Reconfigurable Architectures

11

Mansureh Shahraki Moghaddam, Jae-Min Cho, and Kiyoung Choi

Abstract

Reconfigurable architecture is a computer architecture combining some of the flexibility of software with the high performance of hardware. It has configurable fabric that performs a specific data-dominated task, such as image processing or pattern matching, quickly as a dedicated piece of hardware. Once the task has been executed, the hardware can be adjusted to do some other task. This allows the reconfigurable architecture to provide the flexibility of software with the speed of hardware. This chapter discusses two major streams of reconfigurable architecture: Field-Programmable Gate Array (FPGA) and Coarse Grained Reconfigurable Architecture (CGRA). It gives a brief explanation of the merits and usage of reconfigurable architecture and explains basic FPGA and CGRA architectures. It also explains techniques for mapping applications onto FPGAs and CGRAs.

Acronyms

ALAP	As Late As Possible
ALM	Adaptive Logic Module
ALU	Arithmetic-Logic Unit
ASAP	As Soon as Possible
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
ASMBL	Advanced Silicon Modular Block
CCE	Configuration Cache Element
CDFG	Control-/Data-Flow Graph
CGRA	Coarse Grained Reconfigurable Architecture

M.S. Mansureh (✉) • J.-M. Cho • K. Choi
Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea
e-mail: mansureh@dal.snu.ac.kr; jaemincho@dal.snu.ac.kr; kchoi@dal.snu.ac.kr

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_12

CLB	Configurable Logic Block
DFG	Data-Flow Graph
DPR	Dynamic Partial Reconfiguration
DRAA	Dynamically Reconfigurable ALU Array
DRESC	Dynamically Reconfigurable Embedded System Compiler
DSP	Digital Signal Processor
EGRA	Expression Grained Reconfigurable Array
EMS	Edge Centric Modulo Scheduling
ESL	Electronic System Level
FDS	Force-Directed Scheduling
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GOPS	Giga Operations Per Second
GPP	General-Purpose Processor
HLS	High-Level Synthesis
II	Initiation Interval
ILP	Integer Linear Program
IMS	Iterative Modulo Scheduling
IOE	I/O Element
I/O	Input/Output
LAB	Logic Array Block
LE	Logic Element
LLVM	Low-Level Virtual Machine
LS	List Scheduling
LUT	Look-Up Table
MRRG	Modulo Resource Routing Graph
NoC	Network-on-Chip
NRE	Non-Recurring Engineering
PE	Processing Element
PLL	Phase Locked Loop
QEA	Quantum-inspired Evolutionary Algorithm
RCM	Reconfigurable Computing Module
RF	Register File
RTL	Register Transfer Level
SDF	Synchronous Data Flow
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SPKM	Split & Push Kernel Mapping
SPMD	Single Program, Multiple Data
SPM	Scratchpad Memory
STMD	Single Thread, Multiple Data
VLIW	Very Long Instruction Word
VLSI	Very-Large-Scale Integration

Contents

11.1 Why Reconfigurable Architectures?

General-Purpose Processors (GPPs) are programmable but not good in terms of performance (or execution time) when compared to Application-Specific Integrated Circuits (ASICs). ASICs are specialized circuits providing large amount of parallelism and thus allowing high performance implementation, but only for a specific application. An ASIC can contain just the right mix of functional units for a particular application and thus can be made fast and compact. They can make a very dense chip, which typically translates to high scalability. As technology has improved over the years, the maximum complexity (and hence functionality) possible in an ASIC has grown from several thousand gates to over millions of gates. But ASICS are not an economic choice for many embedded applications due to higher Non-Recurring Engineering (NRE) cost and longer time to market, except for very large volume applications. Reconfigurable computing systems like FPGAs and CGRAs as an intermediate architecture can provide both performance and flexibility. The performance is from the parallelism of the architecture, and the flexibility is from the configurability of the architecture. While FPGAs provide fine-grained (gate level) reconfigurability, CGRAs provide coarse-grained (register transfer level) reconfigurability.

The world of multimedia processing and telecommunication stack is characterized by increasing speed and performance needs. The required raw compute power has been fed by the ever increasing transistor densities enabled by innovations in the Very-Large-Scale Integration (VLSI) domain. However, this growth has to take into account increasing process/voltage/temperature variations, shorter time to market, and higher NRE cost. That is, it is required to achieve both higher

performance and more efficient design/manufacturing at the same time. These two conflicting requirements have made reconfigurable architectures a popular alternative implementation platform.

FPGA is the first successful reconfigurable architecture. The most popular SRAM-based FPGAs contain many programmable logic blocks that can be reprogrammed many times after manufacturing, although some FPGAs such as the one using the antifuse technology can be programmed only once. It can be used as a test bed to prototype a design before going for a final ASIC design. In this way, FPGAs can be reprogrammed as needed until the design is finalized. The ASIC can then be manufactured based on the FPGA design.

Since an FPGA is basically an array of gates, it also provides a large amount of parallelism and thus allows high performance implementation. Actually, the design phases of FPGAs and ASICs are quite similar except that ASICs lack post-silicon flexibility. ASICs require new fabrication for a new application, and thus fail to satisfy the market's critical time-to-market needs, and are, by definition, unable to satisfy the need for greater flexibility [96]. FPGAs are more flexible with the ability to rapidly implement or reprogram the logic. The general flexibility of an FPGA results in time-to-market advantages since it allows fast implementation of new functions as well as easy bug fixes. One thing to note, however, is that an ASIC is designed to be fully optimized to a specific application or a function. Compared to ASICs, FPGAs consume more power, take more area, and provide lower performance but have much lower NRE cost. Thus FPGAs are in general much more cost-effective than ASICs for low production volumes.

The increase of logic in an FPGA has enabled larger and more complex algorithms to be programmed into the FPGA. Furthermore, algorithms can be parallelized and implemented on multiple FPGAs resulting in highly parallel computing. The attachment of such an FPGA to a modern CPU over a high speed bus, like PCI express, has enabled the configurable logic to act more like an accelerator rather than a peripheral. This has brought reconfigurable computing into the high-performance computing sphere. Of course, the use of FPGAs requires creating the hardware design, which is a costly and labor-intensive task, although the vendors typically provide IP cores for common processing functions [13].

The reconfiguration granularity of CGRAs is larger than that of FPGAs. CGRAs typically have an array of simple Processing Elements (PEs), where the PEs are connected with each other through programmable interconnects. The functionality of each PE is also programmable. Compared to FPGAs, CGRAs have significant reduction in area and energy consumption due to much less amount of configuration memory, switches, and interconnects for programming. Furthermore, because of the low overhead of reconfiguration, CGRAs offer dynamic reconfiguration capabilities, which is not easy for FPGAs. That makes CGRAs attractive for area-constrained designs.

Processors are considered to be most flexible in that any kind of application with complicated control and data dependencies can be easily compiled and mapped onto the architecture. However, realizing the flexibility requires a rich set of instructions and the supporting hardware, which incurs a significant overhead in terms of area cost and power consumption. Moreover, with a single GPP, it is difficult to exploit

the parallelism in the application because of the complexity in the architecture. There have been abundant researches and developments to enhance the performance of GPPs by exploiting parallelism; actually, there have been startling progresses in architectures supporting instruction-level parallelism such as superscalar and Very Long Instruction Word (VLIW) architectures. However, it seems no longer possible to make such a progress in that direction due to the rapid growth of area cost and power consumption (area cost or number of transistors is less a concern today, but it still matters in many applications that consider cost, form factor, leakage current, etc.). GPPs, Digital Signal Processors (DSPs), and Application-Specific Instruction-set Processors (ASIPs) (For the details of ASIP, refer to ▸ Chap. 12, "Application-Specific Processors".) belong to this category, although DSPs and ASIPs are in general less flexible than GPPs.

On the other hand, a new architecture has come to importance, named as multi-core or many-core architecture depending on the number of processor cores integrated on a chip. The processor cores are connected by a bus, or by a Network-on-Chip (NoC) when there are too many cores to be connected by a bus. Such an architecture can exploit task-level parallelism through proper scheduling of tasks, while exploiting instruction-level parallelism available in a task if the processor cores are capable of doing it. Although such an architecture can better exploit parallelism with a better scalability, the processor cores are still very expensive, and the on-chip communications incur additional costs in terms of area and power consumption.

Different applications place unique and distinct demands on computing resources, and applications that work well on one processor architecture will not necessarily map well to another; this is true even for different phases of a single application. As yet another architecture, GPUs are inexpensive, commodity parallel devices with huge market penetration. They have already been employed as powerful accelerators for a large number of applications including games and 3D physics simulation. The main advantages of a GPU as an accelerator stem from its high memory bandwidth and a large number of programmable cores with thousands of hardware thread contexts executing programs in a Single Program, Multiple Data (SPMD) (The model of GPUs executing the same kernel code on multiple data is called differently in the literature. Examples other than SPMD include Single Instruction, Multiple Data (SIMD), Single Instruction, Multiple Threads (SIMT), and Single Thread, Multiple Data (STMD).) fashion. GPUs are flexible and relatively easy to program using high-level languages and APIs which abstract away hardware details. Changing functions in GPUs can be done simply via rewriting and recompiling code. However, this flexibility comes at a cost. For the flexibility, GPUs rely on the traditional von Neumann architecture that fetches instructions from memory, although the SPMD model can execute many threads in parallel to process many different data with a single-thread program fetch. Thus, when the application cannot generate many threads having the same program sequence, the architecture may result in waste of resources and inefficiency in terms of area cost and power consumption. Figure 11.1 briefly expresses the positioning of reconfigurable architectures in terms of efficiency versus flexibility compared to other technologies/architectures including ASIC, GPP, and others.

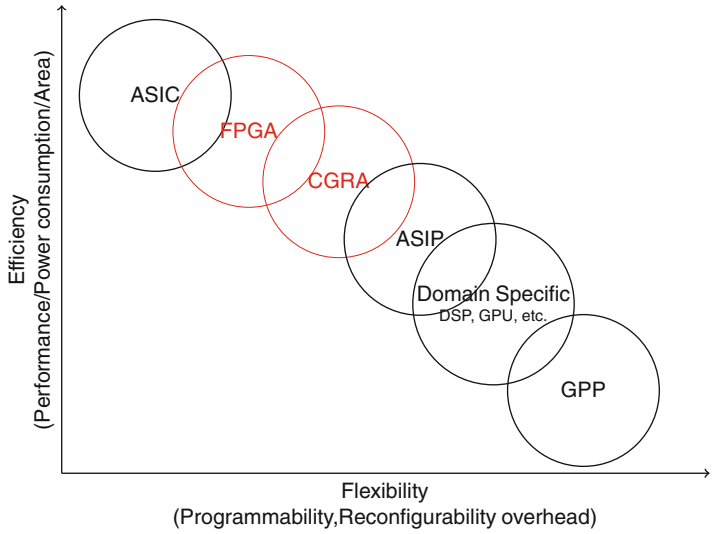

Fig. 11.1 Comparison between implementation platforms [20]

11.2 FPGA Architecture

Field-Programmable Gate Arrays (FPGAs) went far beyond the peripheral position in early days and are now occupying central positions in highly complex systems. Over the 30 years, FPGAs have increased capacity by more than a factor of 10,000 and increased speed by a factor of 100. Cost and energy consumption per unit function have also decreased by more than a factor of 1,000 [105]. An FPGA device provides millions of logic cells, megabytes of block memory, thousands of DSP blocks, and gigahertz of clock speed [72]. FPGAs are getting more complex with the advances in semiconductor technology and are now found in various systems, such as network, television, automobiles, etc., due to their merits compared to other state-of-the-art architectures. For example, an Altera Arria 10 FPGA [2] with DSP blocks that support both fixed and floating-point arithmetic can perform up to 1 TFLOPS [57]. An FPGA platform with four Virtex-5 FPGAs [114] offers performance comparable to a CPU or a GPU with 2.7–293 times better energy efficiency on the BLAS benchmark [46].

Also, since FPGAs have the capability of reconfiguration, multiple applications can be implemented on a small device, and thus the gap between FPGAs and ASIC designs in terms of area and power can be reduced [28]. FPGA has also established a large area of research on self-adaptive hardware systems. Self-adaptive hardware systems are capable of exchanging, updating, or extending hardware functionality during run time.

11.2.1 Building Blocks

FPGAs contain a large amount of logic elements and interconnects among them. There are also interconnects for clock distribution. The bitstream of configuration data is downloaded into an FPGA to configure the logic elements and the interconnects to implement a required behavior on the FPGA.

11.2.1.1 Logic Elements

The types of Logic Elements (LEs) are different depending on the manufacturing companies. Most common LE types are registers, Look-Up Table (LUT), block RAMs, and DSP units. Modern FPGAs combine such logic elements into a customized building block for architectural scalability. For example, the LUTs in Xilinx 7 series FPGA [113] can be configured as either a 6-input LUT with one output or two 5-input LUTs with separate outputs but common addresses or logic inputs. Each 5-input LUT output can optionally be registered in a flip-flop. Figure 11.2 shows a simplified diagram of a sub-block (dotted box) consisting of an LUT, two flip-flops, and several multiplexers. The logic circuit from Cin to Cout in the diagram is used to build a carry chain for an efficient implementation of an adder/subtracter. Four copies of such a sub-block form a slice (dashed box); thus a slice contains four LUTs and eight flip-flops in total together with multiplexers and carry logic. Among the eight flip-flops in a slice, four (one per LUT) can optionally be configured as latches. Two slices form a Configurable Logic Block (CLB); each slice in a CLB is connected to a switch matrix as shown in Fig. 11.3. One of the reasons for this specific CLB structure, which is common to Spartan-6 and Virtex-6, is to simplify design migration from the Spartan-6 and Virtex-6 families to the 7 series devices [115].

As shown in Fig. 11.4, the CLBs are arranged in columns in the 7 series FPGAs (see Fig. 11.3) to form the Advanced Silicon Modular Block (ASMBL) architecture, which uses flip-chip packaging to place pins anywhere (not only along the periphery). With the architecture, the number of I/O pins can be increased arbitrarily without increasing the array size as shown in Fig. 11.5 [112]. It also enhances on-chip power and ground distribution by allowing power and ground lines to be placed at proper locations in the chip as shown in Fig. 11.6. The ASMBL architecture enables an FPGA platform to optimize the mixture of resource columns to an application domain. In Fig. 11.4, for example, applications in domain A require lots of logic, some memory blocks, and small number of DSP blocks, and thus they fit well with platform A since it has a mixture of columns optimized to such applications. Each CLB block can be configured as a look-up table, distributed RAM, or a shift register.

