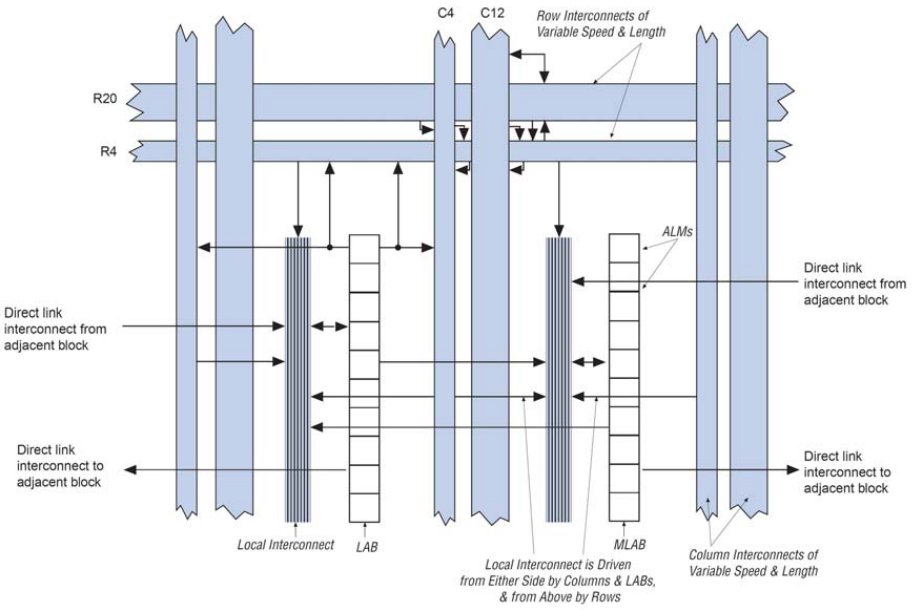


Figure 2.11: Stratix IV LAB Structure

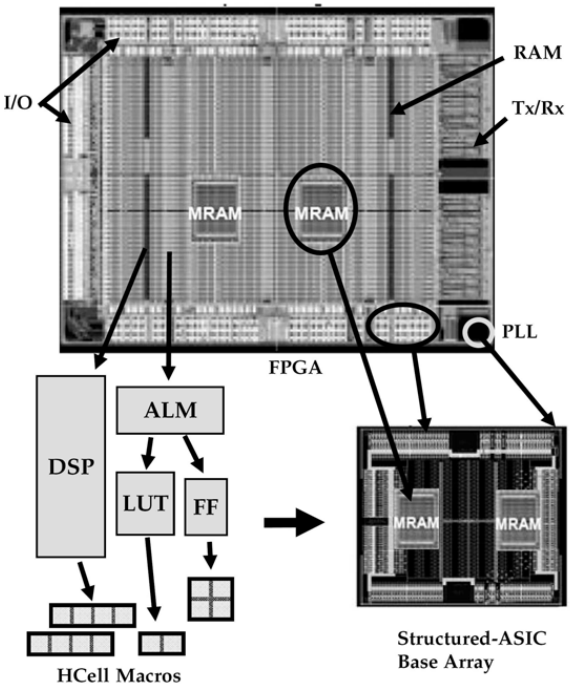


Figure 2.12: FPGA/Structured-ASIC (HardCopy) Correspondence [Hutton et al., 2006]

them drop-in replacements for the FPGAs. Thus, the same system board and softwares developed for prototyping and field trials can be retained, enabling the lowest risk and fastest time-to-market for high-volume production. Moreover, when an application circuit is migrated from Stratix IV FPGA prototype to Hardcopy-VI, the core logic performance doubles and power consumption reduces by 50%.

