# EMBEDDED SYSTEM ARCHITECTURES

ROLF ERNST

*Technische Universität Braunschweig*

*Institut für Datenverarbeitungsanlagen*

*Hans-Sommer-Str. 66*

*D-38106 Braunschweig*

*Germany*

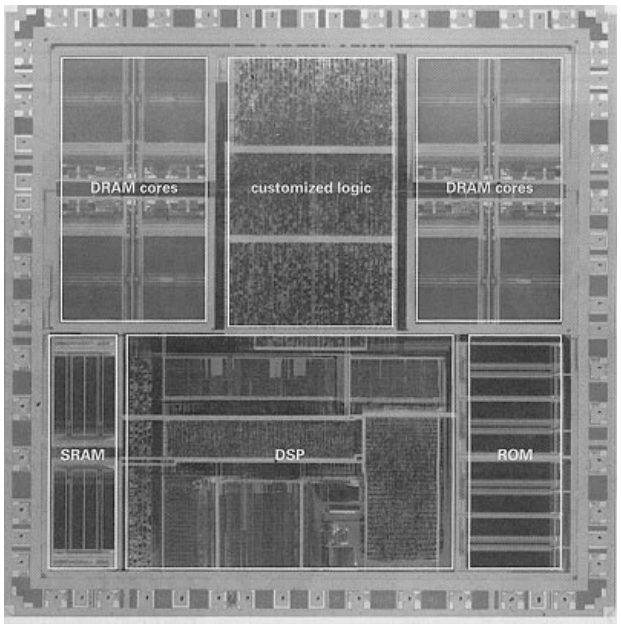
## 1. Introduction

This chapter surveys the target architectures which are typically found in embedded systems and explains their impact on the system design process. Embedded systems have a high impact on the overall industry defining the features of telecommunication equipment, consumer products, or cars, to name just a few industrial sectors where these systems are ubiquitous. There is a large variety of embedded systems architectures which significantly differ from general purpose computer architectures. Despite their importance, these architectures are typically neglected in university curricula.

Embedded system architectures are determined by **circuit technology**, **application requirements**, and **market constraints**. We will briefly review the trends in each of these areas to understand the resulting trends in target architectures and their design.

The trends in circuit technology are best explained using the 1997 SIA roadmap [1]. This roadmap forecasts the development of the semiconductor industry in the next 15 years based on the expertise of a large part of the semiconductor industry. It predicts the reduction in minimum feature size (transistor gate length) from 200nm in 97 to 35nm in 2012. Considering the increasing maximum economically useful chip size, standard microprocessors are expected to grow from 11M transistors today up to 1.4 billion transistors in 2012 while DRAMs will reach 256Gbit capacity.

Besides general purpose processor architectures, memory intensive embedded system applications, in particular multimedia and embedded image processing (3D), high throughput telecommunication, or powerful PDAs are most likely to be the technology drivers for this development. On the way to 2012, an increasing number of complex systems consisting of many processors, coprocessors and memory of different types and sizes will be integrated on a single chip (systems on chip SOC). The integration of dynamic memories and standard logic has been demonstrated in commercial designs (e.g.




SOC with 16 bit DSP, 15k gate equiv. custom logic, SRAM, ROM, 1Mbit DRAM

*Figure 1*: Audio processing system (Siemens) [2]

[2], see fig. 1) and will enable further system integration, but requires new tradeoffs in VLSI system design, as will be explained later. It is not unlikely that memory issues will eventually dominate system design.

In parallel to the growth in system complexity, the design cycle time is expected to decrease to 55% when compared to 97 [1]. To reach this cycle time at increasing system sizes, the individual design productivity must grow 160 times over the next 15 years. This tremendous productivity goal which faces increasing physical design problems seems only possible when more than 90% of a design can be reused from previous designs or from a library [1]. These reused components will come from different sources with exchange across companies. The transfer of intellectual property between companies has coined the term **IP** for these reused parts.

The reuse requirements will have a large impact on the design process, in particular since reuse must be combined with a significant rise in the level of design automation to account for the remaining productivity gain requirements (90% reuse alone provides a maximum productivity gain of 10).

On the other hand, many embedded system applications will not require such a huge circuit capacity. Already today, the vast majority of embedded systems is integrated on single chip microcontrollers with a 4 to 16 bit processor as core. Some of these systems,

however, implement rather complex functions, such as in automotive control, where distributed embedded systems are required to closely cooperate. Active suspension or drive-by-wire are just two examples of such functions. So, a discussion of embedded system architectures should not be restricted to single chip systems. Nevertheless, research should focus on the large circuit technology drivers to enable the required productivity gain.

System complexity will even grow faster than VLSI complexity since an increasing part of embedded system functionality will be implemented in software. Software can be used to adapt an SOC to different applications or for product diversification, i.e. for creating products with different features based on the same hardware platform. Examples are mobile phones with different features or automotive controllers for different motor types and different countries (emission control). The higher volume of the common hardware platform can reduce the overall system cost. Therefore, additional flexibility may account for slightly higher system complexity, but without compromising system constraints, such as minimum battery lifetime. On the other hand, higher system integration gives more opportunities for system specialization and cost reduction. Also, there are new forms of flexibility, such as FPGA components and reconfigurable systems which must be taken into account. Flexibility of system components or subsystems may also be used to enable the inclusion of IP. The controlled use of flexibility will, hence, become an important factor in system design.

Much more than general purpose computer design, embedded system design is affected by non functional market constraints, most importantly:

- **Strict cost margins** Often, the design objective is not only to minimize cost but to reach a certain market segment consisting of a price range and an expected functionality. So, if cost is above the intended price margin, functionality and performance may have to be redefined. This is different from typical general purpose microprocessor design for PC and workstation applications, where performance plays a dominant role.

- **Time-to-market and predictable design time** This constraint also applies to general purpose processor design except that in embedded system design, the specification includes all software functions. Furthermore, the development can lead to a family of similar products or include other embedded products, which might not yet be completely defined.

- **(Hard) time constraints** Many embedded systems are subject to hard time constraints. By definition, when hard time constraints [3] are violated, a major system malfunction will occur. This is often the case in safety critical systems. For such systems, timing predictability is of primary importance. Modern RISC processors with long pipelines and superscalar architectures, however, support dynamic out of order execution which is not fully predictable at compile time. So, the obtained speedup is to a large part useless in the context of hard real-time requirements. This is not an issue in a PC environment.

- **Power dissipation** is certainly also a problem in PC processor design. In portable systems, such as PDAs and mobile phones, however, battery life time is also an important quality factor.
- **Safety** Many embedded applications such as automotive electronics or avionics have safety requirements. There is a host of work in self checking and fault tolerant electronic systems design [4, 5] but, in embedded systems, mechanical parts must be regarded as well. Safety requirements can have very subtle effects on system design when we take EMI (electromagnetic integrity) into account [6]. As an example, the clock frequency may be constrained since (differential mode) radiation of a digital system increases roughly with the square of the clock frequency.
- **Physical constraints**, such as size, weight, etc...

We may assume that designers will only accept computer aided design space exploration and system optimization if the resulting target architectures show a quality comparable to manual designs. We need a thorough understanding of these architectures and their underlying design techniques to be able to match their quality in computer aided co-design. This not only holds for an automated design process, but even more for design space exploration where the designers would expect to have the same design space available as in manual designs. Moreover, target architecture knowledge is a prerequisite for efficient and accurate estimation of performance, cost, power dissipation and safety.

While current system synthesis tools are limited to a small set of architectures and optimization techniques, future CAD tools for 100 million transistor IC designs will have to cover a wide range of architectures for efficient exploration of a huge design space. To explore the inhomogeneous system design space, we will need design transformation, which allows to map a system to different types of target architectures such as a parallel DSP, a microcontroller or an ASIC or an application specific processor. However, we will show that the wide variety of different target architectures and components can be explained with a small set of specialization techniques and communication principles which might be exploited in system design space exploration.

The next section introduces the specialization techniques which will then be applied to explain the different classes of target architectures in use today. We will use small systems which will be sufficient to show the main target architecture characteristics. We will focus on the main principles and will exclude other aspects such as test, fault tolerance, or production. The examples are selected to demonstrate the concepts and shall not imply any judgment on design quality. We will assume a working knowledge in general purpose computer architecture, such as represented e.g. by [7].

The rest of this chapter will be organized as follows. We will first explain how embedded system design exploits specialization techniques to meet the design objective and reduce cost. In this step, we will assume that all system design parameters are free corresponding to an ideal system design from scratch. We will show practical examples of specialization using different techniques for different levels of flexibility using an important application, multimedia processors. In the last section, we will briefly summarize

the influence of IP reuse and the relation between IP reuse and flexibility. We will conclude with proposals for research topics.

## 2. Architecture Specialization Techniques

### 2.1. COMPONENTS AND SYSTEMS

Hardware design develops components and integrates them into larger entities. The size of the components depends on the design task. For the purpose of this paper, system design components shall be defined as function blocks, such as data paths, buses, and memory blocks, or combinations of these elements with a single thread of control, such as core processors, coprocessors or memories. Systems have multiple threads of control requiring load distribution and optimization of control and data flow.

We can classify architecture specialization in 5 independent specialization techniques for components and additional techniques to support load distribution and to organize data and control flow in a system of specialized components. Components shall be defined to have a single control thread while systems may have multiple control threads. The reference system used for comparison is always a general purpose programmable architecture. We start bottom up with specialization techniques for individual components and will then derive specialization techniques for systems.

### 2.2. COMPONENT SPECIALIZATION TECHNIQUES

The first technique is **instruction set specialization**. It defines the operations which can be controlled in parallel. It can be adapted to operations which can frequently be scheduled concurrently in the application or which can frequently be chained. An example for the first case is the next data address computation in digital signal processors (DSPs) which can mostly be done in parallel to arithmetic operations on these data, and an example for the second case are multiply-accumulate or vector operations which can also be found in signal processing applications.

The potential impact of instruction set specialization is the reduction of instruction word length and code size (chaining) compared to a general purpose instruction set for the same architecture which in turn reduces the required memory size and bandwidth as well as the power consumption.

The second technique is **function unit and data path specialization**. This includes word length adaptation and the implementation of application specific hardware functions, such as for string manipulation or string matching, for pixel operations, or for multiplication-accumulation. Word length adaptation has a large impact on cost and power consumption while application specific data paths can increase performance and reduce cost. As a caveat, specialized complex data paths may lead to a larger clock cycle time which potentially reduces performance. This problem, however, is well known from general purpose computer architecture. In embedded systems, however, a slower

clock at the same performance level might be highly desirable to reduce board and interconnect cost or electromagnetic radiation.

The third technique is **memory specialization**. It addresses the number and size of memory banks, the number and size of access ports (and their word length) and the supported access patterns such as page mode, interleaved access, FIFO, random access, etc. This technique is very important for high performance systems where it allows to increase the potential parallelism or reduce the cost compared to a general purpose system with the same performance. The use of several smaller memory blocks can also reduce power dissipation.

The fourth technique is **interconnect specialization**. Here, interconnect shall include the interconnect of function modules, as well as the protocol used for communication between interconnected components. Examples are a reduced bus system compared to general purpose processors to save cost or power consumption or to increase performance due to a minimized bus load. Interconnect minimization also helps to reduce control costs and instruction word length and, therefore, increases the concurrency which is "manageable" in a processor component. One can also add connections to specialized registers (memory specialization) which increases the potential parallelism. Examples include accumulator registers and data address registers or loop registers in digital signal processors.

The fifth technique is **control specialization**. Here, the control model and control flow are adapted to the application. As an example, microcontrollers often use a central controller without pipelining. Pipelining is employed in high performance processors while distributed control can be found in complex DSP processors as we will see later. The individual controller can be microprogrammed or an FSM, fully programmable or hard wired.

## 2.3. SYSTEM SPECIALIZATION TECHNIQUES

System specialization has two aspects, **load distribution** and **component interaction**.

Unlike load distribution in parallel systems, embedded systems are typically heterogeneously structured as a result of component specialization. Unlike load distribution in real-time systems (which may be heterogeneous as well), the target system is not fixed. In hardware/software co-design, the distribution of system functions to system components determines how well a system can be specialized and optimized. *So, hardware/software co-design adds an extra dimension to the cost function in load distribution, which is the potential specialization benefit.*

Two load distribution objectives seem to be particularly important to embedded system design, control decomposition and data decomposition.

