2. Coarse Grained Reconfigurable Architectures

While the first systems for reconfigurable computation featured fine grained FPGAs, it was soon discovered, that FPGAs bear different disadvantages for computational tasks. First, due to the bit-level operations, operators for wide datapaths have to be composed of several (bit-level) processing units. This includes typically a large routing overhead for the interconnect between these units and leads to a low silicon area efficiency of FPGA computing solutions. Also, the switched routing wires use up more power than hardwired connections.

A second drawback of the fine granularity is the high volume of configuration data needed for the large number of both processing units and routing switches. This implies the need for a large configuration memory, with an according power dissipation. The long configuration time, which is implied by this problem, makes execution models relying on a steady change of the configuration impossible.

As a third disadvantage, application development for FPGAs is very similar to VLSI design due to the programmability at logic level. The mapping of applications from common high-level languages is difficult compared to the compilation onto a standard microprocessor, as the granularity of the target FPGA does not match that of the operations in the source code. The standard way of application specification is still a hardware description language, which requires a hardware expert. In the following design process, the large number of processing units leads to a complex synthesis, which uses up much computation time.

Coarse grained reconfigurable architectures try to overcome the disadvantages of FPGA-based computing solutions by providing multiple-bit wide datapaths and complex operators instead of bit-level configurability. In contrast to FPGAs, the wide datapath allows the efficient implementation of complex operators in silicon. Thus, the routing overhead generated by having to compose complex operators from bit-level processing units is avoided.

Regarding the interconnects between processing elements, coarse grain architectures also differ in several ways to FPGAs. The connections are multiple bits wide, which implies a higher area usage for a single line. On the other hand, the number of processing elements is typically several orders of magnitude lower than in an FPGA. Thus, much fewer lines are needed, resulting in a globally lower area usage for routing. The lower number and higher granularity of communication lines allows also for communication resources, which would be quite inefficient for fine grained architectures. Examples for such resources are time-multiplexed buses or global buses, which connect every processing element.

In the past years, several approaches for coarse grained reconfigurable architectures have been published. In this chapter, several example architectures will be presented to give an overview over the developments in the area of coarse grain reconfigurable computing. A time line of the presented architectures is given in Table 2-1, together with basic properties of each architecture. The properties considered are:

- The basic interconnect structure
- The width of the datapaths
- The reconfiguration model

Publi-cation Year	Name of System	PE Arrangement	Reconfiguration Model	Datapath Width
1990	PADDI	Crossbar	Static	16 bit
1993	PADDI-2	Crossbar	Static	16 bit
1994	DP-FPGA	Mesh	Static	4 bit
1995	KressArray	Mesh	Static	32 bit
1996	Colt	Mesh	Dynamic	16 bit
1996	RaPiD	Linear Array	Mostly Static	16 bit
1996	MATRIX	Mesh	Dynamic	16 bit
1997	Garp	Mesh	Static	2 bit
1997	Raw	Mesh	Static	32 bit
1998	PipeRench	Linear Array	Dynamic	4 bit
1998	REMARC	Mesh	Static	16 bit
1998	MorphoSys	Mesh	Dynamic	16 bit
1999	CHESS	Mesh	Static	4 bit

Table 2-1: Time line of example coarse-grain reconfigurable architectures

The basic interconnect structure is determined by the global arrangement of the processing elements. This in turn is often motivated by the targeted applications for the architecture or by implementation considerations. Obviously, the type of the communication architecture has a direct impact on the complexity of application mapping. The predominating structure of the architectures presented here is a two-dimensional mesh. In other architectures, the processing elements are arranged in one or more linear arrays. The third interconnect structure found is a crossbar switch being used to connect processing elements. This allows basically arbitrary connections.

The width of the datapath ranges from two bits to 32 bits. The selection of the datapath width for an architecture is a trade-off between flexibility and efficiency. While a wide datapath allows an efficient implementation of the processing element for operations on a whole data word, the execution of operations on parts of a data word, which occur in certain applications, requires typically several execution steps, including the extraction of the relevant bit range. A smaller datapath allows the direct execution of such operations or implies less extraction overhead. On the other hand, operators for wider data words have to be composed of several processing elements, which inflicts routing overhead.

The reconfiguration model determines, when a new configuration is loaded into the architecture. For architectures featuring static reconfiguration, a configuration is loaded at the beginning of the execution and stays during the computation phase. When a new configuration has to be loaded, the execution must stop. The dynamic reconfiguration model allows a new configuration to be loaded while the application is running. This includes the case, that the execution relies on steady reconfiguration of the processing elements. Some architectures use normally static reconfiguration, but allow dynamic reconfiguration for special purposes like the configuration of unused parts for the next task, while a computation is running. Such architectures are denoted as static, as this remains the main reconfiguration model.

The presentation of the example systems is ordered by the interconnect structure. In the next subsection, nine mesh-based architectures are described. Then, two systems based on linear arrays are presented, and finally, two systems employing a crossbar switch are shown.

2.1 Mesh-Based Architectures

Mesh-based architectures arrange their processing elements in a rectangular array, featuring horizontal and vertical connections. This structure allows efficient parallelism and a good use of communication resources. However, the advantages of a mesh are traded for the need of an efficient placement and routing step. The quality of this step can have a remarkable impact on the application performance. Though, due to the relative low number of processing elements, the placement and routing is often much less complex than for e.g. FPGAs.

The arrangement of the processing elements encourages nearest neighbor links between adjacent elements as an obvious communication resource. Typically, longer lines are added with different lengths, which allow connections over several processing elements.

The first architecture called DP-FPGA [ChLe94] was published by D. Cherepacha and D. Lewis from the University of Toronto. This approach shows clearly its relation to FPGAs, featuring bit-sliced ALUs and a routing architecture quite similar to its fine-grained ancestors. The KressArray-I [Kres96] by R. Kress of University of Kaiserslautern is an approach with a very wide datapath, which required to reduce the communication resources in order to achieve a feasible chip design. However, the KressArray features also a serial bus as a high-level connection, which connecs all processing elements, and is globally scheduled. The KressArray-I was accompanied by a control unit and integrated into the Xputer prototype MoM-3 [Rein99]. The Colt [BAM96] architecture by R. Bittner, P. Athanas and M. Musgrove from Virginia State University features a single integer multiplier in addition to the mesh of processing elements. This architecture is a study for the wormhole runtime-reconfiguration computing paradigm, which relies on heavy dynamic reconfiguration. The MATRIX [MiDe96] system by E. Mirsky and A. DeHon of MIT is the first one to extend the processing elements to small processors, featuring local instruction memories in each element as well as small data memories. While the previous systems resembled stand-alone architectures (sometimes accompanied by a control unit), the Garp [HaWa97] architecture by J. Hauser and J. Wawrzynek from University of Berkeley is a system composed of a normal microprocessor with a reconfigurable coprocessor. It features lookup-table based processing elements with only two-bit wide datapaths. In fact, this architecture comes very close to an FPGA and is often considered to be fine-grained. It appears in this chapter, as the basic unit of reconfiguration is a whole row of processing elements, which form a kind of reconfigurable ALU. Though, the available routing resources of this approach classify it as a mesh-based architecture. The Raw machine paradigm [WTSS97], which has been developed by E. Waingold et al. from MIT features the most complex processing elements with a complete RISC processor, data and instruction memory, and a programmable switch for time-multiplexed nearest neighbor interconnects. In contrast to other approaches, which try to provide a more complex FPGA, this architecture tries to realize a mesh of less complex microprocessors, with the main responsibility for proper execution moved to the software environment at compile time. The REMARC [MiOl98a] architecture by T. Miyamori and K. Olukotun from Stanford University is another co-processor, consisting of 64 small processors with memory. The MorphoSys [LSLB99] machine by G. Lu et al. from Irvine University is also a hybrid system consisting of a RISC processor with a reconfigurable coprocessor. Here, the reconfigurable mesh is accompanied by a frame buffer for intermediate data. The last architecture shown is the CHESS [MVS99] array by A. Marshall et al. from Hewlett Packard Laboratories. This architecture features a unique PE arrangement in the form of a chess board, with embedded memories to support multimedia applications.

2.1.1 The Datapath FPGA

In 1994, Cherepacha and Lewis introduced the DP-FPGA (Datapath FPGA) [ChLe94], [ChLe96]. The DP-FPGA is intended for implementing regularly structured datapaths like the ones found in digital signal processing, communications, circuit emulation and special-purpose processor applications. The system comprises three basic components: A control logic, the datapath and memory. The control logic resembles a general purpose architecture adequate for implementing random logic. The memory is useful for large datapath designs. The datapath, which also occupies the most area of the design, is unique for the DP-FPGA and will be described in more detail in the following.

The targeted applications for the DP-FPGA may contain several datapaths of various widths, where also a small number of irregularities in some bit-slices is allowed. To provide both flexibility like bit-level devices and area advantages like ALU level architectures, the DP-FPGA features a medium granularity of four bits for both logic and routing resources. Thus, the basic logic block is made of four bit-slices, which are programmed identically using the same set of configuration bits. Each bit-slice comprises a lookup table, a carry chain and a four-bit register, the latter inspired by an experimental study by Rose et al [RFLC90].

An overview on the routing architecture of the DP-FPGA is shown in figure 2-1. In contrast to other architectures, the DP-FPGA features separate routing resources for data (horizontal) and control signals (vertical), both using four-bit buses. Vertical data routing is needed for shifts of multiples of four bits, connections to the I/O pads and connections spanning several rows, when the entire bit-slice does not fit in one row. However, such vertical routing connections are only possible in a limited way by using control wires for data transport.

To enable bit-granular shifts and irregularities in datapaths in spite of the four-bit granularity, shift blocks are included in the architecture, which can perform up and down shifts from zero to three bits. As these shifters can access data lines as well as control lines for source and destination, limited vertical data routing is also possible.