Altera Stratix V FPGA [4] devices use a building block called enhanced Adaptive Logic Module (ALM) to implement logic functions more efficiently. The enhanced ALM has a fracturable LUT with eight inputs, two dedicated embedded adders, and four dedicated registers as shown in Fig. 11.7. The ALM in Stratix V packs 6% more

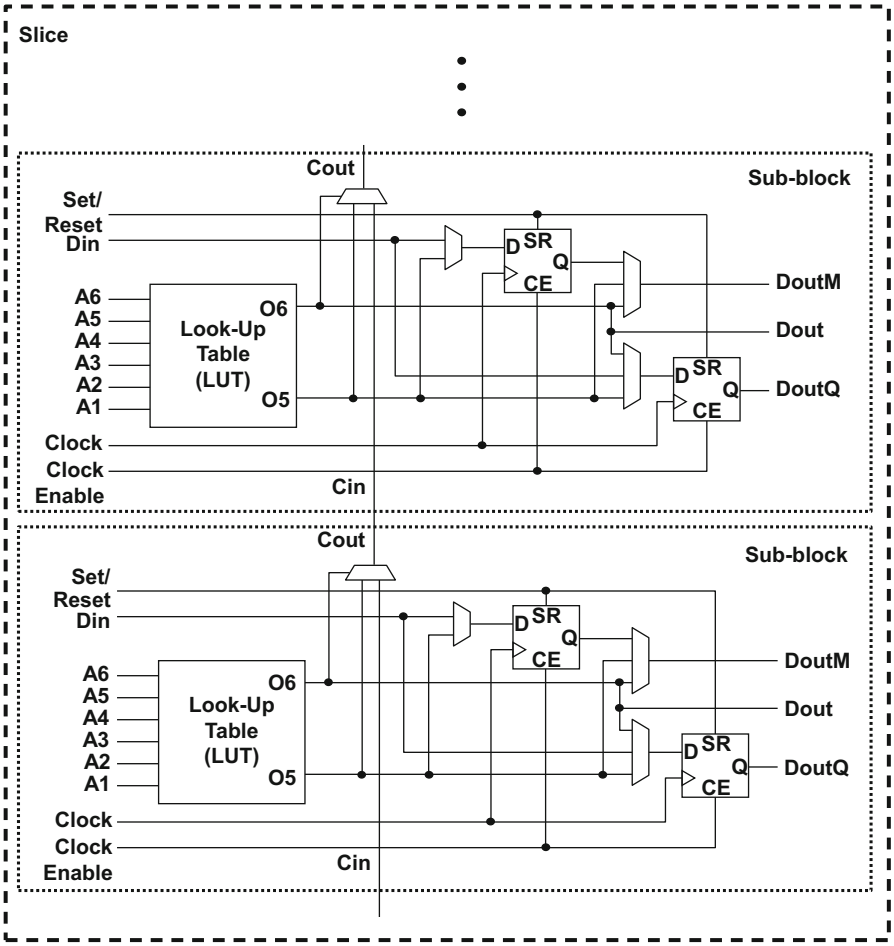

Fig. 11.2 Sub-blocks in a slice

logic compared to the previous-generation ALM found in Stratix IV devices. An ALM can implement some 7-input LUT-based function, any 6-input logic function, two independent functions with a smaller-sized LUT (such as two independent 4-input LUT-based functions), and two independent functions that share some inputs as shown in Fig. 11.8; this is the reason for calling LUT as a fracturable LUT. This enables Stratix V devices to maximize core performance at higher core logic utilization and provide easier timing closure for register-rich and heavily pipelined designs.

11.2.1.2 Interconnects

In Xilinx UltraScale architecture [72], extra connectivity (bypass connections shown in Fig. 11.9) eliminates the need to route through an LUT to gain access to the

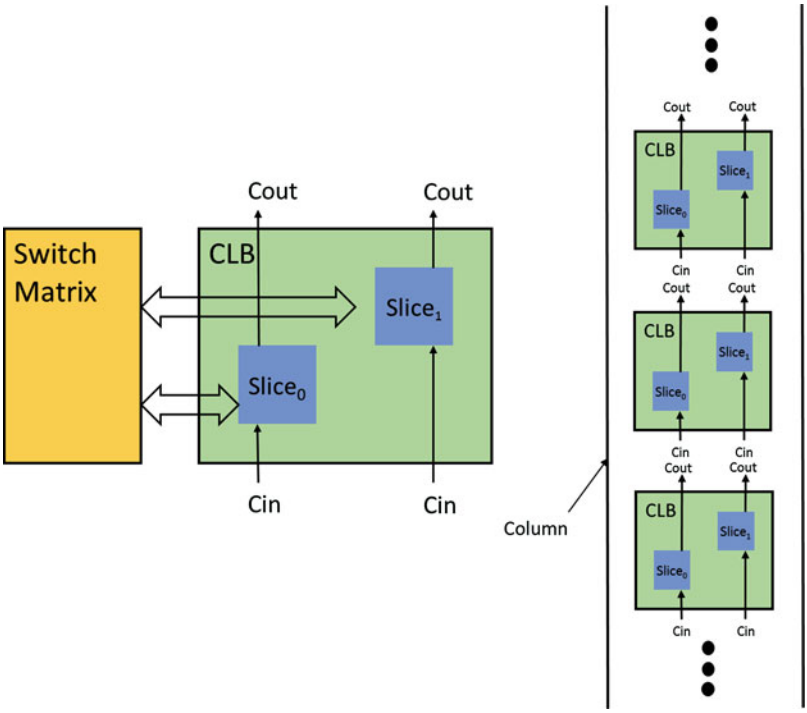

Fig. 11.3 CLB block diagram

associated flip-flops. The flip-flops in the CLB of the architecture benefit from several flexibility enhancements such as inversion attributes. The inversion attributes are used to change the active polarity of each pin. When set to 1, it changes the pin to behave active-low rather than active-high. Having more control signals with increased flexibility provides the software with additional flexibility to use all the resources within each CLB in the architecture.

Old FPGA generations have used central clock spine to distribute the various clocks throughout the FPGA. As a result, clock skew always grows larger when clock sources are away from the center of the device. In Xilinx UltraScale architecture, segmented clock networks allow the center of clock network of a logic block to be placed at the geometric center of the logic block. This technique reduces the clock skew and also improves the performance. The clock segments can also switch on and off when needed. This scheme eliminates unnecessary transistor switchings and reduces the amount of power required to run the on-chip clock networks.

The high-performance Altera Stratix architecture also consists of vertically arranged LEs, memory blocks, DSP blocks, and Phase Locked Loops (PLLs) that are surrounded by I/O Elements (IOEs) as depicted in Fig. 11.10. Speed-optimized interconnects and low-skew clock networks provide connectivity between these

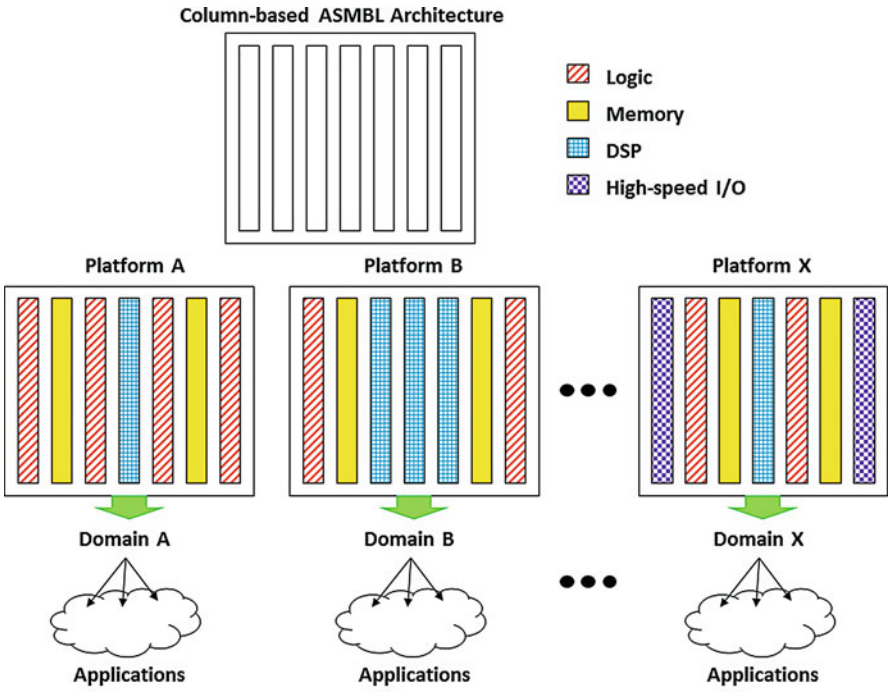

Fig. 11.4 Xilinx ASMBL architecture

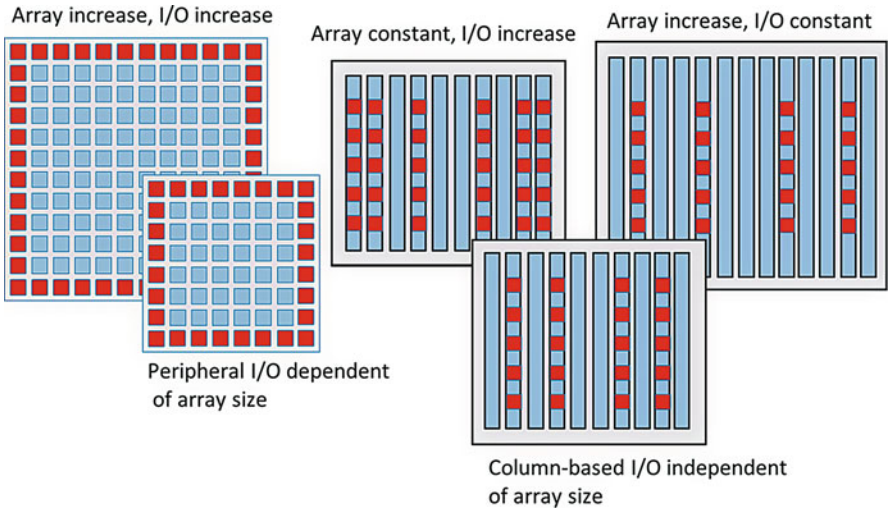

Fig. 11.5 Column-based I/O, enabled by flip-chip packaging technology

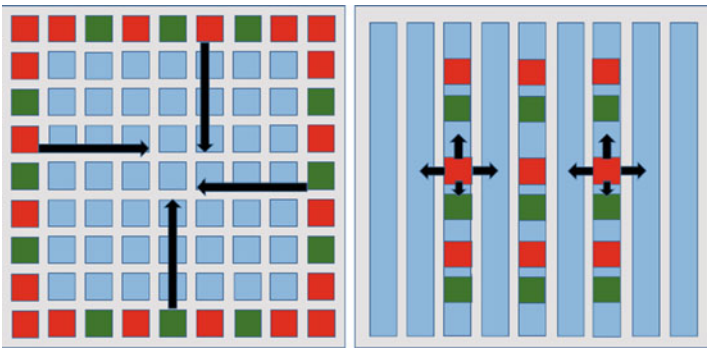

Fig. 11.6 Power and ground distribution in traditional and ASMBL architecture

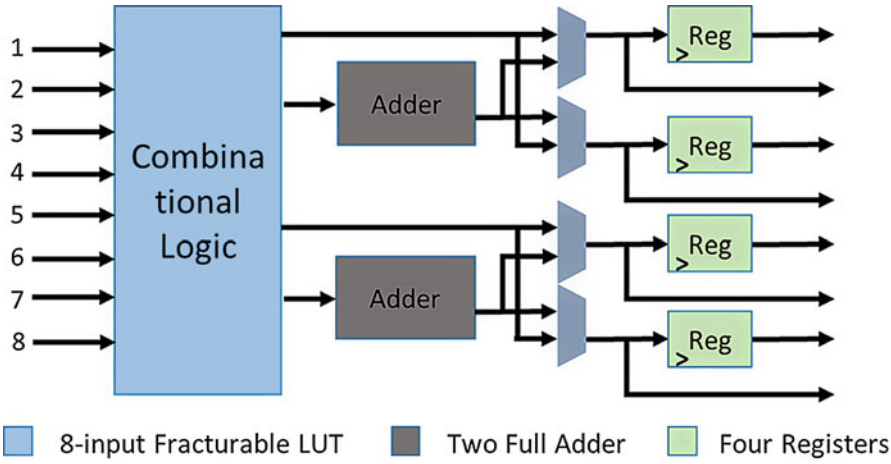

Fig. 11.7 Altera ALM block diagram

structures for data transfer and clock distribution. Stratix FPGAs are based on the MultiTrack interconnect with DirectDrive technology [84]. The MultiTrack interconnect consists of continuous, performance-optimized routing lines of different lengths used for communication within and between distinct design blocks. It also gives more accessibility to any surrounding Logic Array Block (LAB) with much fewer connections, thus improving performance and reducing power. MultiTrack interconnect structure also provides accessing up to 22 clock domains per region. Each Stratix device features up to 16 global clock networks. The DirectDrive technology is a deterministic routing technology, which simplifies the system integration stage of block-based designs by eliminating the often time-consuming system re-optimization process that typically follows design changes and additions.