The basic logic elements in an SRAM-based FPGA comprise of LUTs and Flip-Flops. The logic functionality is implemented on these LUTs which are optionally registered. On the other hand, the basic logic unit of HardCopy is termed as HCell. It is similar to FPGA logic cell (LAB) in the sense that the fabric consists of a regular pattern which is formed by tiling one or more basic cells in a two dimensional array. The difference is that HCell has no configuration overhead. Different HCell candidates can be used, ranging from fine-grained NAND gates to multiplexors and coarse-grained LUTs. An array of such HCells candidates, and a general purpose routing network which interconnects them is laid down on the lower layers of the chip. Specific layers are then reserved to form via connections or metal lines which are used to customize the generic array into specific functionality. Figure 2.12 illustrates the correspondence between an FPGA floorplan and a compatible structured ASIC base array. There is a one to one layout-level correspondence between MRAMs, phase-lock loops (PLLs), embedded memories, transceivers, and I/O blocks. The soft-logic DSP multipliers

and logic cell fabric of the FPGA are re-synthesized to structured ASIC fabric (HCells). However, they remain functionally and electrically equivalent in FPGAs and HardCopy ASICs.

2.2.5 Configurable ASIC Cores (cASIC)

Configurable ASIC Core (cASIC) [Compton and Hauck, 2007] is a reconfigurable device that can implement a limited set of circuits which operate at mutually exclusive times. cASICs are intended as accelerator in domain-specific systems-on-a-chip, and are not designed to replace the entire ASIC-only chip. The host would execute software code, whereas compute-intensive sections can be offloaded to one or more cASICs. For that reason, cASICs implement only data-path circuits and thus supports full-word blocks only (such as 16-bit wide multipliers, adders, RAMS, etc). Since the set of circuits supported by a cASIC are limited, cASICs are significantly smaller than an FPGA implementation. As hardware resources are shared between different netlists, cASICs are even smaller than the sum of the standard-cell based ASIC areas of individual circuits.

Automatic generation of cASIC cores occurs in two phases. The logic phase determines the computation needs of the given set of application netlists. Different computational components are generated which may include ALUs, RAMs, multipliers, registers, etc. These logic resources are shared by all the application netlists. The logic components are properly ordered along the one-dimensional datapath so that minimum routing resources are required. The routing phase then creates wires and multiplexors to connect logic and I/O components. The objective of routing phase is to minimize area by sharing wires between different netlists while adding as few multiplexors/demultiplexors as possible. Different heuristic algorithms are used to maximize wire sharing. Experiments show that configurable ASIC hardware is on average 12.3x smaller than an FPGA solution with embedded multiplier, and 2.2x smaller than a standard cell implementation of individual circuits.

2.2.6 FPGA based processors

Considerable amount of FPGA area can be reduced by incorporating a microprocessor in an FPGA. A microprocessor can execute any less compute intensive task, whereas compute-intensive tasks can be executed on an FPGA. Similarly, a microprocessor based application can have huge speed-up gains if an FPGA is attached with it. An FPGA attached with a microprocessor can execute any compute intensive functionality as a customized hardware instruction. These advantages have compelled commercial FPGA vendors to provide microprocessor in their FPGAs so that complete system can be programmed on a single chip. Few vendors have integrated fixed hard processor on their FPGA (like AVR Processor integrated in Atmel FPSLIC [ATMEL, 2010] or PowerPC processors embedded in Xilinx Virtex-4 [Xilinx, 2010]). Others provide soft processor cores which are highly optimized to be mapped on the programmable resources of FPGA. Altera's Nios [NIOS, II] and Xilinx's Microblaze [MicroBlaze, 2010] are soft processor meant for FPGA designs which allow custom hardware instructions. [Peter Yiannacouras and Rose, 2007] have shown that considerable area gains