2.1.2 The KressArray-I

In 1995, the KressArray-I [Kres96] was introduced under the name rDPA (reconfigurable DataPath Architecture). The architecture was originally targeted as a reconfigurable ALU (rALU) for the MoM-3 prototype [Rein99] of the Xputer

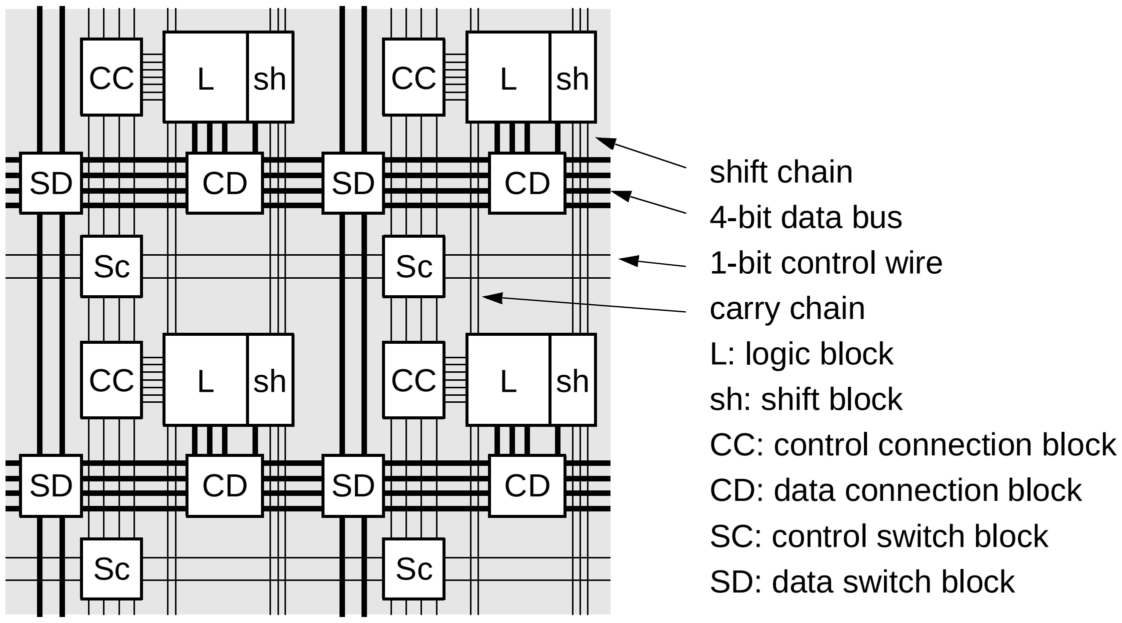

Figure 2-1: The DP-FPGA architecture

machine paradigm [HHW90], [HHRS91], [Hirs91], [AHRS94]. As Xputers are based on a strict separation of control path and datapath, the KressArray-I was used exclusively for application datapaths, which can include branches as well as while and do-while loops. Although the KressArray-I is dynamically partially configurable, the typical datapath would not change during execution.

The general structure of the KressArray-I system is depicted in figure 2-2. It consists of the array itself, resembling a mesh of processing elements called rDPUs (reconfigurable DataPath Units) and a controller unit. The controller manages the input and output of data streams to and from the array as well as the configuration process. For data I/O, the controller interfaces between the array and an external bus to the memory. Data streams are provided over this bus, controlled by the MoM-3 machine. For synchronization of input and output data streams, a data buffer register is provided. Further, the status of the array, like the finishing of the computation, is signaled by a status generation unit. The data words of the I/O streams may have to be distributed to arbitrary rDPUs. For this purpose, every rDPU can be addressed individually. An address generation unit delivers the correct addresses for the rDPU registers, so the data can be written to the correct location. For intermediate results of the computation, a register file with 64 entries of 34 bits each is integrated, which can also be used as a smart cache to save memory cycles. The global control of all the parts is done by the KressArray-I control unit, which executes a control program. The instruction set includes operations for data transfer between the external bus, the register file, and specific rDPUs in the array. The control program is loaded during the configuration time by the configuration unit, which manages also the configuration of the rDPU array. The controller is connected to the rDPU array by a serial global bus leading to every single rDPU, which is used for data transfers and for configuration.

The KressArray-I itself resembles a rectangular array of rDPUs. In the prototype implementation, a subarray of three by three rDPUs was implemented on one chip. The whole array was composed of two by four chips, resulting in an array of six by twelve (72) rDPUs. Each single rDPU contains an ALU with a datapath of 32 bits wide, featuring all integer operators of the C programming language. The operators are microprogrammed, so the operator repertory can be extended if necessary. The KressArray architectures are based on transport-triggered execution. That means, an operator starts its computation as soon as all input operands are available.

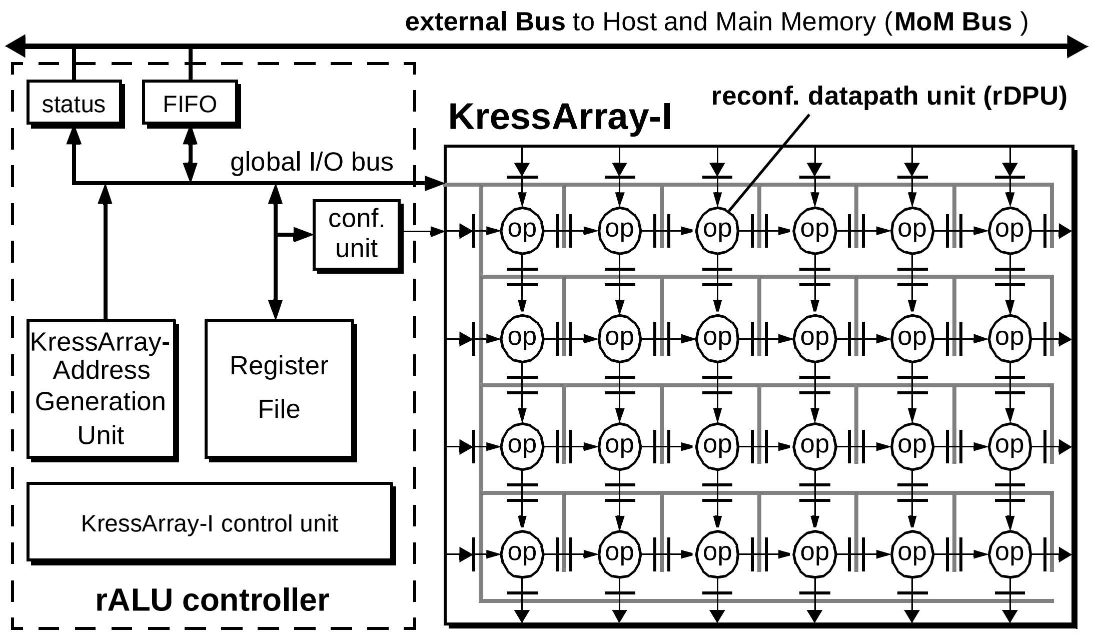
external Bus to Host and Main Memory (MoM Bus)
status
FIFO
global I/O bus
conf. unit
KressArray-Address Generation Unit
Register File
KressArray-I control unit
rALU controller
KressArray-I
reconf. datapath unit (rDPU)
op

Figure 2-2: KressArray-I architecture

The routing architecture of the KressArray-I features nearest neighbor connections from each rDPU to its southern and eastern neighbors. As an additional routing resource for longer connections, the serial global bus is available, which is also used for data I/O and the transfer of intermediate results between the array and the register file in the controller. This global bus is controlled by a static schedule, which is determined at compile-time. The rALU controller uses this schedule to manage the data transfers over the global bus.

2.1.3 The Colt System

The Colt system [BAM96], [BiAt97], presented in 1996 by Bittner, Athanas and Musgrove, combines concepts from FPGAs and data flow computing. Colt is mainly targeted for DSP applications, which are implemented by configuring pipelines or parts of pipelines onto the architecture. These pipelines are then used to process data streams. Thus, the Colt architecture can be seen as a Pipenet as discussed by Hwang [Hwan93]. The granularity of the data paths for routing and operations is 16 bits, with only limited possibilities for finer-grained operations. The Colt architecture relies highly on runtime reconfiguration (RTR). It uses Wormhole-RTR as an execution model. Hereby, the data streams to be processed carry a header with configuration information. This header holds the configuration data for both the routing and the functionality of all processing elements the data stream encounters on its way. Thus, the model employs the wormhole routing method known from computing networks [DaSe87].

The general structure of the Colt architecture is shown in figure 2-3. The main components are a mesh of IFUs (Interconnected Functional Units), a crossbar switch, an integer multiplier, and six data ports. The mesh of IFUs is the main computational facility. Each IFU features a 16-bit ALU with all logical operations with two inputs as well as a carry chain for add, subtract and negation. Furthermore, each IFU contains a barrel shifter to support multipliers and floating point arithmetic, a conditional unit for data-dependent flow control, and optional delay units for synchronization of pipelines. In addition to the ALU output, there is an auxiliary output passing through one of the inputs to support floating point arithmetic. Thus, the core of an IFU has two inputs and two outputs. The IFUs are connected by unidirectional nearest neighbor links in each direction with one outgoing and one incoming port at each side of the IFU. Additionally, each row and column features a so-called skip bus, which is a segmented bus featuring bidirectional local links, which can be merged. The IFUs in the northern and southern rows of the mesh differ from the inner IFUs in regard of their links at the edge, which are connected to the crossbar. In the northern row, each IFU has two inputs at the north side coming from the crossbar, while in the southern row, each IFU features a single output at the south side going to the crossbar. At the eastern and western edge, the two nearest neighbor links and the skip bus are connected to the opposing edge, thus forming a torus structure. The dedicated integer multiplier has two 16 bit inputs and two 16 bit outputs for the high and low word of the 32 bit result. The six data ports are bidirectional, each 20 bits wide with 16 bits for data and four bits for stream flow control. The crossbar is used to make nearly arbitrary connections between the mesh, the multiplier and the data ports.