**Control decomposition** (which could also be called **control clustering**) partitions system functions into locally controlled components *with the objective to specialize component control*. As a result, some of the components are fully programmable processors, some are hard coded or hard wired functions without a controller (e.g. filters and periph-

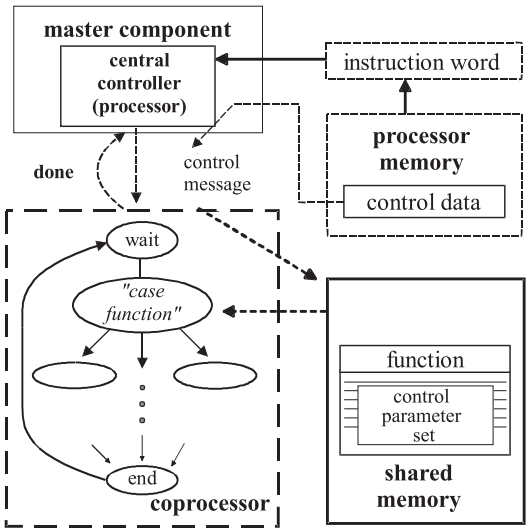



*Figure 2*: Dependent coprocessor

erals) and some components are implemented with a set of selectable, predefined control sequences, possibly with chaining. An example for the latter case are floating point coprocessors, DMA coprocessors or graphics coprocessors. The components control each other which requires a global control flow. Again, the global control flow is specialized to the system and is heterogeneous in general. We will explain the global control flow techniques in the next section.

**Data decomposition** (or **data clustering)** partitions the system functions and their data with the objective to specialize data storage and data flow between components. This leads to different, specialized memory structures within and between the components, as we have seen in the context of component memory specialization.

In the next sections, we will have a closer look at the techniques of global control flow and global system communication which are needed to support global specialization techniques.

## 2.4. SYSTEM SPECIALIZATION TECHNIQUES - A CLOSER LOOK

Control specialization which is restricted to individual components results in a system of communicating but otherwise *independently controlled components*. Control specialization in a system can go further and minimize component control such that it becomes dependent on other components. Such asymmetric control relationships require global control flow. Again, there is a large variety of global control mechanisms, but they adhere to few basic principles. We will distinguish independently controlled components, depend-

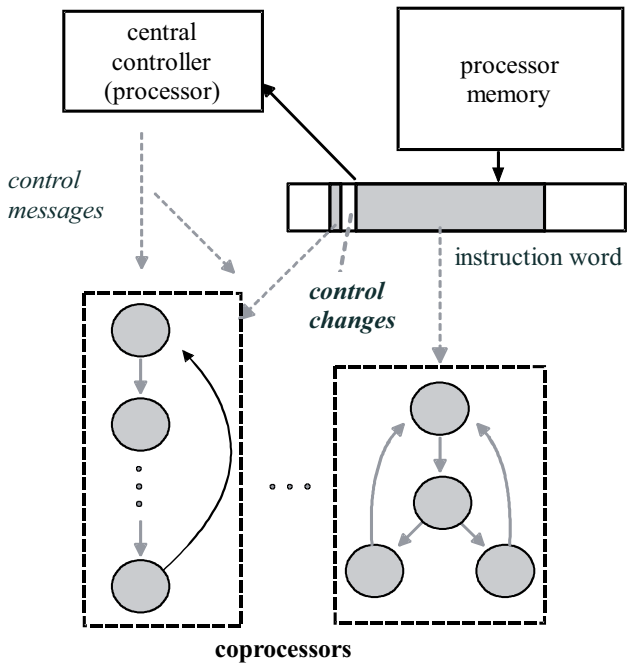



*Figure 3*: Incrementally controlled coprocessors

ent coprocessors, incrementally controlled coprocessors and partially dependent coprocessors.

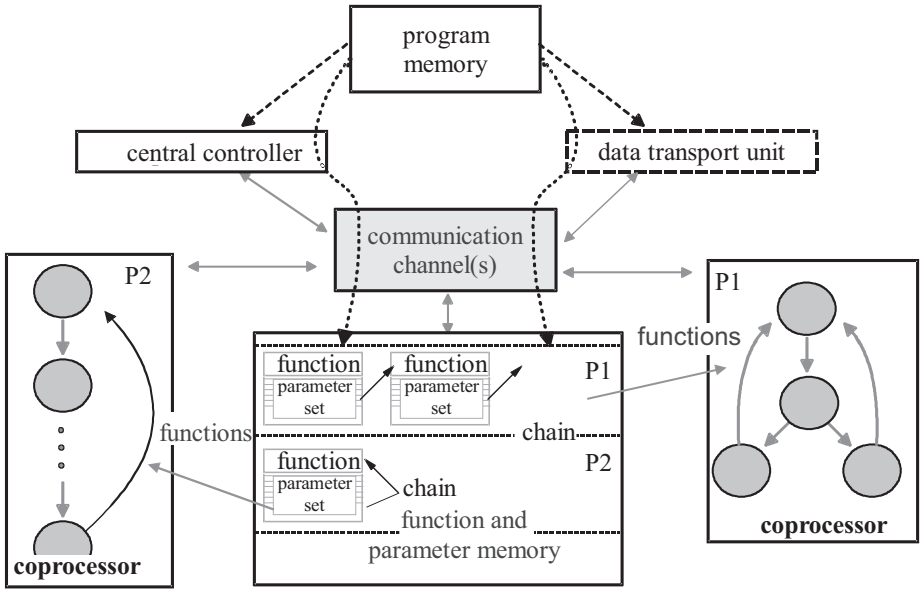
The first class are **independently controlled components**. If the components are programmable processors, we would classify the system as MIMD [8]. There is no global control flow required between such components.

On the other extreme are components which completely depend on the controller of another component. They shall be called **dependent coprocessors**. The controlling component shall be the *master component*, usually a processor, with the *central controller*, usually a processor control unit. Fig. 2 shows the typical implementation of the global control flow required for dependent coprocessors. The master component sends a control message to the *coprocessor* which it derives from its own state (in case of a processor: instruction). As a reaction, the *coprocessor* executes a single thread of control and then stops to wait for the next *control message*. So, the *coprocessor* behavior can be modeled with a "wait" and a "case" statement as shown in fig. 2.

If needed, additional control parameters (control data) are passed in a *control parameter* set, typically via *shared memory* communication. This alleviates the *master component* from control data transmission. Examples are small peripheral units such as UARTs (serial I/O) or PIOs (parallel I/O). From an instruction set point of view, we can consider the coprocessor functions as complex, multi-cycle microcoded processor functions.

Master component execution can continue in parallel to the coprocessor (multiple system control threads) or mutually exclusive to the coprocessor (single system control thread). Dependent coprocessors can also be found in many general purpose architectures, such as floating-point coprocessors, or graphics coprocessors [9]. It is also the coprocessor model of many co-synthesis systems, such as COSYMA [10].

The second class shall be called **incrementally controlled coprocessors**. Incrementally controlled coprocessors can autonomously iterate a control sequence. A central controller (the master component frame is omitted for simplicity) only initiates changes in the control flow and in the control data of the *coprocessor(s)* (fig. 3). This technique reduces the amount of control information for locally regular control sequences. In case of such regularity, the same instruction word can be shared among the main component (e.g. a *processor* ) and several coprocessors such that the control "effort" is shifted between the units in subsequent instructions using different instruction formats. A good illustration of this approach is found in some digital signal processors with incrementally controlled concurrent address units which exploit highly regular address sequences (see examples below).




(also: coprocessors with program)

*Figure 4*: Partially dependent coprocessors

Both dependent coprocessors and incrementally controlled coprocessors are a part of a control hierarchy. Even if components are independent with their own control memory, there can still be a control hierarchy where a master component defines the program to

be executed by a coprocessor. Such coprocessors shall be called *partially dependent coprocessors*. Partially dependent coprocessor implementation is strikingly similar in very different architectures, such as digital signal processors and complex peripheral controllers. The central controller typically downloads a linked set of function blocks each consisting of a function designator and a parameter block. The links can form a circle for iterative execution. The mechanism only allows very simple round robin linear sequences of functions but each function may correspond to a complex (micro)program with data dependent control flow. This is outlined in fig. 4. Obviously, these independent coprocessors are well suited for shared memory architectures.

The control hierarchy can be extended to several levels as we will see in some of the more complex examples.

### 2.5. MEMORY ARCHITECTURES

Memory architecture design is subject to system specialization techniques as well. The application data set can be decomposed and mapped to several specialized memories to optimize cost, access time or power consumption. A priori knowledge of application data structures and data flow can be used in several ways. Besides extending the component memory specialization techniques to systems, **system memory specialization** exploits the data flow within a system. A priori knowledge of data flow is used to insert and optimize buffers, assign memories for local variables and distribute global data such as to minimize access time and cost. This is known from DSP design, such as the TMS320C80 [11], which will be discussed as an example, or the VSP2 [12], but has recently also drawn some attention in general purpose processor design, where part of the memory architecture is specialized to multimedia application, e.g. using stream caches in addition to standard caches. There are even formal optimization techniques, in particular for static data flow systems [13]. One can also exploit knowledge of data access sequences either to reduce memory access conflicts with memory interleaving or e.g. stream caches reduce address generation costs by taking advantage of linear or modulo address sequences typical for many signal processing or control algorithms. Again, there are optimization approaches, e.g. [13, 14]. Reducing memory conflicts, however, can lead to less regular address sequences. So, regular address sequences and memory access optimization are potentially conflicting goals.

## 3. Target Architectures and Application System Classes

### 3.1. THE SPECIALIZATION PROBLEM

The main difference between general purpose highest volume microprocessors and embedded systems is specialization. We have seen a variety of specialization techniques on the component and system levels to optimize an embedded system for a given target application or a class of target applications. Nevertheless, efficient specialization requires suitable system properties. So, given the specialization techniques, a designer should identify the specialization potential of a target application with respect to the specializa-

tion techniques, before a decision on the architecture is made. Moreover, specialization must not compromise flexibility, i.e. it must possibly cover a whole class of applications (DSP) and include some flexibility for late changes. This also limits specialization. What is needed is a target application analysis which

- identifies application properties that can be used for specialization,
- quantifies the individual specialization effects, and
- identifies and takes flexibility requirements into account.

Unfortunately, there are no formal methods today which would support the three tasks, except for quantitative techniques from computer architecture [7], which may help in the second issue. Therefore, system specialization and design is currently for the most part based on user experience and intuition.

3.1.1. *Application System Classes and Examples*
To give an impression on how specialization is done in practice, we will discuss target architectures for several important but very different application system classes. We selected high volume products which have been used for a few years, where we may assume that a large effort went into careful manual system optimization, and where the design decisions have already been validated by a larger number of applications. We will distinguish four application system classes,

- **computation oriented systems** such as workstations, PCs or scientific parallel computers,
- **control dominated systems**, which basically react to external events with relatively simple processes,
- **data dominated systems** where the focus is on complex transformation or transportation of data, such as signal processing, or packet routing, and
- **mixed systems**, such as mobile phones or motor control which contain extensive operations on data as well as reaction to external events. Here, we will only look at the latter three classes.

## 4. Architectures for Control Dominated Systems

Control dominated systems are reactive systems showing a behavior which is driven by external events. The typical underlying semantics of the system description, the "input model of computation", are coupled FSMs or Petri-Nets. Examples are StateCharts or Esterel processes ([15]). Time constraints are given as maximum or minimum times. These shall be mapped to communicating hardware and software components possibly after a transformation which decomposes and/or merges part of the FSMs (of the Petri-Net).

