Chapter 11

MORPHWARE AND CONFIGWARE

Reiner Hartenstein
TU Kaiserslautern

Abstract

This chapter introduces morphware as the basis of a second machine paradigm, which mainly has been introduced by the discipline of embedded system design, targeting the system on chip (SoC). But more recently SoC design is adopting more and more computer science (CS) mentality and also needs the services of computer science (CS) professionals. CS is going to include the morphware paradigm in its intellectual infrastructure. The time has come to bridge the traditional hardware–software chasm. A dichotomy of two machine paradigms is the road map to upgrade CS curricula by evolution, rather than by revolution. This chapter mainly introduces morphware platforms as well as their models and architectures.

1 INTRODUCTION

Morphware [1] [2] is the new computing paradigm, the alternative RAM-based general-purpose computing platform model. The traditional hardware–software chasm distinguishes software running on programmable computing engines (microprocessors) *driven by instruction streams* scanned from RAM, as well as application-specific fixed hardware like accelerators that are not programmable after fabrication. The operations of such accelerators are primarily *driven by data streams*. Such accelerators are needed because of the microprocessor's performance limits caused by the sequential nature of its operation—by the *von Neumann bottleneck*.

John von Neumann's key achievement has been the simple common model called the *von Neumann machine paradigm* ([3, 4], von Neumann has not invented the computer). His model provides excellent guidance in CS education and also narrows the almost infinite design space. However, the contemporary common model of computing systems is the cooperation of the (micro)processor and its accelerator(s), including an interface between both (Figure 11.1). This model

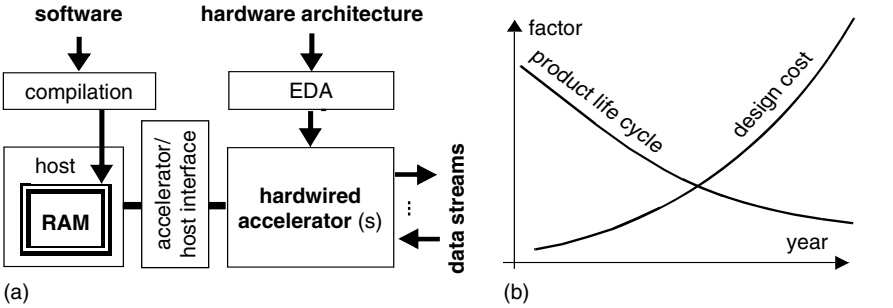


Figure 11.1. The common model of computer systems. (a) Embedded microprocessor model; (b) impact of the second design crisis.

holds not only for embedded systems but also for the PC needing accelerators not only for running its own display. Accelerators are a kind of slaves. The operating system and other software are running on the microprocessor, which is the host and master of the accelerators. The host may send parameters (for example, mode select, start, stop, reset, etc.) and receive interrupts and some result data.

The host operation is *instruction–stream driven*. The instruction stream is managed by the program counter inside the host processor. The accelerator usually has no program counter; its operations are *data–stream driven* (see *data stream* interface in Figure 11.1). Not only in terms of efficiency, this model especially makes sense for data-intensive applications, where *multiple data streams* are interfaced to the accelerator Figure 11.1. Only a few very sophisticated architectures are difficult to map onto this model. In the case of computation–intensive applications with very low data traffic to/from the accelerator, a single data stream generated by the host may be sufficient. This model (for details, see the next section and Section 3.1 ff. is as simple as the host's von Neumann (vN) model, which is also important for educational purposes (for details, see also Section 3.1).

By the way, *data–stream–driven computing (or flowware–based computing*: this term will be defined later) had already been used implicitly by the first programmers. In a von–Neumann–based, instruction–stream–driven environment, the less efficient *detour* over the application control-structures has been the only viable solution. However, by avoiding the (vN) bottleneck, a data–stream–driven environment permits much more direct and efficient solutions. For more detailed explanations, see Section 11.3.

vN processor programming is supported by compilers, whereas traditional accelerator development has been and is done with electronic design automation (*EDA*), tools [5]—for acronyms, see Figure 11.2.

More recently, however, such accelerator design has been affected by the second *design crisis* (Figure 11.1b). Compared with microprocessor design, the SoC design productivity in terms of gates per day is slower by a factor of about 10^{-4} [6]. Another symptom of increasing design implementation problems and the *silicon technology crisis* has been the drastically decreasing number of wafer starts for newer technology fabrication (Figure 11.3a) and the still decreasing low number of *application-specific IC* design starts (Figure 11.3c). Another major cost fac-

AM	anti-machine (DS machine)	ISP	instruction stream processor
AMP	data stream (AM) processor	LSI	Large Scale ICs
ASIC	application-specific IC	LUT	Look-Up Table
asMB	autosequencing Memory Bank	MCGA	Mask-Configurable Gate Array
BIST	Built-In Self-Test	MPGA	(see MCGA)
CFB	Configurable Function Block	MSI	Medium Scale ICs
CLB	Configurable Logic Block	MW	Morphware
COTS	commodity off the shelf	PC	Personal Computer
CPU	"central" processing unit: DPU (with instruction sequencer)	PS	Personal Supercomputer
		pSoC	programmable SoC
cSoC	configurable SoC	rDPU	reconfigurable DPU
CW	Configware	rDPA	reconfigurable DPA
DAC	Design Automation Conference	RA	reconfigurable array
DPA	data path array (DPU array)	RAM	random access memory
DS	data stream	rAMP	reconfigurable AMP
DPU	data path unit (without sequencer)	RC	reconfigurable computing
ecDPU	emulation-capable DPU	rGA	reconfigurable gate array
EM	evolutionary methods	RL	reconfigurable logic
EDA	electronic design automation	RTR	run-time reconfiguration
EH	evolvable morphware ("evolvable hardware")	SoC	(an entire) System on a Chip
		SSI	Small Scale ICs
FPGA	field-programmable gate array	SW	Software
FRGA	field-reconfigurable gate array	System	C C dialect f.Hw/Sw co–design
FW	Flowware	UML	Unified Modeling Language
GNU	GNU's Not Unix (consortium)	Verilog	a popular C-like HDL
HDL	Hardware Description Language	VHDL	VHSIC Design Language an HDL)
HPC	High-Performance Computing		
HW	Hardware	VHSIC	Very High Speed ICs
IC	integrated circuit	VLSI	Very Large Scale ICs
IP	intellectual property	vN	von Neumann

Figure 11.2. Acronyms

tor of the application-specific silicon, needed for accelerators, is increasing mask cost (Figure 11.3b), driven by growing wafer size and the growing number of masks needed. ASIC stands for mask–configurable gate arrays and similar methodologies [7] that need fewer masks than *full custom ICs* requiring the full mask set of the fabrication process [8].

1.1 Morphware

Illustrated by Makimoto's wave model [9, 10], the advent of morphware is the most important revolution in silicon application since the introduction of the microprocessor [11]. Emerging in the 1980s and now having moved from a niche market to mainstream, this third class of platforms now fills the gap between vN–type procedural compute engines and application–specific hardware. It is *morphware*, the fastest growing segment of the semiconductor market. (for terminology, see also Figure 11.5). The most important benefit of morphware is the opportunity to replace hardwired accelerators by RAM–based reconfigurable accelerators so that application–specific silicon can be mostly avoided, as is well-known from running software on the vN–type microprocessor. This will be

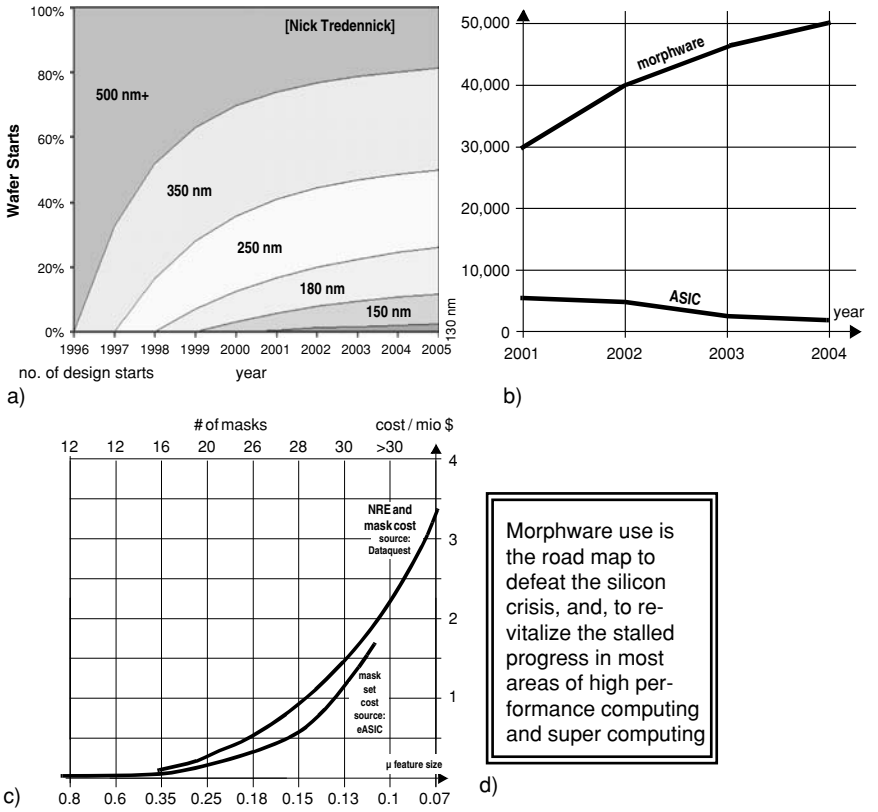


Figure 11.3. The second design crisis (silicon crisis). a) Decreasing number of wafer starts, b) growing number of morphware-based design starts [13] vs. declining number of ASIC design starts [13]; demonstrating that morphware already has reached mainstream status; c) increasing mask set cost and total NRE cost; d) providing the road map on the way out of the silicon crisis.

explained in the following paragraphs. The very high and still increasing number of morphware–based design starts (Figure 11.3c) demonstrates the benefit of using replacement morphware platforms instead of ASICs, where the backlog of design starts over morphware has exceeded a factor of more than 10 and is growing further.

> von Neumann's key achievement is the simple common model called the ***von Neumann machine paradigm.***

Morphware is structurally programmable hardware, where the interconnect between logic blocks and/or functional blocks, as well as the active functions of such blocks, can be altered individually by *downloading configware*, down to the *configuration memory (configuration RAM)* of a morphware chip (also compare Figure 11.6e). So we need two kinds of input sources: Traditional *software* for programming instruction streams, and *configware* for structural reconfiguration of morphware.

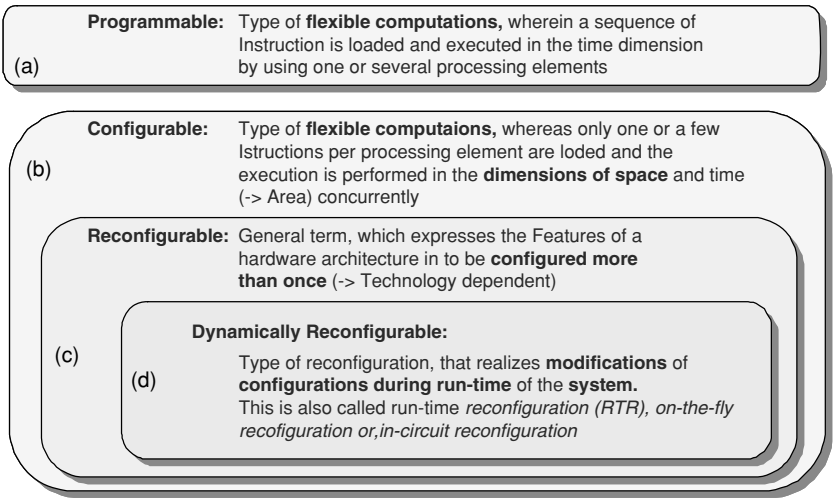


Figure 11.4. Contributions to terminology [12]: Programmable vs. (re)configurable.

Platform		Program source	Machine paradigm
Hardware		(not program-mable)	(None)
Mor-phware	Fine grain morphware	***Configware***	
	Coarse grain morphware (data-stream-based)	***Configware & flowware***	Anti-machine*
Hard-wired proces-sor	Data-stream-based computing	***Flowware***	
	Instruction-stream-based computing	Software	Von Neu-mann

* see Section 19.3.6 and Figure 11.23.

Figure 11.5. Terminology.

Before going into more detail, we should take a first step in clarifying the *terminology* around reconfigurability [12]. To highlight the key issues in distinguishing the classical vN paradigm from morphware, we should define the term *reconfigurable*, because *reconfiguration* in general has many different meanings. In computing sciences, the terms *programmable* refer to the *time domain*, where *programming* means *instruction scheduling* (Figure 11.4a). The term *configurable* introduces the *space domain*, where *configuration* means *the setup of structures* and preadjustment of logic blocks or function blocks (Figure 11.4b). *Reconfiguration* means, that a platform can be configured several times for different structures (Figure 11.4c), whereas Mask–Configurable Gate Arrays (see Section 2) can be configured only once. Configuration or reconfiguration usually is *impossible during run time*. But *dynamically reconfigurable* (Figure 11.4d) means that partial reconfiguration may happen at run time. A warning to educators: Dynamically reconfigurable

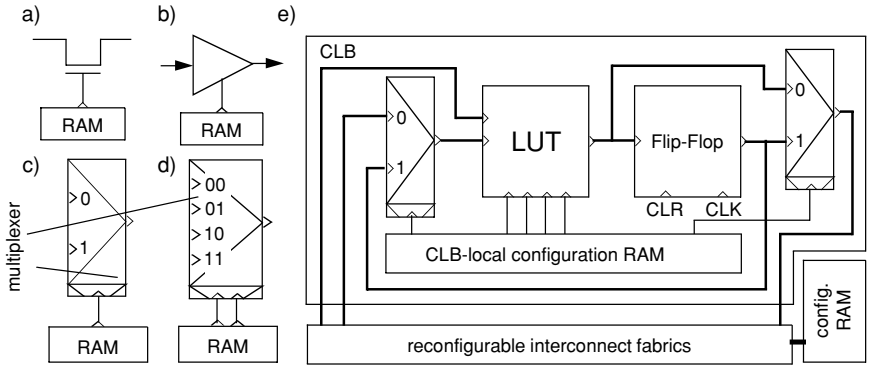


Figure 11.6. Programmable switches and blocks used in FRGAs. a) Pass transistor; b) Tristate buffer; c) two–way multiplexer; d) four–way multiplexer, e) simplified example of a Configurable Logic Block (CLB).

or *self–reconfigurable* systems are more bug prone than others and are more difficult to explain and to understand.