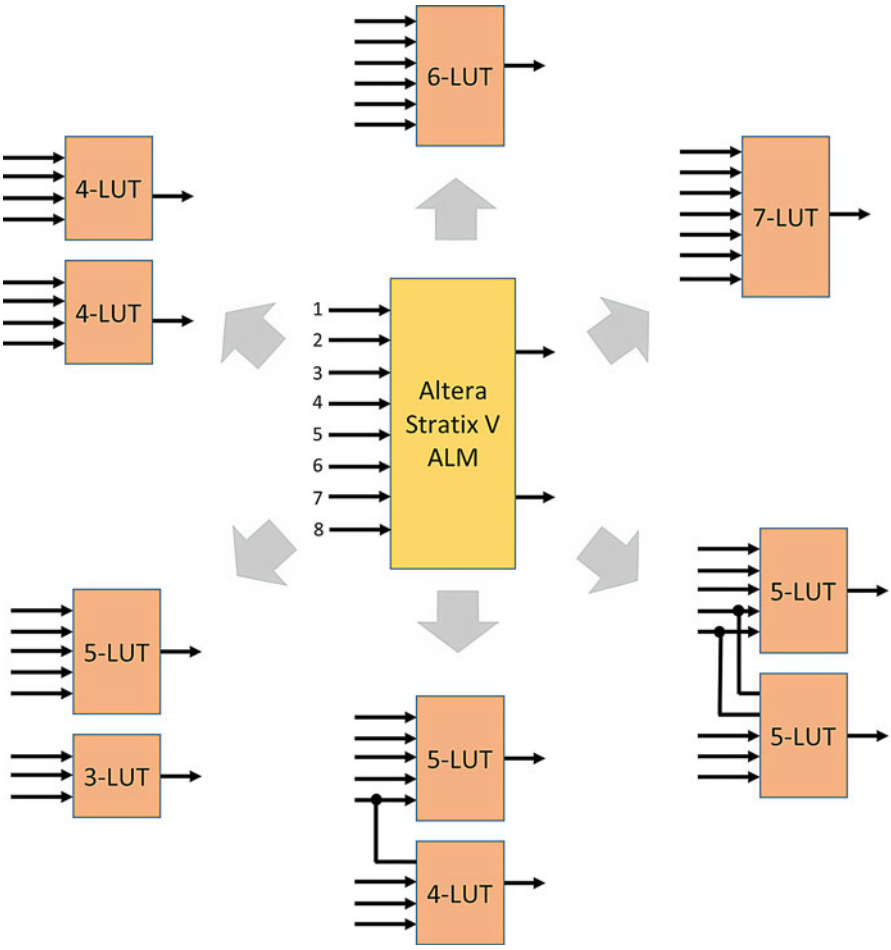

Fig. 11.8 Fracturability of an Altera ALM

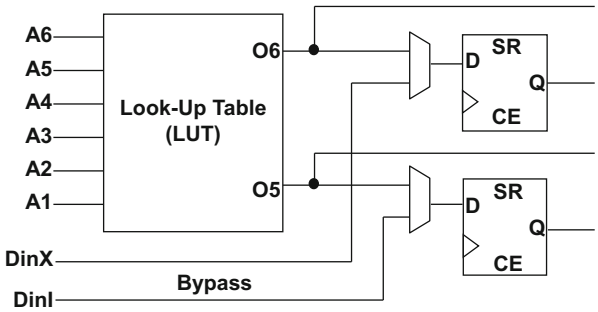

Fig. 11.9 Direct connections to flip-flops bypassing LUT

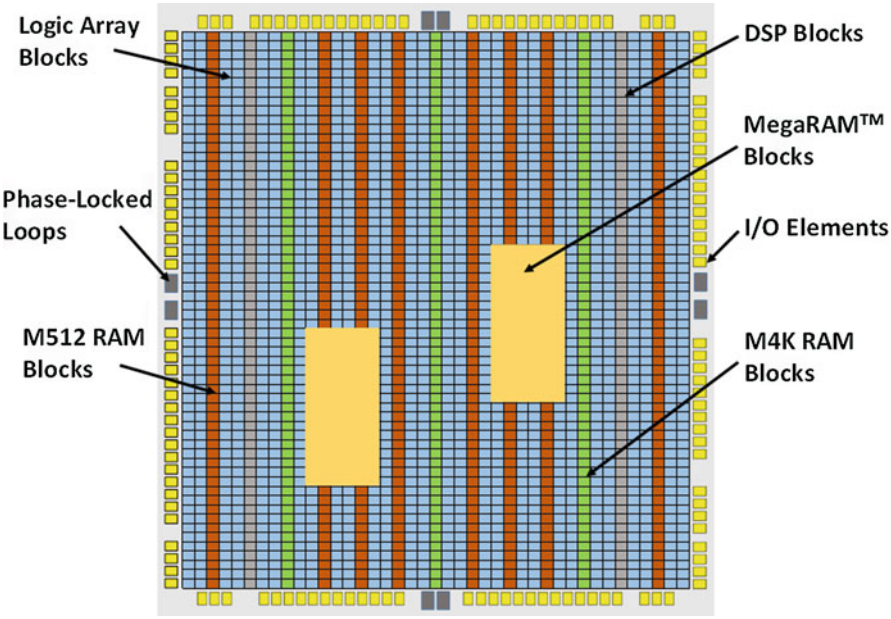

Fig. 11.10 Stratix device architecture

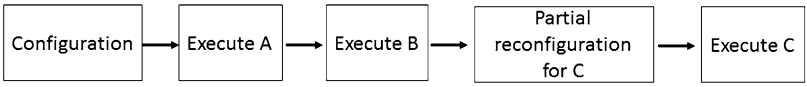

Fig. 11.11 Static partial reconfiguration

11.2.2 Partial Reconfiguration in FPGA

Partial reconfiguration is a feature of modern FPGAs that allows reconfiguration of only a part of the logic fabric of an FPGA. Normally, reconfiguring an FPGA requires it to be held in reset while an external controller reloads a design onto it. Partial reconfiguration allows for critical parts of the design to continue operating while a controller either on the FPGA or off of it loads a partial design into a reconfigurable module. Partial reconfiguration can also be used to save space for multiple designs by only storing the partial designs that change between designs. Partial reconfiguration of FPGAs is a compelling design concept for general purpose reconfigurable systems for its flexibility and extensibility. Partial reconfiguration can be divided into two groups: dynamic partial reconfiguration [68, 97] and static partial reconfiguration.

In static partial reconfiguration, the device is not active during the reconfiguration process. In other words, while the partial data is sent into the FPGA, the rest of the device is stopped and brought up after the configuration is completed, as shown in Fig. 11.11. Dynamic Partial Reconfiguration (DPR), also known as active partial

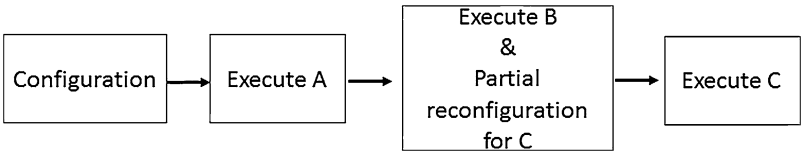

Fig. 11.12 Dynamic partial reconfiguration

reconfiguration, permits to change a part of the device while the rest of an FPGA is still running as illustrated in Fig. 11.12. Nowadays, Xilinx and Altera FPGA vendors support DPR technique in their products [29,69]. The technique can be used to allow the FPGA to adapt to changing hardware algorithms, improve fault tolerance, and achieve better resource utilization. DPR is especially valuable where devices operate in a mission critical environment that cannot be disrupted while some subsystems are being redefined. Placing reconfigurable modules for the partial reconfiguration can be done in different styles such as island, slot, or grid style [28]; depending on the style, a different DPR technique is used. Not all the techniques are supported by FPGA vendors such as Xilinx and Altera, but there are active researches on handling such techniques.

11.2.2.1 Island-Style Reconfiguration

As shown in Fig. 11.13, there are different configuration styles depending on the arrangement of the regions for partial reconfiguration. In the island-style approach, the configurable region is capable of hosting one reconfigurable module exclusively per island. A system might provide multiple islands, but if a module can only run on a specific island, it is called single-island style [54]. If modules can be relocated to different islands, it is called multiple-island style. While the island style can be ideal for systems where only a few modules are swapped, it typically suffers from waste of logic resources due to internal fragmentation in most applications. It happens when modules with different resource requirements exclusively share the same island. For example, if a large module taking a big island is replaced by a smaller one, there will be a waste of logic resources in the reconfigurable region. To alleviate the problem, one can reduce the size of each island, but in that case, a large module may not fit into an island. However, hosting only one module per island makes it simple to determine where to place a module.

11.2.2.2 Slot-Style Reconfiguration

The island-style reconfiguration suffers from a considerable level of internal fragmentation. We can improve this by tiling reconfigurable regions into slots. This results in a one-dimensional slot-style reconfiguration as shown in Fig. 11.13b. In this approach, a module occupies a number of tiles according to its resource requirements, and multiple modules can be hosted simultaneously in a reconfigurable region. Figure 11.13 shows how the tiling influences the spatial packing of modules into a reconfigurable region. In general, however, partial reconfiguration should also consider packing of modules in the time domain. In order to improve the utilization

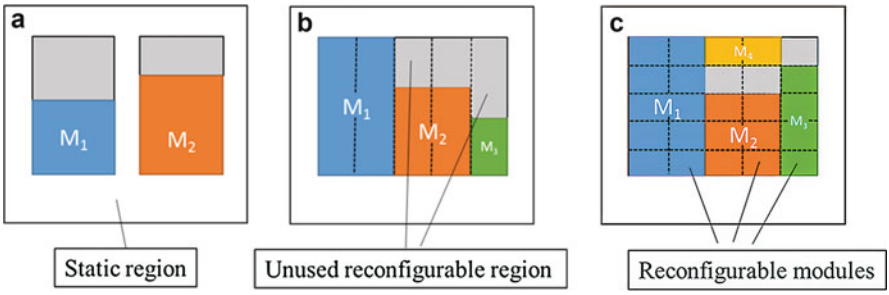

Fig. 11.13 Different placement styles of reconfigurable modules. (**a**) Island style. (**b**) Slot style. (**c**) Grid style

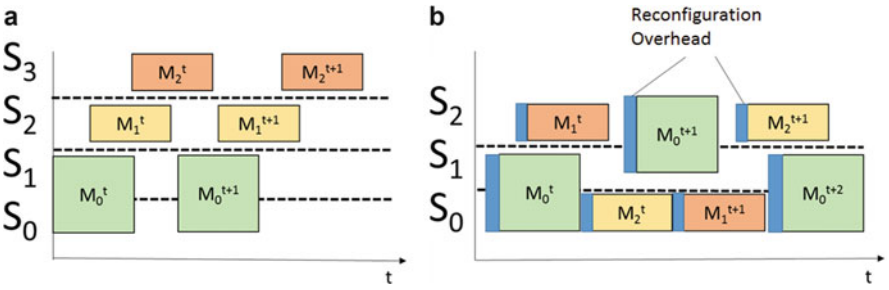

Fig. 11.14 Packing of modules into a system using slot-style reconfiguration. (**a**) Island style. (**b**) Slot style

of the reconfigurable resources by the slot-style reconfiguration [55], modules can be made relocatable to different slots. This is similar to the multiple-island reconfiguration. Figure 11.14 gives an example of how module relocation helps to better fit modules into a reconfigurable region over time. Tiling the reconfigurable region is considerably more complex as the system has to provide communication to and from the reconfigurable modules as well as the placement of the modules. The placement should also consider that FPGA resources are in general heterogeneous. For example, there are different primitives like logic, memory, and arithmetic blocks on the fabric as we mentioned in the last section. Moreover, depending on the present module layout, a tiled reconfigurable region might not provide all free tiles as one contiguous area, which is called external fragmentation. Such an external fragmentation can be removed by defragmenting the module layout which is called compaction.

11.2.2.3 Grid-Style Reconfiguration

The internal fragmentation of a reconfigurable region that is tiled with one dimensional slots can still be large. In particular, the dedicated multiplier and memory resources can be affected much by this since a module typically needs only a few among many resources arranged in columns on the FPGA fabric. Thus

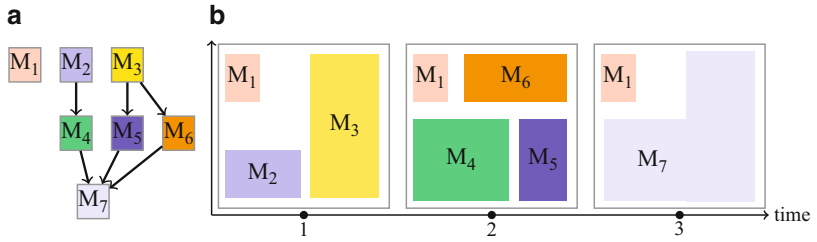

Fig. 11.15 Grid style module packing. (**a**) Task graph. (**b**) Space-time packing

it is beneficial if another module can use the remaining resources by tiling the vertical slots in Fig. 11.13b in horizontal direction. This results in a two-dimensional grid-style reconfiguration [53] as shown in Fig. 11.15b. The implementation and management of such a system is even more complex than the slot-style reconfiguration approach. Note that it requires a communication architecture that can carry out the communication of the modules with the static part of the system and also the communication between reconfigurable modules, and the communication must be established within a system at run time in the absence of sophisticated design tools. Together with the time domain, the two-dimensional grid style placement becomes a three-dimensional packing problem as visualized in Fig. 11.15. The packing should perform scheduling while satisfying the constraints on resource availability and the dependency between the modules. This packing, considering also fragmentation, has to be managed at run time.

11.3 CGRA Architecture

Coarse Grained Reconfigurable Architectures (CGRAs), also known as coarse grained reconfigurable arrays, emerged in 1990s targeting DSP applications [30]. Whereas FPGAs feature bitwise logic in the form of LUTs and switches, CGRAs feature more energy-efficient and area-conscious wordwide PEs, Register Files (RFs), and their interconnections. The cycle-by-cycle reconfigurability of CGRAs along with multiple-bit data-paths has made them superior to FPGAs for repetitive, computation-intensive tasks consisting of various word-level data-processing operations. Wider data-paths in CGRAs allow more efficient implementation of complex operators in silicon. Also the feature of cycle-by-cycle reconfigurability allows customizing the PEs and their connections for every computation and communication, making the performance closer to that of ASIC for word-level operations. Compared to FPGAs, CGRAs have lower delay characteristics and less power consumption. They have much shorter reconfiguration time (cycle level), and thus much more flexible like a programmable processor. On the other hand, gate-level reconfigurability is sacrificed, and thus they are less efficient for bit-level operations.

RaPiD [30], one of the first CGRAs, was developed in 1996. It is a linear array of cells, where each cell comprises of two 16-bit integer ALUs, a 16-bit integer multiplier, six registers, and three small local memories. The interconnections between the functional units are made by segmented buses. It works on 16-bit signed/unsigned fixed-point data. The two 16-bit ALUs in each cell can make a pipelined 32-bit ALU. It runs at 100 MHZ frequency and can perform a sustained rate of 1.6 Giga Operations Per Second (GOPS). Of course, more recent CGRAs can operate at much higher clock frequencies, provide higher power efficiency, and have more PEs in the array [102]. There are plenty of CGRAs designed and implemented in the last two decades; RaPiD, MATRIX [78], Chimaera [41], Raw [110], Garp [42], MorphoSys [100], REMARC [79], CHESS [70], HSRA [106], PipeRench [34], DReAM [8], AVISPA [65], PACT XPP [98], ADRES [73], DAPDNA-2 [99], MORA [58], Chameleon [101], SmartCell [67], FLoRA [62], ReMAP [111], SYSCORE [92], and EGRA [5] are some of THE well-known CGRAs reported to date. A detailed review can be found in survey papers by Hartenstein [39], Todman et al. [104], Choi [20], Tehre et al. [103], and Chattopadhyay [12]. One thing to note is that, in some CGRAs such as ADRES or SRP [51], the architectures work in two modes: CGRA mode and VLIW mode. In such architectures, the VLIW mode executes the control intensive part of the application.

11.3.1 Building Blocks

The main part of CGRA is an array of PEs, RFs, and their interconnections. The array is connected to data and configuration memories as well as a host processor. A PE is basically a unit that performs ALU operations mostly for executing innermost loop kernels. Typically, a PE has its own registers to save temporary data. The host processor may be a VLIW processor (e.g., ADRES), a DSP processor (e.g., Montium), or a general-purpose microprocessor (e.g., MOLEN) to execute non-loop or outer loop code. It also controls the reconfiguration of the array. Data memory works as a communication medium between PE array and the host. Reconfiguration bitstreams reside in the configuration memory and are fed to the array for reconfiguration. The reconfiguration can be done every cycle if the required array behavior changes cycle by cycle. Otherwise, the current configuration can stay in the array for a while without any reconfiguration. Figure 11.16 shows the block diagram of FloRA as a sample CGRA.