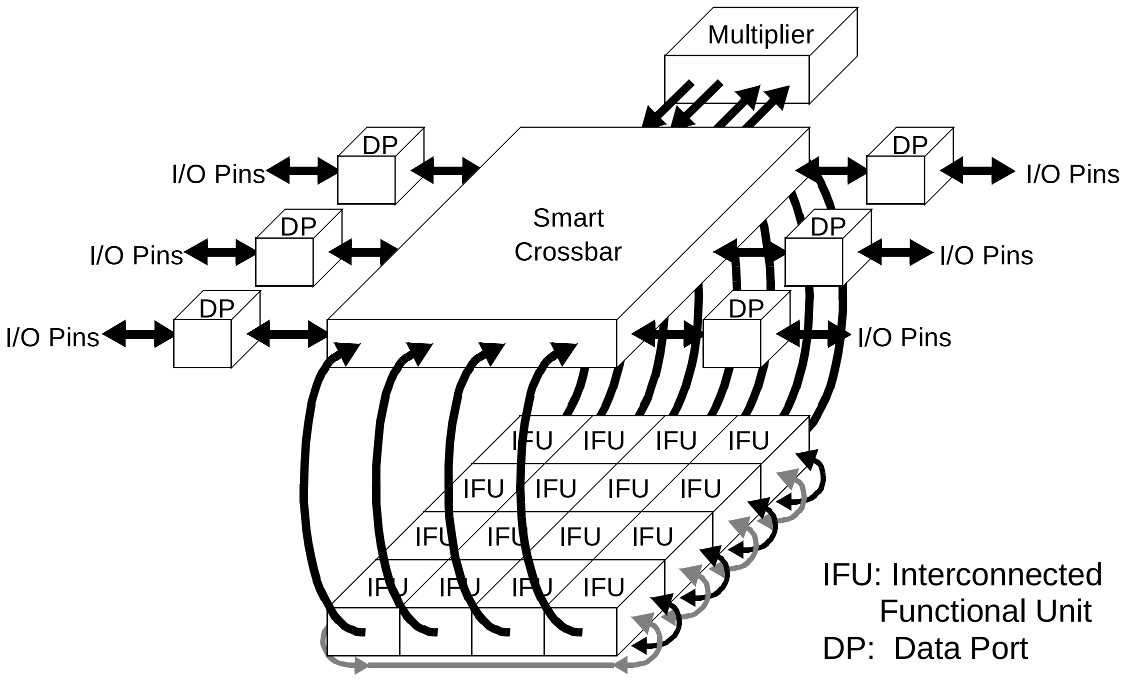

Figure 2-3: General structure of the Colt architecture

2.1.4 The MATRIX architecture

The MATRIX architecture by Mirsky and DeHon [MiDe96], [Mirs96], [Mirs97] was introduced in 1996. It is aimed at general computing. The main idea is to make the amount of the basic resources of a computing system, namely computational units, instruction storage and data memory, adaptable to application requirements. Although the reconfiguration is meant to happen at a per task basis, the processing elements can be programmed using instruction storage to change their behavior and data sources on a cycle-by-cycle basis.

The MATRIX comprises an array of identical Basic Functional Units (BFUs). Each BFU contains an 8-bit ALU, a 256 words of 8-bit memory and a control logic. The ALU features the standard set of arithmetic and logic functions and a multiplier. A configurable carry-chain between adjacent ALUs can be used to cascade ALUs for wide-word operations. The control logic can generate local control signals by several ways. A pattern matcher can be used to generate control signals from the ALU output data. A reduction network can be employed for control generated from neighbor data. Finally, a 20-input, 8-output NOR block may be used as half of an PLA to produce control signals. According to these features, a BFU can serve as an instruction memory, a data memory, a register-file-ALU combination, or an independent ALU function. Due to the routing resources of the MATRIX, instructions may be routed over the array to several ALUs. Thus, a high compression rate for instructions can be achieved.

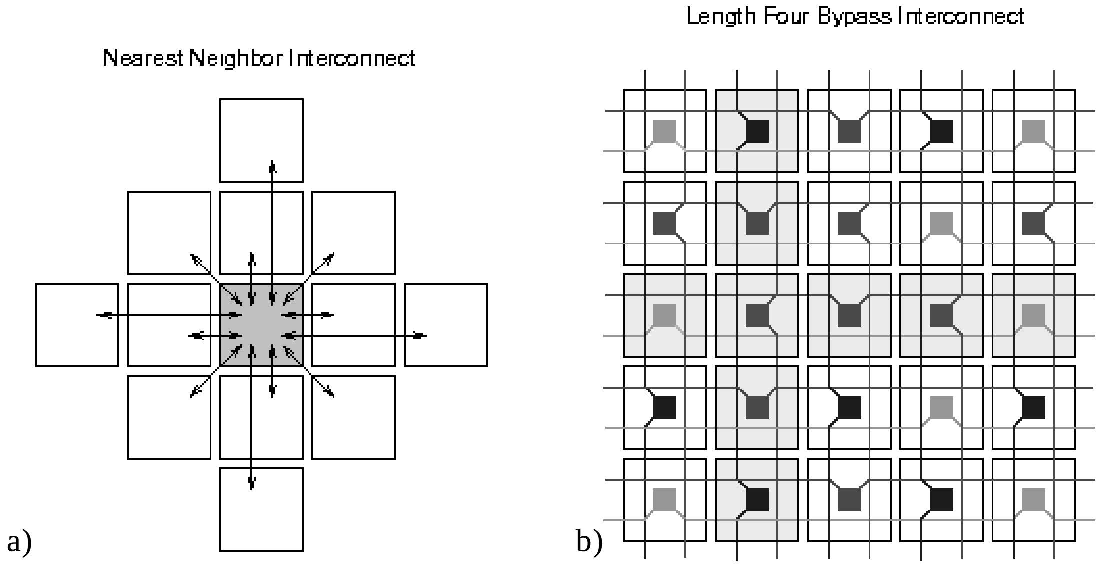
Nearest Neighbor Interconnect
Length Four Bypass Interconnect
a)
b)

Figure 2-4: Local interconnect and length four bypass interconnect of the MATRIX architecture
a) Nearest neighbor interconnect within distance two
b) Length four bypass interconnect

The routing architecture of the MATRIX, which is sketched in figur e2-4, consists of a hierarchical network with three levels, each consisting of 8-bit buses. The levels are nearest neighbor connections, length four bypass connections, and global lines. The nearest neighbor connections link each BFU to all of its neighbors within manhattan distance two (see left side of figure 2-4). The length four bypass connections (right side of figure 2-4) allows corner turns, local fan-out, medium distance interconnect, and some data shifting and retiming. The global lines span an entire row or column. There are four of them in every row and column.

2.1.5 The Garp Architecture

The Garp architecture concept [HaWa97] first published in 1997 by Hauser and Wawrzynek, consists of a standard Processor accompanied by a reconfigurable architecture, aiming to combine the advantages of both concepts. By including a microprocessor, the Garp is targeted for ordinary processing environments, with the reconfigurable array being only activated for acceleration of specific loops or subroutines. Although the processing elements of Garp resemble basically LUTs with a two bit datapath, the architecture is considered to be medium-grained, as the basic unit of reconfiguration is a cluster of several processing elements.

Garp consists of a main processor resembling a MIPS-II attached to a reconfigurable array. The instruction set of the processor has been extended with instructions to configure and control the array. An execution of the reconfigurable array is initialized by the processor specifying the number of clock cycles the computation in the array should last. Hereby, the processor clock and the array clock do not need to be the same.

The reconfigurable array consists of at least 32 rows of processing elements (called blocks) arranged in 24 columns. An overview of this structure is given in figure 2-5. The array is connected to the processor by vertical buses, with four 2 bit buses in each column. These buses are used for data I/O between the array and the memory or the processor. Memory accesses can be initiated by the reconfigurable array, but the data connection to memory is restricted to the central 16 columns. However, the array can access the same memory hierarchy as the main processor. The blocks in the leftmost column are dedicated control blocks, which are used for interfacing tasks like interrupting the processor or initiating the mentioned memory accesses. The basic unit of reconfiguration is one row, which can be seen to as a kind of reconfigurable ALU, being formed from relatively fine-grained blocks. To allow fast switching of configurations, the array also features a distributed cache with depth four, which stores the least recently used configurations. Although the array is partially reconfigurable, there can be only one active configuration at a time.

Each block in the reconfigurable array features a datapath of 2 bits width and can implement a function of up to four 2 bit inputs and two outputs. Wider datapaths can be formed by joining blocks in the same row. A carry chain is provided to support the build of wide adders, shifters or similar functions. The blocks comprise two-bit lookup tables with three inputs to implement the functions as well as two clocked registers. Three of the four inputs are fed into the lookup-tables, forming one output value, while the second output is a direct copy of the fourth input. The two outputs can be sent to three different paths leading to other blocks.

The routing architecture of the reconfigurable array consists of horizontal and vertical lines of different length. Conforming to the datapath of the blocks, these lines are 2 bits wide. Unlike other architectures, the horizontal and vertical lines are not segmented the same way, but are arranged in different patterns, being optimized for different purposes. Several short horizontal lines with segments spanning 11 blocks are provided to support multi-bit shifts along a row, while vertical wires with different segment lengths are used for connecting functional units laid out horizontally. In addition to this, some long horizontal lines span the whole array to enable the broadcast of control signals to the blocks of a single multi-bit operation.