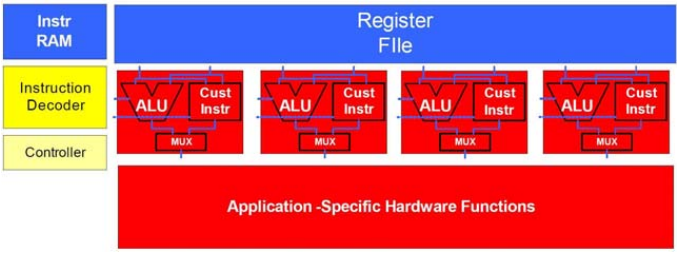


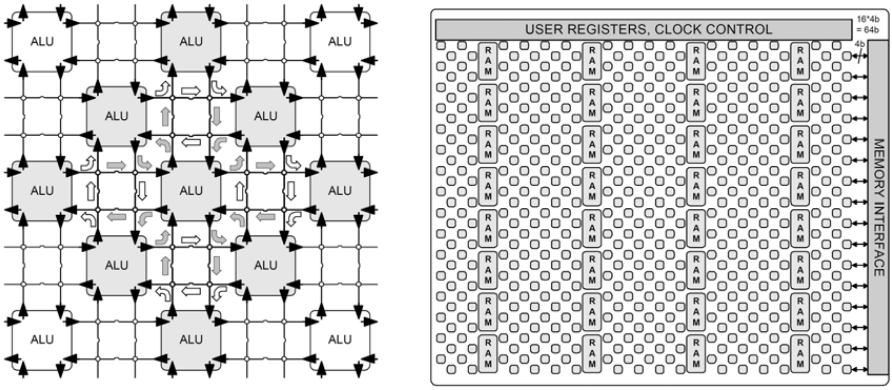
Figure 2.13: A VLIW Processor Architecture with Application Specific Hardware Functions [Jones et al., 2005]

can be achieved if these soft processors for FPGAs are optimized for particular applications. They have shown that unused instructions in a soft processor can be removed and different architectural tradeoffs can be selected to achieve on average 25% area gain for soft processors required for specific applications.

Reconfigurable units can also be attached with microprocessors to achieve execution time speedup in software programs. There is a famous 90/10 rule, which states that 90% of the program's execution time is spent in 10% of the code. So the major aim is to convert the 10% code into hardware logic and implement it as a hardware function to achieve very large gains. Whereas the remaining 90% of the code runs on a microprocessor. [Callahan et al., 2000], [Sima et al., 2001] and [Jones et al., 2005] have incorporated a reconfigurable unit with microprocessors to achieve execution-time speedup.

[Jones et al., 2005] have proposed a VLIW processor which can support application specific customized instructions through a reconfigurable hardware block. VLIW processors have a single instruction controller that dispatches different operations to several functional units that share a single register file. All these functional units execute in parallel. Ideally, many instructions can execute in parallel. But the application must also exhibit high Instruction Level Parallelism (ILP), so that control and data dependencies do not limit the performance improvements. The proposed architecture, illustrated in Figure 2.13, is a 4 way VLIW processor (4 functional units), with hardware resources meant for implementing application specific hardware functions. All the four functional blocks and the hardware functions are linked together through the register file. The hardware block is able to read 16 operands from any of the 32 registers and write back 8 results into any of the registers.

A compilation process is developed for the VLIW processor with hardware functions. The C code to be implemented is profiled to find the computational intensive kernels. Behavioral or high-level synthesis is applied to automatically transform them into combinatorial logic. The remaining code is transformed by the VLIW compiler and assembler, and implemented on the VLIW processor. The VLIW processor and the hardware block are implemented on Altera Stratix II FPGA due to its support for high-speed DSP blocks. The VLIW processor with hardware functions have shown a maximum speedup of 230 times, and an average speedup of 63 times for computational kernels from a set of DSP benchmarks. The overall maximum



(a) CHESS layout and nearest neighbour wiring

(b) A 512 ALU CHESS array with embedded RAMs

Figure 2.14: A Reconfigurable Arithmetic Array for Multimedia Applications

speedup of 30 times and an average speedup of 12 times is shown for DSP benchmarks from MediaBench.