11.3.1.1 Processing Elements

CGRAs mostly consist of a 2D (e.g., 8×8) array of cells (PEs) although RaPiD has a linear (1D) array of cells. A cell usually implements a single execution stage but may also include an entire execution unit (RaPiD) or can even be a general-purpose processor (Raw). Figure 11.17a, b show the difference in computing cells between FPGA and CGRA; while a basic cell in FPGA can execute a bit-level operation, the same in CGRA can execute a word-level operation.

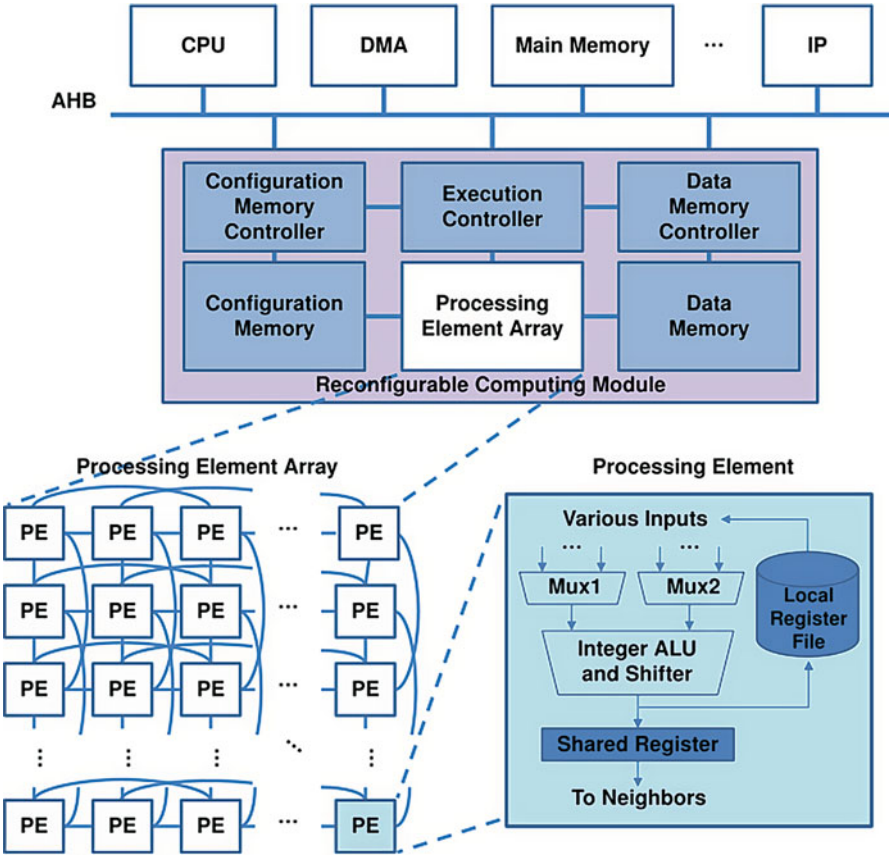

Fig. 11.16 FloRA block diagram

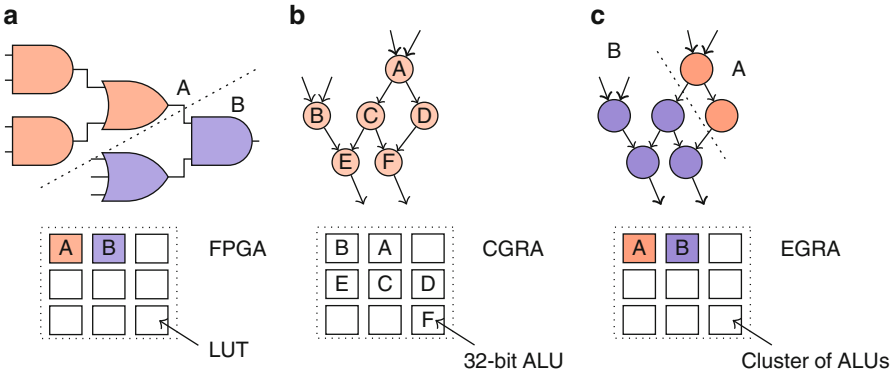

Fig. 11.17 Operation granularity comparison among (**a**) FPGA, (**b**) CGRA and (**c**) EGRA [5]

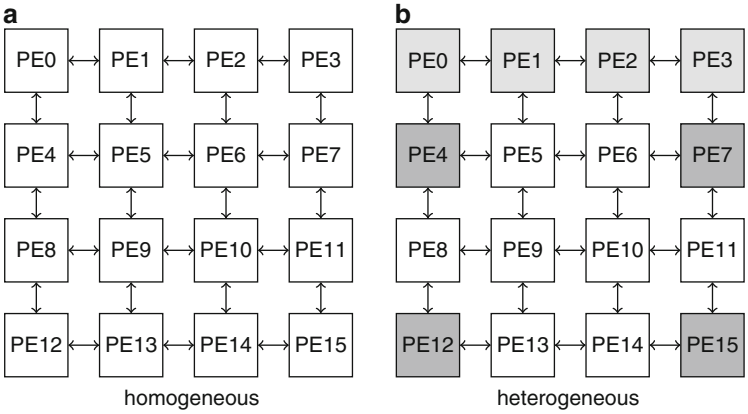

Fig. 11.18 Homogeneous vs. heterogeneous CGRAs

Although some CGRAs have different names for the computing cells, each cell is commonly called a processing element or a PE in short. PEs in different CGRAs support different set of operations (e.g., 16 instructions for CHESS). The number of PEs in a CGRA array varies from 16 in SRP to 64 in FloRA and even more to 24 rows of 16 cells in ReMAP. The PEs are either homogeneous or heterogeneous. While the homogeneity provides a uniform architecture and thus is easier to use, the heterogeneity targets better resource utilization and therefore less power and area consumption. CGRAs such as MORA, MorphoSys, REMARC, and SYSCORE are homogeneous as shown in Fig. 11.18a. Each PE has an ALU, a multiplier, and a register file and is configured through a 16- or 32-bit context word [100]. On the other hand, BilRC, MATRIX, XPP, and SRP are heterogeneous (Fig. 11.18b). Some PEs in SRP support scalar operations while some other support vector operations as well. Besides that, the type and number of operations are not the same among different PEs in SRP. Another aspect of heterogeneity is in accessing the data memory by using read and write operations; only a few PEs, called load/store PEs have access to the data memory [49]. For example, a CGRA array may have one load/store PE per row, while one or two PEs per row may provide multiplications [38]. A heterogeneous architecture may allow normal PEs to share expensive resources (like multipliers), which leads to less area and energy consumption [90]. The sample heterogeneous CGRA illustrated in Fig. 11.18b has three different kinds of PEs in the array. For example, PE0, PE1, PE2, and PE3 can be load/store PEs; PE4, PE7, PE12, and PE15 can contain expensive functional units; other PEs are normal ones.

In architectures like FloRA [38], each PE contains its own RF. But in some other CGRAs like SRP, a register file is shared among a number of PEs. PipeRench, FloRA [44], and SRP are among a few CGRAs that their PEs support floating-point operations besides the integer operations. MorphoSys is a CGRA architecture that works in a SIMD format and is also appropriate for systolic array kind of

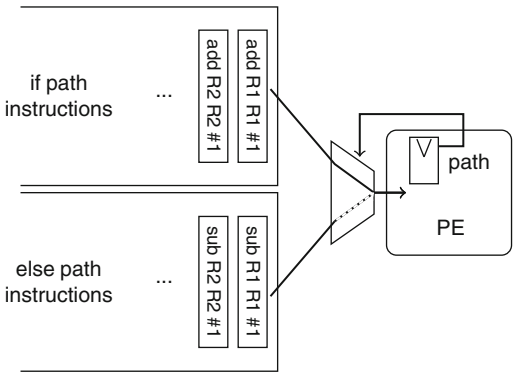

Fig. 11.19 Branch predicate techniques in FloRA

operations. Predicated execution in some new CGRAs such as SRP and FLoRA enables accelerated execution of control flows on a CGRA. Figure 11.19 shows the implementation of predicated executions in FloRA [38].

Raw [110] is another architecture in this category. The main element in the array is called a tile, which contains instruction and data memories, an arithmetic logic unit, registers, configurable logic, and a programmable switch that supports both dynamic and compiler-orchestrated static routing. On the other hand, Garp [42] is something between FPGA and CGRA, having 2-bits wide operations for the logic blocks. Template Expression Grained Reconfigurable Array (EGRA) [5] in Fig. 11.17c is another example of a coarse-grained array. RAC, a complex cell at the heart of the arithmetic in EGRA, supports efficient computation of an entire sub-expression, as opposed to a single operation. In addition, RACs can generate and evaluate branch conditions and be connected either in a combinational or a sequential mode. Figure 11.17c illustrates how a complete expression can be mapped to a cell in EGRA.

The processing element in the reMORPH array is a tile built using DSP and RAM blocks which are already available in an FPGA platform for ALUs and local data/code memories, respectively. Each tile can implement arithmetic and logic operations along with direct and indirect addressing to the data in memory. This enables complete C style loops to be executed on a PE. Memory locations are reused to store the intermediate results. In each iteration, the same set of instructions can be executed by updating the base addresses of the registers to read new data using register indirect addressing. As the reconfiguration of reMORPH array is done at the task level, it is sometimes considered as a many-core architecture rather than a CGRA.

11.3.1.2 Interconnects

In general, CGRAs execute only loops; therefore, they need to be coupled to a host processor. While the array executes the kernel loops, the host processor can execute other parts of the application. Therefore, interconnections in CGRAs can be discussed in two different levels: intra-connections and inter-connections.

Inter-connections define the connections between the array and the host processor, and intra-connections define connections among PEs in the array.

Figure 11.16 shows that a PEs array is connected to the host processor using a common bus, which also connects the PEs array to the main memory. However, the array is connected to its own data memory using direct connections. On the other hand, in some other architectures like ADRES, there is no separate host processor, but part of the array works in a VLIW mode for the role of the host processor.

As the intra-connection, which defines the connections inside a PE array, segmented buses are used among the functional units in RapiD. The most common connection topology in a 2D array of PEs is a mesh connecting a PE to its four nearest neighbors (Fig. 11.20a). Such a mesh is the base interconnection topology in architectures like MorphoSys and ADRES. Figure 11.20 illustrates some other interconnection topologies among PEs including next hop (Fig. 11.20b), buses (Fig. 11.20c), and extra (Fig. 11.20d). Some CGRAs combine mesh interconnects with next-hope connections to provide more routing capabilities among PEs

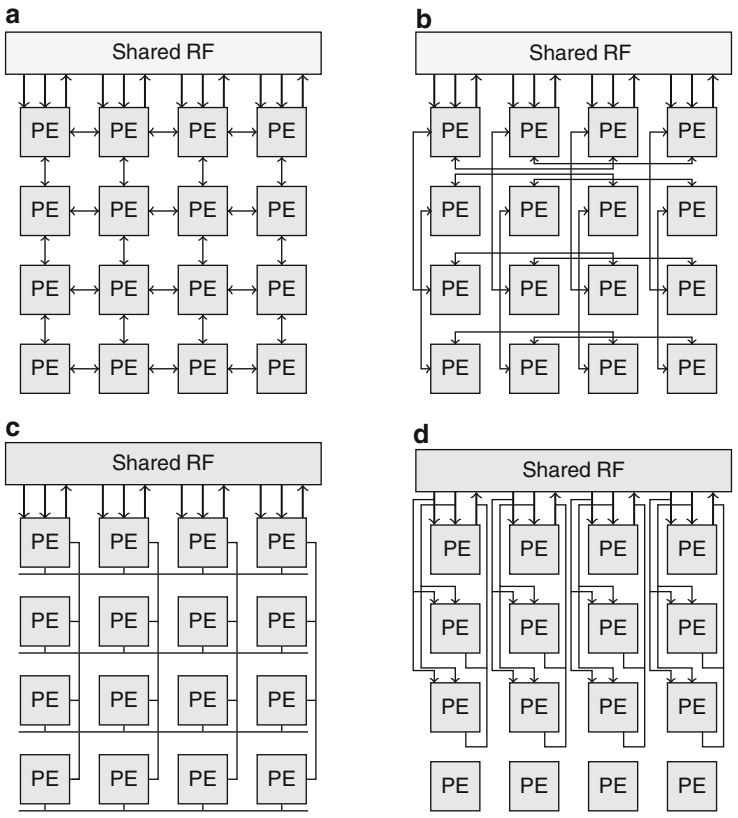

Fig. 11.20 Basic interconnects that can be combined [25]. (**a**) Nearest neighbor. (**b**) Next hop. (**c**) Buses. (**d**) Extra

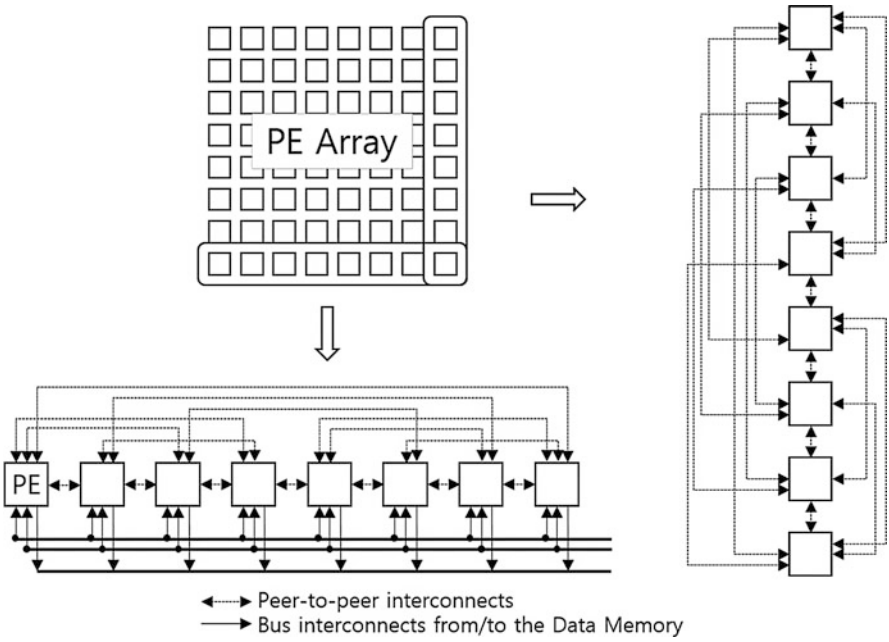

Fig. 11.21 FloRA interconnect network [76]

(e.g., FloRA). Horizontal and/or vertical buses are other common interconnects among PEs in some architectures as shown in Fig. 11.21 [25]. While Fig. 11.20 shows examples of flat interconnection, there are CGRAs with a hierarchical structure supporting multi-level interconnects among PEs. As an example, the PE array in SRP has a 2-level hierarchical interconnect topology. A cluster of PEs form a minicore with a full connection between them. Besides that there is a full connection between minicores [51].