By introducing morphware, we obtain a new general model of embedded computers (Figure 11.7a): *The accelerator has become reconfigurable*. It has been changed from hardware (Figure 11.1a) to morphware. As mentioned previously, accelerator operation is usually data–stream–based. Because of its *non–von Neumann machine principles*, an accelerator has *no von Neumann bottleneck* and may be interfaced to a larger number of data streams (Figure 11.23c, d). With a

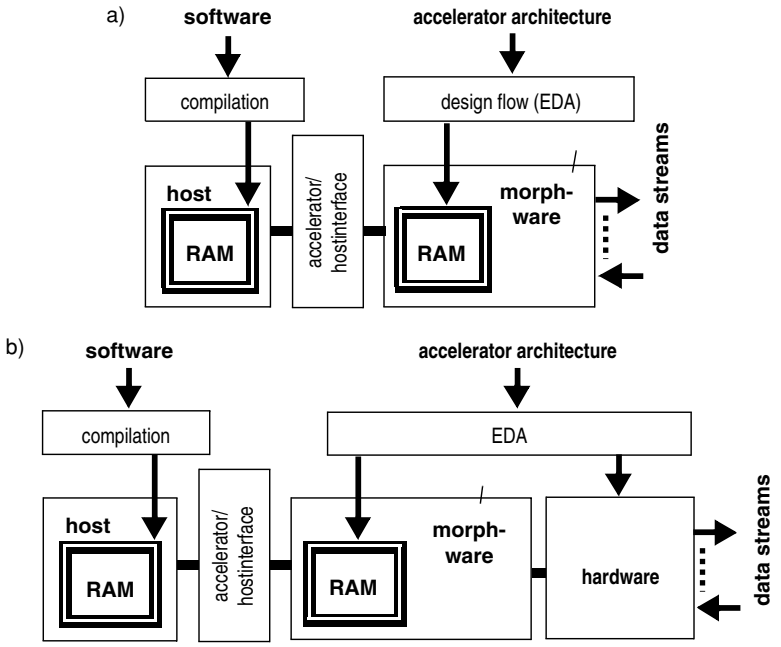


Figure 11.7. Traditional embedded computing design flow. a) morphware based; b) morphware/ hardware–based.

morphware accelerator (Figure 11.7a), the host may also use the host/accelerator interface to organize the reconfiguration process (this will be explained later). Also, mixed–type accelerators are possible (Figure 11.7b): Hardware and morphware. However, a few architectures include morphware directly inside the vN microprocessor. Here the morphware is used for *flexible instruction set extensions* [14, 15], a modern version of the vN model only, where morphware is connected to the *processor bus* (Figure 11.8a). Also most *network processors* use instruction set extensions [16]. This situation is different from the common model shown in Figure 11.7, where morphware is just connected to the host's *memory bus* (Figure 11.8b).

1.2 Two RAM-based machine paradigms

We now have two different RAM–based input source paradigms: One for scheduling (programming) the instruction streams, to be scanned *from RAM program memory during run time* by sequences of instruction fetches, and the other for configuring structures by downloading configware code *to the configuration RAM before run time*. Downloading configware code is a kind of pseudo–instruction fetch (but here *not at run time*) where, however, such "instructions" or expressions may be much more powerful than microprocessor instructions. The configuration RAM is often called *hidden RAM*, because it is not nicely concentrated into a matrix, as in typical RAM components sold by IC vendors. Physically, the individual memory cells in a morphware device are located close to the switch point or connect point they are controlling (see the flip–flops *FF* in Figure 11.10c and d). Also, the addressing method used by morphware for downloading reconfiguration code is often different from that of classical RAM.

> Data–stream–driven computing had already been used implicitly by the first programmers.

It was recognized rather early that morphware had introduced a fundamentally new machine paradigm. Field–reconfigurable Custom Computing Machines (FCCM) [17], the name of an annual conference series, is an indication. A major

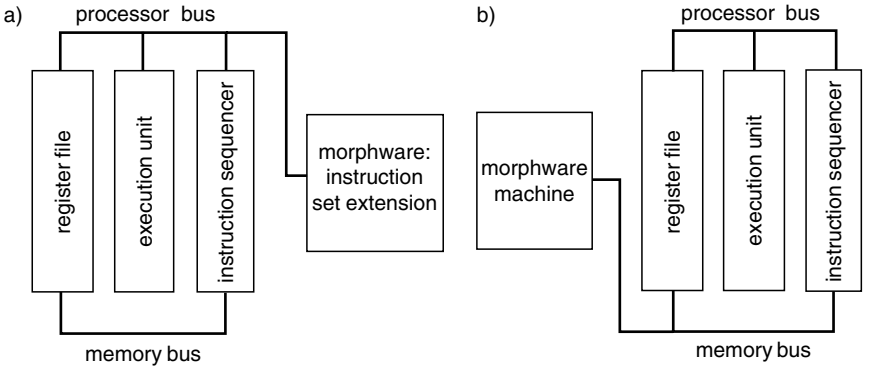


Figure 11.8. Alternative morphware applications. a) von Neumann processor with morphware-based instruction set extension; b) von Neumann host with morphware–based coprocessor.

number of experimental computing machines of this kind have been implemented, mostly from academia (for a survey covering the years 1995 and earlier, see [18]).

As mentioned earlier, the use of commodity off–the–shelf (COTS) morphware for acceleration can avoid the very costly need for application–specific silicon. Both kinds of platforms support *rapid downloading of patches, upgrades, or even new applications* down to the RAM program memory, even via the Internet. The consequence is a change of the business model for accelerators. *Personalization before fabrication*, typical of hardwired accelerators, can be replaced by the business model of the microprocessor, using *personalization after fabrication*—at the customer's site.

It is very important to distinguish, that the personalization source for vN microprocessors is *software*, and for morphware it is *configware*. Because of the growing importance of configware we currently observe a growing *configware industry*—a kind of emerging competitor to the *software industry*. Morphware has become an essential and indispensable ingredient in SoC (System on a Chip) design and beyond. Morphware meanwhile is used practically everywhere, so this chapter has no room for a survey to mention all uses. A good reading source is the volumes of proceedings (published by Springer in its LNCS series [19]) of Field–Programmable Logic [20], the annual international conference on Field–Programmable Logic and its applications, and the largest conference in this area.

2 FINE-GRAIN MORPHWARE

Since their introduction in 1984, *Field–Reconfigurable Gate Arrays* (*FRGAs*, often also called FPGAs), or *reconfigurable Gate Arrays* (*rGAs*) have become the most popular implementation media for digital circuits. For a reading source on the role of rGAs (providing 148 references), see [21]. The very high and increasing number of design starts on FRGAs demonstrates that the mask–configurable ASICs were already the losers years ago (Figure 11.3c). The technology-driven progress of FRGAs (for key issues, see [22]) is much faster than that of microprocessors. FRGAs with 50 mio system gates are coming soon [23]. It is well known that the growth rate of the integration density of microprocessors is much slower than Moore's law. However, because of the high degree of layout regularity, the integration density of FRGAs is moving at the same speed as Moore's law [9]. But because of the high percentage of wiring area, the transistor density of FRGAs is memory behind by two orders of magnitude [9]. However, the number of transistors per chip on FRGAs had surpassed that of microprocessors already by the early 1990s and is now higher by two orders of magnitude [9].

2.1 The Role of rGAs

We may distinguish two classes of morphware: *Fine-grain* reconfigurable morphware, and *coarse–grain* reconfigurable morphware. Reconfigurability of fine granularity means that the functional blocks have a datapath width of about one bit. This means that programming, at a low abstraction level, is logic design. Practically all products on the market are *FPGAs* (field–programmable gate arrays, better called *FRGAs* or *rGAs*: (*(field–)reconfigurable gate arrays*),

although some vendors prefer different terms as kinds of brand names, like, for instance, Programmable Logic Device (PLD), or reconfigurable logic device LD. Morphware platforms and their applications have undergone a long sequence of transitions. First, FPGAs appeared as cheap replacements for *MPGAs* (or *MCGAs*: Mask-Configurable Gate Arrays). Even today, FRGAs are the reason for the shrinking ASIC markets (Figure 11.3c), since for FPGAs no application-specific silicon is needed—a dominating cost factor in low production volume products. (ASIC fabrication cost is much lower—only a few specific masks are needed than that of other integrated circuits.) Later, the area proceeded into a new model of computing possible with FRGAs. The next step was making use of the possibility for debugging or modifications during the last day or week, which also led to its adoption by the *rapid prototyping* community which also has led to the introduction of *ASIC emulators* faster than simulators. The next step is direct *incircuit execution* for debugging and patching at the last minute.

Meanwhile, morphware is used practically everywhere.

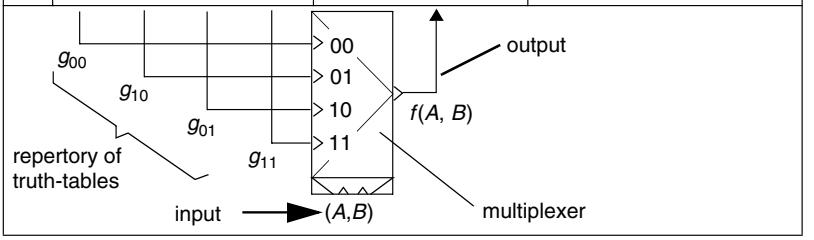
From a terminology point of view, the historic acronyms *FPGA* and *FPL* are a bad choice, because programming, i.e., scheduling, is a *procedural* issue in the *time domain*. The term *PLD* is also a bad choice and should be replaced by *rLD*. A program determines a *time sequence* of executions. In fact, the FP in FPGA and in FPL (the acronym for *field–programmable*), actually means *field reconfigurable*, which is a structural issue *in the space domain: configuration in space*. For a clearly consistent terminology, it would be better to use *FRGA (field–reconfigurable gate array)* or *rGA* instead of FPGA. Throughout this chapter the term *rGA* or *FRGA* will be used instead of *FPGA*. For terminology, see Figure 11.2, Figure 11.5, and Sections 2.5 and 4.1.

The most important architectural classes of rGAs are (see [24]) island architecture (Xilinx), hierarchical architecture (Altera), and row–based architecture (Actel). A more historic architecture is *mesh–connected*, sometimes also called *sea of gates* (introduced by Algotronix) [25]. A simple example of Configurable Logic Block block diagram is shown in Figure 11.6. Its functional principles by multiplexer implementation are shown in Figure 11.9a and b, where in CMOS technology, only 12 transistors are needed for the fully decoded multiplexer (Figure 11.9c). The island architecture is illustrated in Figure 11.10a. Figure 11.10b show details of *switch boxes* and *connect boxes*. Figure 11.10c shows the circuit diagram of a *cross point* in a switch box, and, Figure 11.10d shows the same from within a connect box. The thick wire in Figure 11.10b illustrates how these interconnect resources are configured to connect a pin of one CLB with a pin of another CLB. The total configuration of all wires of an application is organized by a *placement and routing* software. Sometimes more interconnect resources are needed than are available, so for some CLB not all pins can be reached. Due to such *routing congestion*, it may happen that a percentage of CLBs cannot be used.

2.2 Commercially available FRGAs

A wide variety of fine-grain morphware products is available from a number of vendors, such as the market leader Xilinx [26], the second largest vendor Altera

a) #	Configuration bits g_{ab} g_{00}	g_{01}	g_{10}	g_{11}	$f(A, B)$	Function
0	0	0	0	0	0	constant 0
1	0	0	0	1	*A* and *B*	and
2	0	0	1	0	*B* disables *A*	if *B* then 0 else *A*
3	0	0	1	1	*A*	identity *A*
4	0	1	0	0	*A* disables *B*	if *A* then 0 else *B*
5	0	1	0	1	*B*	identity *B*
6	0	1	1	0	*A* exor *B*	antivalence
7	0	1	1	1	*A* or *B*	or
8	1	0	0	0	*not*(*A* or *B*)	nor
9	1	0	0	1	*A* coin *B*	equivalence
10	1	0	1	0	not(*B*)	negation of *B*
11	1	0	1	1	*B* implies *A*	if *A* then 1 else ¬ *B*
12	1	1	0	0	not(*A*)	negation of *A*
13	1	1	0	1	*A* implies *B*	if *B* then 1 else ¬ *A*
14	1	1	1	0	not(*A* and *B*)	nand
15	1	1	1	1	1	constant1



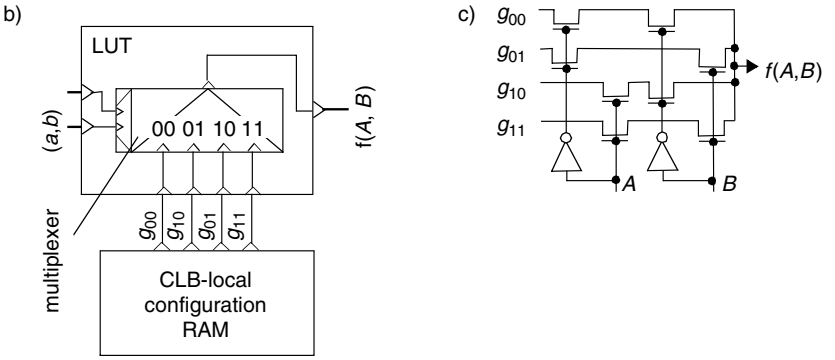


Figure 11.9. Illustrating LUT implementation by multiplexer: example for functions of two variables. a) illustration of the function generator; b) multiplexer circuit; c) illustration of LUT (look-up table) block use within CLB (compare Figure 11.6e).

[27], and many others. A variety of evaluation boards and prototyping boards is also offered. COTS (commodity off the shelf) boards for FRGA–based developments are available from Alpha Data, Anapolis, Celoxica, Hunt, Nallatech, and others, to support a broad range of in–house developments. As process geometries have shrunk into the deep–submicron region, the logic capacity of FRGAs has greatly increased, making FRGAs a viable implementation alternative for larger and larger designs. FRGAs are available in many different sizes and prices

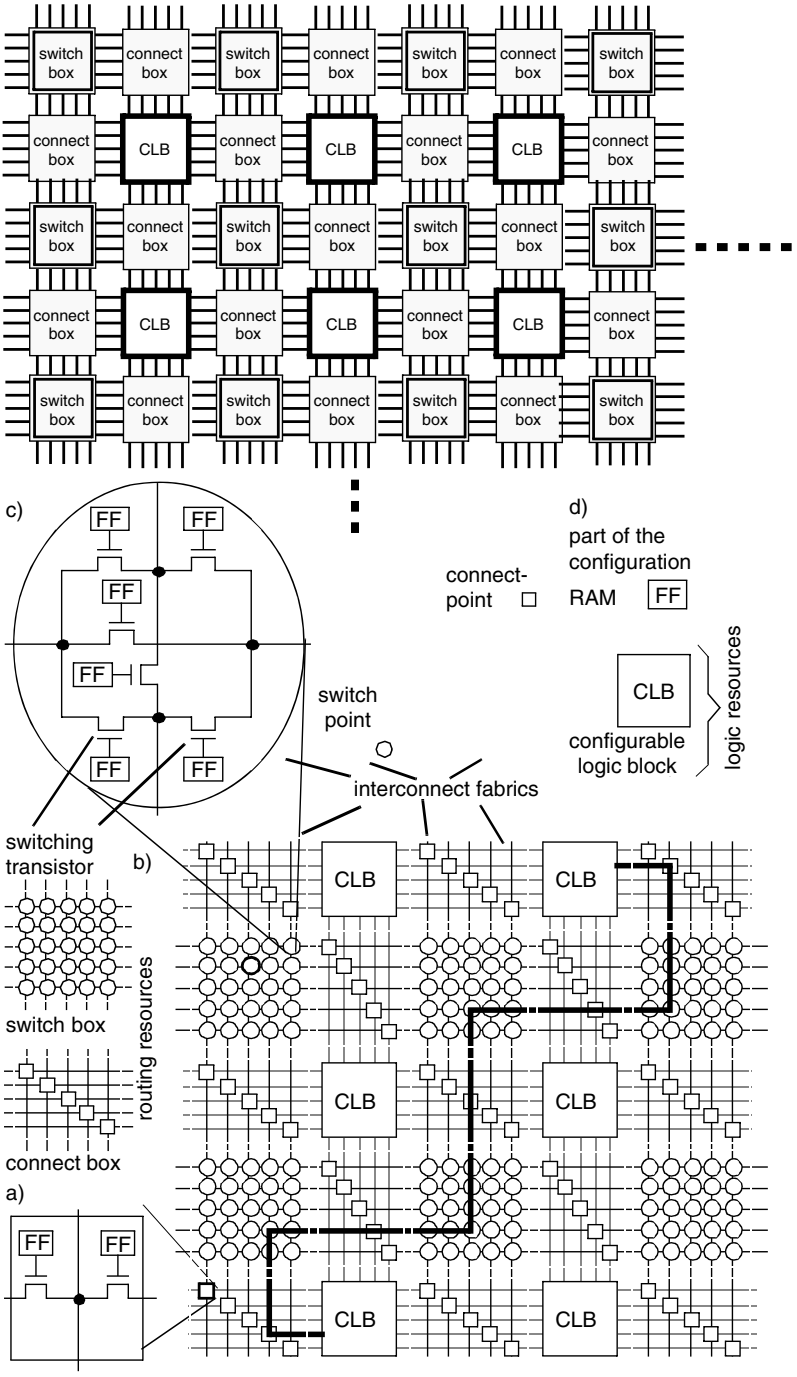


Figure 11.10. Illustrating FRGA island architecture fine-grain morphware resources. a) Global view of interconnect fabrics; b) detailed view (only one configured "wire" shown); c) connect point circuit of a switch box; d) connect point circuit of a connect box.