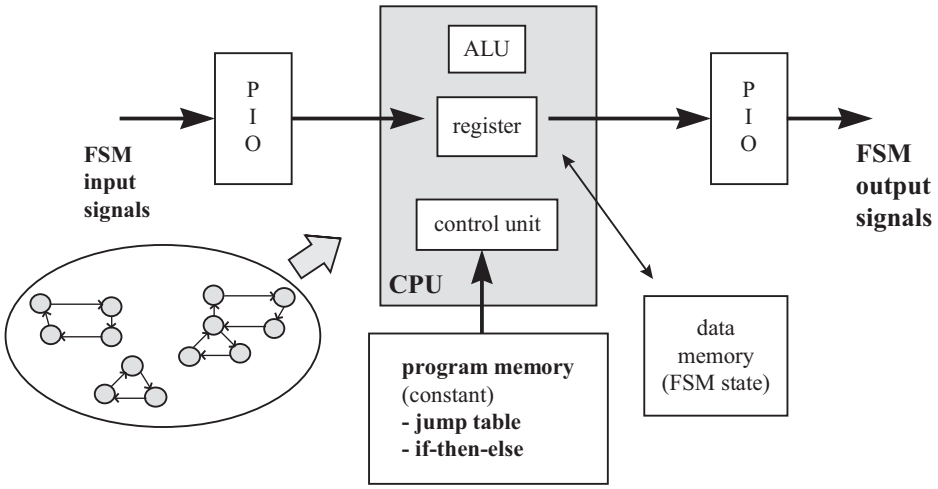



*Figure 5*: Microcontroller FSM implementation

The resulting co-design problems are the execution of concurrent FSMs (or Petri Net transitions) reacting to asynchronous input events, which have an input data dependent control flow and are subject to time constraints. Very often, control applications are safety critical (automotive, avionics). So, correctness (function and timing), FSM transition synchronization and event scheduling are the major problems while computation requirements are typically very low.

Fig. 5 shows the example of a minimal software implementation of a set of coupled FSMs on a **microcontroller**. Microcontrollers are small systems-on-a-chip used in many low-cost applications from simple functions in automotive electronics to dish washer control. Parallel input and output units (PIO) which can be found in any microcontroller latch the FSM input and output signals. FSM transitions must be serialized on the processor which is a resource sharing problem. After transformation, each FSM is mapped to a process. These processes consist of a small variable set containing the FSM state and the FSM transition functions encoded as short program segments in chains of branches or in jump tables. Program size grows with the number of states and transitions. In microcontrollers, program memories often require more space than the CPU. So, code size is an important issue. In microcontroller architectures, FSM communication uses shared memory. Since each FSM transition typically requires a **context switch**, context switching times are critical.

This short application overview reveals a number of specialization opportunities:

- most of the operations are boolean operations or bit manipulation,
- there are few operations per state transition,
- the data set containing the FSM states is typically small.

We will see in an example that this observation is used to apply all component specialization techniques plus control decomposition with different types of coprocessors.

## 4.1. 8051 - AN 8 BIT MICROCONTROLLER ARCHITECTURE

The 8051 is a quasi standard 8 bit architecture which is used as a processor core (CPU) in numerous microcontroller implementations by many companies. It is probably the highest volume processor architecture overall. Its main application are control dominated systems with low computing requirements. Typically, it is combined with a number of peripheral units most of them having developed into standard coprocessor architectures. It has 8 bit registers and a complex instruction set architecture (CISC).

We will characterize the 8051 architecture and its peripheral coprocessors using the specialization techniques derived above.

**Instruction set specialization**. The 8051 (e.g. [16, 17]) is a 1 address accumulator architecture [7]. It has 4 register banks with only 8 registers each and uses a highly overlaid data address space for 8 bit data addresses. Both features together enable very dense instruction coding of 1 to 3 bytes per instruction. This corresponds to the need for dense encoding and makes use of the observation that the variable set is typically small. To avoid a performance bottleneck and to allow short programs, the single address architecture and address overlay are complemented with a rich variety of data addressing modes (direct, indirect, immediate for memory-memory, memory-register, and register-register moves) which make it a CISC architecture [7]. Unlike other CISC architectures, however, there are no complex pipelined multi-cycle operations making interrupt control easy and fast. Also, instruction execution times are data independent making running time analysis easier. It is rather slow with 6 to 12 clock cycles per instruction cycle and clock frequencies up to currently 30Mhz. Among the more application specific operations are registered I/O with direct I/O bit addressing for compact transition functions and branching relative to a "data pointer" as a support of fast and compact branch table implementation.

| | | | |
|---|---|---|---|
| MOV | P0,#8FH; | load P0 with #8FH | (2 bytes) |
| CPL | P0_1; | complement bit no. 1 of port P0 | (2 bytes, 1 cycle) |
| JB | P0_2,100; | jump to PC+100 if P0_2 is set | (3 bytes) (P0_2 is input) |

*Figure 6*: 8051: Direct I/O bit addressing

Fig. 6 shows three examples with timing data taken from [18]. The first instruction is an immediate store to a registered 8 bit output: The output port P0 is loaded with the hex value 8F. Execution time is 1 instruction cycle. The second operation is a registered I/O operation to a single output bit, where a value of an output bit is complemented. Again, the execution time is a single instruction cycle only. And finally, in the third example, a

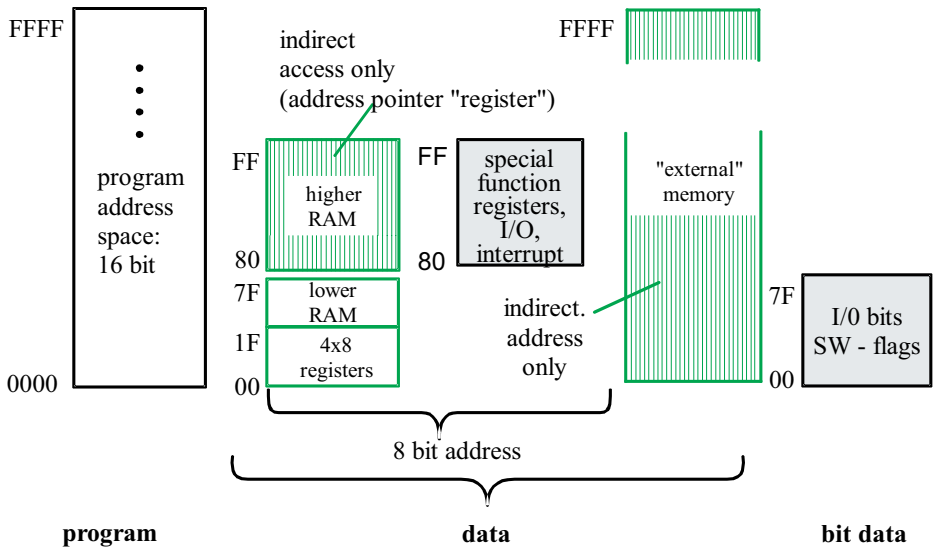



*Figure 7*: 8051 address space

conditional jump is executed on a single bit of an input port, executed in 2 instruction cycles. Especially this last instruction shows how the I/O signals are directly included in the control flow, very much like a carry bit of an accumulator. Such an integration of I/O events into the instruction set reflects the FSM view of the processor illustrated in fig. 5.

**Function unit and data path specialization**. The data path is only 8 bit wide. This is completely sufficient for boolean operations and bit manipulation required for FSM implementation and simple I/O protocols. Multiply and division functions are software implemented. Some of the typical peripheral coprocessors have specialized data paths.

**Memory specialization**. Fig. 7 shows that the lower 8 bit *address space* is overlaid with 5 different memory spaces. Which address space is selected depends on the operation and the address mode. The CPL operation e.g. always addresses one of the 128 *I/O bits* and *SW flags*, while MOV operations to the lower 128 addresses access the data RAM and register banks and the higher addresses either access the I/O ports, interrupt registers or extended data RAM, interestingly with higher priority to I/O. The register set consists of 4 banks of 8 bit registers. Bank selection is part of the program status word, so switching between banks is done with interrupt or subroutine calls in a single cycle. Complementing the short instruction execution times, this allows an extremely fast context switch for up to 4 processes. To give an example, even the slow instruction cycle times [18] still lead to context switching times of less than a microsecond. All processes use a single memory space and share the same local memory such that communication between software FSMs can easily be implemented in shared memory.

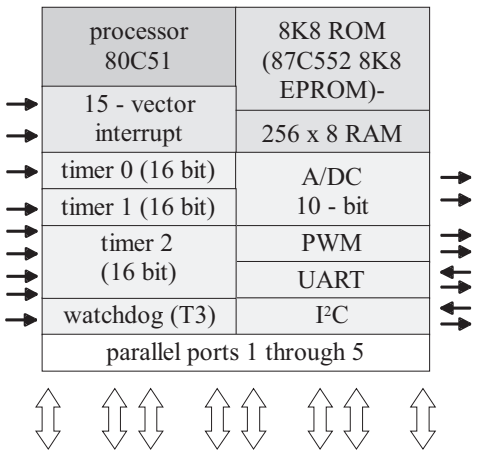



*Figure 8*: Philips 80C552

**Interconnect specialization**. Except for small 8 bit word length buses, interconnect is not particularly specialized but is based on a single bus due to the low data transportation requirements in such a system.

**Control specialization**. Component control is specialized to low power consumption and fast reaction to external events. Registered input signals can directly be linked to vectored interrupts which supports fast FSM process invocation on external events. Several power down modes which disable part of the microcontroller function can be applied to reduce power consumption.

Fig. 8 shows a complete 8051 microcontroller [17]. Two memories are on chip, an 8 kByte *EPROM* and a small 256 byte data *RAM*. The small size of both memories reflects the extreme cost sensitivity of high volume control dominated applications. The 5x8 bit *parallel I/O ports* and the universal asynchronous receive transmitter (*UART*) can be considered as dependent coprocessors. 3 *timers* can be used to capture the time between events or to count events, or they can generate interrupts for process scheduling. A pulse width modulation (*PWM*) unit is an FSM with a counter and a comparator which can generate pulses of a programmable length and period. Both can be classified as *incrementally controlled coprocessors*. PWM is a popular technique to control electrical motors. Combined with a simple external analog low pass filter, it can even be used to shape arbitrary waveforms where the processor repeatedly changes the comparator values while the PWM unit is running. There is also an $I^2C$ unit to connect microcontrollers via the $I^2C$ bus. A peripheral *watchdog* is a timer coprocessor which releases an interrupt signal when the processor does not write a given pattern to a register in a given time frame. It can be classified as an independently controlled component. Such watchdogs are used to detect whether the program is still executed as expected which is a helpful feature for safety critical systems. Finally, there is an analog digital converter (*ADC* ) on the chip (a dependent coprocessor).

There are two types of system communication mechanisms in this microcontroller. The processor uses shared memory communication by reading and writing from or to memory locations or registers. Only in few cases, point-to-point communication is inserted between coprocessors, e.g. between the interrupt and timer coprocessors, i.e. almost all of the system communication relies on processor operations. This is flexible since the system can easily be extended but it limits performance.

Since several software processes can access a single coprocessor, there are potential resource conflicts requiring process synchronization and mutual exclusion. Peripheral resource utilization is an optimization problem and is, e.g. addressed in [19].

## 5. Architectures for Data Dominated Systems

### 5.1. DATA DOMINATED SYSTEMS

The term **data dominated systems** can be used for systems where data transport requirements are dominant for the design process, such as in telecommunication, or where a stream of data shall be processed, such as in digital signal processing. In this overview, we want to focus on the latter application. Often, such systems are described in flow graph languages, such as in signal flow graphs which are common in digital signal processing or control engineering (see e.g. [20]). Similar to data flow graphs, signal flow graphs represent data flow and functions to be executed on these data.

Such data dominated applications typically have a demand for high computation performance with an emphasis on high throughput rather than short latency deadline. Large data variables, such as video frames in image processing, require memory architecture optimization. Function and operation scheduling as well as buffer sizing are interdependent optimization problems [13].

The flow graph "input model of computation" offers many opportunities for specialization:

- Periodic execution often corresponds to a (mostly) input data independent system data flow.
- The input data is mostly generated with a fixed period, the sample rate.

Such systems can be mapped to a static schedule thereby applying transformations for high concurrency, such as loop unrolling. The resulting static control and data flow allows to specialize control, interconnect function units and data path while, at most, a limited support of interrupts and process context switch is required. Furthermore, static scheduling leads to regular and a priori known address sequences and operations which gives the opportunity to specialize the memory architecture and address units.

**Digital signal processors (DSP)** are the most prominent examples of target architectures for this class of applications. The various signal processing applications have led to a large variety of DSPs, with word lengths of 16 to 64 bit, fixed or floating point

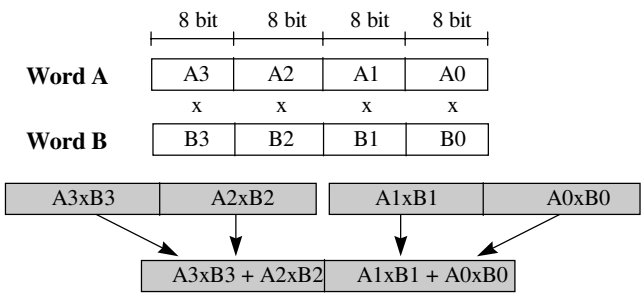



*Figure 9*: MMX subword operation: Multiply-add

arithmetic, with different memory and bus architectures targeting high performance, low cost or low power.