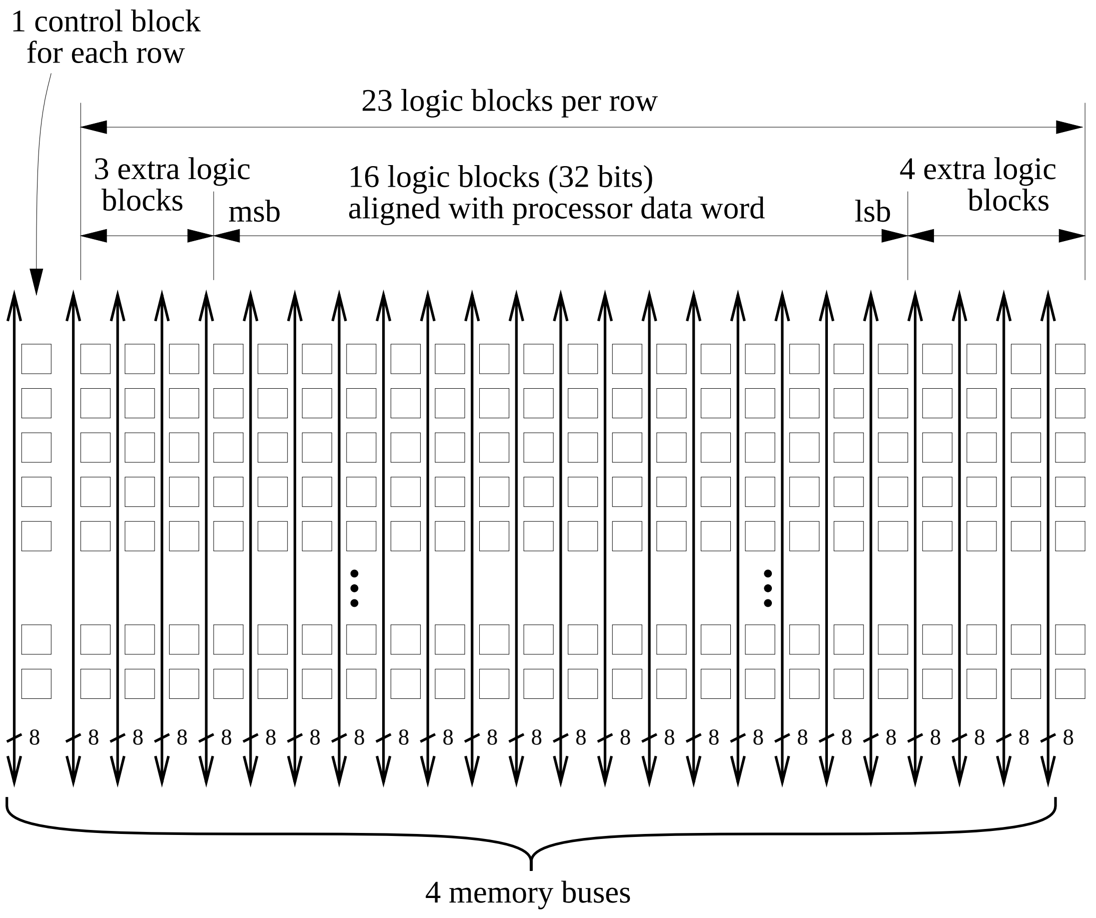
1 control block for each row
23 logic blocks per row
3 extra logic blocks
msb
16 logic blocks (32 bits) aligned with processor data word
lsb
4 extra logic blocks
8
4 memory buses

Figure 2-5: The Garp reconfigurable array architecture

2.1.6 The Reconfigurable Architecture Workstation

The Reconfigurable Architecture Workstation (RAW) project by the Computer Architecture Group of MIT was first published in 1997 [WTSS97], [AABF97] [Tayl99]. The idea of RAW is to provide a simple, highly parallel computing architecture composed of several repeated tiles connected among each other by nearest neighbor connections. The tiles comprise computation facilities as well as memory, thus implementing a distributed memory model. All architectural details are disclosed to the compilation framework. Especially, the processors lack hardware for register renaming, dynamic instruction issuing or caching, which is found in current superscalar processors. Due to the lack of these features, the execution model uses statically scheduled instruction streams generated by the compiler, thus moving the responsibility for all dynamic issues to the development software. However, RAW provides the possibility of flow control as a backup dynamic support, if the compiler should fail to find a static schedule.

An overview on a typical RAW system consisting of a RAW microprocessor with attached external memory is shown in figure 2-6. A RAW microprocessor is a homogeneous array of processing elements called tiles. The prototype chip features 16 tiles arranged in a 4 by 4 array. Each tile comprises a simple RISC-like processor consisting of ALU, register file and program counter, SRAM-based instruction and data memories, and a programmable switch supplying point-to point connections to nearest neighbors. Early concepts of the architecture [WTSS97] included also configurable logic to allow customized instructions and overcome the limitations of a wide datapath. This concept has been abandoned in the prototype implementation [Tayl99]. The CPU in each tile of the prototype is a modified 32 bit MIPS R2000 processor [Henn90] with an extended 6-stage pipeline, a floating point unit, and a register file of 32 general purpose and 16 floating point registers. Both the data memory and the instruction memory consist of 32 Kilobyte SRAM. While the instruction memory is uncached, the data memory can operate in cached and uncached mode. If the data memory should be too small for an application, the virtualization of memories has to be done by software, which is generated by the compiler.

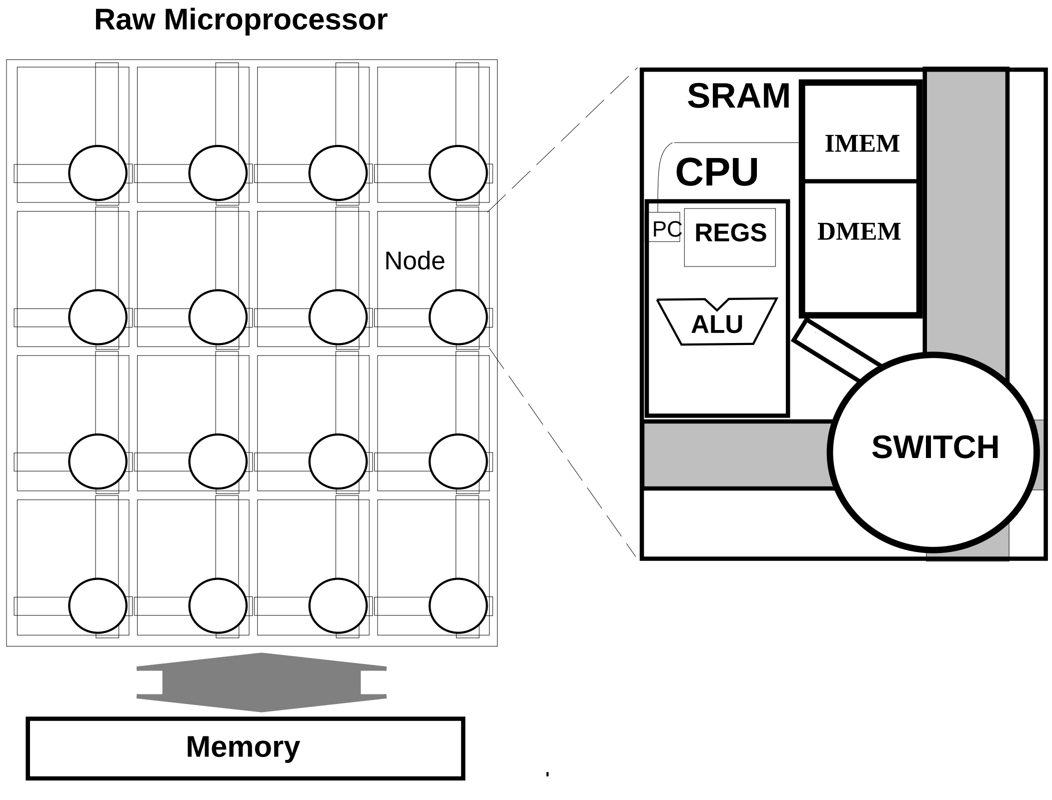

Figure 2-6: Overview on a RAW system comprising a RAW Microprocessor and external RAM

The interconnect of the RAW architecture is done over nearest neighbor connections being optimized for single data word transfer. Communication between tiles is pipelined over these connections and appears at register level between processors, making it different from multiprocessor systems. A programmable switch on each tile connects the four nearest neighbor links to each other and to the processor.

The RAW architecture provides both a static and a dynamic network, which are multiplexed on the physical structure described above. The static network is realized by a dedicated processor in the switch, which controls data transfers. The compiler has to generate code for this switch processor based on a static schedule for the data transfers. Thus, in the static network each data transfer is exactly determined at compile-time. For cases where a static schedule is not possible, a dynamic network is provided, which can perform data transfers in free cycles not used by the static network. This network uses wormhole routing for the data forwarding, with the routing information contained in a header put before the data to be transmitted.

2.1.7 The Reconfigurable Multimedia Array Coprocessor

The Reconfigurable Multimedia Array Coprocessor (REMARC) [MiOl98a], [MiOl98b] by Miyamori and Olukotun was introduced in 1998. The architecture is a reconfigurable coprocessor aimed at multimedia applications like video compression, video decompression, and image processing. The REMARC is tightly coupled to a MIPS-II ISA [Kane88] RISC processor, which supplies support for coprocessors. The processor has been extended by special instructions to access the REMARC architecture in the same way as a floating-point coprocessor is accessed.

An overview on the REMARC architecture is given in figure 2-7. The REMARC consists of an 8 by 8 array of processing elements (also called nanoprocessors), which is attached to a global control unit. The control unit manages data transfers between the main processor and the reconfigurable array and controls the execution of the nanoprocessors. It comprises an instruction RAM with 1024 entries, 64 bit data registers and four control registers.