per piece, ranging from 10 US–dollars to FRGAs with many more than a million usable gates for more than 1000 US–dollars. Xilinx has preannounced FRGAs with 50 *mio* system gates around 2005 [23]. Modern FRGAs support mapping entire systems onto the chip by offering on board all components needed, such as several memory banks for user data; one or several microprocessors like ARM, PowerPC, MIPS, or others; a major number of communication interfaces (WAN, LAN, BoardAN, ChipAN etc.) supporting contemporary standards; up to several GHz bandwidth; JTAG boundary scan circuitry to support testing; sometimes even multipliers

Also, FRGAs featuring low power dissipation [28] or better radiation tolerance (for aerospace applications) are offered. Several major automotive corporations have contracts with FRGA vendors to develop morphware optimized for this branch of industry. Some commercially available FRGAs also support partial columnwise reconfiguration so that different talks may reside in the array and may be swapped individually. This setup may also support *dynamic reconfiguration* (*RTR*: run–time reconfiguration), where some tasks may be in the execution state, while at the same time other tasks are being reloaded. Dynamic reconfiguration, however, tends to be tricky and difficult to understand and to debug. But static reconfiguration is straightforward and easier to understand. Because reconfiguration is slow multi–context morphware has also been discussed, but is not yet available commercially. Multicontext morphware features several alternative internal reconfiguration memory banks, for example two or four banks, so that reconfiguration can be replaced by an ultrafast context switch to another memory bank.

2.3 Applications

Morphware is used practically everywhere, so this section can mention only a few examples. Most early FRGA applications have been rapid prototyping [25, 29, 30], rather than directly implementing products on morphware platforms. *Rapid prototyping* and *ASIC emulation* are still important for the development of hardwired integrated circuits. Since, in IC design, flow simulation may take days or even weeks, a remedy has been *ASIC emulation*, using huge emulation machines called *ASIC emulators*.

Earlier such machines included racks full of boards equipped with masses of FRGAs of the low density available at that time. Through acquisitions the three major EDA vendors now offer ASIC emulators, along with compilers: Cadence has acquired Quickturn, Synopsys has acquired IKOS, and Mentor Graphics has bought Celaro, also offering such service over the Internet. Another R&D scene and market segment calls itself *Rapid Prototyping*, where for smaller designs less complex emulation boards are used, such as Logic emulation PWB (based on the Xilinx Virtex FRGA series, which can emulate up to 3 million gates), and the DN3000k10 ASIC Emulator from the Dini Group.

The terminology is reconfigurable vs. programmable. The semantics is structural vs. procedural.

Another morphware application area is scientific high–performance computing (HPC) where often the desired performance is hard to attain through "traditional" high–performance computing. For instance, the gravitating *n*-body problem is one of the grand challenges of theoretical physics and astrophysics [31, 32]. Hydrodynamic problems fall into the same category, where often numerical modeling can be used only on the fastest available specialized hardware.

Analytical solutions exist for only a limited number of highly simplified cases. For example interpretation of dense centers of galactic nuclei, observed with the Hubble Space Telescope, by uniting the hydrodynamic and the gravitational approach within one numerical scheme. The maximum particle number was limited until recently to about 10^5 even on the largest supercomputers. For astrophysics, the situation improved thanks to the GRAPE special purpose computer [33]. To improve flexibility, a hybrid solution has been introduced with AHA-GRAPE, which includes auxiliary morphware [31]. Other morphware–based machines such as, WINE II, MDGRAPE [34], and MDM (Modular Dynamics Machine) [35–37] are also used for modeling and simulation in molecular dynamics [31, 33, 38].

Because of the availability of high–density FRGAs, the scenario has drastically changed. The trend is to deliver the FRGA–based solution directly to the customer, at least for lower production volumes. Not only microcontrollers or simple logic circuits are easy to transfer onto a FRGA platform; practically everything can migrate onto morphware. A single FRGA type may replace a variety of IC types. Design and debugging turn–around times can be reduced from several months to weeks or days. Patches or upgrades may take only days, hours, or even minutes, and may even be carried out at the customer's site or remotely over the Internet or wireless communication, which means a change of the business model—an important benefit for innovative efforts in remote diagnosis and other customer services.

A future application of emulation may serve to solve the long–term microchip spare-part problem in areas such as industrial equipment, military, aerospace, automotive, etc., with product lifetimes up to several decades [39]. The increasing spare-part demand stems from the increasing number of embedded systems, the limited lifetime of microchip fabrication lines (mostly less than 7–10 years), and the decreasing lifetime of unused microchips. When a modern car with several dozen embedded microchips needs electronic spare–parts 10 or 15 years later, the microchip fab line no longer exists, and a major percentage (or all) of the parts kept in spare–parts storehouses have faded away. The hope of keeping an old fab line alive that could deliver long–lasting robust products at low NRE cost seems to be an illusion. *Retro emulation* might be the only viable solution, where reverse engineered products are emulated on FRGAs, since application–specific silicon will not be affordable due to low microchip production volumes in these areas and rapidly increasing mask cost.

Fortunately now, with FRGAs, a new kind of IC platform is available so that we can switch from hardware to morphware, which can be "rewired" at run time. Because of their general–purpose properties, FRGAs are a suitable platform for reverse engineering of required but unavailable spare parts. Morphware is the fastest growing segment of the IC market [Dataquest]. Also for industries such as the automotive, aerospace, military, or industrial electronics such a common morphware platform would be a promising route to avoid very high mask costs,

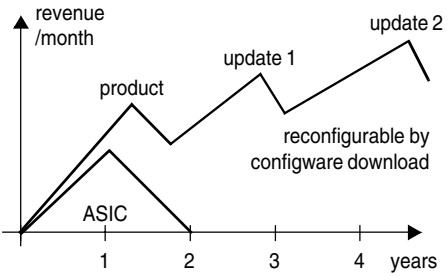


Figure 11.11. accelerator longevity [40].

to reduce the number of IC types needed, to accelerate IC time to market, and to solve long–term spare–part supply problems by retro emulation.

In morphware application, the lack of algorithmic cleverness is an urgent educational problem.

The new business model of morphware brings a new dimension to digital system development and has a strong impact on SoC design (System–on–Chip). Performance by parallelism is only one part of the story. The time has come to fully exploit morphware flexibility to support very short turn–around time for real–time, in–system debugging, profiling, verification, tuning, field maintenance, and field upgrades. One of the consequences of the new business model is the adoption of a computer science mentality for developing all kinds of electronics products, where patches and upgrades are carried out at the customer's site (Figure 11.11), or even via the internet using Run–Time Reconfiguration (RTR). This approach is also an important remedy to the current embedded system design crisis, caused by skyrocketing design cost coinciding with decreasing product lifetime, by providing product longevity (Figure 11.1b).

2.4 Application Development support

Morphware is the fastest growing segment of the integrated circuit (IC) market, currently relying on a growing large user base of HDL–savvy designers. A number of books are available that give an introduction to application development using FRGAs [29, 41–45]. Not only is the configware industry rapidly growing, offering IP cores [46] and libraries for morphware platforms but also a rapidly growing branch of the EDA industry offers tools and design environments to support configware development. Complete design flows from HDL sources such as VHDL [47] are offered by Mentor Graphics [48], Synplicity [49], Celoxica [50], and others. A key issue is the integration of IP cores into the design flow. At DAC [51], a task force has been set up to solve standards problems.

Sloppy terminology is a severe problem which torpedoes diffusion and education.

There are also design flows [52, 53] from Matlab sources [54], and a tool to generate HDL description from Unified Modeling Language has been reported [55]. An emerging trend is going to input sources of higher abstraction levels like the languages Handel–C by Celoxia, Precision–C from Mentor Graphics, SystemC [56, 57], a C dialect [58] by Synopsys [59] targeting HW/CW/SW co-design. Matlab indicates a tendency to go to even higher abstraction level of mathematical formulas. The emerging use of *term rewriting systems (TRS)* for design is another indication of this trend [60–63].

Also, a wide variety of vendors are offering tools not covering the entire design flow, such as those for debugging, timing estimation [64], simulation, verification, placement and routing, and other tasks, as well as soft IP cores. Examples include the CoreConnectBus (Xilinx), Parameterizes Processor (Xilinx), IPbus interface (Xilinx), embedded software development tools (Wind River, GNU, and others), Integrated Bus Analyzer (Xilinx), board support package for interface software (Xilinx), and over 40 processor IP models (Xilinx), [23]. Still a research area is morphware operating systems, to load and coordinate multiple tasks to be resident in a single FRGA. PACT has this sort of an OS for its XPP coarse–grain reconfigurable array (see Section 3), which can be *partly reconfigured* rapidly in parallel while neighboring reconfigurable data path units (rDPUs) are still processing data. Reconfiguration is triggered externally or even by special event signals originating within the array, *enabling self–reconfiguring designs* [65]. In general, there is still room for new tools and design flows offering improved quality and designer productivity. Key issues for the performance of FRGAs implemented in deep–submicron processes are the following three factors: the quality of the CAD tools used to map circuits into the FRGA, the quality of the FRGA architecture, and the electrical (i.e., transistor–level) design of the FRGA. In order to investigate the quality of different FRGA architectures, we need EDA tools capable of automatically implementing circuits in each FRGA architecture of interest.

19.2.5 Education

Education is an important area of application development support because it prevents a shortage of qualified professionals. In morphware application, the lack of algorithmic cleverness is one of the urgent educational problems. For instance, how can we implement a high–performance application for low-power dissipation on 100 datapath units running at 200 MHz, rather than on one processor running at 20 GHz? An example is the migration of an application from a very fast digital signal processor to a low power implementation on FRGA, yielding speedup factors between 5 and 22 [66]. The transformation of the algorithm from the software domain to fine–grain morphware required an enormous effort by the student-in-charge of this project, because such algorithmic cleverness is not yet taught within typical curricula.

> The data stream paradigm has been around for almost three decades. Software uses it indirectly through inefficient instruction–stream implementations. Due to poor synthesis methodology, its direct use by systolic arrays remained a niche until the mid–1990s.

CS education is becoming more and more important for embedded system development, because SoC design has rapidly adopted CS mentality [67]. The amount of program code implemented for embedded systems doubles every 10 months and will reach 90% of all codes being written by the year 2010 [68]. Currently, a typical CS graduate with von–Neumann–only mentality does not have the skills needed for HW/CW/SW partitioning decisions nor the algorithmic cleverness needed to migrate an application from software onto an FRGA. The failure to teach the important skills, needed to map applications onto morphware in our CS curricula will cause a major disaster. Our current graduates are not qualified for the IT labor market of the near future [72].

Terminology is a key issue. It is very important to maintain a clear and consistent terminology. Sloppy terminology is a severe problem that torpedoes diffusion, education, and efforts to bridge communication gaps between disciplines. Too many experts using their own nonconsensus terminology are creating massive confusion: their colleagues often do not know what they are really talking about.

I have had my own frustrating experiences with contradictory terminology when teaching VHDL and Verilog in the same course [73]. Students have been confused by most of the terminology because, for almost each important term in this area, there have been usually three different definitions: 1) what the student associates with the term when hearing it for the first time, 2) how the term is used by VHDL experts, and 3) how it is used by Verilog experts.

Terminology should be tightly linked with common models. In both hardware and software, the *design space* has almost infinite size. Not only students get lost in this space without any guidance by models that narrow the design space. A machine paradigm is needed. The *von Neumann paradigm* has been highly successful for 50 years; but now because of the *dominance of morphware* we need a new, second, machine paradigm that can be used as a general model for guidance due to: (1) Its well–defined terminology, and (2) its simplicity: the *anti–machine paradigm* (see Section 3.6). The term *reconfigurable* has too many different meanings in too many different areas, including everyday life. For this reason the term *morphware* is often much better. Because terminology is so domain specific, you can guess a person's field by his or her use of terminology. When somebody associates *blacksmith* with *hardware*, you know this person is an IT professional. When somebody associates downloading drivers or other software into the RAM of a von Neumann machine with *reconfiguration*, you know that this person is not familiar with morphware and its environment.

2.6 Innovative Applications

Terms like *evolvable hardware* (EH) or in fact, *evolvable morphware (EM)*, *Darwinistic Methods* for system design, or *biologically inspired system design* point to a newer research area stimulated by the availability of fine-grain morphware. Also, retro emulation is an innovative application. It is an efficient way of re–engineering unavailable electronics parts for replacement to solve the long-term microchip spare-part problem in areas such as industrial equipment, military, aerospace, automotive, etc., with product lifetimes up to several decades. But in the future, reverse engineering can be avoided, it the implementation of all IC architectures are FRGA based from the beginning.

The antimachine has no von Neumann bottleneck. No caches are needed.

FRGAs may be good platforms to achieve *fault tolerance* by self–healing mechanisms [74, 75]. Partial rerouting can circumvent wires or CLBs found to be faulty. A NASA single–chip spacecraft has been discussed (breaking many paradigms in spacecraft design [76]), which is based on a high–density FRGA, does not need an operating system, and uses fault tolerance to reduce the need for radiation hardening. Currently available commercial FRGA architectures insufficiently support such rearrangements at run time. More research is required to obtain better architectural support [74, 77].

Another interesting area deals with *soft CPUs*, also called *FRGA CPUs*, i.e., microprocessors implemented by mapping their logic circuits onto an FRGA. Examples are the *MicroBlaze*, a 32–bit Harvard architecture from Xilinx [78], Altera's *Nios* processor [27], the ESA *SPARC LEON* open source core [79, 80], the *LEON2* processor [81], which is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture, and the *Dragonfly* 8-bit core [78]. Of course, soft processors run about a factor of 3 to 5 times slower than their hardwired versions. By the way, designing soft CPUs is a popular subject of lab courses offered by a large number of universities.

2.7 Scalability and Relocatability

Relocation, even dynamically at run time, of configware macros is subject of the new area of configware operating systems [69–71]. Some FRGAs are so large that more than 100 soft CPUs can be mapped onto such a single chip. Will future giga–FRGAs permit the mapping of practically everything, including large rDPAs, onto a single morphware chip? This leads to the question of FRGA scalability. For instruction set processors, the von Neumann bottleneck guarantees full relocatability of code. Within very large FRGAs, however, relocatability might be limited by routing congestion (Figure 11.12a). But *Structured Configware Design* (a design philosophy derived from structured VLSI design [82]) is a promising approach to solve the relocatability problem (Figure 11.12b), so that FRGAs may be universal as microprocessors.