In CGRAs with shared resources and heterogeneous PEs, the connections between PEs and shared resources can follow different topology than the connections between PEs of same or different type. But in most CGRAs, PEs in a row or column share the same resources. In heterogeneous architectures, The load/store PEs may follow the same connection topology as other PEs but they have separate dedicated connections to the ports of the data memory.

11.3.2 Reconfiguration in CGRAs

The reconfigurability of CGRA arrays can be categorized into static reconfiguration, partially dynamic reconfiguration, and fully dynamic reconfiguration.

KressArray is a statically reconfigurable CGRA. In the architecture, the array is configured before a loop is entered. So the mapping is spatial and no reconfiguration

takes place during the loop execution. In such architectures each resource is assigned a single task for executing the loop. Therefore, the associated compiler performs task mapping and data routing, which is similar to the place & route process in FPGA. The spatial mapping in such CGRAs leads to less power consumption, but a large loop cannot be mapped onto the array.

ADRES, Silicon Hive, and MorphoSys support fully dynamic reconfiguration. In such architectures, One full reconfiguration takes place for every execution cycle. Therefore, more than one task can be assigned to a resource during the loop execution lifetime and thus the loop size is not a problem. In this case, the CGRA is treated as a 3D spatial-temporal array, with time (or cycles) as the third dimension. The power consumption of the configuration memories is one drawback for these architectures. SIMD structure of MorphoSys decreases power consumption overhead by fetching one configuration code for all the PEs in a row (or a column). Another technique to reduce power consumption overhead is to pipeline the current configuration of a column to the next column for the next execution cycle [48]. Compressing configuration memory content is another solution to reducing power consumption as well as required memory capacity [86].

PACT and RaPiD feature a partial dynamic reconfiguration, such that part of the configuration bitstream is downloaded to the array statically while the other part is invoked and downloaded onto the array dynamically by using a sequencer. PACT CGRA can initiate events to invoke (partial) reconfiguration [25].

11.4 Mapping onto FPGAs

Traditional mapping of an application onto an FPGA is at the logic level mostly involving technology mapping of logic operations to FPGA logic blocks. As the systems become more complex, however, it is preferred to start the design process at a higher abstraction level such as Electronic System Level (ESL), where high-level programming languages such as C, C++, or SystemC are used to describe the system behavior and then the High-Level Synthesis (HLS) technique is used to automatically generate the Register Transfer Level (RTL) structure that implements the behavior. Figure 11.22 describes the HLS flow.

The compilation, which is the first step of the flow, transforms the input behavioral description into a formal representation. This first step may include various code optimizations such as false data dependency elimination, dead-code elimination, and constant folding. The formal model produced by the compilation exhibits the data and control dependencies between the operations. Data dependencies can be easily represented with a Data-Flow Graph (DFG) in which nodes represent operations and the directed arcs between the nodes represent the input, output, and temporary variables for data dependencies. Such a simple representation does not support branches, loops, and function calls and thus it is extended by adding control dependencies to obtain Control-/Data-Flow Graph (CDFG). There are various ways of combining data flow and control flow into a CDFG. For example, a CDFG can be a hierarchical graph where each node is a DFG that represents a

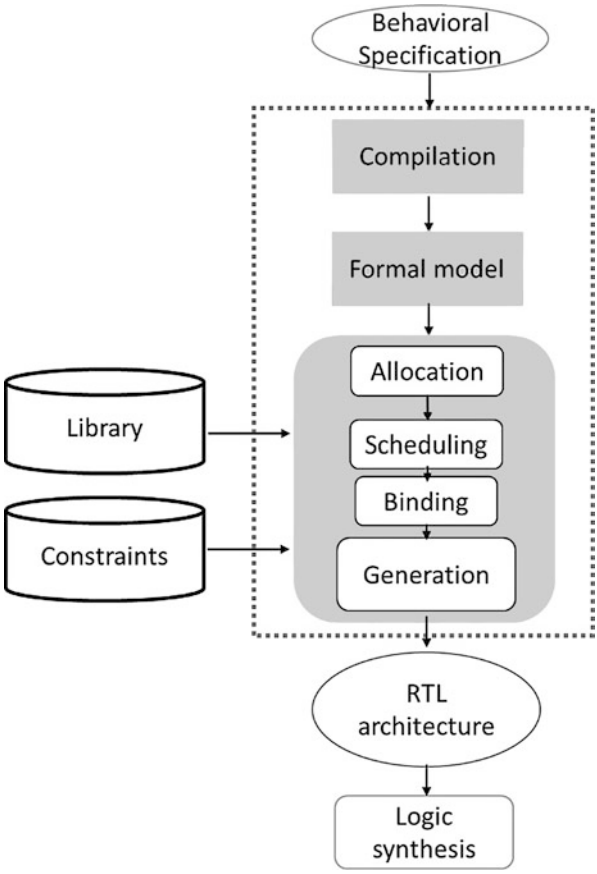

Fig. 11.22 High-level synthesis (HLS) procedures

basic block and edges between the nodes represent control dependencies. Once the CDFG has been built, additional analyses or optimizations can be performed mostly focusing on loop transformations including loop unrolling, loop pipelining, loop fission/fusion, and loop tiling. These techniques are used to optimize the latency or the throughput. To the optimized CDFG, a typical HLS process applies three main steps, namely, allocation, scheduling, and binding. We will discuss those steps in the following subsections. Several HLS tools have been developed for FPGAs targeting specific applications. GAUT is a high-level synthesis tool that is designed for DSP applications [24]. GAUT synthesizes a C program into an architecture with a processing, communication, and memory unit. It requires the user supply specific constraints, such as the pipeline initiation interval. ROCCC is an open-source HLS tool that can create hardware accelerators from C [108]. ROCCC is designed to accelerate kernels that perform repeated computation on streams of data such as FIR filters in DSP applications. ROCCC supports advanced optimizations such as systolic array generation, temporal common subexpression elimination, and it can generate Xilinx PCore modules to be used with a Xilinx MicroBlaze processor [77].

```
void FIR(short*y, short c[N], short x){
  ...
  acc=0;
  Shift_Accum_Loop: for (i=N-1;i>=0;i--){
    if (i==0){
      shift_reg[0]=x;
      data = x;
    }
    else{
      shift_reg[i]=shift_reg[i-1];
      data = shift_reg[i];
    }
    acc+=data*c[i];;
  }
  *y=acc;
}
```

Fig. 11.23 FIR filter C code

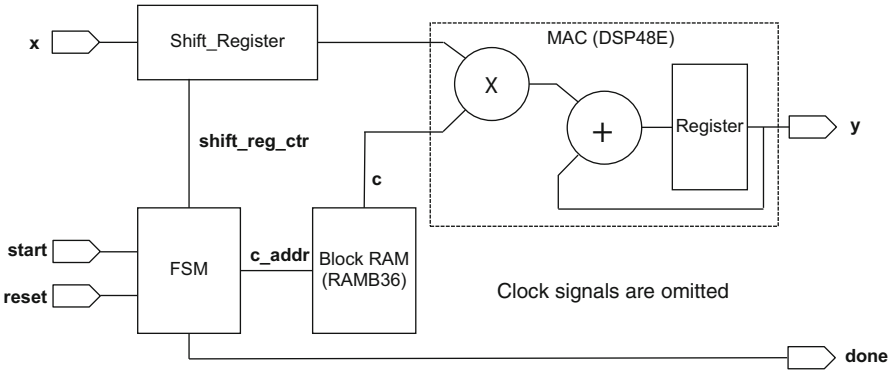

Fig. 11.24 FIR filter block diagram generated by Xilinx Vivado HLS tool

Xilinx developed the Vivado HLS tool [109] based on AutoPilot (a commercial version of xPilot [17]), a product of AutoESL which was acquired by Xilinx. It uses a Low-Level Virtual Machine (LLVM) [59] compilation infrastructure and optimizes various parameters such as interconnect delays, memory configurations, and I/O ports/types for different implementation platforms. It can automatically generate RTL code from an untimed or partially timed C, C++, or SystemC description. Figure 11.23 shows an example of C code for an FIR filter with a 16-bit wide data path, and Fig. 11.24 shows the RTL block diagram generated by the Vivado HLS tool from the C code.

The LegUp [11] is an open-source HLS framework that aims to provide the performance and energy benefits of hardware, while retaining the ease-of-use associated with software. LegUp automatically compiles a standard C program to target a hybrid FPGA-based software/hardware system-on-chip, where some

program segments execute on an FPGA-based 32-bit MIPS soft processor and other program segments are automatically synthesized into FPGA circuits – hardware accelerators – that communicate and work in tandem with the soft processor. LegUp also uses LLVM compiler framework for high-level language parsing and its standard compiler optimizations.

11.4.1 Allocation

Allocation defines the type and the number of hardware resources (functional units, storage, or connectivity components) needed to implement the behavior while satisfying the design constraints. Depending on the HLS tool, some components may be added during scheduling or binding [23]. For example, functional units such as adders or multipliers can be added during scheduling or binding if the given performance constraint cannot be met with the allocated resources. The components are selected from the RTL component library. It is important to select at least one component for each operation type used in the behavioral specification. The library must also include component characteristics such as area, delay, and power consumption.

11.4.2 Scheduling

Scheduling algorithms automatically assign control steps to operations subject to design constraints. These algorithms can be classified into two types: exact algorithms and heuristics. Exact algorithms like the one based on Integer Linear Program (ILP) [33, 43] provide an optimal schedule but take prohibitively long execution time in most practical cases. To cater to the execution time issue, various algorithms based on heuristics have been developed. For example, an algorithm may make a series of local decisions, each time selecting the single best operation-control step pairing without backtracking or look-ahead. So it may miss the globally optimal solution, but can quickly produce a result that is sufficiently close to the optimum and thus acceptable in practice. Examples of basic heuristic algorithms for HLS include As Soon as Possible (ASAP), As Late As Possible (ALAP), List Scheduling (LS), and Force-Directed Scheduling (FDS).

FDS and LS are constructive heuristic algorithms, and the quality of the results may be limited in some cases. To further improve the quality, an iterative method can be applied to the result of constructive method. In [87], for example, they adopt the concept of Kernighan and Lin's heuristic method for solving the graph-bisection problem [45] to reschedule operations into an earlier or later step iteratively until maximum gain is obtained. There are many other iterative algorithms for the resource constrained problem including genetic algorithm [7], tabu search [6, 94], simulated annealing [10, 19], and graph theoretic and computational geometry approaches [3].

11.4.3 Binding

Every operation in the specification or CDFG must be bound to one of the functional units capable of executing the operation. If there are several units with such capability, the binding algorithm must optimize this selection. Each variable that carries values from an operation to another operation across cycles (or control steps) must be bound to a storage unit. In addition, each data transfer from component to component must be bound to a connection unit such as a bus or a multiplexer. Ideally, high-level synthesis estimates the connectivity delay and area as early as possible so that later steps of HLS can better optimize the design. An alternative approach is to specify the complete architecture during allocation so that initial floor planning results can be used during binding and scheduling.

There are many algorithms proposed, but some of the basic ones include clique partitioning, left-edge algorithm, and iterative refinement. In the clique partitioning-based binding [83], the operations and variables are modeled as a graph. Cong and Smith [14] present a bottom-up clustering algorithm based on recursive collapsing of small cliques in a graph. Kurdahi and Parker [56] solved the register binding problem for a scheduled data-flow graph by using the left-edge algorithm. Chen and Cong [18] propose the k-cofamily-based register binding algorithm targeting multiplexer optimization problem.

11.4.4 Technology Mapping

Most modern FPGA devices contain programmable logic blocks that are based on a K-input look-up table (K-LUT) where a K-LUT contains 2^K truth table configuration bits so it can implement any K-input function. Thus, any logic circuit can be implemented with one K-LUT, provided that the circuit has only one output and the number of inputs is not larger than K; the internal complexity of the circuit does not matter.

The number of LUTs needed to implement a given circuit determines the size and cost of the FPGA-based realization. Thus one of the most important phases of the FPGA CAD flow is the *technology mapping* step that maps a circuit description into a LUT network presented in the target FPGA architecture, while minimizing the number of LUTs used for the mapping and the critical path delay. The process of technology mapping is often treated as a covering problem. For example, consider the process of mapping a circuit onto a network of LUTs as illustrated in Fig. 11.25. Figure 11.25a illustrates the original gate-level circuit and a possible covering with three 5-LUTs. Figure 11.25b illustrates a different mapping of the circuit through overlapped covering. In the mapping, the gate labeled X is duplicated and covered by both LUTs. Gate duplication like this example is often necessary to minimize the number of LUTs used for the mapping [22].

There are several methods for technology mapping including graph-based and LUT-based methods. Chen et al. [16] introduce graph-based FPGA technology mapping for delay optimization. As a preprocessing phase of this work, a general

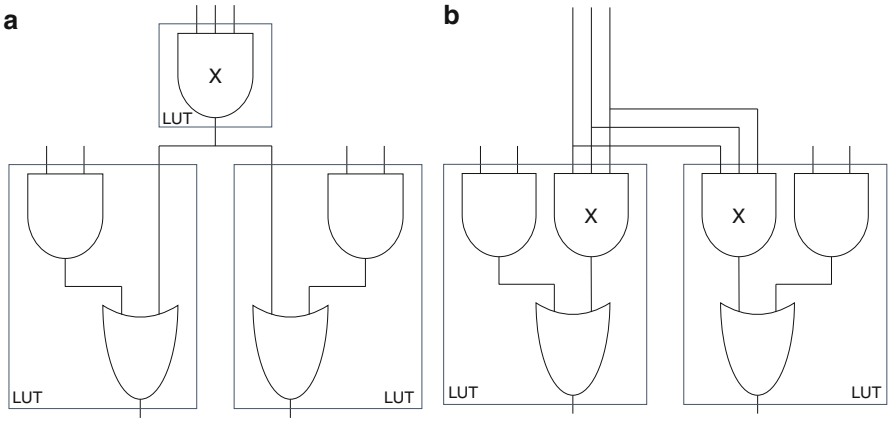

Fig. 11.25 Technology mapping as a covering problem. (**a**) Gate-level circuit and mapping. (**b**) Better mapping with duplication

algorithm called DMIG transforms an arbitrary n-node network into a network consisting of at most two-input gates with only an O(1) factor increase in network depth. A matching-based technique that minimizes area without increasing network delay is used in the post-processing phase. Cong and Minkovich [21] present LUT-based FPGA technology mapping for reliability. As device size shrinks to the nanometer range, FPGAs are increasingly prone to manufacturing defects, and it is important to have the ability to tolerate multiple defects. One common defect point is in the LUT configuration bits, which are crucial to the correct operation of FPGAs. This work presents an error analysis technique that efficiently calculates the number of critical bits needed to implement each LUT. It allows the design to function correctly when implemented on a faulty FPGA.