The nanoprocessors consist of a local instruction RAM with 32 entries, an ALU with 16 bit datapath, a data RAM with 16 entries, an instruction register, eight data registers, four data input registers, and one data output register. The ALUs can execute 30 instructions, including add, subtract, logical operations, shift instructions, as well as some operations often found in multimedia applications like minimum, maximum, average and absolute and add. Each ALU can use data

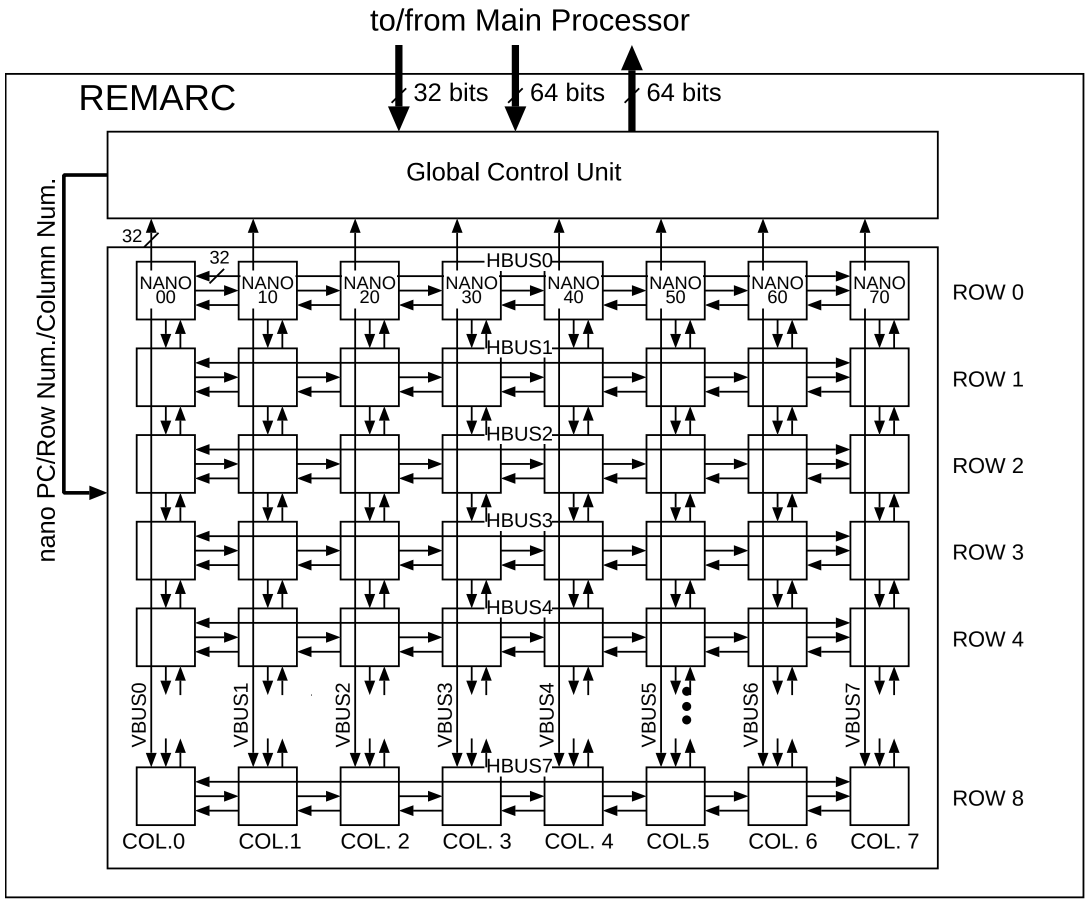

Figure 2-7: Overview on the REMARC architecture

from the data output registers of the adjacent processors via nearest neighbor connect, from a data register, a data input register, or from immediate values as operands. The result of the ALU operation is stored in the data output register.

The communication lines consist of nearest neighbor connections between adjacent nanoprocessors and additional horizontal and vertical buses in each row and column. The nearest neighbor connections allow the data in the data output register of a nanoprocessor to be sent to any of its four adjacent neighbors. The horizontal and vertical buses have double width (32 bits) and allow data from a data output register to be broadcast to processors in the same row or column respectively. Further, the buses can be used to transfer data between processors, which are not adjacent to each other. As can be seen in figure 2-7, the eight vertical buses are also used to connect the processor array to the global control unit and to transfer data from or to the 64 bit data registers inside the controller. Hereby, the 64 bit data word can be split up onto the eight buses.

During execution, the global control unit issues a global program counter value each cycle, which is broadcast to all nanoprocessors. Each nanoprocessor then uses this global value to index its local instruction memory and execute the according operation. To support SIMD operations for multimedia applications, it is possible to configure a whole row or column of processors to execute the same operation with a single instruction, thus saving instruction memory in the nanoprocessors.

2.1.8 The Morphoing System

The Morphoing System (MorphoSys) reconfigurable processing architecture [LSLB99], [SLLK98], [LSLB00], first published in 1998 is a parallel system on one chip comprising a standard processor tightly coupled to a reconfigurable co-processor. Though MorphoSys can be used for general computing due to the included processor, it is targeted to applications with inherent parallelism and a high level of granularity, which can be accelerated by the reconfigurable part. In spite of a preference for word-level applications, which is caused by the coarse granularity of the MorphoSys processing elements, the system is also flexible enough to support operations at bit level.

The complete MorphoSys chip comprises a core processor, a frame buffer, a DMA controller, a context memory, and an array of 8 by 8 reconfigurable cells. The core processor is a MIPS-like TinyRISC CPU with an extended instruction set for manipulation of the DMA controller and the reconfigurable array. The frame buffer is an internal data memory for blocks of intermediate results, similar to a data cache. This memory is logically organized into two sets, which can be used independently, so overlapping load and store is possible. Each set is itself divided into two banks with 64 rows of 8 bytes each, resulting in 128 times 16 bytes for the whole memory. The frame buffer is connected by a horizontal bus of 128 bits width to the reconfigurable array, so each row of cells gets a segment of 16 bits from the frame buffer. The DMA controller is used for both loading configuration data from the main memory and for data transfers between the main memory and the frame buffer. The context memory is used to store the configuration data for the reconfigurable array. This memory supports multiple contexts, which can be changed dynamically during execution. The configuration model of the reconfigurable cell array is restricted to the concurrent configuration of all cells in either one row or one column of the array by broadcasting a configuration word to all cells in the according row or column, which will then perform the same operation. Thus, the context memory features two blocks (one for the rows, one for the columns), each block containing eight configuration sets (one for each row or column) with sixteen context words.

The general structure of the configurable cell array is shown in figure 2-8 (the figure shows the same array twice, with different parts of the routing structure). The reconfigurable part of MorphoSys comprises an 8 by 8 array of mesh connected processing elements, also called reconfigurable cells. The array is divided into four quadrants of 4 by 4 cells each. A single reconfigurable cell features a 16-bit datapath, comprising an ALU-Multiplier, a shift unit, two input multiplexers, a register file with four 16 bit registers, and a 32 bit context register for storing the configuration word. The ALU-multiplier can perform the standard arithmetic and logical operations as well as a multiply-accumulate operation in a single cycle. It has four inputs, two over the input multiplexers, one from the output register, and one connected to the context register to load a 12 bit operand contained in the configuration word. The multiplexers are used to select operands from the register file, the data bus from the frame buffer, or from other cells over the interconnect structure.

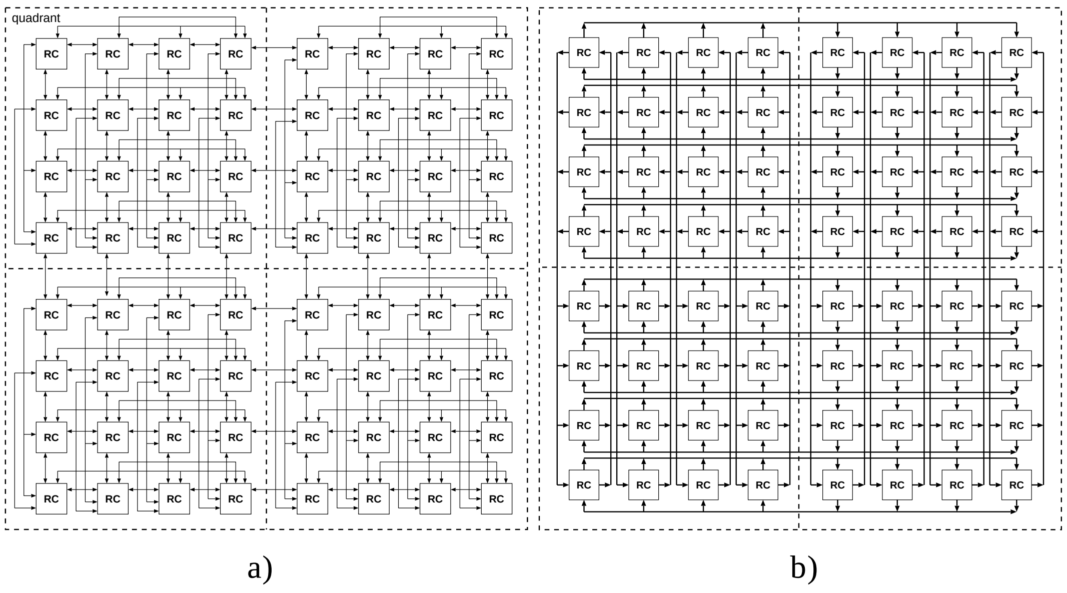

Figure 2-8: Architecture of the MorphoSys reconfigurable cell array:
a) Nearest neighbor connects and inter-quadrant lines
b) Same architecture showing buses spanning the whole array

The interconnect network of MorphoSys features three layers. First, all cells are connected to their four nearest neighbors. Second, each cell can access data from any other cell in the same row or column of the same array quadrant. These first two layers of interconnect are depicted in figure 2-8a. The third layer of interconnect, which is shown in figure 2-8b, consists of buses spanning the whole array and allowing transfer of data from a cell in a row or column of a quadrant to any other cell in the same row or column in the adjacent quadrant. In addition to the shown interconnect resources, two horizontal 128 bit buses connect the array to the frame buffer: An operand bus for data transfer to the array, and a result bus for data transfer to the frame buffer. Like mentioned in the description of the frame buffer, the 128 bits of the buses are shared among the eight rows of the array, giving each row access to a 16 bit segment of each bus.