2.2.7 A Reconfigurable Arithmetic Array for Multimedia Applications, CHESS

[Marshall et al., 1999] have proposed a reconfigurable arithmetic array for multimedia applications. Due to its layout, this architecture is also termed as CHESS. The principal goal of CHESS was to increase arithmetic computational density, to enhance the flexibility, and to increase the bandwidth and capacity of internal memories significantly beyond the capabilities of existing commercial FPGAs. These goals were achieved by proposing an array of ALUs with embedded RAMs as shown in Figure 2.14. Each ALU is 4-bit wide and supports 16 instructions. No run time configuration is required, as ALU instruction can be changed dynamically. The CHESS layout of ALUs ensures strong local connection with 8 adjacent ALUs, as shown in Figure 2.14(a). A switch box lies between adjacent ALUs. The switch box contains 64 connections, which can also act as 16W*4 RAMs. Dedicated RAM blocks are placed in columns at equal distance with each other, as shown in Figure 2.14(b). The major advantage of CHESS is that routing network takes only 50% of area.

2.2.8 Reconfigurable Pipelined Data paths, Rapid

Reconfigurable pipelined data paths (known as Rapid) [Ebeling et al., 1996] is a coarse-grained, field programmable architecture for constructing deep computational pipelines. Rapid architecture can efficiently implement applications related to media, signal processing, scientific computing and communications. Rapid consists of a linear array of functional units that are interconnected through a programmable segmented bus network. These

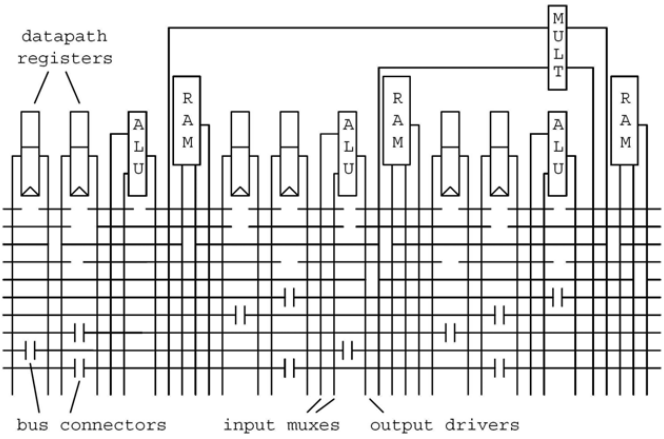


Figure 2.15: A basic cell of RaPiD [Ebeling et al., 1996]

coarse-grained functional units and the bus interconnect are used to implement a data path circuit. The rapid datapath is usually divided into identical units, called as cells, that are replicated to form a complete datapath. A typical rapid cell is shown in Figure 2.15. Each cell can contain hundreds of functional units ranging in complexity from simple general purpose register to multi-output booth-encoded multipliers. All functional units are linearly arranged and are connected to a segmented bus network that runs the entire length of datapath. The functional unit outputs are connected to optional delay units which can be configured to 0 to 3 register delays. This feature allows creation of deep pipelines. The input multiplexors on the segmented bus network are used to give input to functional unit. The Rapid architecture is configured through programming bits that are divided into soft bits and hard bits. The hard bits are the SRAM bits which do not change during the execution of a single application. The soft bits can change after each clock cycle. A pipelined control path generate these soft bits to control the datapath circuit. The control logic achieved through static and dynamic bits substantially reduces the control overhead as compared to FPGA-based and general-purpose processor architectures. Rapid architecture is programmed through Rapid-C [Ebeling, 2002], a C-like language with extensions to specify parallelism, partitioning and data movement. Rapid-C programs may contain several nested loops. Outer loops are transformed into sequential code for address generators, inner loops are mapped on Rapid cells.

2.2.9 Time-Multiplexed Signals

Time-Multiplexed signals can schedule different signals on the same routing wires. [Essen et al., 2009] have analyzed the design tradeoffs involved in static vs time-multiplexed routing for coarse-grained reconfigurable arrays. Unlike commercial FPGA architectures, where routing resources are configured in exactly one way for the entire run of a single

application, time-multiplexing allows routing configuration to be changed on a cycle by cycle basis. Iterating through the schedule of configurations, a scheduled channel changes its communication pattern on each cycle. Time-multiplexing of scheduled signals requires additional configuration memories and some control circuitry. These extra resources make time-multiplexing less attractive for commercial FPGA architectures where generally single-bit routing wires are used. However, when applied to coarse-grained reconfigurable arrays, these extra resources can be amortized across word-wide resources.