11.5 Mapping onto CGRAs

Despite the enormous computation power, the performance of CGRAs critically hinges on a smart compiler and mapping algorithm. The target applications of these architectures often spend most of their time executing a few time-critical loop kernels. So the performance of the entire application may be improved considerably by mapping these loop kernels onto an accelerator. Moreover, these computation-intensive loops often exhibit a high degree of inherent parallelism. This makes it possible to use the abundant computation resources available in CGRAs. The programmer or the compiler for a CGRA may find these computation-intensive loops through profiling and/or analysis and directs the computation-intensive segments to CGRA and control-intensive part to the host processor.

The first compilation attempts were focused on ILP but failed to better exploit the parallelism than VLIW [74]. Success of software pipelining techniques encouraged researches to examine modulo scheduling. Modulo scheduling is a software

pipelining technique used in VLIW to improve parallelism by executing different loop iterations in parallel. The objective of modulo scheduling is to engineer a schedule for one iteration of the loop such that the same schedule is repeated at regular intervals with respect to intra- and inter-iteration dependencies and resource constraints. This interval is termed Initiation Interval (II), essentially reflecting the performance of the scheduled loop. It is determined by several parameters, and the reader is directed to [95] for the details.

Modulo scheduling on coarse-grained architectures is a combination of three subproblems: placement, routing, and scheduling. Placement determines on which PE to place one operation. Scheduling determines in which cycle to execute that operation. Routing connects the placed and scheduled operations according to their data dependencies [25, 74]. In the worst case, II is equal to the schedule length (iteration length), and in the best case, it is equal to one, which means that the entire loop is mapped onto the CGRA at once (static mapping). In case of II≥2, PEs need to be reconfigured several times to execute the entire loop.

Dynamically Reconfigurable Embedded System Compiler (DRESC) [74] uses a modulo scheduling algorithm based on simulated annealing [52]. It begins with a random placement of operations on the PEs, which may not be a valid modulo schedule. Operations are then moved between PEs until a valid schedule is achieved. The random movement of operations in the simulated annealing technique can result in a long convergence time for loops with modest numbers of operations [89]. SPR [31] is a mapping tool that uses Iterative Modulo Scheduling (IMS), besides using simulated annealing placement with a cooling schedule inspired by VPR [9] as well as PathFinder [71] and QuickRoute [66] for pipelined routing.

To have a better mapping, it is required to consider scheduling, placement, and routing at the same time. Graph-based algorithms [74, 75, 116, 117] are able to do the job just by modeling CGRA as a graph including time as the third dimension. Therefore, the mapping problem becomes mapping the loop kernel DFG onto the CGRA Modulo Resource Routing Graph (MRRG). Figure 11.26 shows how a loop kernel DFG is mapped onto the CGRA, where three subsequent iterations of the DFG are mapped. As shown in this example, II is 2 while the schedule length is 4.

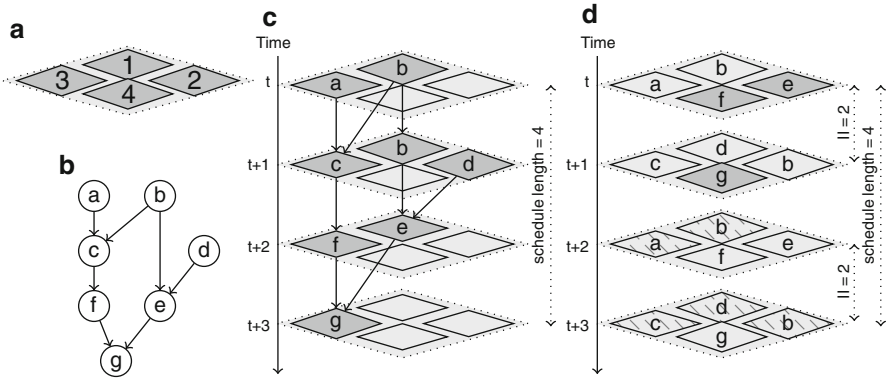

Fig. 11.26 Mapping example [35]

Figure 11.26d shows that the same PE which is doing operation "b" in cycle "i," acts as routing PE in cycle "i+1," to route operation "b" to cycle "i+2." In this way, assigning some PEs for routing provides more connectivity for the communications between nodes of a DFG.

One of the parameters defining II is the recurrence-constrained lower bound. Oh et al. [85] introduced a recurrence cycle-aware scheduling technique for CGRAs. Their modulo scheduler groups operations belonging to a recurrence cycle into a clustered node and then computes a scheduling order for those clustered nodes. Deadlocks that arise when two or more recurrence cycles depend on each other are resolved by using heuristics that favor recurrence cycles with long recurrence delays. Whereas previous approaches had to sacrifice either compilation speed or quality of the result, this is no longer necessary with the recurrence cycle-aware scheduling technique. Traditional schedulers are node-centric in that the focus is assigning operations to PEs. The straightforward adaptation of this approach is operation placement followed by operand routing to determine if the assignment is feasible. Park et al. [89] have shown that node-centric approaches are poor for CGRA. They proposed an Edge Centric Modulo Scheduling (EMS) approach. This approach focuses on mapping edges instead of nodes.

Shared resources [47], data memory limitation [26, 49], and register file distribution (REGIMap [36]) are also important constraints that must be considered for the mapping of a DFG onto a CGRA. ILP can be used to obtain an optimal solution to a mapping problem considering such constraints. However, since the ILP approach is slow in general, it is used to obtain an optimal solution for problems of small size; the solution is used to check the quality of other heuristic-based (non-ILP) approaches. We briefly introduce some of the ILP-based and heuristic-based mapping approaches in Sects. 11.5.1 and 11.5.2, respectively.

In cases that there is not enough space to map all the loop kernels of the application, some decision has to be made to find more eligible kernels. Lee et al. [64] have proposed a kernel selection algorithm. If the memory requirement of the application is larger than the available Scratchpad Memory (SPM) size, kernel selection is performed based on detailed statistics such as run-time and buffer-access information of each kernel. Otherwise, all the kernels are mapped to the CGRA.

11.5.1 ILP-Based Mapping Approaches

There have been a few approaches to ILP formulation of the problem of mapping an application to a CGRA. Ahn et al. [1] have formulated the mapping problem in ILP for the first time. Their approach consists of three stages: covering, partitioning, and laying-out. In the covering stage, a kernel tree is transformed to the configuration tree such that each node of the configuration tree represents a configuration for each PE and can cover and execute one or more operations. The partitioning stage splits the configuration tree to clusters such that each cluster is mapped to one distinct column of the CGRA in laying-out stage. The authors have targeted optimal vertical mapping with the minimum total data transfer cost among the rows of PEs.

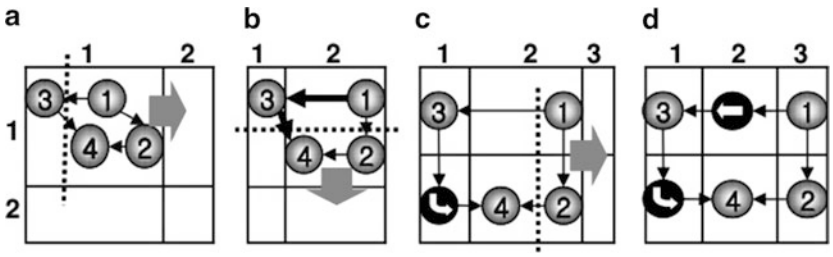

Fig. 11.27 Split & Push heuristic in SPKM [116]

Yoon et al. [117] have developed a graph-based ILP formulation. Then they have used Split & Push Kernel Mapping (SPKM) heuristic to solve the mapping problem within a feasible time. Figure 11.27 shows how the Split & Push Kernel Mapping Algorithm works. It assigns the entire DFG into one PE and then starts splitting it horizontally and vertically. The link between the non-neighboring PEs is fulfilled by using routing PEs. Their formulation takes into account many architectural details of CGRA and leads to minimum number of rows.

Lee et al. [63] have proposed an approach that covers not only integer operations but also floating-point operations implemented by simply using two neighboring tiles. Besides their ILP formulation, they have developed a fast heuristic mapping algorithm considering Steiner points. Details are given in Sect. 11.5.3.

As already mentioned, there are CGRAs like Raw [110] and reMORPH [80, 93] that provide reconfigurations at the task level rather than instruction level. Moghaddam et al. [81, 82] have presented an ILP-based optimal framework to map an application in the form of a task graph onto a tile-based CGRA. They have integrated scheduling, placement, and routing into one mapping problem. The formulation benefits from the reconfigurability feature of the target platform; a large application having more tasks than the number of PEs or even multiple applications can be mapped to the platform.

11.5.2 Heuristic-Based Approaches

There are many heuristic-based mapping approaches for CGRAs including EMS [89], EPIMap [35], and graph-minor approach [15]. We review some of the most referenced ones here.

Lee et al. [60] have developed a generic architecture template, called the dynamically reconfigurable ALU array Dynamically Reconfigurable ALU Array (DRAA). Their mapping approach goes through the following three levels: PE level, line level, and plane level. In the PE level, a DFG is extracted. In the line level, nodes of the DFG are grouped such that each group can be assigned to a distinct row of PEs. And finally in the plane level, the lines are stitched together to form a plane. They take into account the data reuse patterns in loops of DSP algorithms as part of their approach.

Park et al. [88] have presented their modulo graph embedding. Modulo graph embedding is also a modulo scheduling technique for software pipelining. They have modelled the architecture using an MRRG. Their MRRG has only II layers, which makes the problem space smaller, and therefore the mapping algorithm converges to the solution faster. They have later [89] presented an EMS approach, which specifically targets routing of data instead of placement of operations.

Galanis et al. [32] have presented a priority-based mapping algorithm. This algorithm assigns an initial priority to each operation of the DFG. This priority is inversely proportional to the mobility, which is the difference between ALAP and ASAP schedule times. The operations residing on the critical path will be scheduled first.

Hanataka et al. [40] have presented a modulo scheduling algorithm that takes into account "resource reservation" and "scheduling" separately. They have used a resource usage aware relocation algorithm. Their approach uses a compact 3D architecture graph similar to the MRRG used in [88]. This graph is only II times as large as the original two-dimensional graph.

Dimitroulakos et al. [27] have presented an efficient mapping approach where scheduling and register allocation phases are performed in one single step. They have also incorporated modulo scheduling with back tracking in their approach. Their mapping approach minimizes memory bandwidth bottleneck. They have tried to maximize the ILP using a new priority scheme and few heuristics. Their solution covers a large range of CGRAs. They have also developed a simulation framework.

Oh et al. [85] have proposed a scheduling technique that is aware of data dependencies caused by inter-iteration recurrence cycles. Therefore, operations in a recurrence cycle are clustered and considered as a single node. The operations in a recurrence cycle are handled as soon as all predecessors of the clustered node have been scheduled. They have also proposed a modification in the target architecture to further improve the quality of their scheduling approach.

Lee et al. [61] have proposed a mapping approach based on high-level synthesis techniques. They have used loop unrolling and pipelining techniques to generate loop parallelized code to improve the performance drastically.

Patel et al. [91] benefit from systolic mapping techniques in their scheduler. They prepare an Synchronous Data Flow (SDF) graph for the application; they rearrange the graph for systolic mapping, schedule the SDF graph, and then prepare a CDFG for each node of the SDF graph. As the last step, they generate topology matrix and delay matrix which are used for the final systolic mapping.

Kim et al. [50] have proposed a memory-aware mapping technique for the first time. They have also proposed efficient methods to handle dependent data on a double-buffering local memory, which is necessary for recurrent loops.

11.5.3 FloRA Compilation Flow: Case Study

FloRA consists of a Reconfigurable Computing Module (RCM) for executing loop kernel code segments and a general-purpose processor for controlling the RCM, and

these units are connected with a shared bus. The RCM consists of an array of PEs, several sets of data memories, and a configuration memory [47]. Figure 11.16 shows FloRA containing a 8×8 reconfigurable array of PEs and internal structure of a PE. Each PE is connected to the nearest neighboring PEs: top, bottom, left, and right. The size of the array can be optimized to a specific application domain.

The area-critical resources (such as multipliers) are located outside the PEs and shared among a set of PEs. Each area-critical resource is pipelined to curtail the critical path delay, and its execution is initiated by scheduling the area-critical operation on one of the PEs that share this area-critical resource. Thus, each PE can be dynamically reconfigured either to perform arithmetic and logical operations with its own Arithmetic-Logic Unit (ALU) in one clock cycle or to perform multiply or division operations using the shared functional unit in several clock cycles with pipelining. Resource pipelining further improves loop pipelining execution by allowing multiple operations to execute simultaneously on one pipelined resource. Furthermore, pipelining together with resource sharing increases the utilization of these area-critical units. Data memory consists of three banks: one connected to the write bus and the other two connected to the read buses. The connections can also be reconfigured. Each PE has its local Configuration Cache Element (CCE). Each CCE has several layers, so the corresponding PE can be reconfigured independently with different contexts.

FloRA supports floating-point operations by allotting a pair of PEs: one for mantissa and the other for exponent. Mapping a floating-point operation onto the PE array with integer operations may take many layers of cache. If a kernel consists of a multitude of floating-point operations, then mapping it onto the array easily runs out of the cache layers, causing costly fetch of additional context words from the main memory. Instead of using multiple cache layers to perform such a complex operation, some control logic is added to the PEs so that the operation can be completed in multiple cycles but without requiring multiple cache layers. The control logic can be implemented with a small Finite-State Machine (FSM) that controls the PE's existing data path for a fixed number of cycles [44].

Lee et al. have presented two mapping approaches for FloRA: (1) an optimal approach using ILP and (2) a fast heuristic approach using Quantum-inspired Evolutionary Algorithm (QEA). Both approaches support integer-type applications as well as floating-point-type applications. These mapping algorithms adopt HLS techniques that handle loop-level parallelism by applying loop unrolling and loop pipelining techniques. The overall compilation flow is given in Fig. 11.28. The first step is partitioning, which generates two C codes one for the RISC processor and the other for the CGRA.