2.1.9 The CHESS Array

The CHESS architecture [MVS99] by Hewlett Packard Laboratories was introduced in 1999 and is intended for the implementation of multimedia applications. CHESS is aimed as a component of an ASIC or processor datapath.

An overview of the CHESS array is given in figure 2-9. The architecture consists of ALUs and switchboxes. These components are arranged in a chessboard-like pattern with each switchbox being surrounded by four ALUs and each ALU being surrounded by four switchboxes. To support applications with memory requirements for intermediate results, embedded RAMs can be provided organized in vertical stripes inside the array. Additionally, switchboxes can be converted to 16 word by 4 bit RAMs if needed.

The ALUs and all routing resources are four bits wide. The ALUs feature two inputs and one output for four-bit data words, as well as one single-bit input and one output for carry. The instruction set features 16 operations, including add and subtract, nine logical operations, two multiplex operations and three tests using the carry bit as condition output. It is possible to connect the data output of an ALU to the configuration input of another one. Thus, the functionality of an ALU can be changed in a limited way on a cycle-per-cycle basis during runtime by configuration data generated inside the array. However, a way for partial configuration from outside is not provided. In order to enable operations, which cannot be mapped onto an ALU or to make fine-grained interconnections not supported by the four-bit wiring, the RAMs in the switchboxes can also be used as a 4-input, 4-output LUT.

The routing structure of the CHESS array consists of segmented four-bit buses of different length. There are 16 buses in each row and column of the array. Four buses are for local connections, spanning one switchbox. Due to the chessboard layout of the array, these local connections are sufficient to link an ALU to all of its eight surrounding neighbors, as shown in figure 2-9 by the grey arrows. In addition to the four local buses, there are two buses of length 1, crossing one ALU and half a switchbox on either side. Longer buses have lengths of powers of 2. A bus of length 2^n crosses 2^n ALUs and half a switchbox on either side. These buses are laid out in pairs with the ends of one bus in the middle of the other one. There are four buses of length 2, and two buses of length 4, 8 and 16 respectively.

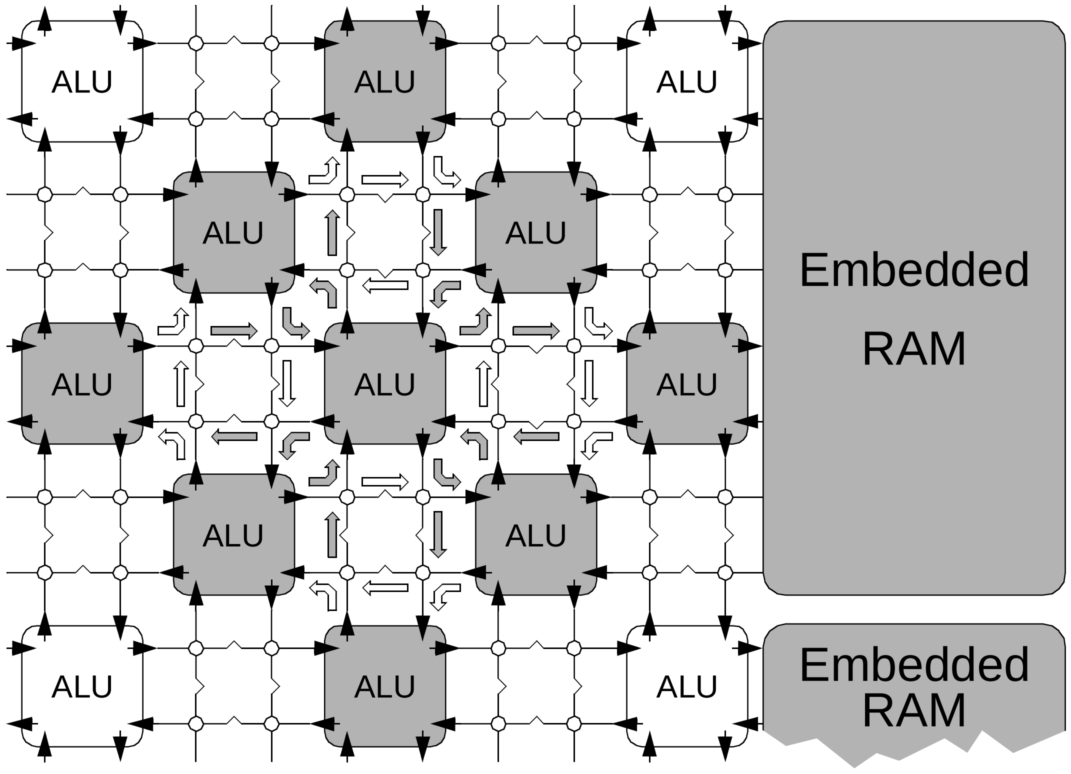

Figure 2-9: CHESS array architecture overview (excerpt)

2.2 Architectures Based on Linear Arrays

A more uncommon arrangement of processing elements is one or several linear arrays, typically featuring connections between the neighbors. This structure is motivated by the idea to map pipelines onto it, with each processing element featuring one pipeline stage. This works well for linear pipelines without forks. If there are forks in the pipeline, which would need a two-dimensional realization, additional routing resources are needed, which are normally provided by longer lines spanning the whole or a part of the array, often being segmented. The linear structure allows a direct mapping of pipelines, with the mentioned problems for forks.

Two architectures appear as examples for a linear array structure. The RaPiD architecture [ECFF96] by C. Ebeling, D. Cronquist, and others, is based on the idea to provide a number of different computing resources, like ALUs, RAMs, Multipliers, and Registers. Unlike most mesh-based architectures, the resources are not evenly distributed. Instead, the fact that the architecture extends only in one direction allows to provide e.g. three times as much ALUs than multipliers. While RaPiD uses a mostly static reconfiguration model, the PipeRench architecture [CWGS98] relies on dynamic reconfiguration, allowing the configuration of a processing element to change in each execution cycle. Besides mostly unidirectional nearest neighbor connects, this architecture provides also a global bus for data transfer.

2.2.1 The Reconfigurable Pipelined Datapaths

The Reconfigurable Pipelined Datapath (RaPiD) described in [ECFF96], [ECF96], [CFFF99] was developed in 1996 by Ebeling et al. It is aimed to speed up highly regular, computation-intensive tasks by implementing deep application-specific pipelines.

RaPiD provides a number of computational resources like ALUs, multipliers, registers and memory modules, which can implement application-specific datapaths. These resources are arranged in a linear array, which is configured to form a mostly linear pipeline. This array is divided into identical basic cells, which are replicated to form a whole array. The RaPiD-1 prototype features 16 cells, each cell containing an integer multiplier, three integer ALUs, six general-purpose datapath registers and three local memories with 32 entries. All datapaths as well as the memories are 16 bits wide. The ALUs are capable of the usual logical and arithmetic integer operations and can be chained for wide-integer operations. The multiplier processes two 16 bit input data words to one 32 bit output word. The datapath registers can be used to store constants or temporary values, to implement additional multiplexers, to support routing, and for additional pipeline delays. The local memories can be used for block oriented data processing. Each memory has a specialized datapath register featuring an incrementing feedback path. An overview on the general basic cell architecture is given in figure 2-10.

The routing architecture of the RaPiD consists of several parallel segmented 16 bit buses, which span the whole array. The length of the bus segments varies in different tracks. In some tracks, adjacent bus segments can be merged by configurable bus connectors (see figure 2-10), which can also implement pipeline delays if needed. The functional units can access all buses for reading and writing data. For reading from the bus, each functional unit features an input multiplexer of *n*:1. One multiplexer input is reserved for constant zero, another one for a feedback line (the latter one does not appear any more in [CFFF99]). Thus, *n*-2 inputs are for the buses, and therefore the number of buses is also *n*-2. The output drivers can put the result of a functional unit onto one or more bus segments.