Specialization for data dominated systems has lead to a large variety of architectures. We will select an important application area, image processing for multimedia, and will discuss examples with a different degree of specialization. We will start with multimedia extensions of general processors and take MMX as an example. Next we will look at the Hitachi SH-DSP, following a completely different concept of a combination of RISC processor and digital signal processor specializing to a wider range of applications. Then, we will analyze a VLIW processor with specialization and coprocessors, namely the Philips TriMedia. Finally, we will look at multiprocessors with specialized data flow and memory architectures using the TI320C80 as an example.

## 5.2. MMX

MMX is an extension of the Intel (Pentium) architecture (IA). It defines parallel subword operations on the floating point registers. Each 64 bit floating point register can be split in 2, 4, or 8 subwords of 32, 16, or 8 bits. Additional instructions control the execution of parallel operations on the subword operands. Fig. 9 shows an example of a multiply-add operation. Controlled by a first instruction, the 8x8 bit subwords of two 64 bit registers are multiplied as 4 independent operations resulting in 4x32 bit results in two registers. With a second instruction the adjacent subwords of the two registers are added to 2x32 bit subwords which are then stored in a single floating point register. MMX introduces a total of 57 additional IA instructions to control arithmetic, boolean, and shift operation on floating point register subwords as well as move operations to support the conversion between subword and standard data representations. The combination of subword operations of a register is fixed for each instruction. This allows to control up to 8 parallel subword operations without changing the instruction format. As a tradeoff, branching on results of individual subword operations is not possible any more. Instead, boolean operations can be used to mask operations on subwords leading to conditional subword operations. This approach is known from SIMD array processing. MMX, furthermore, supports specific digital signal processing arithmetic, such as saturation. All operations are executed in an extended floating point unit.

These MMX operations are well suited to the matrix and vector operations on pixel arrays in image processing and image coding, e.g. for the JPEG or MPEG standards. The typical pixel size in these applications is 8 to 16 bits which fits the subword length. For these applications, performance improvements between 1.5 (MPEG 1 decoding) and 5.8 (matrix-vector product) have been reported [21]. As an additional effect, the code size is drastically reduced which may lead to fewer cache misses.

MMX is an example of function unit and data path specialization combined with instruction set specialization. In [21], the authors emphasize that the register model and the control for all other operations (including interrupts) remains unchanged which is required to stay compatible to the standard IA. MMX is specialized to a certain but time consuming part of image processing algorithms which is demonstrated by the difference in speedup between the complete MPEG 1 decoding and a matrix-vector operation which is an MPEG 1 subfunction (DCT). To summarize, MMX is a highly specialized but very area efficient extension of a general purpose processor.

This type of extension is not unique, but can be found in many architectures, such as the MAX and VIS extensions of the hp PA or the Sun Ultra SPARC architecture [22, 23].

## 5.3. DIGITAL SIGNAL PROCESSORS - HITACHI SH-DSP

While MMX extends the data path, it leaves most of the processor architecture unchanged. In digital signal processors (DSP), specialization goes further and includes the control unit, busses and memories. There are many different DSP architectures. For easy comparison with MMX, we pick the SH-DSP, a standard RISC microcontroller which is enhanced by a DSP unit.

The SH-DSP [24, 25] is based on the Hitachi SH instruction set architecture (ISA) which was extended in each generation, from the SH1 to the recent SH4 which is a superscalar architecture. The SH-DSP is a branch of this ISA supporting instructions which are not found in the SH4. Fig. 10 illustrates the processor architecture. The CPU consists of 2 main units, the CPU core with 32 bit integer arithmetic and a 16 bit DSP unit with extended internal word length. The SH-DSP runs at a clock cycle of 20 MHz.

The SH-DSP is targeted to single chip systems, but with higher computation requirements than the 8051. There are several dependent and partially dependent coprocessors for peripheral functions connected by a peripheral bus. For larger systems or for system development, up to 4 Mbytes of external memory can be accessed through an external bus interface (BSC). The DSP extension has lead to a quite complex bus architecture and memory access procedure. There is no on-chip cache function as some of the earlier SH2 architecture versions but 4 single-cycle on-chip memory banks (X and Y) and 3 processor busses. The CPU core is responsible for instruction fetch and program control. Program and data can reside anywhere in the 4 memory blocks. Instruction fetch and data transfers for the CPU core share the 32 bit I bus which would become a bottleneck when operating on array variables such as in digital signal processing. To avoid such conflicts, the DSP unit uses the 16 bit X and Y buses to transfer a maximum 2 data words in one cycle in parallel to the instruction fetch on the I bus, as long as the 3 trans-

fers use different memory blocks (Harvard architecture). While the I bus supports 32 bit addresses in a unified address space, the X and Y busses use 15 bit addresses only to address a 16 bit word in the X and Y memory blocks. 15 bit addresses allow for a maximum of 32 K words in each memory block which closely fits the internal memory size. The DSP unit can also access the I bus but at the cost of conflicts with instruction fetch. The extra memory blocks and the X and Y buses were added to the SH architecture as specialized memory and interconnect closely focused on internal memory access. Concurrent memory access requires concurrent address generation. Since the CPU core would be idle during DSP operations, its ALU is used to generate the X memory address while an additional 16 bit ALU generates the Y memory address. This minimizes address generation overhead and stays close to the original CPU core architecture. It also allows to use the general register file for address generation which should simplify compilation. Address generation is incrementally controlled by the instruction word supporting increment of the type *addr = (addr + incr) mod m*, where *incr* is either 2 or 4 or an explicitly specified integer value. Such addressing patterns are typical digital signal processing, such as in the Fast Fourier Transform. The modulo operation is used to generate address patterns in 2 dimensional arrays.

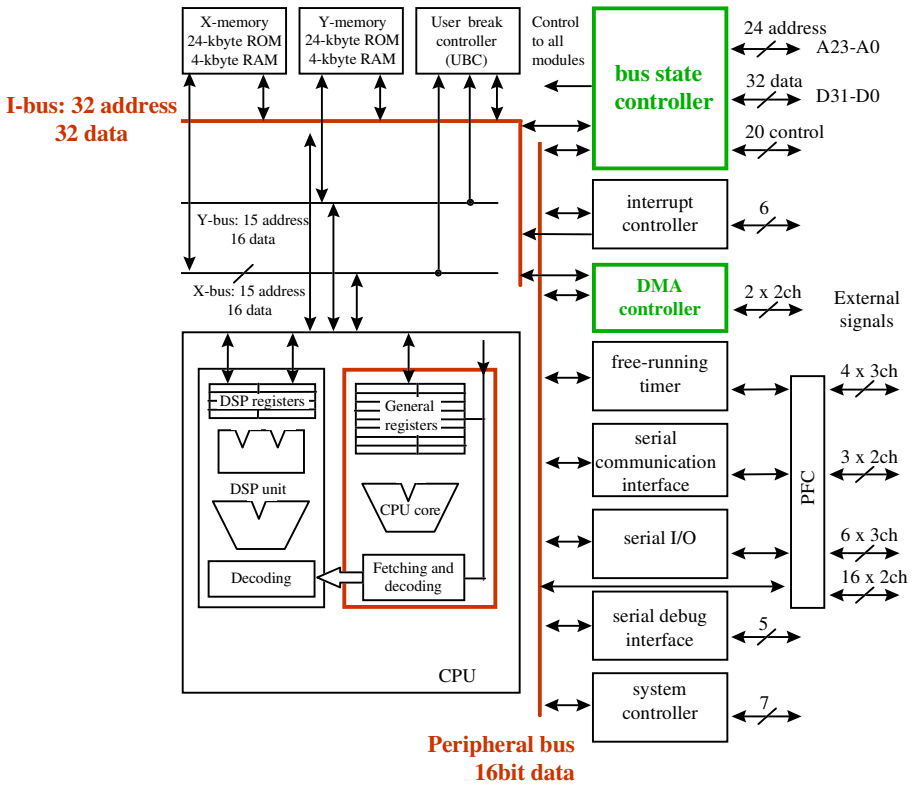



*Figure 10*: Hitachi SH-DSP

The internal memories are far too small to store larger array variables such as complete video images. Such applications require external memories which, in turn, requires fast memory I/O. The SH-DSP bus interface (BSC) contains an external memory controller supporting several memory types including DRAM. It can be configured to different memory word lengths and wait states with differing row and column access times. A direct memory access controller (DMA) can transfer a coherent block of data between the external memory and the internal memory blocks as a dependent coprocessor. It mainly consists of a set of address counters supporting 4 independent memory transactions (i.e. 4 DMA channels). Since it uses the I bus, only one memory transfer can be active at a time which is, furthermore, in conflict with CPU core memory access. This is a limitation for high throughput applications but fits slow external DRAMs in low cost systems. Both DMA and external memory interface are versatile enough to be able to optimize the external memory to a given low cost application but are less specialized to the DSP extensions. The overhead for higher throughput external memory access was obviously found to be inadequate for the SH-DSP class of applications.

The DSP unit has an internal register file of six 32 bit registers and two 40 bit registers. It can execute 32 or 16 bit operations on these registers using fixed point arithmetic. Like MMX, it supports saturation and conditional execution. Here, conditional execution rather than branching avoids an increase in the control delay of the CPU core which controls all branches. Also, for many DSP applications conditional operations are more appropriate, such as in conditional shift operations to normalize operands in fixed point arithmetic. Other specialized DSP unit operations are shift-by-n (barrel shift) and most significant bit detection, both needed for fast normalization of arrays to minimize rounding errors.

The 32 bit DSP instruction word allows combinations of up to 4 instructions, one X memory load or store operation, one Y memory load or store operation, a multiplier and an ALU operation. The load and store operations can be connected with a simple address increment/decrement of 1, 2, or 4, while other linear address sequences reduce the available parallelism, just as explained for incremental controlled units. This type of instruction level parallelism (ILP) is much more flexible than the matrix-vector type subword parallelism of the MMX allowing many different memory access patterns and operation combinations at different levels of parallelism, but this comes at the price of an extended specialization including memories, busses, data paths and instructions.

In general purpose computing, an ILP of 4 would already lead to a high impact of control hazards due to conditional branches [7]. In signal processing, most of the branches are found in loops with fixed iteration counts. Instead of using conditional branches, DSPs use a specialized control unit and loop instructions. It requires specialized control registers and a specialized control unit. At loop entry, a loop start address register is loaded with the start address of a loop, a loop end register is loaded with the last address of a loop and a loop counter is loaded with the number of loop operations. Then, the instructions between loop start and loop end are executed thereby decrementing the loop counter for each iteration. So, there is no loop overhead except for the loop instructions at the beginning. While the other operations of the DSP unit use one of the register files, the loop operation uses dedicated registers which must be saved for nested loops. Dedicated registers with special interconnect usually make compilation harder [26] and are therefore avoided in general purpose computing. The loop operation is an example of specialized control.

To summarize, the SH-DSP introduces a dedicated DSP unit which effectively operates as a dependent coprocessor which has no direct influence on the control sequence. The SH-DSP exploits all specialization techniques to optimize the DSP unit and the global architecture for maximum ILP in fixed loop operations on 16 bit fixed point arithmetic with normalization. The memory architecture assumes either small data arrays or high locality of the operations. We will see even higher specialization in a later example.

In the specialization process, the SH instruction set is maintained as well as the overall CPU core architecture and its memory architecture. So, the added functionality does not reduce the flexibility of the original core and keeps binary compatibility, like in case of MMX, but leads to circuit overhead which increases cost.