The optimal tradeoff between scheduled and static resources depends on the word-width of the interconnect, since the overheads associated with some techniques may be much larger in a 1-bit interconnect than with a 32-bit interconnect. To explore this possibility, the effect of different word-widths is measured on the area and power consumption of the time-multiplexed coarse-grained architectures. It is found that for 32-bit word-wide interconnects, going from 100% statically configured to 100% scheduled (time-multiplexed) channels reduce the channel width to 0.38x the baseline. This in turn reduces the energy to 0.75x, the area to 0.42x, and the area-energy product to 0.32x, despite the additional configuration overhead. This is primarily due to amortizing the overhead of a scheduled channel across a multi-bit signal. It is important to note that as the datapath width is reduced, approaching the single bit granularity of an FPGA, the scheduled channel overhead becomes more costly. It is found that for datapath widths of 24-, 16-, and 8-bit, converting from fully static to fully scheduled reduces area-energy product to 0.34x, 0.36x, and 0.45x, respectively.

Another factor that significantly affects the best ratio of scheduled versus static channels is the maximum degree of time-multiplexing supported by the hardware, i.e. its maximum Initiation Interval (II). Supporting larger II translates into more area and energy overhead for scheduled channels. It is shown that for a 32-bit datapath, supporting an II of 128 is only 1.49x more expensive in area-energy than an II of 16; a fully scheduled interconnect is still a good choice. However, for an 8-bit datapath and a maximum II of 128, 70% static (30% scheduled) achieves the best area-energy performance, and fully static is better than fully scheduled.

2.2.10 Time-Multiplexed FPGA

Time-multiplexed FPGAs increase the capacity of FPGAs by executing different portions of a circuit in a time-multiplexed mode. [Trimberger et al., 1997] have proposed a time-multiplexed FPGA architecture. A large circuit is divided into sub-circuits; each sub-circuit is sequentially executed on a time-multiplexed FPGA. Each sub-circuit runs as a separate context on the FPGA. Such an FPGA stores a set of configuration bits for all contexts. A context is shifted simply by using the SRAM bits dedicated to a particular context. The combinatorial and sequential outputs of a sub-circuit that are required by other sub-circuits are saved in context registers which can be easily accessed by sub-circuits at different times.

Time-Multiplexed FPGAs increase their capacity by actually adding more SRAM bits rather than more CLBs. These FPGAs increase the logic capacity by dynamically reusing the hard-