The data to be processed enters and leaves RaPiD via I/O streams at each end of the datapath. For this purpose, the linear array is accompanied by a stream generator, which also contains the memory interface. The stream generator

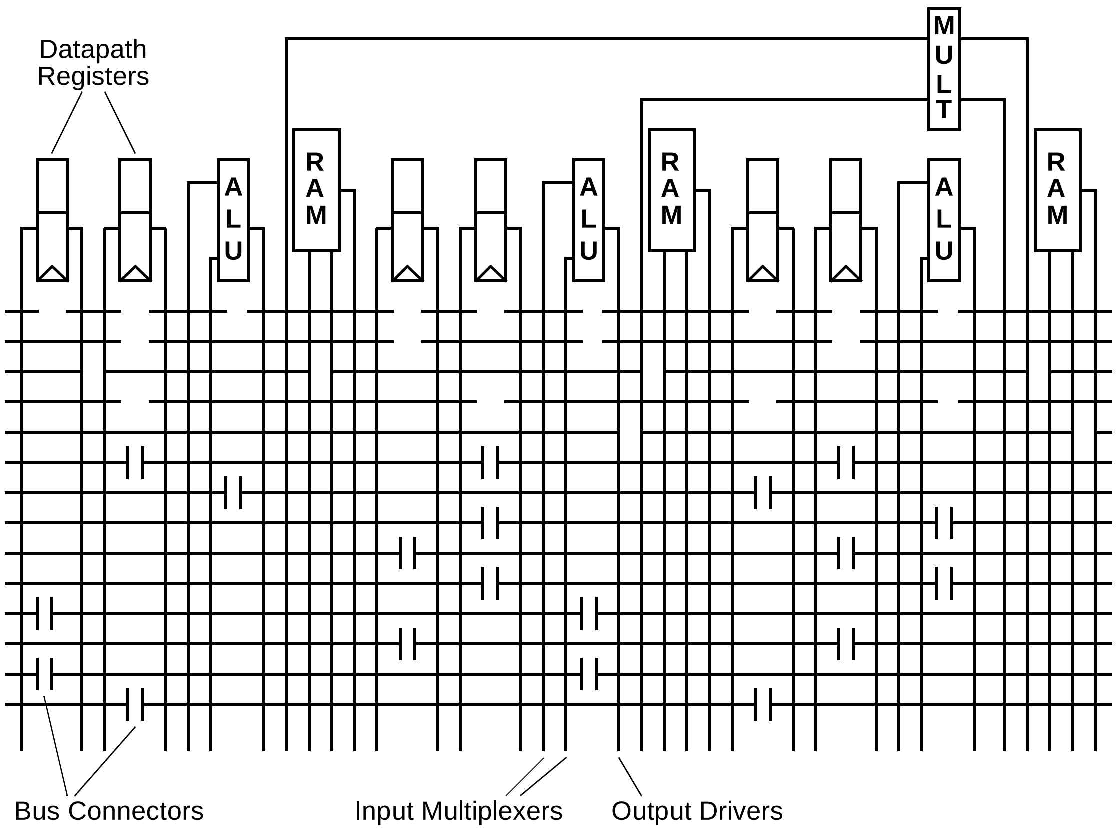

Figure 2-10: Overview on the general basic cell architecture of RaPiD

contains address generators, which are optimized for nested loop structures, associated with FIFOs. The address sequences for the generators are determined at compile-time.

For configuration, the RaPiD architecture uses a mix of static configuration and dynamic control. The biggest part of control signals (e.g. for tristate drivers and bus connectors) is determined at compile time and configured statically, while about a quarter of the signals (e.g. for multiplexers and ALUs) are determined dynamically during run time. This is done by a programmed controller along with a configurable path with buses and some logic gates to distribute the instruction bits over the architecture. The configurable control path follows the same scheme like the routing network for the data, consisting of several segmented buses.

2.2.2 The PipeRench Architecture

The PipeRench architecture [CWGS98], [GSMB99], [BuGo99] is a coprocessor to speed up pipelined applications, introduced in 1998 by Goldstein, Schmit et al. It aims to adapt the concept of virtual memory to reconfigurable hardware, resembling a virtual hardware. The idea is to provide a hardware comprising several reconfigurable pipeline stages (called stripes in PipeRench terminology). An application pipeline is mapped onto the PipeRench, whereby the physical hardware is kept transparent to the application. This is done by removing temporary unused pipeline segments from the hardware, if necessary, thus implementing a time multiplexing of the physical computation resources. According to this concept, the PipeRench relies highly on fast partial dynamic pipeline reconfiguration as well as run time scheduling of both configuration streams and data streams.

An overview of the architecture is given in figure 2-11. The PipeRench consists of a reconfigurable fabric, a configuration memory, a state memory, an address translation table (ATT), four data controllers, a memory bus controller and a configuration controller. The configuration memory is on-chip and connected to the fabric with a wide bus, enabling one complete pipeline stage to be configured with one memory read. In a prototype described in [CWGS98], the configuration memory has a capacity of 256 entries with 1024 bits each. The state memory is used to save the current register contents of a stripe, if it is swapped out. The address translation table is used for storing the address in the state memory for the state of a given stripe. The four data controllers are connected to the fabric by four global buses. Each of the buses is dedicated to either the storing of a stripe state, the restoring of a state, the input of data or the output of data. The controllers may all be used for data input or output where they interface between the fabric and the memory bus controller. For the task of state storing and restoring, only the two rightmost controllers may be employed, which act as interface between the fabric and state memory, when used this way. The data controllers are also responsible for the generation of address sequences for both input and output data streams. This task includes the run time scheduling of memory accesses. The address sequences that can be generated are affine functions of the loop index. The data controllers may also contain caches to exploit locality or FIFOs to enhance burst mode memory accesses. The memory bus controller handles access of the data controllers to off-chip memory by arbitrating accesses to a single memory bus. This unit is also used to load the configuration memory. The configuration controller is responsible for the tasks of interfacing the fabric to the host, mapping of the configuration words to the hardware, run time scheduling (including time multiplexing and handling of the virtualization), and managing the configuration memory.

The reconfigurable fabric of the PipeRench allows the configuration of a pipeline stage in every cycle, while concurrently executing all other stages. The architecture of the reconfigurable fabric is sketched in figure 2-12. The fabric consists of several (horizontal) stripes. Each stripe is composed of interconnect and processing elements (PEs in figure 2-12), which contain registers and ALUs. The ALUs are implemented as 3-input lookup tables (one for each datapath bit).

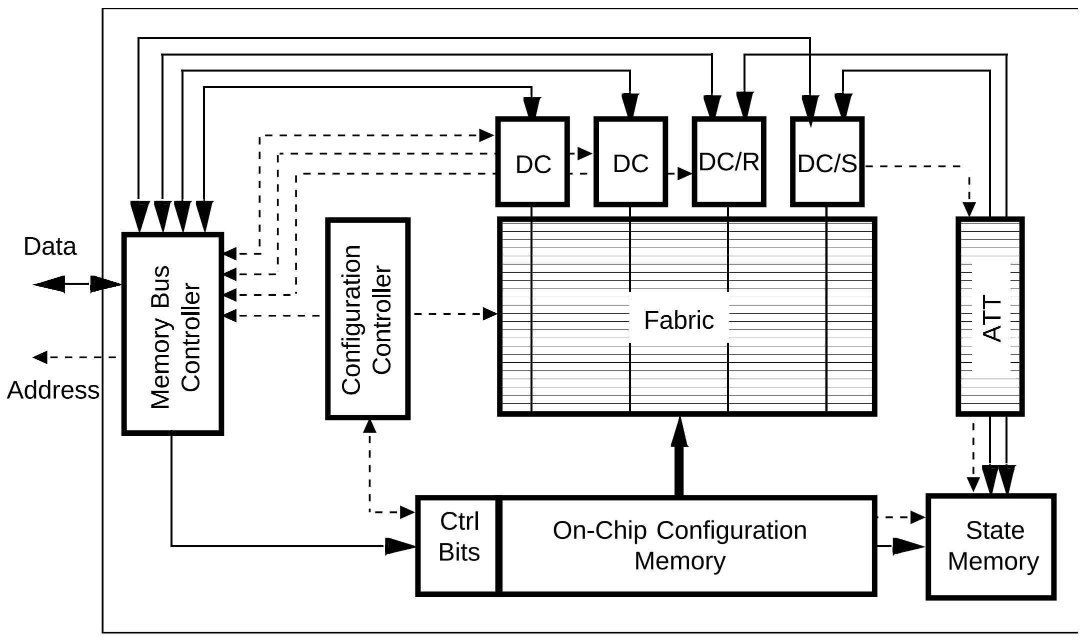

Figure 2-11: Overview on the PipeRench architecture

The three inputs of each look-up table are divided into 2 data inputs and one control input. The ALU in each PE is accompanied by a barrel shifter for left shift and extra circuitry for carry chains, zero detection, etc. The carry lines of neighboring ALUs in one stripe can be cascaded to form wider ALUs. Furthermore, In the prototype implementation, a stripe provides a total datapath of 128 bits, which is divided into 32 ALUs with 4 bits each. The whole fabric features 28 stripes.

The interconnect scheme of PipeRench features local interconnect inside a stripe as well as local and global interconnect between stripes. The global interconnect is realized by the four global buses and is used for data input and output to and from the pipeline. The local interconnect is implemented by an according network provided in each stripe. This network, which is implemented as a word-level full crossbar, allows each PE to access operands from outputs of the previous stripe as well as from any other PE in the same stripe. However, no connections lead to a previous stripe, thus no long feedback loops are possible, as any cycle must be contained within one stripe.

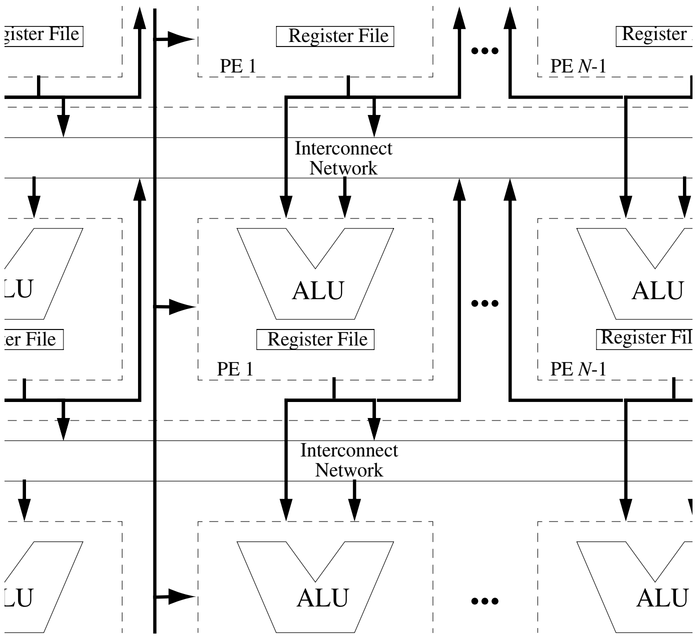

Figure 2-12: Architecture of the PipeRench reconfigurable fabric

2.3 Crossbar-Based Architectures

A full crossbar switch allows arbitrary connections between processing elements, making it the most powerful communication network. The routing task is thus a simple operation. However, due to the high implementation cost of a full crossbar, the two architectures presented in this section implement a reduced crossbar, which again implies the need for a routing process to connect the processing elements.