3 COARSE-GRAIN MORPHWARE

In contrast to fine-grain morphware using CLBs of smallest datapath width (~1 bit), coarse-grain morphware uses rDPUs (reconfigurable Data Path Units) with wide data paths, e.g., 32 bits wide. Instead of FRGAs, we have rDPAs (reconfigurable DPU Arrays). As an example, Figure 11.13 shows the result of mapping an image-processing application (SNN filter) onto a primarily mesh-based KressArray [83] with 160 rDPUs of 32-bit path width. This array is interfaced to 10 data streams: nine input streams and one output stream. Figure 11.14 shows some details of the XPU (xtreme processing unit), a commercially available rDPA from PACT AG [84–87]. Figure 11.15 illustrates the differences in the execution mechanisms. At vN execution (Figure 11.15a), exactly one operation is

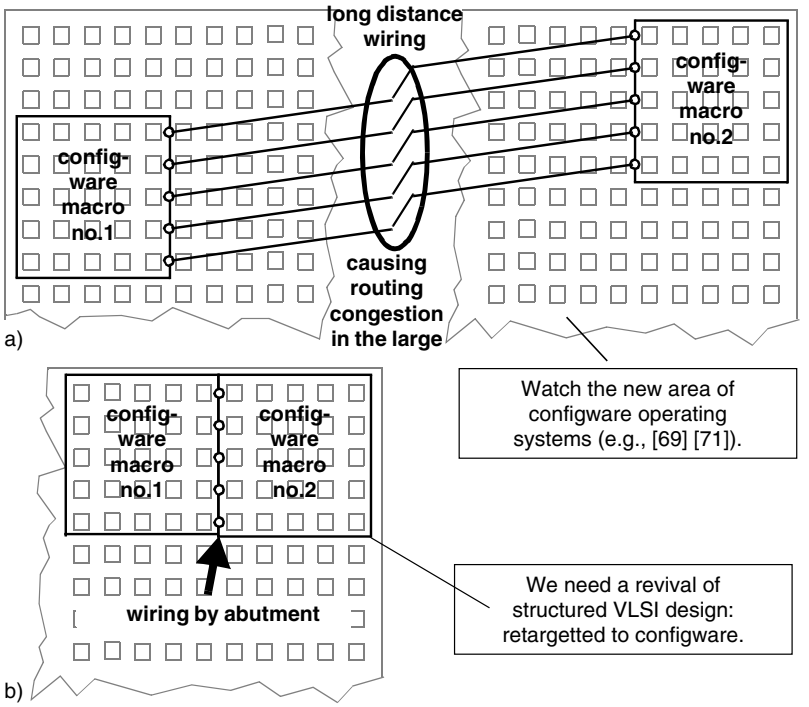


Figure 11.12. Solving a) the FRGA scalability problem b) automated structured configware design.

carried out per clock cycle. Intermediate results are stored in registers. For migration of such an algorithm from vN to an rDPA like PACT XPP (Figure 11.15b), a subsequence is mapped from time to space and executed in parallel on the array.

As soon as this operation is completed, the next chunk of parallelized code is executed. Intermediate results may be communicated by a buffer (see Figure 11.15b).

Usually an rDPA is a pipe network, not a multiprocessor or multicomputer network, since DPUs do not show a program counter (for details, see later sections of this chapter). Coarse–grain morphware has been a research area for more than a decade (for a survey, see [88, 89]). Since it plays an important role in wireless communication [90, 91], *software–defined radio* [92], and multimedia processing, not only performance but also MIPS / mW are key issues. Figure 11.16 shows that FRGAs just fill the efficiency gap and the flexibility gap between hardwired platforms and instruction set processors. Coarse-grain arrays, however, almost attain the efficiency of hardwired platforms (Figure 11.16), when mesh-based architectures using wiring by abutment are used so that no separate routing areas are needed [9]. Also, configuration memory being an order of magnitude smaller than that of FRGAs, contributes to this area/power efficiency [9].

> Breaking away from the current mindset requires more than traditional technology development and infusion. It requires managerial commitment to a long-term plan to explore new thinking [96].

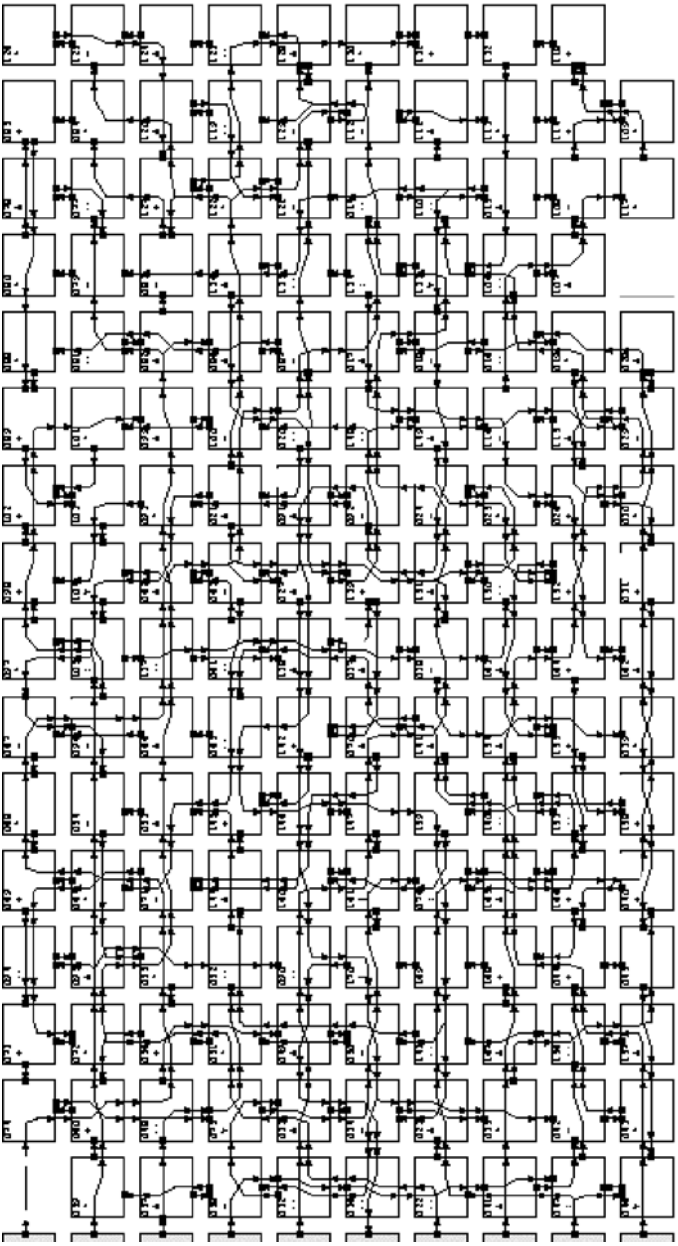


Figure 11.13. Example of mapping an application (image processing: SNN filter) onto a (coarse, grain) KressArray,

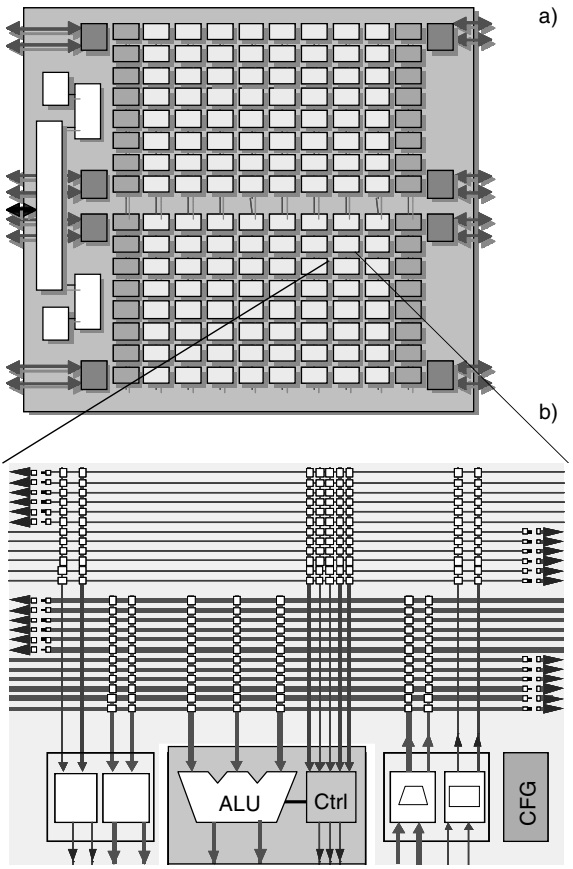


Figure 11.14. Configurable XPU (xtreme processing unit from PACT. a) Array structure; b) rDPU.

3.1 Pipe Networks and Flowware

We have to distinguish between two different domains of programming in time: *Instruction scheduling* and *data scheduling*. The programming code for von Neumann–like devices is an *instruction schedule*, compiled from *software* (Figure 11.17b). The programming code for resources like systolic arrays and other DPA (arrays of DPUs) is a *data schedule*, which can be compiled from *flowware* defining, which data item has to appear at which port at which time. Such data schedules manage the flow of *data streams*. This is illustrated in Figure 11.7a, showing a typical *data stream* notation introduced with *systolic arrays more than 20 years ago*.

The first *flowware–based* paradigm, the systolic array, got stuck in a niche for a long time (throughout the 1980s and beyond) because of the wrong synthesis method—until the *supersystolic array* made it viable for morphware. This

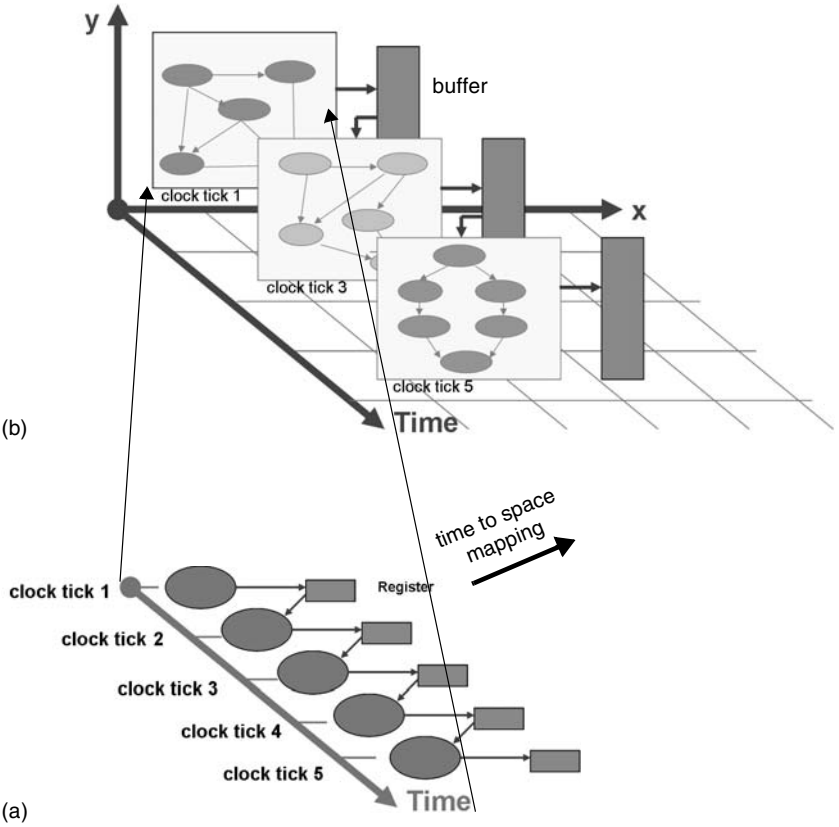


Figure 11.15. Migration to a) PACT XPP from b) von Neumann.

will be explained later. A systolic array [93–95] is a pipe network. The term *systolic* reminds us of the multiple data streams clocked into and out of such a pipe network and of its similarity to the heart and the bloodstreams entering and leaving it. Its DPUs never have instruction sequencers. The mode of DPU operation is transport triggered by data items. If synchronization is done by handshake instead of clocking, a systolic array may be also called a *wavefront array*.

The traditional systolic array could be used only for applications with strictly regular data dependencies, because array synthesis methods used linear projections or algebraic methods resembling linear projections. Such synthesis methods yield only strictly uniform arrays with linear pipes. The Data Path Synthesis System (DPSS) [83], however, uses simulated annealing (the mapper in Figure 11.17c), which removes the traditional application limitations, enabling the synthesis of *supersystolic arrays* featuring and also any kind of nonuniform arrays with any freeform pipes, such as zigzag, spiral, completely irregular, and many others. Due to this drastically improved flexibility, reconfigurable arrays (rDPAs) also make sense. The KressArray Xplorer, including a mapper, has been implemented as a design space explorer to optimize rDPU and rDPA architectures [97–99]. For more details on Xplorer, see Section Figure 3.4.

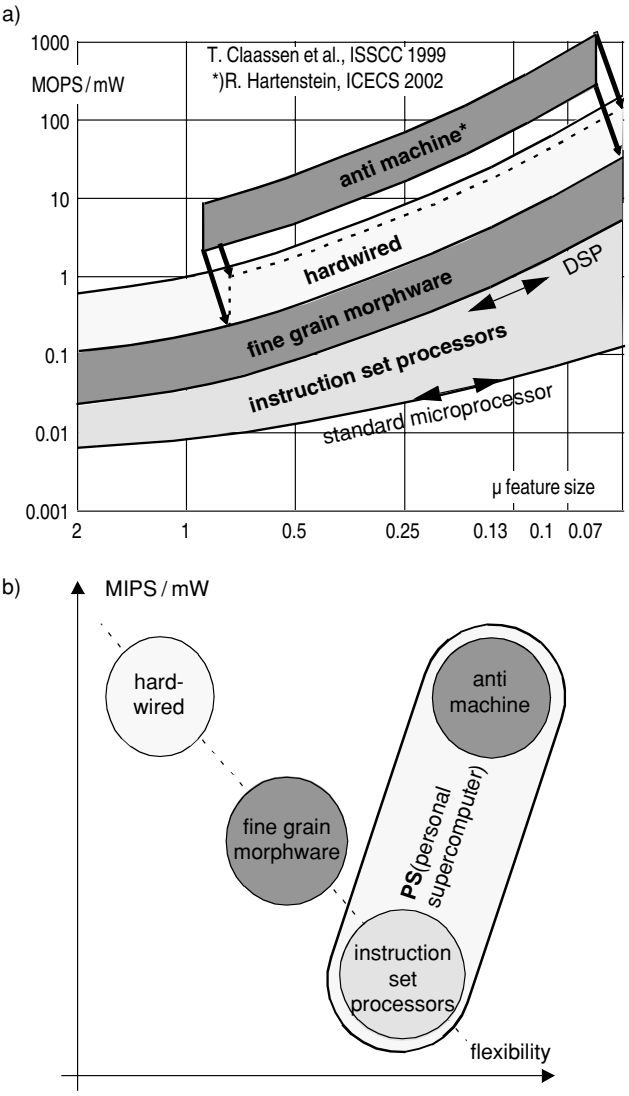


Figure 11.16. Performance and energy efficiency: a) vs. technology generation; b) vs. flexibility.