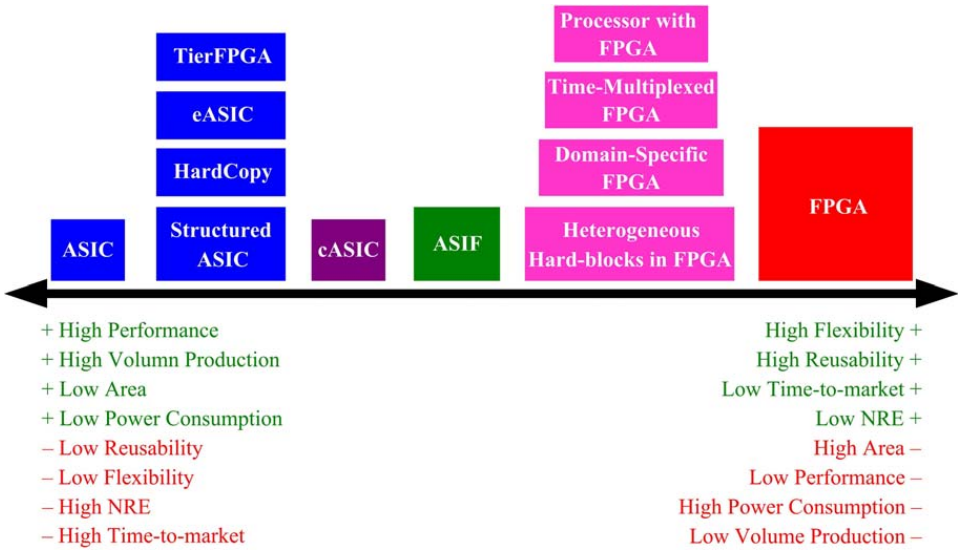


Figure 2.16: Comparison of different solutions used to reduce ASIC and FPGA drawbacks

ware. The configuration bits of only the currently executing context are active, the configuration bits for the remaining supported contexts are inactive. Intermediate results are saved and then shared with the contexts still to be run. Each context takes a micro-cycle time to execute one context. The sum of the micro-cycles of all the contexts makes one user-cycle. The entire time-multiplexed FPGA or its smaller portion can be configured to (i) execute a single design, where each context runs a sub-design, (ii) execute multiple designs in time-multiplexed modes, or (iii) execute statically only one single design.

Tabula [Tabula, 2010] is a recently launched FPGA vendor that provides time-multiplexed FPGAs. It dynamically reconfigures logic, memory, and interconnect at multi-GHz rates with a Spacetime compiler.

2.3 Conclusion

This chapter has initially presented a brief introduction of a traditional FPGA architecture, and related software flow to program hardware designs on the FPGA. It has also described various approaches that have been employed to reduce few disadvantages of FPGAs and ASICs, with or without compromising upon their major benefits. Figure 2.16 presents a rough comparison of different solutions used to reduce the drawbacks of FPGAs and ASICs. The next few chapters of this book will focus on the exploration of FPGA architectures using hard-blocks, application specific Inflexible FPGAs (ASIF), and their automatic layout generation methods.

This work presents a new environment for the exploration of heterogeneous hard-blocks in an FPGA. Hard-blocks are used in commercial FPGA architectures to reduce area, power and performance gaps between FPGAs and ASICs. Specialized hard-blocks, their architectural floor-planning, and specialized routing network can also be used to design domain specific FPGA architectures [Ebeling et al., 1996] [Marshall et al., 1999]. Unlike existing exploration environments [Luu et al., 2009] [Lagadec, 2000], the heterogeneous exploration environment proposed in this work can perform automatic optimization of architecture floor-planning for a given set of application circuits.

Altera [Altera, 2010] has proposed a new idea to prototype, test, and even ship initial few designs on an FPGA, later the FPGA based design can be migrated to Structured-ASIC (known as HardCopy) for high volume production. Other commercial vendors such as eASIC [eASIC, 2010] and TierLogic [TIERLOGIC, 2010] also propose a similar solution. However, migration of an FPGA-based product to Structured-ASIC supports only a single application design. HardCopy, eASIC and TierFPGA totally lose the quality of an FPGA to use the same hardware for executing multiple applications at different times. An ASIF retains this property, and can be a possible future extension for the migration of FPGA-based applications to Structured-ASIC. The concept of an ASIF is similar to a cASIC [Compton and Hauck, 2007], which can execute multiple applications at different times. However unlike the bottom-up insertion technique for the generation of cASIC, the top-down removal technique of an ASIF can be applied to any existing FPGA architecture. Thus when an FPGA-based product is in the final phase of its development cycle, and if the set of circuits to be mapped on the FPGA are known, the FPGA can be reduced to an ASIF for the given set of application designs.

This work also presents automatic layout generation techniques to reduce time-to-market and NRE costs of domain-specific FPGA and ASIF architectures.

http://www.springer.com/978-1-4419-7927-8

Application-Specific Mesh-based Heterogeneous FPGA Architectures

Parvez, H.; Mehrez, H.

2011, XVII, 150 p., Hardcover
ISBN: 978-1-4419-7927-8