The example architectures were both published by J. Rabaey et al. from University of Berkeley. The PADDI architecture [ChRa90] was meant for the fast prototyping of DSP datapaths, featuring eight processing elements connected by a multilayer crossbar. The PADDI-2 [YeRa93], which is a successor to the PADDI, featured 48 processing elements. While in the PADDI, all PEs were connected to the crossbar, such an implementation would be too expensive for 48 PEs. Thus, a hierarchical interconnect structure was chosen, featuring linear arrays of PEs forming clusters, and a restricted crossbar for the interconnect of the clusters. This hierarchical interconnect structure has again an impact on the routing for this architecture.

2.3.1 The Programmable Arithmetic Device for Digital Signal Processing

The PADDI (Programmable Arithmetic Device for Digital Signal Processing) architecture was first described in 1990 by Chen and Rabaey [ChRa90], [ChRa92], [CGNS92a]. PADDI addresses the problem of rapid prototyping for computation-intensive DSP data paths.

The PADDI architecture consists of clusters of eight arithmetic execution units (EXUs), which are connected via a crossbar switch. Each EXU is accompanied by a local controller. Clusters can communicate with each other over 128 I/O pins connected to the crossbar switch. The prototype implementation of PADDI featured four clusters on one chip. An overview of the PADDI architecture is given in figure 2-13.

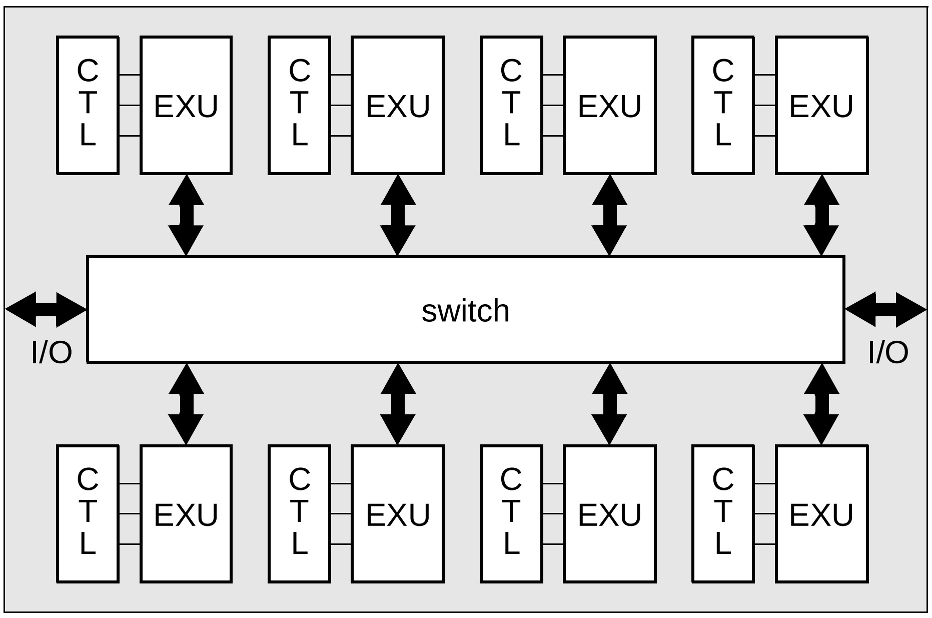

Figure 2-13: Overview on the PADDI architecture

The EXUs contain the computational units as well as storage facilities. Each EXU features one 16 bit ALU, which supports addition, subtraction, accumulation, comparison, maximum-minimum and right shifting using a logarithmic shifter. Arithmetic can be performed for signed or unsigned numbers.To affect program flow, a status flag (a Š b) is provided. At the ALU inputs, two 16 bit wide register files with six registers each are provided, which can be used to store temporal data. The registers are dual-ported and can be configured as delay lines for pipelining and retiming of operations. At the output of each EXU, an optional pipeline register is available. Two EXUs can be concatenated for double arithmetic accuracy of 32 bits.

For the interconnection of the EXUs, a crossbar switch is employed, which is used for routing both data and status flags. While the data routing can be changed in each cycle dynamically at run time, the routing of the status flags is static and determined at compile time.

The control of the architecture is done hierarchically in order to handle the relatively high instruction bandwidth for all eight EXUs. On a first level, an external global controller broadcasts 3 bit global instructions to each EXU. The instructions from the global controller are decoded on a second level by local control units in each EXU into a 53 bit instruction word. The local controllers in each EXU are SRAM-based nanostores, which are serially configured at setup-time. Each SRAM contains eight words, allowing eight different operations to be performed. The execution units and the external controller communicate status information to affect both the local and global control flow.

2.3.2 The PADDI-2 Architecture

In 1993, a successor to the PADDI architecture, called PADDI-2 [YeRa93], [Yeun95], [YeRa95] was developed at Berkeley University. This architecture featured a data-driven execution mechanism. Although the basic architectural principles of the original PADDI were kept, the PADDI-2 has several differences. An overview of the PADDI-2 is shown in figur e2-14.

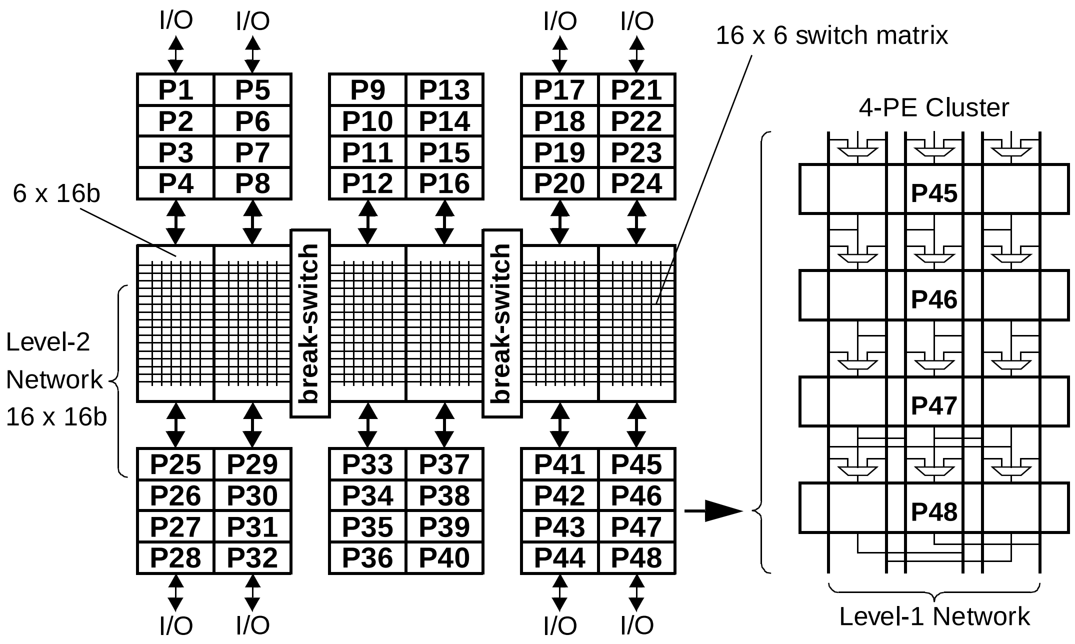
I/O
16 x 6 switch matrix
4-PE Cluster
P1 P5 P2 P6 P3 P7 P4 P8
P9 P13 P10 P14 P11 P15 P12 P16
P17 P21 P18 P22 P19 P23 P20 P24
6 x 16b
break-switch
Level-2 Network 16 x 16b
P25 P29 P26 P30 P27 P31 P28 P32
P33 P37 P34 P38 P35 P39 P36 P40
P41 P45 P42 P46 P43 P47 P44 P48
P45
P46
P47
P48
Level-1 Network

Figure 2-14: The PADDI-2 architecture

The PADDI-2 has been implemented on a chip featuring 48 processing elements (or EXUs in original PADDI terminology) with a 16 bit datapath. Each processing element features a 16 bit ALU with 12 instructions including add, add with carry-in, subtract, subtract with carry-in, logical or, logical and, logical xor, logical not, arithmetic up shift, arithmetic down shift, booth multiply and select. Multiplication can be done in eight cycles on a single processor. Alternatively, eight processors can be pipelined, resulting in one multiplication product in every cycle. Using the carry-in, the add and subtract operations can be concatenated for datapaths with a width of a multiple of 16 bits. Each processing element provides also six general purpose registers, which can be configured as up to three input queues for the ALU, three scratch pad registers and an instruction memory (nanostore) with eight entries as well as a control unit.

In contrast to the original PADDI architecture, the processing elements of PADDI-2 are packed in 12 clusters of four elements each. Thus, the interconnect is hierarchical with two levels. On the first level, each cluster features six 16 bit data buses and four one bit control buses, providing intra-cluster interconnect. On the second level, 16 data buses and eight control buses can be used for inter-cluster interconnect. They can be broken up into shorter segments by reconfigurable switches. The eight clusters in the corners of the chip provide additional I/O connections to the outside. As the crossbar switch of the original PADDI for arbitrary connections between processing elements has been given up, the placement and routing for PADDI-2 is more sophisticated.

The PADDI-2 architecture employs a distributed data-driven control strategy [YeRa92] implemented by a hardware handshake protocol for synchronization. If a sender or a receiver is not ready for a data transfer, all participants are stalled. By this mechanism, the need for global synchronization and retiming during the mapping process is eliminated.