3.2 Data streams and flowware languages

More recently, data–stream–based computing has been popularized by a number of academic projects, such as SCCC [100], SCORE [101, 102], ASPRC [103], BEE [104, 105], KressArray [97, 98], and more [106]). The specifications of data streams can be expressed by *flowware language*. Data streams are created by executing flowware code on *auto–sequencing memory modules* (asM). Figure 11.19 a shows a distributed memory array of such asM modules driving data streams from/to the rDPA surrounded by the asMs. All enabling architectural resources for flowware execution are available [107, 108, 110, 111]. The new R&D discipline of application–specific distributed memory architectures [107] has arrived just in time

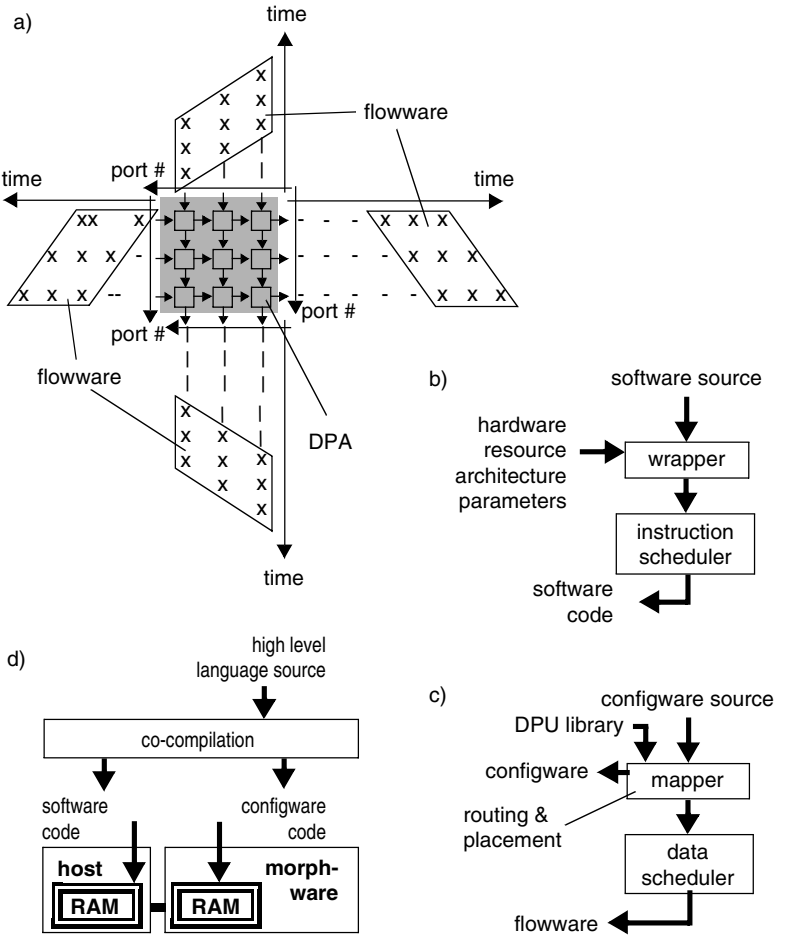


Figure 11.17. Compilation. a) a systolic array example (matrix multiplication) to illustrate flowware and its role; b) compilation for von Neumann platforms; c) configware / flowware compilation for morphware platforms; d) software / configware co/compilation.

to provide a methodology of architectural resources for processing flowware. Two alternative memory implementation methodologies are available [107, 112, 113], either specialized memory architectures using synthesized address generators (e.g., APT by IMEC [107]) or flexible memory architectures using programmable general–purpose address generators [109, 114]. Performance and power efficiency are supported especially by sequencers, which do not need memory cycles even for complex address computations [107], having been used also for the smart memory interface of an early antimachine architecture [114, 115].

Flowware may also be described by higher–level flowware languages [116], which are similar to high level software languages like C (Figure 11.21). Both languages have jumps, loops, and nested loops. The main differences between software and flowware is that, flowware semantics is based on one or several data counters, whereas software refers to only a single program counter. Because of

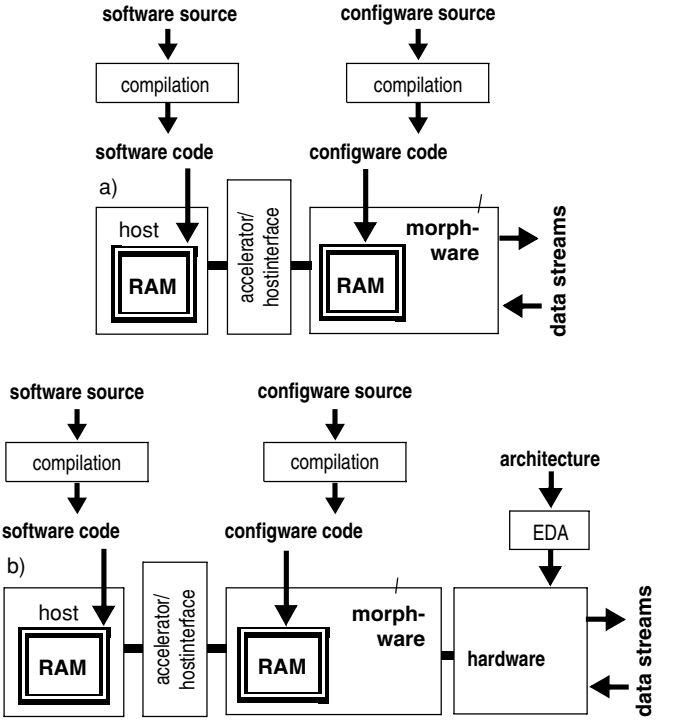


Figure 11.18. Modern embedded computing design flow. a) Software / configware codesign; b) software / configware / hardware codesign.

multiple data counters, flowware also features parallel loops, which are not supported by software languages. Flowware is much simpler because it does not need to express data manipulation.

> Because of the wrong synthesis method, the systolic array, the first flowware–based paradigm, got stuck in a niche for a long time—until the super–systolic array made it viable for morphware.

For good morphware application development support, an integrated synthesis system is useful that efficiently supports configware / flowware codesign, such as, for instance, DPSS [83] (Figure 11.19b), so that the user does not need to care about the configware / flowware interaction. A well–designed dual–paradigm language covering both [116] the flowware paradigm and the configware paradigm, and supporting the communication between both segments, would be useful for designer productivity. Examples for multiple–scope languages are already existing hardware languages like VHDL [47] or Verilog [43], which support the co–description of hardware and software constructs and also alleviate the handling of hardware / software communication. The strong trend within EDA toward higher abstraction levels, heralded by new languages like System–C [56–58] and others, opens a path toward integrated codesign frameworks coordinating all three paradigms covering hardware, morphware, software, configware, and flowware.

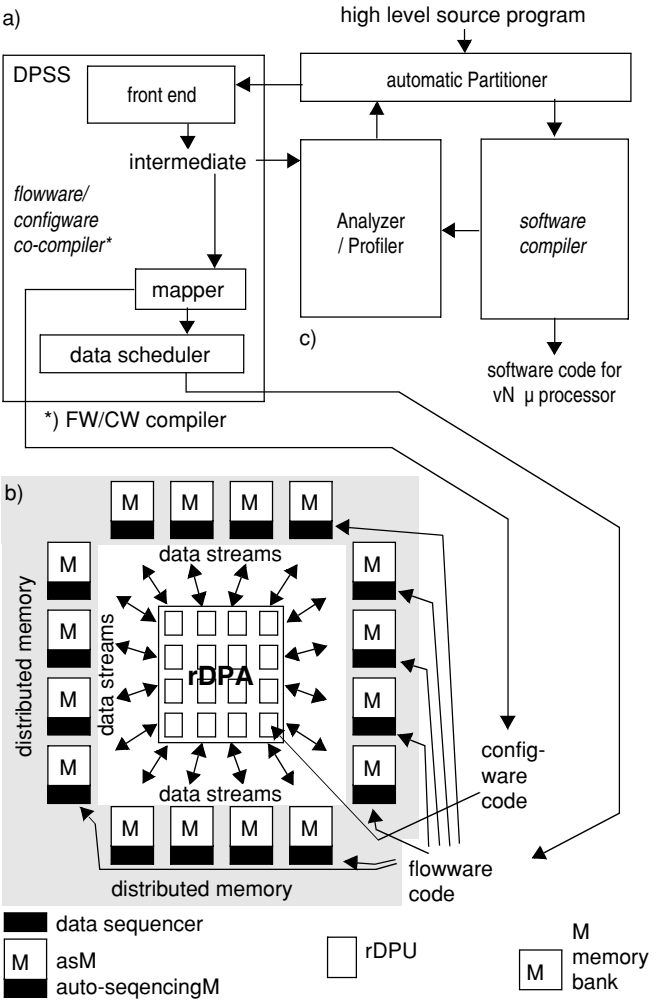


Figure 11.19. Flowware/configware/software cocompilation. a) Becker's partitioning cocompiler; b) antimachine target example.

The flowware–based common model of data–stream–based computing may be used for both hardware and morphware. There is, in principle, no difference, whether DPAs are hardwired or reconfigurable (rDPAs). The only important difference is the binding time of placement and routing before fabrication (hardware) or after fabrication (morphware: Compare Figure 11.27).

3.3 Coarse-Grain Arrays

Because the number of CFBs is by orders of magnitude smaller than that of CLBs in FRGAs, mapping takes only minutes or less instead of hours. Since computational data paths have regular structure potential, full custom designs of *reconfigurable datapath units* (rDPUs) are drastically more area–efficient, Coarse–grained architectures provide operator–level CFBs and very area–efficient datapath routing

switches. A major benefit is massive reduction of configuration memory and configuration time, and drastic complexity reduction of the P&R (placement and routing) problem. Several architectures will be briefly outlined (for details, see [88]).

Primarily mesh–based architectures arrange their PEs mainly as a rectangular 2-D array with horizontal and vertical connections that support rich communication resources for efficient parallelism and encourage nearest neighbor links between adjacent PEs. Typically, longer lines are also added with different lengths for connections over distances larger than one. The *KressArray* [83] is primarily a mesh of rDPUs physically connected through wiring by abutment. *MATRIX* [117] is a multigranular array of 8–bit CFBs (basic with vN microprocessor core) *Reconfigurable Architecture Workstation (RAW)* [118] provides a 4–by–4 array RISC multiprocessor architecture of NN–connected 32–bit modified MIPS R2000 microprocessors. The *(Dynamically Reconfigurable Architecture for Mobile Systems (DReAM)* Array [119]) is for next–generation wireless communication.

Some RAs are based on one or several linear arrays, like *(Reconfigurable Pipelined Datapath)* (*RaPiD*) [120] and *PipeRench* [121]. Architectures using crossbars include *(Programmable Arithmetic Device for DSP) PADDI*, which uses a central reduced crossbar (difficult to rout) and a two level hierarchy of segmentable buses; *PADDI–1* [122, 123], and *PADDI–2* [124]. The *Pleiades* Architecture [66] is a kind of generalized low–power *PADDI–3*.

3.4 Compilation Techniques

The first step in introducing morphware–oriented compilation techniques in application development for embedded systems is the replacement of EDA (Figure 11.7a and b) by compilation also for the morphware part (Figure 11.18a and b). This step of evolution should be accompanied by a clean model that has been introduced in the course of history. Partly in synchrony with Tsugio Makimoto's Wave model [9, 10], Nick Tredennick summarizes the history of silicon application [125] in three phases (Figure 11.22a–c): Hardwired components like SSI, MSI, and LSI circuits which *cannot be programmed*; have fixed resources; and fixed algorithms (Figure 11.22a). The introduction of the microprocessor changes this set up to fixed resources but variable algorithms (Figure 11.22b). We need *only one programming source: Software* (Figure 22e). The advent of morphware has made both resources and algorithms variable (Figure 11.22c). We need *two programming sources: Configware* to program the resources, *and flowware* to program the data streams running through the resources (Figure 11.22f). An early implementation is the DPSS (Figure 11.17c, see also Section 3.1).

3.5 Cocompilation

Separate compilation of software and configware (Figure 11.18a and b) gives only limited support to reach the goal of good designer productivity. Especially to introduce *software / configware / flowware codesign* to CS professionals and CS curricula, we need *cocompilation techniques* to support application development at high abstraction levels. Figure 11.17c shows the typical structure of a *software / configware partitioning / cocompiler* (Figure 11.17c), where the configware part (DPSS in Figure 11.19b) includes both a *configware code generator* and a *flowware*

code generator. CoDe–X was an early implementation of a compiler of this kind, which was a *partitioning cocompiler* (Figure 11.19b and c), accepting C language input (pointers are not supported), which partitions source input to run on a symbiosis of a host and a rDPA [126–128]. This partitioner (Figure 11.19c) was based on the identification of usability of *loop transformations* [129–134]. This partitioner was implemented via *simulated annealing*. An additional *analyzer / profiler* (Figure 11.19c) was used for further optimization. Figure 11.19b shows the flowware / configware compiler (a version of the DPSS) as explained above, which was used as a subsystem inside the CoDe–X co–compiler. Figure 11.20a gives some DPSS details. *ALE–X* is an intermediate form derived from the C language.

Language category	Software languages	Flowware languages
Sequencing managed by	Read next instruction, goto (instruction address), jump (to instruction address), instruction loop, nesting, ***no*** *parallel loops*, escapes, instruction stream branching	Read next data item, goto (data address), jump (to data address), data loop, nesting, *parallel loops*, escapes, data stream branching
Data manipulation	Yes	Not needed
State register	Program counter	Single or multiple data counter(s)
Instruction fetch	Memory cycle overhead	No overhead
Address computation	Massive memory cycle overhead	Drastically reduced overhead

It is time to bridge the hardware / software chasm. We need a Mead-&-Conway–like edu rush [135].

A newer version of DPSS includes *KressArray Xplorer* (Figure 11.20a), a design space explorer to optimize KressArray DPU and rDPA architectures [98, 99]. As shown in Figure 11.20a mapping based on architecture description one yields a different array configuration than that based on architecture description two. Figure 11.20b illustrates the high flexibility of the KressArray family concept accepted by Xplorer. Path width and mode of each nearest neighbor connection can be individually selected. Also, a wide variety of second–level of *back bus interconnect* resources are available (not shown in the figure) featuring highly parallel buses or bus segments. Other design space explorers include *DSEs (Design Space Explorers*, survey: [88]), which use automatic guidance systems or design assistants to give advice during the hardware (and morphware) design flow, e.g., by DPE (Design Planning Environment) [136]; *Clio* [137] (both for VLSI); and DIA (for ASICs) [138]. *Platform Space Explorers (PSEs)* are used to find an optimum vN processor array, as with DSE [139], *Intelligent Concurrent Object-oriented Synthesis (ICOS)* [140], and *DSE for Multimedia Processors (DSEMMP)* [141].