The code segments for the RISC processor are statically scheduled and the corresponding assembly code is generated with a conventional compiler. The code segments for the RCM (generally loop kernels) are converted to a CDFG using the SUIF2 [107] parser. During this process, loop unrolling maximizes the utilization of the PEs. Then HLS techniques are used for the scheduling and binding on one column of PEs. Each column of the CGRA executes its own iteration of the loop to implement loop pipelining.

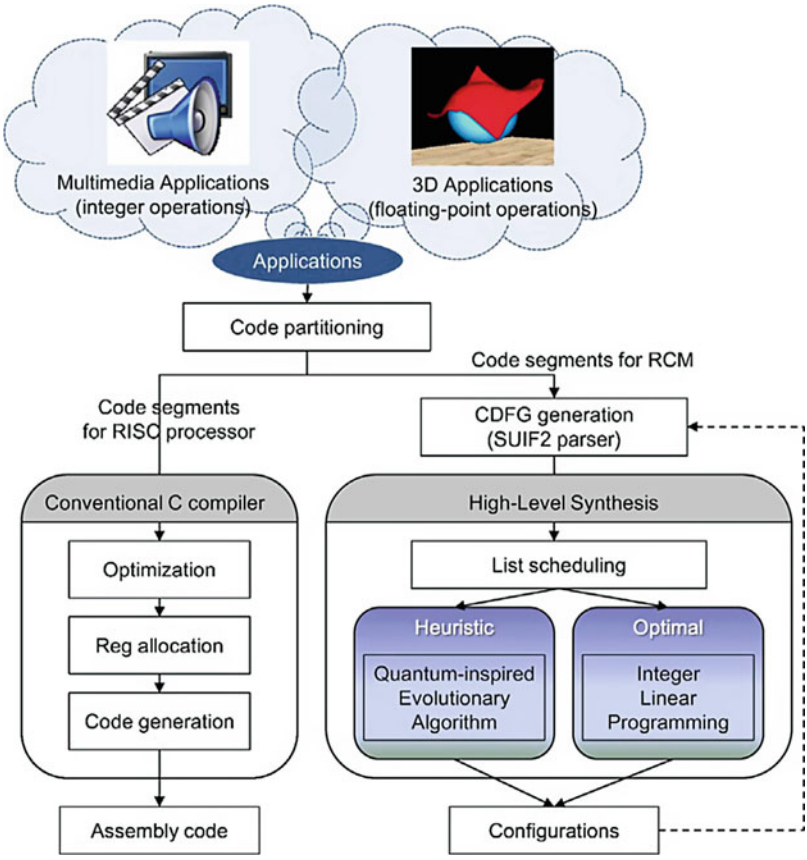

Fig. 11.28 Overall design flow for application mapping onto FloRA [63]

The main objective of the mapping problem is to map a given loop kernel to the CGRA such that the total latency is minimized while satisfying several constraints. Lee et al. have formulated the problem using ILP. ILP-based application mapping yields an optimal solution. However, it takes an unreasonably long execution time to find a solution, making it unsuitable for large designs or for design space exploration. Therefore, they defined a fast heuristic mapping algorithm considering Steiner points. Their heuristic is based on a mixture of two algorithms: List Scheduling and QEA.

11.5.3.1 List Scheduling

First, List Scheduling algorithm topologically sorts the vertices from the sink to the source. If a vertex has a longer path to the sink, then it gets a higher priority. From the sorted list, the algorithm selects and schedules the vertex with the highest priority if all the predecessor vertices have been scheduled and the selected vertex is

reachable from all the scheduled predecessor vertices through the interconnections available in the CGRA. If the vertex is a floating-point vertex, the algorithm checks to see if neighbor PEs are busy, since executing a floating-point operation requires a pair of PEs for several cycles. Mapping a vertex onto a PE considers interconnect constraint and shared resource constraint. If there is no direct connection available for implementing a data dependency between two PEs, a shortest path consisting of unused PEs which work as routers is searched. Another constraint to be considered is the constraint set by sharing area-critical functional units. For example, if there is only one multiplier shared among the PEs in a row, two multiply operations cannot be scheduled successively but should wait for N (number of PEs in a row) cycles after scheduling one multiply operation, since the multiplier must be used by other PEs in the same row for loop pipelining. In this case, the second multiply operation may need to wait with proper routing of the input data.

11.5.3.2 QEA

QEA is an evolutionary algorithm that is known to be very efficient compared to other evolutionary algorithms [37]. The QEA starts from the List Scheduling result as a seed and attempts to further reduce the total latency. Starting the QEA with a relatively good initial solution tends to reach a better solution sooner than starting it with a random seed. When the schedule and binding of all vertices are determined, it tries to find the routing paths among the vertices – the routing may need to use unused remaining PEs – to see if these schedule and binding results violate the interconnect constraint. In this routing phase, the quality of the result depends on the order of edges to be routed. Thus the priority of edges for the ordering is determined as follows.

- Edges located in the critical path are assigned higher priority.
- Among the edges located in the critical path, edges that have smaller slack (shorter distance) receive higher priority.
- If a set of edges have the same tail vertex, then the set of edges becomes a group and the priority of this group is determined by the highest priority among the group members.

According to the above priority, a list of candidate edges is made and a shortest path for each edge is found in the order of priority with the Dijkstra's algorithm. In this routing phase, a Steiner tree (instead of a spanning tree) for multiple writes from a single source is considered. The heuristic algorithm for finding a Steiner tree tries to find a path individually for each outgoing edge from the source. If some paths use the same routing PE, it becomes a Steiner point. Although this approach may not always find an optimal path, it gives good solutions in most of the cases if not all. Indeed, experimental results show that the approach finds optimal solutions for 97% of the randomly generated examples. Table 11.1 compares the result obtained by the heuristic algorithm for the butterfly addition example with the optimum result obtained by the ILP formulation.

Table 11.1 Experimental result of butterfly addition example

		Latency (cycle)	Mapping time (s)
ILP	Spanning tree	5	1022
	Steiner tree	4	965
Heuristic	Spanning tree	5	13
	Steiner tree	4	9

11.6 Conclusions

Reconfigurable architecture provides software-like flexibility as well as hardware-like performance. Depending on the granularity of configuration, we can consider two types of reconfigurable architecture: fine-grained reconfigurable architecture like FPGA and CGRA. In this chapter, we have surveyed various architectures for FPGAs and CGRAs. We have also surveyed various approaches to mapping applications to the architectures. Compared to pure hardware design or pure software design, there are more opportunities in utilizing such reconfigurable architectures since they support hardware reconfiguration which is controlled by software (For general trade-offs between hardware and software, refer to ▸ Chap. 1, "Introduction to Hardware/Software Codesign".). For example, FPGAs can be better utilized by dynamic partial reconfiguration, which has been mentioned in this chapter. However, such opportunities have not been very well investigated and still require more researches together with the researches on better architectural supports.

References

1. Ahn M, Yoon J, Paek Y, Kim Y, Kiemb M, Choi K (2006) A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In: Proceedings of the design, automation and test in Europe, DATE '06, vol 1, p 6
2. Altera arria 10 FPGA. www.altera.com. Accessed 28 Nov 2015
3. Aletà A, Codina JM, Sánchez J, González A (2001) Graph-partitioning based instruction scheduling for clustered processors. In: Proceedings of the 34th annual ACM/IEEE international symposium on microarchitecture, MICRO 34. IEEE Computer Society, Washington, DC, pp 150–159
4. Altera stratix v FPGA. www.altera.com. Accessed 28 Nov 2015
5. Ansaloni G, Bonzini P, Pozzi L (2011) EGRA: a coarse grained reconfigurable architectural template. IEEE Trans Very Large Scale Integr (VLSI) Syst 19(6):1062–1074
6. Baar T, Brucker P, Knust S (1999) Tabu search algorithms and lower bounds for the resource-constrained project scheduling problem. In: Voss S, Martello S, Osman I, Roucairol C (eds) Meta-heuristics. Springer US, pp 1–18. doi: 10.1007/978-1-4615-5775-3_1
7. Bean JC (1994) Genetic algorithms and random keys for sequencing and optimization. ORSA J Comput 6(2):154–160. doi: 10.1287/ijoc.6.2.154, http://dx.doi.org/10.1287/ijoc.6.2.154
8. Becker J, Glesner M (2000) Fast communication mechanisms in coarse-grained dynamically reconfigurable array architecture. In: The 2000 international conference on parallel and distributed processing techniques and applications (PDPTA'2000), Las Vegas

9. Betz V, Rose J (1997) VPR: a new packing, placement and routing tool for FPGA research. In: Proceedings of the 7th international workshop on field-programmable logic and applications, FPL '97. Springer, London, pp 213–222
10. Bouleimen K, Lecocq H (2003) A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. Eur J Oper Res 149(2):268–281. doi: 10.1016/S0377-2217(02)00761-0. Sequencing and Scheduling
11. Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Anderson JH, Brown S, Czajkowski T (2011) LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the 19th ACM/SIGDA international symposium on field programmable gate arrays, FPGA '11. ACM, New York, pp 33–36. doi: 10.1145/1950413.1950423
12. Chattopadhyay A (2013) Ingredients of adaptability: a survey of reconfigurable processors. VLSI Des 2013:10:10–10:10
13. Che S, Li J, Sheaffer J, Skadron K, Lach J (2008) Accelerating compute-intensive applications with GPUs and FPGAs. In: Proceedings of the symposium on application specific processors, SASP 2008, pp 101–107
14. Chen D, Cong J (2004) Register binding and port assignment for multiplexer optimization. In: Proceedings of the 2004 Asia and South Pacific design automation conference, ASP-DAC '04. IEEE Press, Piscataway, pp 68–73
15. Chen L, Mitra T (2014) Graph minor approach for application mapping on CGRAs. ACM Trans Reconfig Technol Syst 7(3):21:1–21:25
16. Chen KC, Cong J, Ding Y, Kahng A, Trajmar P (1992) Dag-map: graph-based FPGA technology mapping for delay optimization. IEEE Des Test Comput 9(3):7–20. doi: 10.1109/54.156154
17. Chen D, Cong J, Fan Y, Han G, Jiang W, Zhang Z (2005) xPilot: a platform-based behavioral synthesis system. SRC TechCon 5
18. Chen D, Cong J, Fan Y, Wan L (2010) LOPASS: a low-power architectural synthesis system for FPGAs With interconnect estimation and optimization. IEEE Trans VLSI Syst 18(4): 564–577
19. Cho JH, Kim YD (1997) A simulated annealing algorithm for resource constrained project scheduling problems. J Oper Res Soc 48(7):736–744
20. Choi K (2011) Coarse-grained reconfigurable array: architecture and application mapping. IPSJ Trans SystLSI Des Methodol 4:31–46. doi: 10.2197/ipsjtsldm.4.31
21. Cong J, Minkovich K (2010) Lut-based FPGA technology mapping for reliability. In: Proceedings of the 47th design automation conference, DAC '10. ACM, New York, pp 517–522. doi: 10.1145/1837274.1837401
22. Cong J, Wu C, Ding Y (1999) Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays, FPGA '99. ACM, New York, pp 29–35. doi: 10.1145/296399.296425
23. Coussy P, Gajski D, Meredith M, Takach A: An introduction to high-level synthesis. IEEE Des Test Comput 26(4):8–17 (2009). doi: 10.1109/MDT.2009.69
24. Coussy P, Lhairech-Lebreton G, Heller D, Martin E (2010) Gaut–a free and open source high-level synthesis tool. In: IEEE DATE
25. De Sutter B, Raghavan P, Lambrechts A (2010) Coarse-grained reconfigurable array architectures. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of signal processing systems. Springer, Boston, pp 449–484. doi: 10.1007/978-1-4419-6345-1_17
26. Dimitroulakos G, Galanis MD, Goutis CE (2005) Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays. In: 2005 IEEE international conference on application-specific systems, architecture processors (ASAP'05), pp 161–168. doi: 10.1109/ASAP.2005.12
27. Dimitroulakos G, Georgiopoulos S, Galanis MD, Goutis CE (2009) Resource aware mapping on coarse grained reconfigurable arrays. Microprocess Microsyst 33(2):91–105
28. Dirk K (2012) Partial reconfiguration on FPGAs: architectures, tools and applications. Springer, New York

29. Dynamic reconfiguration in Stratix IV devices (2014). Accessed 27 Nov 2015
30. Ebeling C, Cronquist D, Franklin P (1996) Rapid – reconfigurable pipelined datapath. In: Hartenstein R, Glesner M (eds) Field-programmable logic smart applications, new paradigms and compilers. Lecture notes in computer science, vol 1142. Springer, Berlin/Heidelberg, pp 126–135
31. Friedman S, Carroll A, Van Essen B, Ylvisaker B, Ebeling C, Hauck S (2009) SPR: an architecture-adaptive cgra mapping tool. In: Proceedings of the ACM/SIGDA international symposium on field programmable gate arrays, FPGA '09. ACM, New York, pp 191–200. doi: 10.1145/1508128.1508158
32. Galanis M, Dimitroulakos G, Goutis C (2006) Mapping DSP applications on processor/coarse-grain reconfigurable array architectures. In: Proceedings 2006 IEEE international symposium on circuits and systems, ISCAS 2006, p. 4
33. Garfinkel RS, Nemhauser GL (1972) Integer programming, vol 4. Wiley, New York
34. Goldstein S, Schmit H, Budiu M, Cadambi S, Moe M, Taylor R (2000) Piperench: a reconfigurable architecture and compiler. Computer 33(4):70–77
35. Hamzeh M, Shrivastava A, Vrudhula S (2012) EPIMap: using epimorphism to map applications on CGRAs. In: Proceedings of the 49th annual design automation conference, DAC '12. ACM, New York, pp 1284–1291
36. Hamzeh M, Shrivastava A, Vrudhula S (2013) REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In: 2013 50th ACM/EDAC/IEEE design automation conference (DAC), pp 1–10. doi: 10.1145/2463209.2488756
37. Han KH, Kim JH (2004) Quantum-inspired evolutionary algorithms with a new termination criterion, H/sub/spl epsi//gate, and two-phase scheme. IEEE Trans Evol Comput 8(2):156–169. doi: 10.1109/TEVC.2004.823467
38. Han K, Ahn J, Choi K (2013) Power-efficient predication techniques for acceleration of control flow execution on CGRA. ACM Trans Archit Code Optim 10(2):8:1–8:25. doi: 10.1145/2459316.2459319
39. Hartenstein R (2001) A decade of reconfigurable computing: a visionary retrospective. In: Proceedings of the design, automation and test in Europe, Conference and Exhibition 2001, pp 642–649
40. Hatanaka A, Bagherzadeh N (2007) A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In: IEEE international parallel and distributed processing symposium, IPDPS 2007, pp 1–8
41. Hauck S, Fry T, Hosler M, Kao J (2004) The chimaera reconfigurable functional unit. IEEE Trans Very Large Scale Integr (VLSI) Syst 12(2):206–217
42. Hauser J, Wawrzynek J (1997) Garp: a mips processor with a reconfigurable coprocessor. In: Proceedings of the 5th annual IEEE symposium on field-programmable custom computing machines, 1997, pp 12–21
43. Hwang CT, Lee JH, Hsu YC (1991) A formal approach to the scheduling problem in high level synthesis. IEEE Trans Comput-Aided Des Integr Circuits Syst 10(4):464–475. doi: 10.1109/43.75629
44. Jo M, Lee D, Han K, Choi K (2014) Design of a coarse-grained reconfigurable architecture with floating-point support and comparative study. Integr {VLSI} J 47(2):232–241. doi: 10.1016/j.vlsi.2013.08.003
45. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49:291–307
46. Kestur S, Davis J, Williams O (2010) Blas comparison on FPGA, CPU and GPU. In: 2010 IEEE computer society annual symposium on VLSI (ISVLSI), pp 288–293. doi: 10.1109/ISVLSI.2010.84
47. Kim Y, Kiemb M, Park C, Jung J, Choi K (2005) Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In: Design, automation and test in Europe, vol 1 pp 12–17 . doi: 10.1109/DATE.2005.260
48. Kim Y, Mahapatra RN, Park I, Choi K (2009) Low power reconfiguration technique for coarse-grained reconfigurable architecture. IEEE Trans Very Large Scale Integr (VLSI) Syst 17(5):593–603. doi: 10.1109/TVLSI.2008.2006039