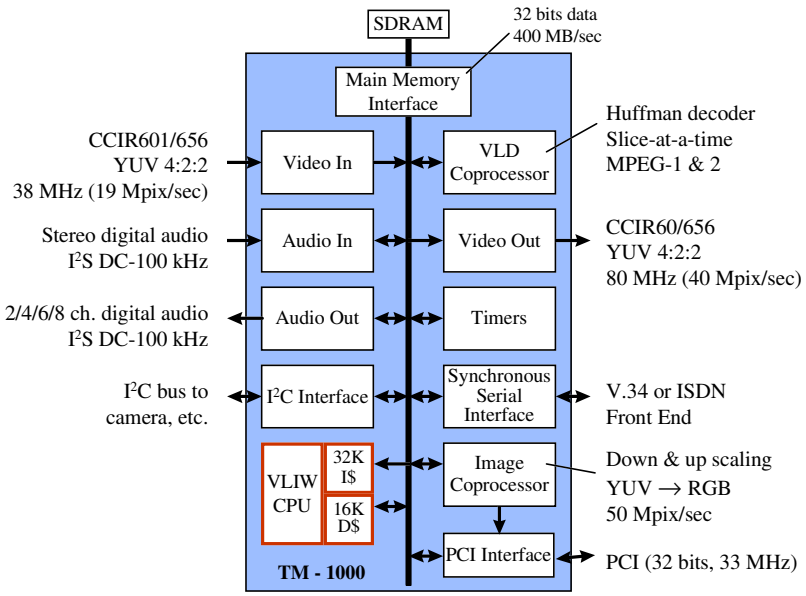



*Figure 11*: Philips TriMedia TM1000

## 5.4. VLIW SIGNAL PROCESSORS - PHILIPS TRIMEDIA TM1000

Adding function units, memory blocks and specialized functions does not arbitrarily scale for higher performance requirements and does not always lead to a cost effective solution. The Philips TriMedia TM 1000 processor [27, 28] is a specialized DSP architecture targeted to multi-media applications but still with emphasis on flexibility to support a wider range of applications in this domain. Fig. 11 gives an overview of the architecture. It consists of a VLIW CPU with cache, an SDRAM main memory interface and a set of peripheral interfaces and coprocessors connected over a 32bit internal bus with block transfer. A PCI bus interface simplifies the use of the TriMedia processor as a media coprocessor in workstations or PC systems. The serial I²C bus interface is standard in Philips processors and can be used for external device control. The processor clock frequency is 100 MHz.

To understand the overall concept, we will first discuss the coprocessors which are highly specialized to the application domain. Most notably in this context is the partially dependent Image Coprocessor, which can resize video images. Resizing requires undersampling or oversampling of video images. An important application is the display of one or more video images on a PC window interface. In the simplest way, this can be done by pixel repetition or omission, while for higher picture quality interpolation or filtering in vertical and horizontal dimensions is required. The microprogrammed Image Coprocessor (ICP) uses a 5 tap digital filter with programmable coefficients for higher

quality requirements. From the TM 1000 manual, we can conclude that in the worst case of a 1024x768 picture size, this coprocessor executes more than 500 Mio. multiplications/accumulations plus the corresponding address generation and scaling operations thereby requiring 150Mbyte/s memory bandwidth for 16 bit operands. The coprocessor can be programmed to scaling factors larger or smaller than 1 and different filter operations in horizontal or vertical directions. For each of these operations, a different memory access sequence must be generated. With a second set of functions, the Image Coprocessor can convert a standard digital video signal stream (YUV 4:2:2 or 4:2:0) to the three RGB signals used in PC monitors which also requires filtering.

The VLIW CPU alone would hardly be able to reach this performance. The same holds for MMX. So, the Image Coprocessor enables the implementation of higher quality picture scaling thereby off-loading the CPU which can now be used for other parts of a video application. New functionality becomes feasible. When we consider flexibility as the set of functions which a system can implement, then the highly specialized Image Coprocessor has increased the flexibility of the TM 1000 over, e.g., an IA with MMX given the domain of consumer video applications. So, in embedded system design, flexibility should be defined in context of the application.

The architecture of the other TM 1000 coprocessors follows the same concept of implementing widely used and static video functions. Programmability is mainly restricted to select between a few function alternatives. The Video In and Video Out as well as the Audio In and Audio Out coprocessors generate or accept different audio and video standards using predefined internal representations in SDRAM. The VLD supports variable length decoding for the MPEG 1&2 video standard which mainly requires table based string operations.

The VLIW CPU (very long instruction word, [7]) is less specialized than a DSP. The architects avoided special registers to simplify compilation and context switch. The CPU contains 27 function units which are summarized in fig. 12. The selection of these function units is based on benchmarks of video and postscript processing applications [27].

The CPU can issue five 42bit instructions per clock cycle. These 5 instructions can be arbitrarily arranged in one VLIW instruction word. Each instruction is extended by a 2 bit instruction type tag leading to a 220 bit instruction word. To reduce memory cost, compile time instruction compression removes unused 42 bit instruction slots. The decompression step simply uses the unmodified instruction type tag to detect and reinstall the unused slots. This way, decompression is very fast and can be executed at instruction fetch time so that even the instruction cache holds compressed instructions.

The CPU is a load/store architecture using a large 128 entry register file. The register file and the CPU buses support 15 register read and 5 register write operations at a time. The processor uses register forwarding to reduce data hazards as known from general purpose processor design. It supports 64 bit floating point and fixed point arithmetic with specialized DSP operations. This architecture has much similarity with a general purpose VLIW architecture, with the main differences in quantitative details such as the selection and data path optimization of the function units, the size of the register file and the number of buses. Such detailed optimization of a general architecture can be ob-

| **functional unit** | **quantity** |
|---|---|
| constant | 5[1] |
| integer ALU | 5 |
| load/store | 2 |
| DSP ALU | 2 |
| DSP multiplier | 2 |
| shifter | 2 |
| branch | 3 |
| int/float multiplier | 2 |
| float ALU | 2 |
| float compare | 1 |
| float sqrt/div | 1 |

[1] The "*constant"* units extract literals from the VLIW instructions.

*Figure 12*: TM 1000 CPU function units

tained with benchmarking [7]. A difference is the missing hardware branch prediction support which can be explained with the fixed loop structure of many signal processing applications while conditional (guarded) operations can be used to handle small if-then-else constructs like in the SH-DSP. The CPU also supports specialized subword instructions for media applications as discussed previously.

Memory specialization is rather different to general purpose processors and the previous examples accounting for the development of fast DRAMs. First, the large multiport register file replaces the multiple memory banks or multiport memories found in many DSPs. The data cache allows program controlled allocation of lines, e.g. for filter coefficients and program controlled prefetch of cache blocks. Both features allow program control of cache behavior which is effective for predictable address and operation sequences, as typically observed in time consuming core parts of DSP algorithms. This way most of the data cache misses which are frequent in large image processing slowing down hardware controlled caches can be hidden [27]. The designers decided to install a complex 8 way set associative cache both for program (32kByte) and data (16kByte) which helps to avoid bottlenecks due to concurrent program and hardware cache control. The memory interface supports synchronous DRAMs (SDRAMs) with 400Mbytes/s maximum throughput.

The designers might have decided to use a DMA unit as discussed before instead of a cache. We will see in the next section that an older processor, the TMS320C80, uses a powerful DMA unit which can generate complex memory access patterns even for sev-

eral processors. Such a unit is appropriate for fast static SRAMs. Today, pipelined DRAMs, such as SDRAM, support block transfers at similar throughput and emerging standards such as SLDRAM [30] and DirectRambus [31] can remove the transfer setup time overhead. The block transfer of a cache utilizes this property, while the additional flexibility of the DMA is of little use. Furthermore, the cache line allocation can use part of the cache as a static RAM. On the other hand, the mixture of hardware and programmable cache control requires user controlled adaptation of the input program to fully exploit the system performance. Moreover, if there are coprocessors which simultaneously access the same memory, the memory throughput can drop due to conflicting row accesses (not allowed for the TriMedia). Such conflicts could be solved by a specialized DMA which is attached to the memory rather than to the processor, like in the case of the TMS320C80. DMA operation, however, is affected by access pattern predictability. In summary, the memory-processor cooperation will most likely be one of the dominant problems of multimedia embedded system design.

## 5.5. SINGLE-CHIP MULTIPROCESSOR-TEXAS INSTRUMENTS TMS320C80 MVP

The fourth example is a single chip multiprocessor, the TMS320C80, shown in fig. 13. The target domain is further narrowed down to pure video processing. This multiprocessor IC consists of four 32 bit integer DSPs (Advanced DSP) with 4 single-cycle RAM blocks of 2 Kbytes each and a 2 Kbytes instruction cache. The processors are connected by a programmable crossbar switch. A single master processor (MP) for general purpose computing and floating-point operations is coupled to this processor array. The transfer controller (TC) is a DMA processor which is programmed through the MP and is responsible for the data and program I/O over a 64 bit external bus. For highly integrated solutions, a video controller (VC) is implemented on chip. The clock frequency is 50MHz.

Data path and function units of the 4 *Advanced DSPs* support subword parallelism and special image processing operations. The ADSP ALU has 3 inputs to support sum operations on arrays. Several multiplexers in the data path allow chaining of operations. Each *Advanced DSP* has 2 address ALUs for index or incremental addressing corresponding to the 2 data buses of each *Advanced DSP*. Again, they are incrementally controlled coprocessors. Like the SH-DSP, the *Advanced DSP* sequencer supports zero overhead loops but adds a loop stack for 3 levels of nested zero overhead loops and a single zero overhead subroutine call. The stack extends the processor state which slows down context switching. There is a second register set for fast context switching between two contexts, e.g. for debugging purposes. This specialization increases performance but introduces many programming constraints.

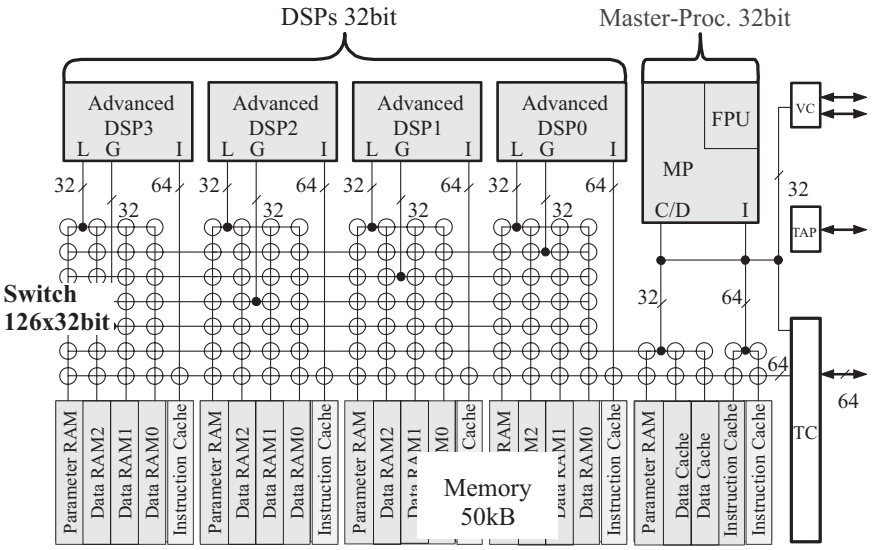



*Figure 13*: TMS 320C80 MVP

All instructions are given in a single 64 bit instruction word, but in a fixed format of 3 independent fields, one of them for the Advanced DSP operation, one for a local data transfer, and one for a global data transfer from the Advanced DSP to any of the memory blocks. Therefore, TI regards the Advanced DSP as a VLIW processor. Each Advanced DSP has a very small cache of 256 instruction words. Unlike the SH-DSP or the TM1000, on-chip program memory is restricted to very small caches. All instruction loads must be routed through the transfer controller.

The *memory architecture* with a crossbar interconnect seems very flexible. Obviously, there was high emphasis on minimum memory access time. On the other hand, the memory architecture is extremely specialized. Depending on the resolution, video frame sizes are on the order of several hundred Kbytes, such that not even a single frame could be stored in all RAM memories together. Since all RAM blocks can only be accessed through the TC bottleneck, this would be an unacceptable limitation in general purpose signal processing. So there must be an algorithmic justification for this memory size. In fact, video algorithms typically use some kind of spatial or temporal locality which is exploited here. As shown in fig. 14, the video signal scans the image, such that algorithms which operate on few neighboring lines can use a line buffer keeping a small number of recently scanned lines. Such a buffer could fit into one RAM block. If the algorithm operates on windows, such as in standard image compression applications [32], then only the current window could be loaded into internal RAM blocks. This also gives an impression of the prospective role of these RAM blocks. They can be used as buffers between the operations of a signal flow graph (SFG) [20] or a similar representation, such as a synchronous data flow graph (SDF) [13], while the operations or functions are

mapped to Advanced DSPs. The crossbar interconnect allows to implement arbitrary flow graph topologies as long as the buffers are large enough. In this context, the TC will have to read and write the windows or lines to or from an external buffer (or the internal video interface) and, at the same time handle potential instruction cache misses.