3.6 A Dichotomy of Two Machine Paradigms

Traditionally hardware experts have been needed for morphware application development (Figure 11.7a, compare Section 2.4). Because of the rapid growth of

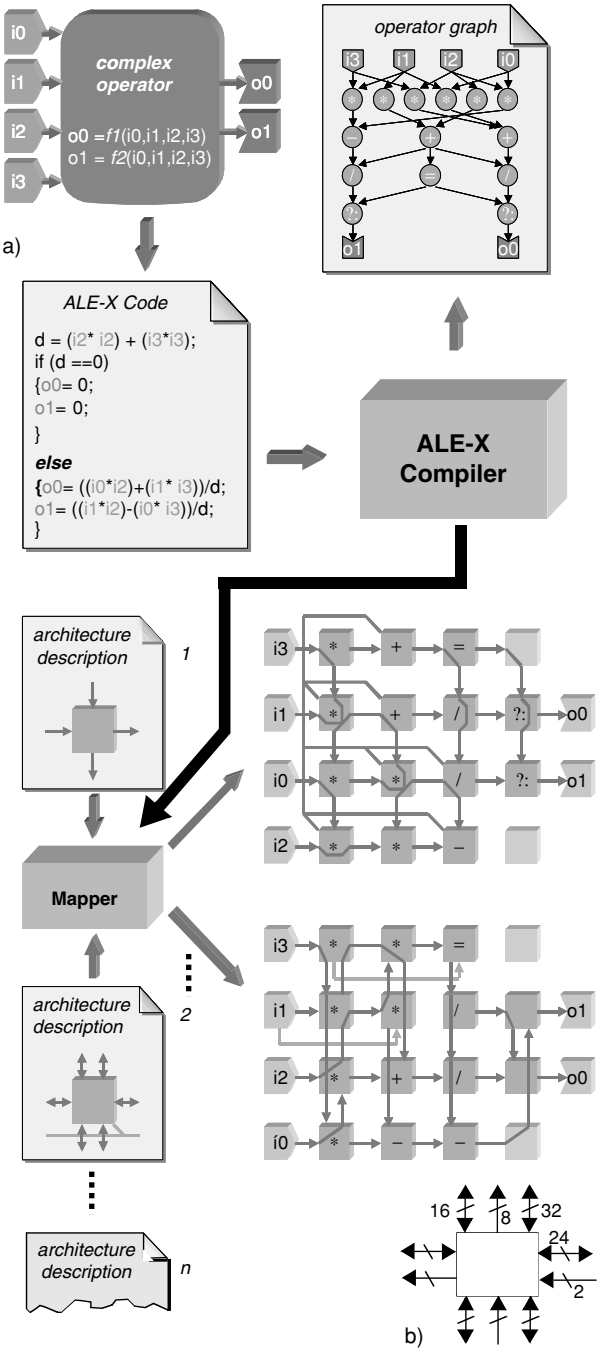


Figure 11.20. KressArray Xplorer (design space explorer [97]). a) Simplified example to illustrate platform space exploration by finding an optimized array depending on rDPU architecture (1 or 2); b) KressArray family rDPU example architecture illustrating flexibility.

Language category	Software Languages	Flowware Languages
Sequencing managed by	Read next instruction, goto (instruction address), jump (to instruction address), instruction loop, nesting, ***no*** *parallel loops*, escapes, instruction stream branching	Read next data item, goto (data address), jump (to data address), data loop, nesting, *parallel loops*, escapes, data stream branching
Data manipulation	Yes	Not needed
State register	Program counter	Single or multiple data counter(s)
Instruction fetch	Memory cycle overhead	No overhead
Address computation	Massive memory cycle overhead	Drastically reduced overhead

Figure 11.21. Software languages versus flowware languages.

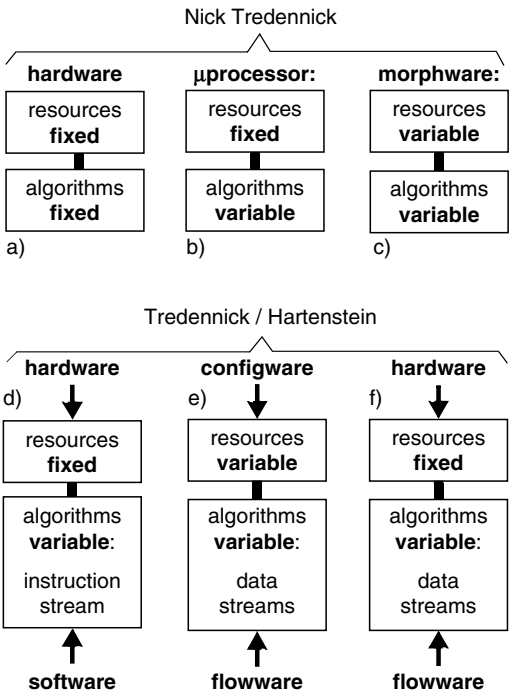


Figure 11.22. Nick Tredennick's digital system classification scheme. a) Hardwired; b) programmable in time; c) reconfigurable; d) von Neumann–like machine paradigm; e) reconfigurable antimachine paradigm, f) Broderson's hardwired antimachine.

the amount of code to be implemented for embedded systems [68], CS graduates are now also needed to handle the amount of work to be done. This expansion is hardly possible without moving to higher abstraction levels. Because it focuses on the design space, as the *von Neumann paradigm* does for software a second machine paradigm is needed as a simple guideline to implement flowware (and configware). This *antimachine paradigm* is summarized in Figure 11.23b–d. In contrast to the von Neumann paradigm (Figure 11.23a), the sequencer (data counter) has moved to the memory (as part of asM, an auto–sequencing memory bank), while the DPU of the antimachine has no sequencer (Figure 11.23b). The anti machine paradigm [141] also supports multiple data streams by multiple asMs providing multiple data counters (Figure 11.23c, d). That's why the antimachine has no von Neumann bottleneck. It does not need caches because of multiple data streams. Caches do not help because new data mostly have new values.

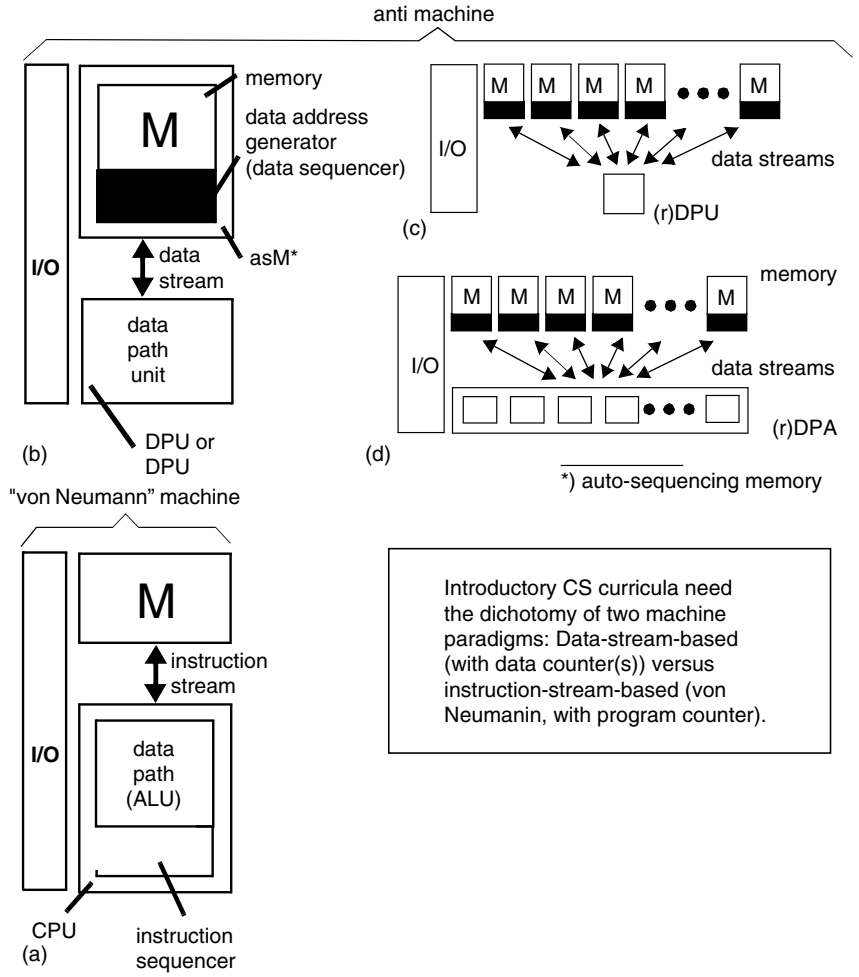


Figure 11.23. Illustrating basic machine paradigms (see Figure 11.19 legend). a) data–stream–based antimachine with simple DPU; b) with rDPU and distributed memory architecture; c) with DPU array (DPA or rDPA); d) von Neumann.

The enabling technologies for the antimachine architecture implementations are available [107, 108, 110–114, 142–144]. Figure 11.26a shows details of an antimachine mapped onto a KressArray, and Figure 11.26b shows the details mapped onto a PACT XPP array. The antimachine paradigm is useful for both morphware-based machines and hardwired machines ([145], etc.). The antimachine should not replace von Neumann: We need both machine paradigms. We need morphware to strengthen the declining vN paradigm.

The antimachine is not a *dataflow machine* [146] because it had been established by an old (now obsolete) research area that focused on an arbitration–driven machine, which checks, for each operator, whether all operands are available. In case of a reject, this operator can be resubmitted later. Such a machine operation is indeterministic, and for an algorithm, the total order of execution cannot be predicted. The execution of the vN machine and of the antimachine, however, is deterministic. However, the dataflow languages that have come along with this indeterministic paradigm [147] could also be useful sources for the antimachine.

4 THE IMPACT OF MORPHWARE ON COMPUTING SCIENCES

As labeled in Figure 11.24(3) the growth rate of algorithmic complexity [148] is higher than that of Moore's law (1), while the growth rate of microprocessor integration density (2) is far behind Moore's law. The improvement of computational efficiency in terms of mA needed per MIPS (5) has slowed down and is moving towards saturation. The performance requirements for wireless communication

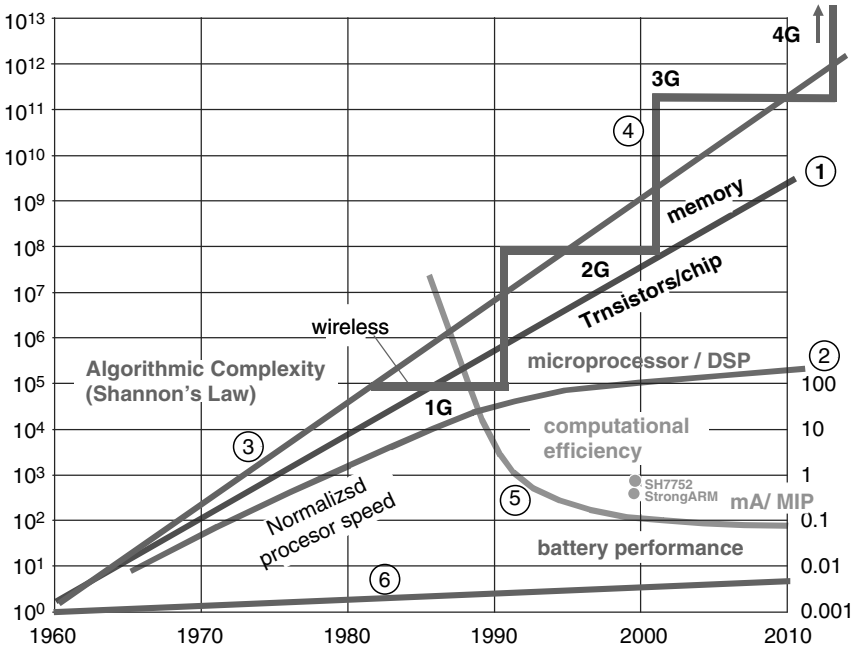


Figure 11.24. Computational requirements are growing faster than Moore's law.

(4) are rising by huge steps from device generation to device generation. Also, a number of other application areas such as multimedia or scientific computing (Section 2.3) suffer from a similar growth of requirements. Traditional HPC needs too much power: about 100W per gigaFLOPS [55]. Forth coming microprocessor generations promise only marginal performance improvements (Figure 11.25). A highly promising alternative is the microprocessor interfaced to a suitable coarse-grain array (Figure 11.17d), maybe for converting a PC into a PS (personal supercomputer). But such a PS will be accepted by the market only when it is accompanied by a good cocompiler (Figure 11.19b and c), the feasibility of which has been demonstrated [126–128].

The future of the microprocessor is no longer very promising: only marginal improvements can be expected for performance area efficiency (Figure 11.25). Power dissipation is becoming worse, generation by generation. The intel Itanium 2 on 130 nm technology with 410 million transistors dissipates 130 Watts at 1.3 Volts operating voltage [91] compared with 130 Watts at 1.6 Volts for the first Itanium. Traditional HPC (High Performance Computing) using such or similar microprocessors needs about 100W per gigaFLOPS [55]. Pipelined execution units within vN machines yield only marginal benefit for the price of sophisticated speculative scheduling strategies. Multithreading needs substantial overhead for any kind of multiplexing [149]. All these bad signs get added to the old limitations like the vN bottleneck [9, 147, 150–154]. Because of the increasing weakness of the microprocessor, we need a new computing paradigm as an auxiliary resource to cooperate with the microprocessor (Figure 11.16b). Morphware has arrived just in time. The future acceptance of the stand-alone operation of morphware is not very likely. Adding an rDPA and a good cocompiler to a microprocessor (Figure 11.17d) enables the PC to become a PS (personal supercomputer).

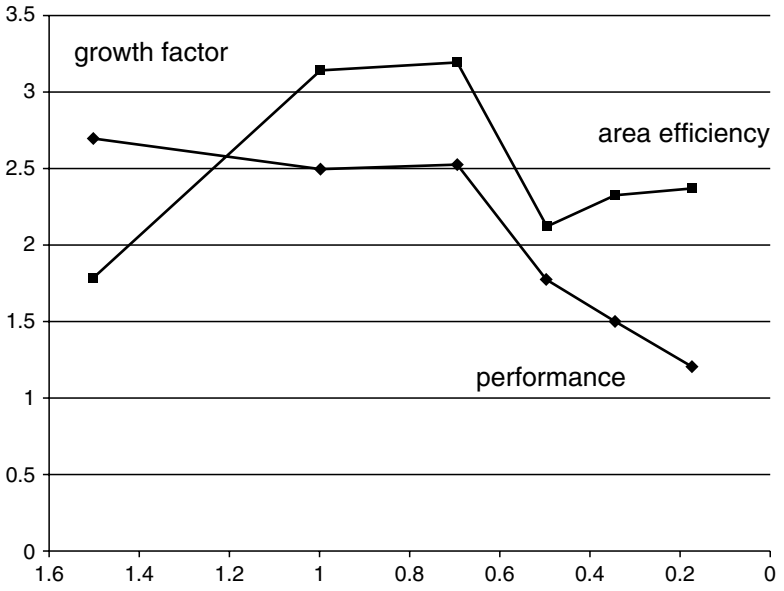


Figure 11.25. Pollack's Law (intel).