49. Kim Y, Lee J, Shrivastava A, Paek Y (2011) Memory access optimization in compilation for coarse-grained reconfigurable architectures. ACM Trans Des Autom Electron Syst 16(4):42:1–42:27. doi: 10.1145/2003695.2003702
50. Kim Y, Lee J, Shrivastava A, Yoon J, Cho D, Paek Y (2011) High throughput data mapping for coarse-grained reconfigurable architectures. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(11):1599–1609
51. Kim C, Chung M, Cho Y, Konijnenburg M, Ryu S, Kim J (2012) ULP-SRP: Ultra low power samsung reconfigurable processor for biomedical applications. In: 2012 international conference on field-programmable technology (FPT), pp 329–334. doi: 10.1109/FPT.2012.6412157
52. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680. doi: 10.1126/science.220.4598.671
53. Koch D, Beckhoff C, Teich J (2009) Minimizing internal fragmentation by fine-grained two-dimensional module placement for runtime reconfigurable systems. In: 17th annual IEEE symposium on field-programmable custom computing machines (FCCM 2009). IEEE Computer Society, pp 251–254
54. Koch D, Beckhoff C, Tørrison J (2010) Advanced partial run-time reconfiguration on spartan-6 fpgas. In: 2010 international conference on field-programmable technology (FPT), pp 361–364
55. Koch D, Beckhoff C, Wold A, Torresen J (2013) Easypr – an easy usable open-source PR system. In: 2013 international conference on field-programmable technology (FPT), pp 414–417
56. Kurdahi F, Parker A (1987) Real: a program for register allocation. In: 24th conference on design automation, pp 210–215. doi: 10.1109/DAC.1987.203245
57. Langhammer M, Pasca B (2015) Floating-point DSP block architecture for FPGAs. In: Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays, FPGA '15. ACM, New York, pp 117–125. doi: 10.1145/2684746.2689071
58. Lanuzza M, Perri S, Corsonello P, Margala M (2007) A new reconfigurable coarse-grain architecture for multimedia applications. In: 2007 second NASA/ESA conference on adaptive hardware and systems, AHS 2007, pp 119–126
59. Lattner C (2002) LLVM: an infrastructure for multi-stage optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana. http://llvm.org/pubs/2002-12-LattnerMSThesis.pdf
60. Lee Je, Choi K, Dutt ND (2003) An algorithm for mapping loops onto coarse-grained reconfigurable architectures. SIGPLAN Not 38(7):183–188
61. Lee G, Lee S, Choi K (2008) Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques. In: Proceedings of the international SoC design conference, ISOCC '08, vol 01, pp I–395–I–398
62. Lee D, Jo M, Han K, Choi K (2009) Flora: coarse-grained reconfigurable architecture with floating-point operation capability. In: 2009 international conference on field-programmable technology, FPT 2009, pp 376–379
63. Lee G, Choi K, Dutt N (2011) Mapping multi-domain applications onto coarse-grained reconfigurable architectures. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(5):637–650
64. Lee H, Nguyen D, Lee J (2015) Optimizing stream program performance on cgra-based systems. In: Proceedings of the 52nd annual design automation conference, DAC '15. ACM, New York, pp 110:1–110:6
65. Leijten J, Burns G, Huisken J, Waterlander E, van Wel A (2003) AVISPA: a massively parallel reconfigurable accelerator. In: Proceedings of the 2003 international symposium on system-on-chip, pp 165–168
66. Li S, Ebeling C (2004) Quickroute: a fast routing algorithm for pipelined architectures. In: Proceedings of the 2004 IEEE international conference on field-programmable technology, pp 73–80. doi: 10.1109/FPT.2004.1393253
67. Liang C, Huang X (2008) Smartcell: a power-efficient reconfigurable architecture for data streaming applications. In: 2008 IEEE workshop on signal processing systems, SiPS 2008, pp 257–262

68. Lie W, Feng-yan W: Dynamic partial reconfiguration in FPGAs. In: 2009 third international symposium on intelligent information technology application, IITA 2009, vol 2, pp 445–448 (2009)
69. Lysaght P, Blodget B, Mason J, Young J, Bridgford B (2006) Invited paper: enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of Xilinx FPGAs. In: FPL, pp 1–6. IEEE
70. Marshall A, Stansfield T, Kostarnov I, Vuillemin J, Hutchings B (1999) A reconfigurable arithmetic array for multimedia applications. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays, FPGA '99. ACM, New York, pp 135–143
71. McMurchie L, Ebeling C (1995) Pathfinder: a negotiation-based performance-driven router for FPGAs. In: Proceedings of the third international ACM symposium on field-programmable gate arrays, FPGA '95, pp 111–117. doi: 10.1109/FPGA.1995.242049
72. Mehta N (2015) Ultrascale architecture: highest device utilization, performance, and scalability, WP455 (v1.2), October 29, 2015
73. Mei B, Vernalde S, Verkest D, De Man H, Lauwereins R (2003) Adres: an architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In: Cheung PYK, Constantinides G (eds) Field programmable logic and application. Lecture notes in computer science, vol 2778. Springer, Berlin/Heidelberg, pp 61–70
74. Mei B, Vernalde S, Verkest D, De Man H, Lauwereins R (2003) Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In: Proceedings of the conference on design, automation and test in Europe – DATE '03, vol 1. IEEE Computer Society, Washington, DC, p 10296
75. Mei B, Vernalde S, Verkest D, Lauwereins R (2004) Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: a case study. In: Proceedings of the 2004 design, automation and test in Europe conference and exhibition, vol 2, pp 1224–1229
76. Mei B, Lambrechts A, Mignolet JY, Verkest D, Lauwereins R (2005) Architecture exploration for a reconfigurable architecture template. IEEE Des Test Comput 22(2):90–101
77. MicroBlaze processor: MicroBlaze soft processor core (2012). http://www.xilinx.com/products/design-tools/microblaze.html
78. Mirsky E, DeHon A (1996) Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In: Proceedings of the IEEE symposium on FPGAs for custom computing machines, pp 157–166
79. Miyamori T, Olukotun K (1998) Remarc: reconfigurable multimedia array coprocessor. In: IEICE transactions on information and systems E82-D, pp 389–397
80. Moghaddam MS, Paul K, Balakrishnan M (2013) Design and implementation of high performance architectures with partially reconfigurable cgras. In: 2013 IEEE 27th international parallel and distributed processing symposium workshops PhD forum (IPDPSW), pp 202–211. doi: 10.1109/IPDPSW.2013.121
81. Moghaddam MS, Paul K, Balakrishnan M (2014) Mapping tasks to a dynamically reconfigurable coarse grained array. In: 2014 IEEE 22nd annual international symposium on field-programmable custom computing machines (FCCM), pp 33–33. doi: 10.1109/FCCM.2014.20
82. Moghaddam M, Balakrishnan M, Paul K (2015) Partial reconfiguration for dynamic mapping of task graphs onto 2d mesh platform. In: Sano K, Soudris D, Hübner M, Diniz PC (eds) Applied reconfigurable computing. Lecture notes in computer science, vol 9040. Springer, pp 373–382
83. Moon J, Moser L (1965) On cliques in graphs. Isr J Math 3(1):23–28. doi: 10.1007/BF02760024
84. MultiTrack interconnect in Stratix III devices (2009)
85. Oh T, Egger B, Park H, Mahlke S (2009) Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. SIGPLAN Not 44(7):21–30
86. Park S, Choi K (2011) An approach to code compression for CGRA. In: 2011 3rd Asia symposium on quality electronic design (ASQED), pp 240–245. doi: 10.1109/ASQED.2011.6111753

87. Park IC, Kyung CM (1991) Fast and near optimal scheduling in automatic data path synthesis. In: 28th ACM/IEEE design automation conference, pp 680–685
88. Park H, Fan K, Kudlur M, Mahlke S (2006) Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In: Proceedings of the 2006 international conference on compilers, architecture and synthesis for embedded systems, CASES '06. ACM, New York, pp 136–146
89. Park H, Fan K, Mahlke SA, Oh T, Kim H, Kim HS (2008) Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques, PACT '08. ACM, New York, pp 166–176
90. Park JJK, Park Y, Mahlke S (2013) Efficient execution of augmented reality applications on mobile programmable accelerators. In: 2013 international conference on field-programmable technology (FPT), pp 176–183. doi: 10.1109/FPT.2013.6718350
91. Patel K, Bleakley CJ (2010) Systolic algorithm mapping for coarse grained reconfigurable array architectures. In: Proceedings of the 6th international conference on reconfigurable computing: architectures, tools and applications, ARC'10, pp 351–357. Springer, Berlin/Heidelberg
92. Patel K, McGettrick S, Bleakley CJ (2011) Syscore: a coarse grained reconfigurable array architecture for low energy biosignal processing. In: 2011 IEEE 19th annual international symposium on field-programmable custom computing machines (FCCM), pp 109–112
93. Paul K, Dash C, Moghaddam M (2012) reMORPH: a runtime reconfigurable architecture. In: 2012 15th Euromicro conference on digital system design (DSD), pp 26–33
94. Pinson E, Prins C, Rullier F (1994) Using tabu search for solving the resource-constrained project scheduling problem. In: Proceedings of the 4th international workshop on project management and scheduling, Leuven, pp 102–106
95. Rau BR (1994) Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proceedings of the 27th annual international symposium on microarchitecture, MICRO 27. ACM, New York, pp 63–74. doi: 10.1145/192724.192731
96. Salefski B, Caglar L (2001) Re-configurable computing in wireless. In: Proceedings of 2001 design automation conference, pp 178–183
97. Sanchez E, Sterpone L, Ullah A (2014) Effective emulation of permanent faults in asics through dynamically reconfigurable FPGAs. In: 2014 24th international conference on field programmable logic and applications (FPL), pp 1–6
98. Sato T, Watanabe H, Shiba K (2005) Implementation of dynamically reconfigurable processor dapdna-2. In: 2005 IEEE VLSI-TSA international symposium on VLSI design, automation and test (VLSI-TSA-DAT), pp 323–324
99. Sato T, Watanabe H, Shiba K: Implementation of dynamically reconfigurable processor dapdna-2. In: 2005 IEEE VLSI-TSA international symposium on VLSI design, automation and test (VLSI-TSA-DAT), pp 323–324 (2005)
100. Singh H, Lee MH, Lu G, Kurdahi F, Bagherzadeh N, Chaves Filho E (2000) Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans Comput 49(5):465–481
101. Smit GJM, Kokkeler ABJ, Wolkotte PT, Hölzenspies PKF, van de Burgwal MD, Heysters PM (2007) The chameleon architecture for streaming DSP applications. EURASIP J Embed Syst 2007(1):1–10
102. SYSTEMS T (2016) Coarse-grained reconfigurable architecture. http://www.trentonsystems.com/blog/intel-cpu-computing/moores-law-pushing-processor-technology-to-14-nanometers/
103. Tehre V, Kshirsagar R (2012) Survey on coarse grained reconfigurable architectures. Int J Comput Appl 48(16):1–7. Full text available
104. Todman T, Constantinides G, Wilton S, Mencer O, Luk W, Cheung P (2005) Reconfigurable computing: architectures and design methods. IEE Proc Comput Digital Tech 152(2):193–207
105. Trimberger S (2015) Three ages of FPGAs: a retrospective on the first thirty years of FPGA technology. Proc IEEE 103(3):318–331

106. Tsu W, Macy K, Joshi A, Huang R, Walker N, Tung T, Rowhani O, George V, Wawrzynek J, DeHon A (1999) HSRA: high-speed, hierarchical synchronous reconfigurable array. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays, FPGA '99. ACM, New York, pp 125–134
107. University S (2016) SUIF compiler system. http://suif.stanford.edu/
108. Villarreal J, Park A, Najjar W, Halstead R (2010) Designing modular hardware accelerators in C with ROCCC 2.0. In: 2010 18th IEEE annual international symposium on field-programmable custom computing machines (FCCM), pp 127–134. doi: 10.1109/FCCM.2010.28
109. Vivado HLS: Xilinx Vivado Design Suite, Inc. (2012). http://www.xilinx.com/products/design-tools/vivado.html
110. Waingold E, Taylor M, Srikrishna D, Sarkar V, Lee W, Lee V, Kim J, Frank M, Finch P, Barua R, Babb J, Amarasinghe S, Agarwal A (1997) Baring it all to software: raw machines. Computer 30(9):86–93
111. Watkins MA, Albonesi DH (2010) ReMAP: a reconfigurable heterogeneous multicore architecture. In: Proceedings of the 2010 43rd annual IEEE/ACM international symposium on microarchitecture, MICRO '43. IEEE Computer Society, Washington, DC, pp 497–508
112. Xcell Journal Issue 52:. http://www.xilinx.com/publications/archives/xcell/Xcell52.pdf
113. Xilinx 7 series FPGA. www.xilinx.com. Accessed 27 Nov 2015
114. Xilinx (2012) Virtex 5 FPGA user guide
115. Xilinx (2014) 7 series FPGAs configurable logic block
116. Yoon JW, Shrivastava A, Park S, Ahn M, Jeyapaul R, Paek Y (2008) SPKM: a novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In: 2008 Asia and South Pacific design automation conference, pp 776–782. doi: 10.1109/ASPDAC.2008.4484056
117. Yoon J, Shrivastava A, Park S, Ahn M, Paek Y (2009) A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. IEEE Trans Very Large Scale Integr (VLSI) Syst 17(11):1565–1578