This is certainly a potential bottleneck even if the internal data flow is larger than the external one. So, again, we observe flexibility in the one place (internally) and constraining effects of specialization on the other, this time in system interconnect. Here, only interconnect and memory specialization together make sense. So when specializing, the designers need a clear understanding of the application domain, which in this case leans itself to data decomposition of a single frame. The small cache size corresponds to view operations on each of the small data objects which is realistic in many video applications. We will see the same assumption in the PADDI and VSP architectures which are discussed later.

To reduce the constraining effects of the single I/O bus, the TC must be a powerful and versatile I/O processor. The TC is programmed in a very compact "Guide Table" where the windows to be transferred are defined as "patches" (fig. 15). Since the TC must manage cache misses and data transfers, it needs an internal context switching mechanism, which must support external memory delays. So, extra buffers are inserted in the TC.

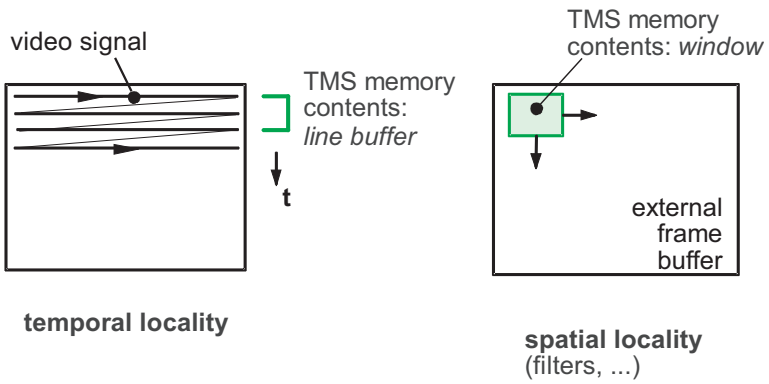



*Figure 14*: Locality in video algorithms

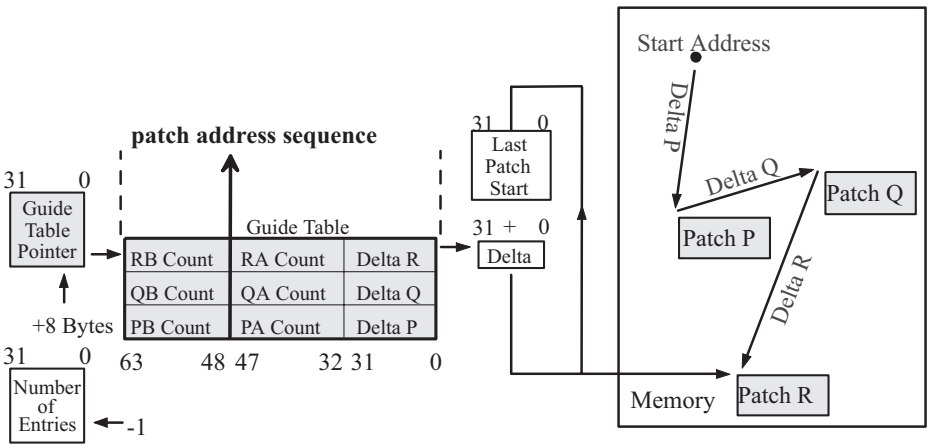



*Figure 15*: Patch addressing in the TMS320C80

The TMS320C80 is a good example for how to apply the classification scheme of system specialization techniques to understand adaptation and flexibility of an architecture.

## 6. **Systematic Approaches to HW/SW Flow Specialization**

### 6.1. DESIGN WITH ARCHITECTURE TEMPLATES

The more interacting components a system has, the larger the design space for optimization and specialization. To make the design manageable, design automation approaches for systems with many interacting components are usually based on architecture templates. These templates support global flow optimization. A well known template is the homogeneous multiprocessor [7] which allows to scale performance with size (in certain limits) and which is supported by parallel compiler and operating system technology. Even though there are different multiprocessor architectures, specialization is very limited and typically restricted to the communication network topology. Embedded system design includes component specialization but the design principle of separating component design and global flow optimization has successfully been applied to embedded system design as well. This is also reflected in the separation of component and system specialization techniques, as described above. The concept of separated component and global flow specialization can easily be extended to a system hierarchy, where a set of interacting components may be a component (i.e. subsystem) of a larger system. For simplicity, we will focus on a single level of hierarchy.

The PADDI [33] and VSP systems [12] address the implementation of static and cyclic signal flow graphs which describe the system data flow at the level of individual operations on data variables. Both architectures consist of small function units with a pro-

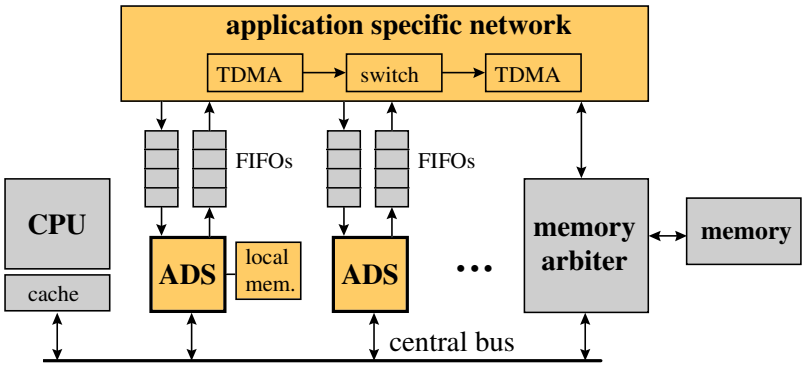



*Figure 16*: Prophid architecture template

grammable interconnect and buffer memories as local variable memories. The VSP2 local programs iterate over a small number of instructions without branches but with conditional execution while PADDI uses a small local nanostore to decode global 3 bit instructions. The second generation VSP2 has been used in commercial designs and integrates 12 ALUs and 4 memories on a single chip using 0.8 micron technology. Like PADDI, the VSP2 can be extended to multichip systems.

PADDI and VSP are specialized multiprocessors which can be programmed by a design automation system which includes component programming as well as global flow optimization. In contrast, the TMS320C80 programming environment contains a compiler for the individual signal processor but leaves global flow optimization and Transfer Controller programming to the user.

## 6.2. PROPHID

In contrast to all previously described systems, the more recent Prophid (fig. 16) is a configurable architecture template rather than a hardware concrete implementation [34, 35]. The Prophid template consists of a single core processor which controls several application specific independent (co)processors, ADSs, via a central standardized bus. This block structure and the communication protocol as well as the communication network topology are fixed, while the ADSs and the communication network parameters are application specific.

The ADS architecture can range from a simple data path to a programmable processor with program memory. Each ADS is equipped with at least one input and one output FIFO which are attached to the application specific communication network.

The application specific interconnect network is, again, generated from a template consisting of an application specific number of busses interconnected by programmable switches (space switch). The ADSs are assigned time slots (time division multiple access - TDMA) in which the FIFOs can read from and write to these busses. The switch ar-

rangement enables point-to-point interconnection between any ADSs (a Clos network) while the communication bandwidth is limited by the number of time slots and the bus bandwidth. In [35], the authors propose an approach to automatic design of this network given the required ADS communication. A memory arbiter controls access to a larger memory.

Scheduling and allocation in the Prophid design automation system are executed at the process rather than on the statement level as in PADDI or VSP2. Several processes can be mapped to one ADS where they are activated by input data in the FIFO memories. In contrast to PADDI or VSP2, the ADSs are activated by incoming data which classifies Prophid as a data flow architecture. This activation rule closely matches dynamic data flow graphs [36] and reflects the data dependent execution times of Prophid, i.e. complex video algorithms, such as MPEG 2 video signal compression.

The Prophid template nicely demonstrates the separation of component specialization and global flow specialization and optimization. Almost any ADS can be inserted, as long as it adheres to the interface protocols of FIFO memories and central bus. This approach also enables component reuse, including reuse of programmable components.

Prophid is an example of a parametrized architecture template which has been targeted to support reuse and design automation. The hardware configurability has increased design space compared to the programmability of the VSP or PADDI architectures without loosing the design automation support. The efficiency of global flow optimization will obviously depend on the a priori knowledge of ADS communication pattern and timing.

### 6.3. EVENT FLOW ARCHITECTURE

The next example is a similar approach for a different domain, i.e. control dominated systems. In contrast to Prophid where a process is activated when all of its input data ("tokens") are available corresponding to an AND function over all input data, process activation in control dominated systems may dependent on arbitrary functions of external events. Events can consist of signal sequences, such as the occurrence of a certain protocol header in a bit stream or the activation condition for fuel injection in a combustion engine requiring the observation of sensor signals. In microcontroller architectures, determination of activating events and the activated process itself are both executed on the same processor. On the other hand, the architecture in [38] used in automotive engine control, employs peripheral coprocessors to detect complex sequential events such as needed for spark plug firing and fuel injection and, even more complex, for sensor failure detection.

The Event Flow Architecture [37] introduces a specialized processor for complex event detection as part of a general architecture template. Fig. 17 highlights the different communication network and (co)processor structure. Like in Prophid, buffers are used to hold input data, but input data evaluation and storage is performed by the specialized programmable event detection unit. The event detection operations are mostly boolean which is exploited to specialize the processor for a high word level parallelism not available in microcontroller cores. This parallelism cuts reaction times (latency) allowing to use programmable architectures in domains which have so far been reserved to hardware

coprocessors. The event flow architecture has been applied to an industrial field bus design.

The event processing unit can range from a simple data path or finite state machine to a programmable microcontroller core which uses the local variable memory for storing and communicating data, activated by the event detection unit. Like in the Prophid architecture, the template allows to include reused components. In our terminology, the event processing unit is an independent coprocessor of the event detection unit which reflects the direction of control. The communication network (TDMA) is simpler than in Prophid due to the lower data throughput requirements of control dominated systems.

The Event Flow Architecture is supported by a design automation system which supports event flow graph mapping and component programming.

## 6.4. RECONFIGURABLE ARCHITECTURES

In all previous examples, the hardware architecture is fixed throughout the lifetime of a product. The hardware structure of a reconfigurable architecture can be changed in the field. Reconfiguration can be used to adapt the architecture specialization to the current application profile, to switch specialization to a different application or simply to update the architecture. It is a possibility to increase system flexibility which can be used in addition or as an alternative to programming. The price for this flexibility is, in general, a larger chip area, larger delay times and/or higher power consumption than a specialized

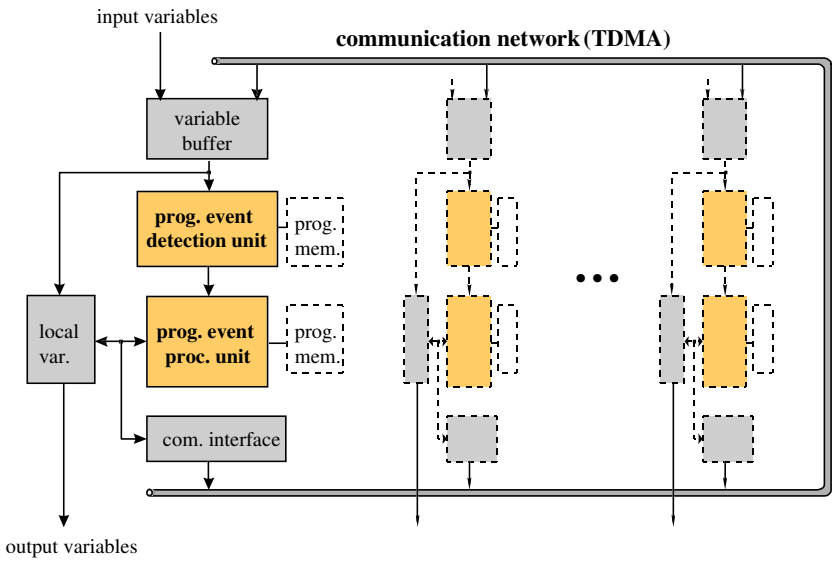



*Figure 17*: Event Flow Architecture template

custom architecture for a single application.