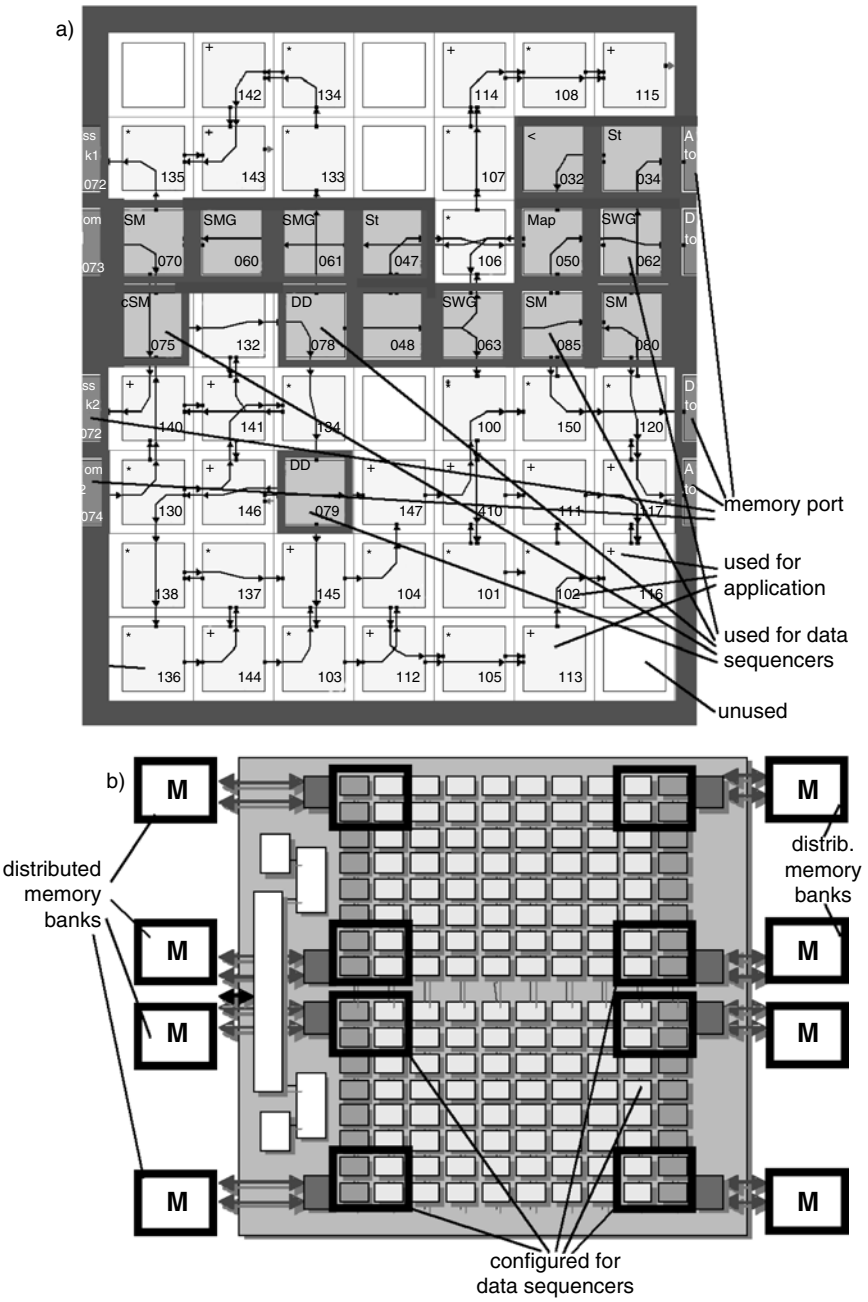


Figure 11.26. Antimachine mapped a) onto KressArray: synthesizable data sequencers mapped by KressArray Xplorer together with an application (linear filter) onto a KressArray; b) onto PACT XPP (another application example, distributed memory shown).

Static reconfiguration is straightforward and easy to understand. But dynamic reconfiguration tends to be tricky and difficult to understand and to debug.

SoC design rapidly adopts a CS mentality [67]. The amount of program code implemented for embedded systems doubles every 10 months and will reach 90% of all codes being written by the year 2010 [68]. Currently, a typical CS graduate with von–Neumann–only mentality does not have the skills needed for HW / CW / SW partitioning decisions, nor the algorithmic cleverness needed to transfer an application from software onto an FRGA. There is a trend to convey the codesign of embedded computing systems from the domain of hardware expertise over to CS methodologies. To cope with this challenge to CS curricula, the new antimachine paradigm and new compilation methods are needed.

The hardware/software chasm in professional practice and in education is causing damage amounting to billions of EURO each year worldwide. It is the main reason for the productivity gap in embedded system design. Meanwhile, it is widely accepted that morphware is a new computing paradigm. Morphware provides the enabling fundamentals to cope with this crisis. It is time to bridge the hardware/software chasm. We need a Mead–&–Conway–like rush [135]. We are already on the way. Scientific computing is using more and more Morphware. The international HPC conference IPDPS is coming along with the rapidly growing Reconfigurable Architectures Workshop (RAW [155, 156]). The number of attendees from HPC coming to conferences like FPL [20] and RAW is rapidly increasing. Special interest groups of professional organizations are changing their scope, e.g., PARS [32, 157–159].

There is sufficient evidence that morphware is breaking through as a new computing paradigm. Breaking away from the current mindset requires more than traditional technology development and infusion. It requires managerial commitment to a long-term plan to explore new thinking [96]. Morphware has just achieved its breakthrough as a second class of RAM–based programmable data processing platforms—a counterpart of the RAM–based von Neumann paradigm. Morphware combines very high flexibility and programmability with the performance and efficiency of hardwired accelerators.

4.1 Reconfigurable Computing versus Parallel Processing

A comprehensive treatment of important issues in parallel computing is provided by *The Sourcebook for Parallel Computing* [150], a key reference giving a thorough introduction to parallel applications, software technologies, enabling technologies, and algorithms. Classical parallelism by concurrent computing has a number of disadvantages over parallelism by antimachines having no von Neumann bottleneck, as is discussed elsewhere [105, 114, 151, 152]. In parallel computing, unfortunately, the scaling of application performance often cannot match the peak speed the resource platforms seem to provide, and the programming burden for these machines remains heavy. The applications must be programmed to exploit parallelism in the most efficient way possible. Today, the responsibility for achieving the vision of scalable parallelism remains in the

hands of the application. Amdahl's Law explains just one of several reasons for inefficient resource utilization [153]. vN–type processor chips are almost all memory, because the architecture is wrong [105]. Here the metric for what is a good solution has been wrong all along [105].

Reconfigurable versus parallel computing is also a very important issue for terminology—to avoid confusion. At the circuit level, all transistors look the same. So the question is how to distinguish switching within a reconfiguration fabric from other switching activities in an IC. The antimachine model introduced in section 3.6 is a good guideline for definition of the term *reconfigurable*. Switching during run time of instruction–stream–based operations, such as, addressing the register file is no reconfiguration. Switching inside a memory address decoder is also not reconfiguration. What about microprogramming? Is it reconfiguration? A microprogrammable instruction-set processor can be modeled by the nested machine model, showing that a microinstruction stream is also an instruction stream [149]. This means that running microcode *is not reconfiguration—it is execution* of a micro instruction stream. The following definitions will help us to avoid confusion. An important difference between reconfigurable computing and concurrent computing is determined by the binding time (Figure 11.27). Another important criterion is whether the code semantics is *structural* or *procedural*.

- The routing of data, addresses, and instructions during run time ***is not*** reconfiguration.
- Loading an instruction–stream–driven device to the program memory ***is not*** *reconfiguration*. It is procedural–style *programming* (instruction scheduling).
- Changing before their run time the effective *structure* of data paths and other resources: ***is definitely*** *reconfiguration*.
- Depending on the method used, dynamic reconfiguration (RTR) may be a hybrid, where parts of the system are running to manage the reconfiguration of other parts. (This chapter has already mentioned that RTR is quite a difficult subject.)

Within reconfigurable computing systems, the "instruction fetch" (i.e., setup of all computational resources and of all related communication paths) happens

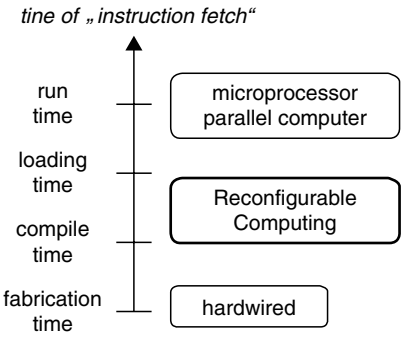


Figure 11.27. "Instruction Fetch."

before run time (Figure 11.27). We call this *reconfiguration* because it changes the effective structure of data paths and similar resources. Within concurrent computing systems, however, the instruction fetch and setup of all related communication paths happens *during run time* (Figure 11.27), which we *do not call reconfiguration*. The main difference with respect to performance is the amount of switching activity at run time, which is low for reconfigurable systems and high for instruction–stream–driven parallel computing. Depending on the application and the architecture, massively parallel concurrent systems may suffer heavily from communication congestion at run time. Because run time is more precious than compilation time, this migration of switching activities over to compile time or leading time is a welcome performance property of the morphware paradigm. Unfortunately, the distinction between parallel and reconfigurable computing is blurred by some projects labeled "reconfigurable" but that, in fact, deal with classical parallel computing on a single chip.

4.2 New Taxonomy Needed

We now live in a time exhibiting a shortage of analysts writing good and comprehensive surveys. What is currently missing and should soon be the subject of research is an all–embracing taxonomy of architectures and algorithms covering both areas, classical parallel computing and supercomputing, as well as reconfigurable computing. We need a taxonomy of architectures providing guidance in designing modern high–performance computing systems using resources from both areas, or to decide which area's resources provide the more promising alternatives. We also need all–embracing taxonomy algorithms to support the migration of applications or parts of applications from one area to another, for instance, from a vN platform to fine-grain morphware, or to coarse–grain morphware, or to mixed platforms. Such a taxonomy of algorithms should also survey the amount of interconnect resources needed by vN to morphware migration. Depending on the algorithm class, the interconnect requirements may show extremely wide variety. Some kinds of algorithms may be very easy to convert into pipelines, whereas others, for instance the parallelized Viterbi algorithm, may require enormously complex interconnect structures. A new taxonomy should be developed rapidly that supports the algorithmic cleverness needed for a good morphware–based designer productivity and for retrieving high–quality design solutions.

> We need a new taxonomy of architectures and algorithms.

We should not hesitate to reform CS and CSE curricula in order to prevent disqualification in the job market in the near future. Introductory undergraduate programming lab courses should not support the development of a procedural-only mindset. Such courses should rather be a guide to the world of embedded systems, requiring algorithmic cleverness for partitioning an application problem into cooperating software, flowware, and configware blocks. The exercises of such courses should feature varieties of tasks, including several subtasks of different nature, such as, (1) software implementation of the problem, (2) flowware

implementation of the problem, and (3) partitioning the problem into (3a) a software part, (3b) a flowware part, and (3c) development of the interface needed for its dual-paradigm coimplementation.

5 CONCLUSIONS

Morphware has become an essential and indispensable ingredient in SoC (System on a Chip) design and beyond. Already HDLs like VHDL (which is an Ada dialect), Verilog (a C dialect), and others are languages of higher abstraction levels and should be taught also to CS students.

The hardware/software chasm in professional practice and in education causes damage amounting to billions of EURO each year worldwide. It is the main reason for the productivity gap in embedded system design. Meanwhile, it is widely accepted that morphware is a new computing paradigm. Morphware provides the enabling fundamentals to cope with this crisis.

But most current work on reconfigurable systems is specialized and is not motivated by long-term aspects—wearing blinders that limit the view to particular applications, architectures, or tools. The long–term view, however, shows a heavy impact of reconfigurable computing upon the intellectual infrastructures of CS and CSE. This chapter has drafted a road map for upgrading CS and CSE curricula and for bridging the gap between a procedural and a structural mentality. The impact of morphware on CS will help to achieve this by evolution, rather than by revolution. You all should be evangelists for the diffusion of the visions needed to take this road and move out of the current crisis.

REFERENCES

[1] http://www.darpa.mil/ipto/programs/pca/vision.htm
[2] http://morphware.net/
[3] A. Burks, H. Goldstein, J. von Neumann (1946): Preliminary discussion of the logical design of an electronic computing instrument. US Army Ordnance Department Report.
[4] H. Goldstein, J. von Neumann, and A. Burks (1947): Report on the mathematical and logical aspects of an electronic computing instrument. *Princeton Institute of Advanced Study*.
[5] D. Jansen et al. (2003): The electronic design automation handbook, Kluwer.
[6] P. Gillick (2003): State of the art FPGA development tools. *Reconfigurable Computing Workshop*, Orsay, France.
[7] M. J. Smith (1997): Application specific integrated circuits, Addison Wesley.
[8] D. Chinnery and K. Keutzer (2002): Closing the gap between ASIC & custom, Kluwer.
[9] R. Hartenstein (invited paper) (1987): The Microprocessor is no more general purpose *Proc. IEEE International Symposium on Innovative Systems (ISIS)*, Austin, Texas.

[10] T. Makimoto (keynote) (2000): The rising wave of field–programmability, *Proc. FPL 2000*, Villach, Austria, August 27–30, Springer-Verlag, Heidelberg/New York.
[11] F. Faggin, M. Hoff, S. Mazor, and M. Shima (1996): The history of 4004. *IEEE Micro*. Dec. 1996.
[12] J. Becker (invited tutorial) (2003): Reconfigurable computing systems. *Proceedings Escola de Microelectrônica da SBC–Sul (EMICRO 2003)*. Rio Grande, Brasil, September.
[13] B. Lewis (2002): Gartner Dataquest, October 28.
[14] P. Athanas (1992): An adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration Ph.D thesis, Brown University, Providence, Rhode Island.
[15] S. Vassiliadis, S. Wong, and S. Cotofana (2001): The MOLEN rm–coded processor. *Proc. FPL*.
[16] M. Iliopoulos, T. Antonakopoulos (2000): Reconfigurable network processors based on field-programmable system level integrates circuits. *Proc. FPL*.
[17] http://www.fccm.org
[18] R. Hartenstein (1995): Custom computing machines. *DMM'95*, Smolenice, Slovakia.
[19] http://www.springer.de/comp/lncs/
[20] http://fpl.org
[21] S. Hauck (1998): The role of FPGAs in reprogrammable systems. *Proc. IEEE*.
[22] V. Betz, J. Rose, and A. Marquardt (eds.) (1999): Architecture and CAD for deep-submicron FPGas. Kluwer.
[23] S. Hoffmann (2003): Modern FPGAs, reconfigurable platforms and their design tools. *Proc. REASON summer school*. Ljubljana, Slovenia, August 11–13.
[24] D. Soudris et al. (2002): Survey of existing fine grain reconfigurable hardware platforms. *Deliverable D9 AMDREL consortium* (*Architectures and Methodologies for Dynamically Reconfigurable Logic*).
[25] J. Oldfield and R. Dorf (1995): Field–programmable gate arrays: Reconfigurable logic for rapid prototyping and implementation of digital systems. Wiley-Interscience.
[26] http://www.xilinx.com
[27] http://www.altera.com
[28] V. George and J. Rabaey (2001): Low–energy FPGAs: Architecture and design. Kluwer.
[29] Z. Salcic and A. Smailagic (1997): Digital systems design and prototyping using field programmable logic. Kluwer.
[30] J. Hamblen and M. Furman (2001): Rapid prototyping of digital systems. Kluwer.
[31] R. Männer and R. Spurzem et al. (1999): AHA-GRAPE: Adaptive hydrodynamic architecture–GRAvity PipE. *Proc. FPL*.
[32] G. Lienhart (2003): Beschleunigung hydrodynamischer N–Körper–simulationen mit rekonfigurierbaren rechensystemen. *Joint 33rd Speedup and 19th PARS Workshop*. Basel, Switzerland, March 19–21.
[33] N. Ebisuzaki et al. (1997): *Astrophysical Journal*, *480*, 432.

[34] T. Narumi, R. Susukita, H. Furusawa, and T. Ebisuzaki (2000): 46 Tflops Special– purpose computer for molecular dynamics simulations WINE-2. *Proc. 5th Int'l Conf. on Signal Processing*. Beijing 575–582.
[35] T. Narumi, R. Susukita, T. Koishi, K. Yasuoka, H. Furusawa, A. Kawai, and T. Ebisuzaki (2000): 1.34 Tflops molecular dynamics simulation for NaCl with a special–purpose computer: MDM. SC2000, Dallas.
[36] T. Narumi, A. Kawai, and T. Koishi (2001): An 8.61 Tflop/s molecular dynamics simulation for NaCl with a special–purpose computer: MDM. SC2001, Denver.
[37] T. Narumi, R. Susukita, T. Ebisuzaki, G. McNiven, and B. Elmegreen (1999): Molecular dynamics machine: Special–purpose computer for molecular dynamics simulations. *Molecular Simulation, 21*, 401–415.
[38] T. Narumi (1998): *Special–Purpose Computer for Molecular Dynamics Simulations Ph D dissertation*, University of Tokyo.
[39] T. Thurner (2003): Trends in der automobile–elektronik; *GI/ITG FG AH - Zielplan–Workshop at FDL 2003*. Frankfurt /Main, Germany.
[40] T. Kean (invited keynote) (2000): It's FPL, Jim–but not as we know it! Market opportunities for the New commercial architectures. *Proc. FPL*.
[41] R. Zeidman (2002): Designing with FPGAs and CPLDs. *CMP Books*.
[42] U. Meyer-Baese (2001): Digital signal processing with field programmable gate arrays (With CD-ROM). Springer-Verlag.
[43] K. Coffman (1999): Real World FPGA design with verilog. Prentice Hall.
[44] R. Seals and G. Whapshott (1997): Programmable logic: PLDs and FPGAs. McGraw-Hill.
[45] G. Martin and H. Chang (ed.) (2003): Winning the SoC revolution: Experiences in real design. Kluwer.
[46] G. Ou and M. Potkonjak (2003): Intellectual property protection in VLSI design. Kluwer.
[47] P. J. Ashenden (2001): The designer's guide to VHDL (2nd Ed.), Morgan Kaufmann.
[48] http://www.mentor.com/fpga/
[49] http://www.synplicity.com/
[50] http://www.celoxica.com/
[51] http://www.dac.com
[52] http://www.mathworks.com/products/connections/product_main.shtml?prod_id=304
[53] http://www.celoxica.com/methodology/matlab.asp
[54] http://www.mathworks.com/
[55] I. Jones (2003): DARPA funded Directions in embedded computing. *Reconfigurable Computing Workshop*. Orsay, France, Sept.
[56] T. Grötker et al. (2002): System design with system-C. Kluwer.
[57] http://www.synopsys.com/products/concentric_systemC/cocentric_systemC_ds.html
[58] http://www.systemc.org/
[59] http://www.synopsys.com/
[60] J. Hoe, Arvind: Hardware synthesis from term rewriting systems. *Proc. VLSI'99*. Lisbon, Portugal.

[61] M. Ayala-Rincón et al. (2003): Efficient computation of algebraic operations over dynamically reconfigurable systems specified by rewriting–logic environments. *Proc. 23rd SCCC. IEEE* CS press.
[62] M. Ayala-Rincón et al. (2003): Architectural specification, exploration and simulation through rewriting-logic. *Colombian J. Comput. 3*(2), 20–34.
[63] M. Ayala-Rincón et al. (2003): Using rewriting–logic notation for functional verification in data–stream–based reconfigurable computing. *Proc. FDL 2003 (Forum on Specification and Design Languages)*. Frankfurt /Main, Germany, September 23–26.
[64] P. Bjureus et al. (2002): FPGA Resource and timing estimation from matlab execution traces *10th Int'l Workshop on Hardware/Software Codesign*. Estes Park, Colorado, May 6–8.
[65] V. Baumgarten, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt (2003): PACT XPP -A self–reconfigurable data processing architecture. *The J. Supercomputing*. *26*(2), Sept. 2003, 167–184.
[66] J. Rabaey (1997): Reconfigurable processing: The solution to low-power programmable DSP. *Proc. ICASSP*.
[67] http://public.itrs. net/Files/2002Update/2002Update.htm
[68] N. N., Department of Trade and Industry (DTI), London, UK, 2001
[69] H. Simmler et al. (2000): Multitasking on FPGA coprocessors. *Proc. FPL*
[70] H. Walder and M. Platzner (2003): Reconfigurable hardware operating systems: From design concepts to realizations. *Proc. ERSA 2003*.
[71] H. Walder and M. Platzner (2004): A runtime environment for reconfigurable hardware operating systems. *Proc. FPL 2004*.
[72] R. Hartenstein (invited paper) (2002): Reconfigurable computing: Urging a revision of basic CS curricula. *Proc. 15th Int'l Conf. on Systems Engineering (ICSENG02)*. Las Vegas, USA, 6–8 Aug. 2002.
[73] course ID=27 in: http://vlsil.engr.utk.edu/~bouldin/C OURSES/HTML/courselist.html
[74] C. Stroud et al. (2002): BIST-based diagnosis of FPGA interconnect. *Proc. IEEE Int'l. Test Conf*.
[75] P. Zipf (2002): *A Fault Tolerance Technique for Field–Programmable Logic Arrays Dissertation*. Univ. Siegen, Germany.
[76] http.//directreadout.gsfc.nasa.gov
[77] M. Abramovici and C, Stroud (2000): Improved BIST–based diagnosis of FPGA logic blocks. *Proc. IEEE Int'l Test Conf*.
[78] http://www.xilinx.com/events/docs/e sc_sf2001_microblaze.pdf
[79] http://www.leox.org/
[80] J. Becker and M. Vorbach (2003): An industrial/academic configurable system– on–chip project (CSoC): Coarse.grain XPP/Leon–based architecture integration. DATE.
[81] http://www.gaisler. com/leonmain.html
[82] C. Mead and L. Conway (1980): Introduction to VLSI systems design. Addison-Wesley.
[83] R. Kress et al.: A datapath synthesis system (DPSS) for the reconfigurable datapath architecture. *Proc. ASP-DAC'95*
[84] http://pactcorp.com

[85] V. Baumgarten et al. (2001): PACT XPP – A self–reconfigurable data processing architecture. ERSA.
[86] J. Becker, A. Thomas, M. Vorbach, and G. Ehlers (2002): Dynamically reconfigurable systems–on–chip: A core-based industrial/academic SoC synthesis project. *IEEE Workshop Heterogeneous Reconfigurable SoC*. Hamburg, Germany, April 2002.
[87] J. Cardoso and M. Weinhardt (2003): From C programs to the configure–execute model. DATE.
[88] R. Hartenstein (2001): A decade of research on reconfigurable architectures. DATE.
[89] W. Mangione-Smith et al. (1997): Current issues in configurable computing research. *IEEE Computer*, Dec 1997.
[90] J. Becker, T. Pionteck, and M. Glesner (2000): An application–tailored dynamically reconfigurable hardware architecture for digital baseband processing. SBCCI.
[91] M. Sauer (2003): Issues in concept development for embedded wireless SoCs. *GI/ITG FG AH -Zielplan-Workshop*. Frankfurt/Main, Germany.
[92] A. Wiesler, F. Jondral (2002): A software radio for second and third generation mobile systems. *IEEE Trans. on Vehicular Technology. 51*, (4), July.
[93] N. Petkov (1992): Systolic parallel processing. North-Holland.
[94] M. Foster, H. Kung (1980): Design of special-purpose VLSI chips: Example and opinions. *ISCA*.
[95] H. T. Kung (1982): Why systolic architectures? *IEEE Computer 15*(1), 37–46
[96] http://directreadout.gsfc.nasa.gov
[97] U. Nageldinger et al. (2000): Generation of design suggestions for coarse-grain reconfigurable architectures *FPL 2000*.
[98] U. Nageldinger (2001): *Coarse–grained Reconfigurable Architectures Design Space exploration Dissertation*, – downloadable from [99]
[99] http://xputers.informatik.uni-kl.de/papers/publications/NageldingerDiss.html
[100] J. Frigo et al. (2001): Evaluation of the streams–C C–to–FPGA compiler: An applications perspective. *FPGA*.
[101] T.J. Callahan: Instruction–level parallelism for reconfigurable computing. *FPL'98*
[102] E. Caspi et al. (2000): Extended version of: Stream computations organized for reconfigurable execution (SCORE). *FPL '2000*.
[103] T. Callahan (2000): Adapting software pipelining for reconfigurable computing. CASES
[104] H. Kwok-Hay So, BEE (2000): *A Reconfigurable Emulation Engine for Digital Signal Processing Hardware M.S. thesis*, UC Berkeley.
[105] C. Chang, K. Kuusilinna, R. Broderson (2002): The biggascale emulation engine. *FPGA*.
[106] B. Mei et al. (2003): Exploiting loop–level parallelism on coarse–grained reconfigurable architectures using modulo scheduling. DATE 2003.
[107] M. Herz et al. (invited paper) (2002): Memory organization for data–stream–based reconfigurable computing *ICECS*.

[108] M. Herz et al. (1997): A novel sequencer hardware for application specific computing. *Proc. ASAP*.
[109] H. Reinig et al. (1995): Novel sequencer hardware for high–speed signal processing. *Proc. Design Methodologies for Microelectronics*, Smolenice, Slovakia.
[110] M. Herz (2001): *High Performance Memory Communication Architectures for Coarse-grained Reconfigurable Computing Systems Ph.D. thesis*, Kaiserslautern – downloadable from: [111]
[111] http://xputers.informatik.uni-kl.de/papers/publications/HerzDiss.html
[112] F. Catthoor et al. (2002): Data access and storage management for embedded programmable processors. Kluwer.
[113] F. Catthoor et al. (1998): Custom memory management methodology exploration of memory organization for embedded multimedia systems design. Kluwer.
[114] M. Weber et al. (1988): MOM–map oriented machine. In (E. Chiricozzi, A. D'Amico (ed.) *Parallel Processing and Applications*. North-Holland.
[115] A. Hirschbiel et al. (1987): A flexible architecture for image processing. *Microprocessing and Microprogramming*. *21*, 65–72.
[116] A. Ast et al. (1994): Data–procedural languages for FPL–based machines. *FPL'94*.
[117] E. Mirsky and A. DeHon (1996): MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. *Proc. IEEE FCCM'96*. April 17–19 Napa, CA, USA.
[118] E. Waingold et al. (1997): Baring it all to software: RAW machines. *IEEE Computer*. 86–93.
[119] J. Becker et al. (2000): Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. *Proc. FCCM'00*. April 17–19, Napa, CA, USA.
[120] C. Ebeling et al. (1996): RaPiD: Reconfigurable pipelined datapath. *Proc. FPL'96*.
[121] S. C. Goldstein et al. (1999): PipeRench: A coprocessor for streaming multimedia acceleration. *Proc. ISCA'99*, May 2–4 Atlanta.
[122] D. Chen and J. Rabaey (1990): PADDI: Programmable arithmetic devices for digital signal processing. *VLSI Signal Processing IV*, IEEE Press.
[123] D. C. Chen and J. M. Rabaey (1992): A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high–speed DSP data paths. *IEEE J. Solid–State Circuits*. *27*(*12*).
[124] A. K. W. Yeung and J. M. Rabaey (1993): A reconfigurable data–driven multiprocessor architecture for rapid prototyping of high throughput DSP algorithms. *Proc. HICSS-26*. Jan. Kauai, Hawaii.
[125] N. Tredennick (1995): Technology and business: Forces driving microprocessor evolution. Dec. *Proc. IEEE*.
[126] J. Becker et al. (1998): Parallelization in co–compilation for configurable accelerators. *Proc. ASP-DAC'98*.
[127] J. Becker (1997): *A partitioning compiler for computers with Xputer–based Accelerators Ph.D. Dissertation*, University of Kaiserslautern. downloadable from [128].

[128] http://xputers.informatik.uni-kl.de/papers/publications/BeckerDiss.pdf
[129] L. Lamport (1974): The parallel execution of Do-loops. *C. ACM 17*, *2*, Feb.
[130] D. Loveman (1977): Program improvement by source–to–source transformation. *J. ACM 24*, 1.
[131] W. Abu-Sufah, D. Kuck, and D. Lawrie (1981): On the performance enhancement of paging systems through program analysis and transformations. *IEEE-Trans. C-30*(5).
[132] U. Banerjee (1979): *Speed–up of ordinary programs; Ph.D. Thesis*, University of Illinois at Urbana-Champaign, Oct. DCS Report No. UIUCDCS-R-79-989.
[133] J. Allen, K. Kennedy (1984): Automatic loop interchange. *Proc. ACM SIG–PLAN'84, Symp. on Compiler Construction*, Montreal, Canada, SIGPLAN Notices June 19, 6.
[134] J. Becker and K. Schmid (1998): Automatic parallelism exploitation for FPL–based accelerators. *Hawaii Int'l. Conf. on System Sciences (HICSS'98)*, Big Island, Hawaii.
[135] http://xputers.informatik.uni-kl.de/staff/hartenstein/eishistory_en.html
[136] D. Knapp et al. (1991): The ADAM design planning engine. *IEEE Trans CAD*.
[137] J. Lopez et al. (1992): Design assistance for CAD frameworks. *Proc. EURODAC'92*. Hamburg, Sept. 7–10, Germany.
[138] L. Guerra et al. (1998): A methodology for guided behavioral level optimization. *Proc. DAC'98*, June 15–19, San Francisco.
[139] C. A. Moritz et al. (1999): Hot Pages: software caching for RAW microprocessors. MIT. LCS-TM-599, Aug. Cambridge, MA.
[140] P.-A. Hsiung et al. (1999): PSM: An object–oriented synthesis approach to multiprocessor design. *IEEE Trans VLSI Systems 4/1*. March.
[141] J. Kin et al. (1999): Power efficient media processor design space exploration. *Proc. DAC'99*. June 21–25, New Orleans, http://anti-machine.org.
[142] K. Schmidt et al. (1990): A novel ASIC design approach based on a new machine paradigm. *J. SSC* -invited reprint from *Proc. ESSCIRC*.
[143] W. Nebel et al. (1984): PISA, a CAD package and special hardware for pixel-oriented layout analysis. *ICCAD*.
[144] R. Hartenstein et al. (1990): A novel paradigm of parallel computation and its use to implement simple high performance hardware. *Future Generation Computer Systems 791/92*, -invited reprint fr. Proc. InfoJapan'90 (Int'l Conf. Commemorating the 30th Anniversary Computer Society of Japan), Tokyo, Japan.
[145] C. Chang et al. (2001): The biggascale emulation engine (Bee). summer retreat UC Berkeley.
[146] D. Gajski et al. (1982): A second opinion on dataflow machines. *Computer*, Feb.
[147] J. Backus (1978): Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, August, *20*(8), 613–641.
[148] J. Rabaey (keynote) (2000): Silicon Platforms for the Next Generation Wireless Systems. *Proc. FPL*.

[149] G. Koch et al. (1975): The universal bus considered harmful. *Proc. 1st EUROMICRO Symposium on the microarchitecture of computing systems*. Nice, France, North Holland.
[150] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White (ed.) (2002): The sourcebook of parallel computing. Morgan Kaufmann.
[151] Arvind et al. (1983): A Critique of Multiprocessing the von Neumann Style. *Proc. ISCA*.
[152] G. Bell (keynote) (2000): All the chips outside. The architecture challenge. *Proc. ISCA*.
[153] G. Amdahl (1967): Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings*. (30).
[154] J. Hennessy (1999): ISCA25: Looking backward, looking forward. *Proc. ISCA*.
[155] http://www.ece.lsu.edu/vaidy/raw04/
[156] http://xputers.informatik.uni-kl.de/raw/index_raw.html
[157] http://www.iti.uni-luebeck.de/PARS/
[158] http://www.speedup.ch/
[159] http://www.hoise.com/primeur/03/articles/monthly/AE-PR-04-03-61.html