Reconfigurable architectures are dominated by field programmable gate arrays (FPGAs) [39], since these circuits allow to change the structure down to the logic level. FPGAs can be used to implement complete systems or system components, mainly constrained by the available logic level resources and memory. This logic level reconfiguration does not add new architectures but enables the implementation of alternative architectures on the same circuit.

Reconfiguration can also occur at the function block level. In the Pleiades architecture

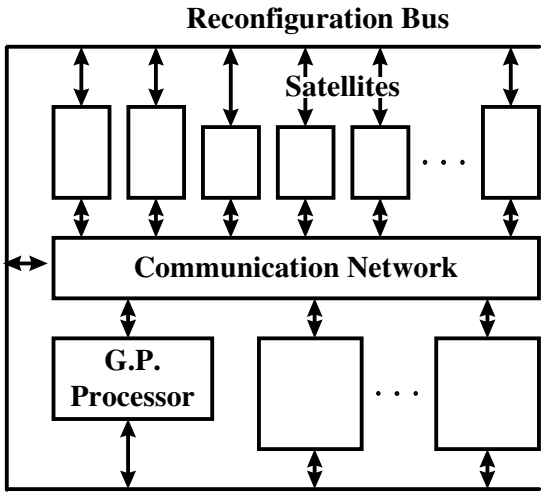



*Figure 18*: Pleiades architecture template

template [40], application specific coprocessors (Satellites) are interconnected through a reconfigurable communication network (fig. 18). The satellites can be fixed components or reconfigurable data paths.

Focusing on reconfiguration for low power consumption, the authors were able to demonstrate significant improvements compared to FPGA or programmable processor implementations [41].

The Pleiades satellites may be fixed or reconfigurable components which must be adapted to the set of applications similar to the ADSs of the Prophid architecture. On the other hand, the assignment of time slots and bus switches in Prophid can be considered as reconfiguration since it is not part of the instruction set architecture. Comparing Prophid and Pleiades, it becomes clear that programming and reconfiguration are closely related. In [40], the Pleiades authors introduce a classification which distinguishes logic, data path and data flow reconfiguration and where programming is just another type of

coarse grain reconfiguration. Despite the similarities, we will keep the distinction of programming and reconfiguration, here, since the instruction set architecture, today, is the major dividing line between hardware and software defining binary code compatibility. We note, however, that reconfiguration is a very active research area which has lead to complex architectures, such as proposed by [42]. Detailed examples and discussions are beyond the scope of this chapter.

## 7. **System Architectures with Complex IP and Flexibility Requirements**

### 7.1. MOTIVATION

While microcontrollers show a clear distinction between a programmable core and relatively simple peripheral coprocessors, the larger multimedia architectures include flexible coprocessors, such as the TriMedia ICP or specialized multiprocessors such as the TMS320C80 or core processor extensions such as in the Hitachi SH-DSP. We have seen that the specialization techniques are typically more effective when used in combination. System level specialization, e.g. to optimize memory access, gives additional freedom which further complicates design decisions. A comparison of MMX and TriMedia also shows that an extension of the application domain favors a completely different architecture, while narrowing down the TriMedia application domain to reduce cost might lead to a Prophid type of architecture. So, while a microcontroller can be used for a wide domain of applications typically only limited by the selected coprocessors and the available memory size, domain specific design for multimedia applications requires a thorough analysis of the application domain.

Moreover, microcontroller components can easily be reused. Reuse of multimedia processor components is less obvious since they were specialized in context of the other system components.

Given the complex design problem, we might feel inclined to simplify the design task by restricting specialization and focusing on a single application rather than on a domain. The comparison of MMX, SH-DSP and TriMedia demonstrates that restricting specialization cuts the available performance and increases cost and will, therefore, not be competitive in practice. A set-top box using general purpose processors would simply be too expensive or too slow. Reducing the domain to a single application reduces the volume and limits the future reuse of parts of the architecture, both increasing cost. Finally, this approach does not account for products with different variants which exploit hardware flexibility to minimize the number of hardware platforms. Examples were given in the introduction.

As a consequence, we will need co-design and system synthesis approaches which create architectures for a set of applications rather than for a single application thereby exploiting hardware flexibility.

So far, we have assumed complete freedom in system specialization. Introducing larger IP components for reuse constrains the design space. Such components have been spe-

cialized in a preceding design process with possibly different design objectives and constraints.

Reused IP components can be fixed or adaptable. Components can be adapted by hardware modification, by programming, or by reconfiguration. Hardware modification changes the original specialization while programming keeps the original specialization and makes use of component flexibility. Reconfiguration makes use of component flexibility which is not represented in the original instruction set architecture.

Obviously, reusability and flexibility are closely related. In the remainder of this chapter, we will develop a taxonomy of flexibility and adaptability which we will then use to analyze the current design automation approaches and propose improvements.

## 7.2. FLEXIBILITY AND ADAPTABILITY - A TAXONOMY

For clarity, we will need some definitions. In the following, the system or component **behavior** shall include its **timing**.

**Dynamic adaptability** shall contain all possible run-time behaviors of a component or system with no change of the hardware structure or program. Examples for dynamic adaptability include instruction reordering, caching, or speculative computation [7]. This definition of dynamic adaptability also includes the case that a hardware realizes a set of different behaviors with dynamic selection at run time dependent on system state and input data. Examples are the simple dependent coprocessors of the 8051 which select from different behaviors depending on control register contents, or a filter circuit which selects between different filter structures depending on the input signal characteristics.

Component or system **programmability** shall contain all possible behaviors of a component or system with no change of the hardware structure and instruction set architecture. Please, note that the inclusion of timing is necessary since otherwise processor behavior is only delimited by determinism and memory, as known from computational theory. This definition of programmability includes dynamic adaptability at run time, which seems intuitive.

Component or system **configurability** shall contain all possible behaviors of a component or system as a result of hardware parameter modification **at design or production time**. Examples are memory size, word length, or bus width. This definition also includes (EP)ROM configuration and all board level configurations used to adapt a larger system at production time.

Component or system **reconfigurability** shall contain all possible behaviors of a component or system as a result of reconfiguration without change of the physical hardware. Reconfiguration is limited to changing the logic structure or interconnect of a component or system as seen from the instruction set model while the physical hardware remains unchanged. Examples can be found in the previous section on system specialization.

Dynamic adaptability, programmability and reconfigurability shall be summarized in the term **flexibility,** while **adaptability** shall summarize all four definitions.

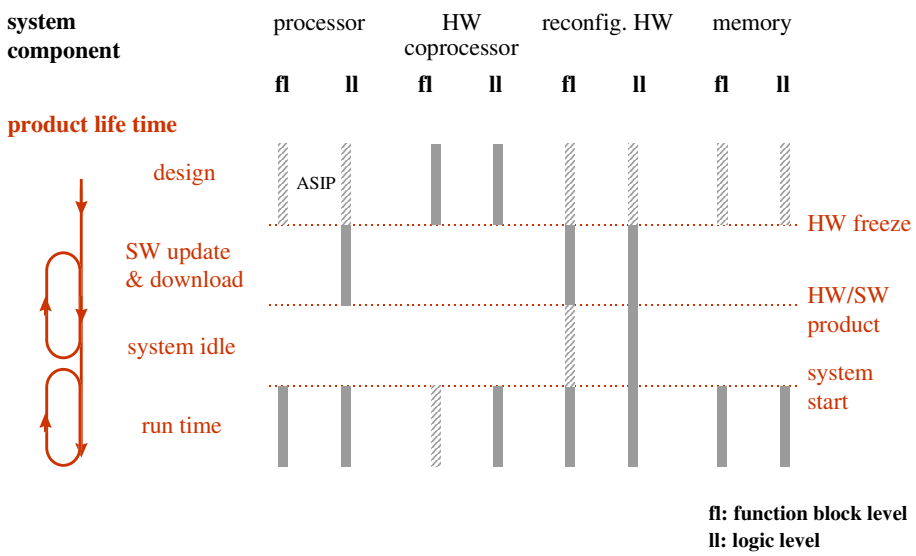



*Figure 19*: Adaptability during product life time

A closer look at the definitions shows that we have just introduced sets of behaviors. To be a little more formal, we define the terms **dynamic adaptability set**, **programmability set**, etc.. In practical cases, these sets are finite but hard to determine. As a next step, we should restrict the system behavior in the sets. The comparison of MMX and TM1000 suggested that we should define an application set of behaviors which is of interest to the intended embedded system plus the required constraints (max. power consumption, etc.) and then build cut sets with the adaptability set, programmability set, etc.. What we obtain shall be called **application programmability set**, **application flexibility set**, etc..

The effects of IP adaptability and adaptability requirements on system optimization are comprehensive. For clarity, we will take a two step approach. We will, first, consider the adaptability of different architecture in context of the product life cycle and then outline their impact on the design process.

We use a simplified model of the product life cycle and a simple classification of component types. The product life cycle starts with the design which may be a co-design process or a conventional sequential process. At some time in the design process, the hardware is fixed ("hardware freeze") and deployed for production. The software can still be modified until it is downloaded for product delivery. The software can also be used to create several variants of a product (see introduction), in which case there are several software variants for production. After product delivery, the software is fixed ("SW freeze"), but may be updated later, e.g. as part of a maintenance procedure.

After the product is delivered, the embedded system can be idle (turned off) or running. These two phases will alternate after delivery. When the system is idle, it can be reconfigured. It is, as well, possible to reconfigure at run time, such that the configuration itself becomes part of the run-time system function. A closer look reveals that even then the respective system part usually goes through a (short) idle period. Similarly, software can be downloaded, compiled and executed at run time ("just-in-time"), but this is not the standard case in embedded systems and shall be omitted, here.

Fig. 19 summarizes the relation between component adaptability and product life cycle. It distinguishes function block level (*fb*) and component level (*cl*). The bars in the figure denote adaptability in the respective phase of a product life cycle, a dotted bar shows that only part of the components are adaptable.

We will distinguish 4 classes of components, processors, hardware coprocessors, reconfigurable hardware, and memory. **Processors** are either given as fixed, non configurable cores or are configured or developed as part of the design process for the specific application (Application Specific Processors: **ASIP**s). Processors are programmable, but not reconfigurable. The individual function blocks of a processor (ALU, CU, ...) are not programmable but their function adapts dynamically to the current control messages (derived from the instruction).

The simplest case of a **hardware coprocessor** has a fixed, data independent control and data flow, such as a filter processor consisting of multipliers and adders. Its functionality is fixed at the function block and the component levels. Coefficients and data may change but the set of operations to be executed remains unchanged. A coprocessor can, however, change its behavior dependent on control messages and data, just as explained above when discussing the specialization techniques. Hardware coprocessors are developed during the design process possibly using IP function blocks.

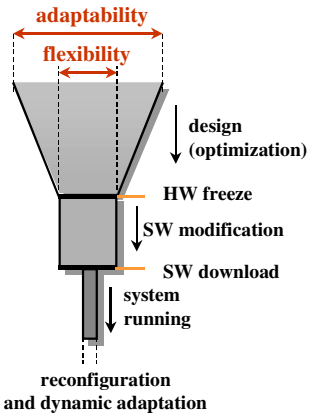
**Reconfigurable hardware** summarizes FPGA based designs and reconfigurable processors. They can be based on fixed components with fixed interconnect, such that there is no adaptability effective in hardware design. The FPGA function is downloaded similar to a software program, both for the function block level and the component level, while in some reconfigurable computers, the function block level is fixed and only program and interconnect are reconfigured. Since an FPGA can be used to implement a simple filter, the reconfigurable hardware might have no dynamic adaptability.

Finally, **memories** must not be omitted from this consideration. Memories can be used as fixed entities or can be configurable at design time. They are usually not programmable, but complex dynamic memory circuits, such as SLDRAMs can even have reconfigurable bank assignment and protocols, as we have seen. Dynamic adaptability is, e.g. found in cache memories.

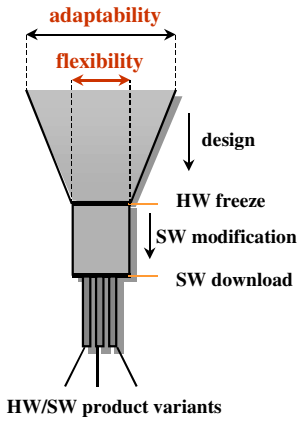
Obviously, the different forms of adaptability are not equivalent and lead to different target system properties with respect to market value, reuse, maintainability, etc.. Fig. 19 suggests that an embedded system specification should, therefore, contain adaptability requirements with respect to the different phases of a product life cycle. To be used in optimization and verification, these requirements should be formal. That this is not the

case, today, can easily be explained with the observation that most specification approaches originate in software engineering.

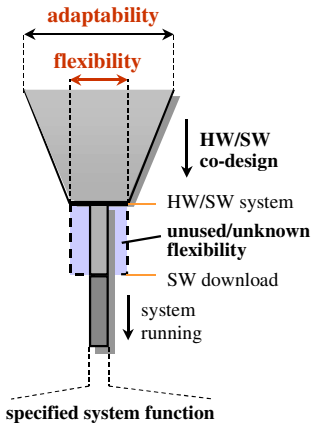
## 7.3. IMPACT ON THE DESIGN PROCESS AND DESIGN OBJECTIVES



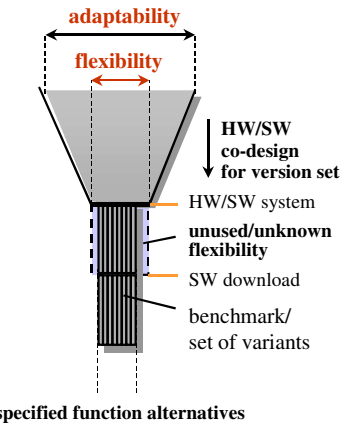

**20a:** Flexibility and adaptability- a pictorial representation




**20b:** Design with product variants




**20c:** Flexibility and co-design




**20d:** Design with mutual exclusive specifications

*Figure 20*: Flexibility and adaptability - a pictorial representation

As a second step, we will consider the impact of adaptability on te design process. Configurability is a requirement for components to be reused, e.g. in libraries, but not for the target embedded system to be delivered, which requires flexibility. So, rather than discussing the challenges of IP based design (e.g. synthesis with IP), we will consider the impact of target architecture **flexibility requirements** as a **design objective** or even **design constraint** on the design process. We will, hence, consider the questions:

- How do the current approaches to co-design influence the target system flexibility?
- What techniques shall be used in the design process to control flexibility?

We will use the picture in fig. 20 to visualize the problem. Fig. 20a introduces the pictorial representation. The design process adapts the system components and architecture to the application. We have seen that this corresponds to a specialization i.e. a reduction of the adaptability. When the HW design is forwarded to production, the adaptability is reduced to what we defined as flexibility of the system. Depending on the remaining flexibility, the hardware platform can still be used for many purposes including the intended application. Only when the software is downloaded, the embedded system is finalized, and what is left are reconfigurability and dynamic adaptability. If designed correctly, the resulting system implements the required behavior. The software can still be changed or updated or it can be used to create different embedded system variants which is shown in fig. 20b.

The current co-design process starting from a single specification does not take such mutual exclusive software variants or late changes into account. The flexibility of the design, which comes from programmable processors or reconfigurable units, is not fully explored, not even documented. This holds for manual designs, but even more so for automated co-synthesis [43]. Fig. 20c visualizes this situation.

In contrast, general purpose computing has always aimed at system optimization under flexibility requirements. For this purpose, it uses benchmarks, which are sets of representative applications which are, one at a time, analyzed as running on the computer [7]. The systematic definition and evaluation of benchmarks ("quantitative" design approach) was found to be a major driver for progress in computer architecture. The benchmark approach accounts for software variants as well as for changes after product delivery which is the idea of a general purpose computer. So, as a first step, it is a good idea to define and use benchmarks for embedded system design. Fig. 20d demonstrates the effects on our abstract model.

There are however, significant differences. First, in general purpose computer design, the target system is the hardware platform with a rough idea on the type of software which will later run on the system. Embedded system design, however, aims at hardware/software systems which means that all constraints (timing, power, ..) must be analyzed and verified including the software. There is no such application verification in general purpose computers, except for specialized system parts, e.g. multi-media function accelerators. Interestingly, most papers on multimedia accelerators for general purpose computers (e.g [21]) give much space to the selection and detailed manual analysis of the intended applications rather than relying on benchmark simulations.

The second issue is the selection of benchmarks. It seems obvious that the higher the specialization, the larger the effect of benchmark selection on the target architecture and the more risky the exclusive use of benchmarks as a quantitative measure in processor design. Since benchmarks measure system properties, which are defined in the system specification, they become part of the system specification. So, we can *incompletely specify* a system with a benchmark set which misses important examples, we can unnecessarily *constrain* system design by an extended set of benchmarks, and we can *incorrectly weigh* the benchmark set if the members of the set overemphasize a certain system property. It would, e.g., be easy to define a benchmark set where the TM1000 Image Coprocessor appears almost useless if we just select enough non-standard video algorithms.

One approach to a verifiable benchmark set is to enumerate all variants of a system including the expected changes. In [44], such a solution was proposed for ASIP design, derived from a very similar solution for systems with concurrent processes but without variants [45]. Different tasks with timing requirements were serialized into a single large sequential process which was, then, synthesized such that all timing requirements of all tasks were met. Notably, synthesis turned out to be very sensitive to the order in which the variants (here: the different applications) were applied which underlines the challenges of a design process based on benchmarks. Similarly, the authors of [46] optimize different functions of an audio/video codec separately but define metrices to identify commonalties between functions which are then used for hardware optimization. Again, they use heuristics to select an appropriate variant order for co-synthesis.

Unfortunately, these solutions are not easily extendible to more complex systems with multiple concurrent processes. Also, with more complex systems, enumeration of potential system variants becomes prohibitive.

A more general approach is to explicitly describe mutually exclusive system variants in the specification. The process model in [47] uses boolean equations to describe sets of system parts which are executed together. This can be used to for scheduling of systems with different system modes (dynamic adaptation and reconfiguration) but equally well to describe mutually exclusive system variants. Originally developed for scheduling, this model can also include incomplete process descriptions where only data flow requirements or estimated timing with lower and upper bounds are provided. This can be used to summarize expected late changes in not yet fully specified parts. Such process models might become a step towards a more formal description of flexibility and adaptability requirements.

## 8. Selected Co-Design Problems

The classification of system specialization techniques has given structure to the confusing variety of seemingly unrelated embedded architectures. Understanding embedded target architectures is a prerequisite for useful and accurate modeling and estimation.

We should also find ways to apply this knowledge to system synthesis. On the one hand, the system specialization techniques which were derived from manual designs could be applied as optimization strategies of a synthesis and code generation process. Current research approaches include **instruction set definition** [48, 49], **instruction encoding** for code compression [50] and **memory optimization** [51, 52, 53]. The other techniques are less examined, in particular the system level techniques are largely unexplored. Just to highlight one of the problems, **global control and data flow optimization** would be a worthwhile topic which goes beyond the work in data dominated systems which has been mentioned above. Related to this problem are **communication channel selection and communication optimization**. **Component interface synthesis** is already an active research topic [54, 55, 19]. Specialization for **power minimization** would clearly be another topic of interest. All problems should be approached under **flexibility requirements**.

On the other hand, the classification of specialization techniques could be applied to support **component selection** and **component reuse**. In this case, we could characterize a component with respect to the specialization techniques and then compare if this specialization fits the application. Such an abstract approach could also include abstract flexibility requirements while synthesis would require a more formal definition of flexibility.

## 9. Conclusion

There is a large variety of target architectures for hardware/software co-design. This variety is the result of manual architecture optimization to a single application or to an application domain under various constraints and cost functions. The optimization leads to specialized systems. Target system optimization can therefore be regarded as a specialization process. We have identified a set of system specialization techniques for components and systems which can serve to describe and to classify the resulting target systems. A number of target system examples for control dominated and data dominated application domains with different models of computation have been analyzed based on this classification. We identified various degrees of specialization and discussed the conflict of specialization and flexibility. Finally, we analyzed the system design process with respect to the related issues of reuse and system flexibility system discussed several approaches to include flexibility requirements in the design process. We identified many open problems which require further research in target architectures.

## 10. References

1 Semiconductor Industry Association (1997) *The national technology roadmap for semiconductors.*

2 Konkurrenzlos Preisgünstig - Siemens (1998) Embedded DRAM für Consumer- Applikationen (in German). *Journal Markt&Technik,* **3**, 24-27.

3 Stankovic, J.A. and Ramamritham, K. (1988) *Tutorial on Hard Real-Time Systems*. IEEE.

4 Lala, P.K. (1985) *Fault Tolerant & Fault Testable Hardware Design*. Prentice Hall.

5 Pradhan, D.K. (1986) *Fault Tolerant Computing Theory and Techniques* **I** and **II**. Prentice-Hall.

6 Ott, H.W. (1988) *Noise Reduction Techniques in Electronic Systems*. Wiley&Sons.

7 Hennessy, J.L. and Patterson, D.A. (1996) *Computer Architecture - A Quantitative Approach.* 2nd edition. Morgan-Kaufmann.

8 Flynn, M.J. (1972) Some Computer Organizations and their Effectiveness. *IEEE Trans. on Comp*., 948-960.

9 DECchip 21030 (1994) Reference Manual. Digital Equipment.

10 Österling, A. et al. (1997) The COSYMA System, in J. Staunstrup, W. Wolf (eds.), *Hardware/Software Co-Design*, Kluwer, 263-282.

11 TMS320C80 (MVP) (1995) Collection of User's Guides. Texas Instruments.

12 Vissers, K.A. (1995) Architecture and Programming of two Generations Video Signal Processors. *Microprocessing and Microprogramming*, 373-390.

13 Lee, E.A. and Messerschmidt, D.G. (1987) Static Scheduling of Synchronous Data Flow Graphs for Digital Signal Processing. *IEEE Trans. on CAD*, 24-35.

14 Lippens, P., van Meerbergen, J., van der Werf, A., Verhaegh, W., Mc-Sweeney, B., Hiusken, J., McArdle, O. (1991) PHIDEO: A Silicon Compiler for High Speed Algorithms. *Proc. EDAC 91*, 436-441.

15 Jerraya, A. et al. (1997) Languages for System Level Specification and Design, in J. Staunstrup, W. Wolf (eds.), *Hardware/Software Co-Design*, Kluwer, 235-261.

16 SAB 80C515 Users Manual. Siemens, 1992.

17 80C552 Product Specification. Philips, 1993.

18 SAB 80C515 Users Manual. Siemens, 1992.

19 Chou, P., Ortega, R., Borriello, G. (1995) Interface Co-Synthesis Techniques for Embedded Systems. *Proc. ICCAD 95*, 280-287.

20 Oppenheim, A.V. and Schafer, R.W. (1997) *Digital Signal Processing*. Prentice Hall.

21 Peleg, A. and Weiser, U. (1996) MMX technology extension to the Intel architecture. *IEEE Micro*, Aug. 96, 42-50.

22 Tremblay, M. and O'Connor, J.M. (1996) VIS speeds new media processing. *IEEE Micro*, Aug. 96, 10-20.

23 Lee, R. (1996) Subword parallelism with MAX-2. *IEEE Micro*, Aug. 96, 51-59.

24 SH-DSP Programming. Hitachi Micro Systems, 1997.

25 SH-DSP Overview. Hitachi Micro Systems, 1996.

26 Liem, C. and Paulin, P. (1997) Compilation Techniques for Embedded Processor Architectures, in J. Staunstrup, W. Wolf (eds.), Hardware/Software Co-Design, Kluwer, 149-192.

27 Rathnam, S. and Slavenburg, G. (1998) Processing the new world of interactive media. *IEEE Signal Processing Magazine*, March 98, 108-117.

28 TM 1000 preliminary data book. Philips, 1997.

29 Riemens, A.K., Schutten, R.J., and Vissers, K.A. (1997) High speed video de-interlacing with a programmable TriMedia VLIW core. *Proc. ICSPAT 97*, 1375-1380.

30 Gillingham, P. and Vogley, B. (1997) SLDRAM: high performance, open-standard memory. *IEEE Micro*, vol 16, no 6, Dec. 97, 29-39.

31 Crisp, R. (1997) DirectRambus technology: the next main memory standard. *IEEE Micro*, vol 16, no 6, Dec. 97, 18-28.

32 Jack, K. (1996) Video Demystified. HighText Publications.

33 Chen, D. and Rabaey, J. (1992) A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithm Specific, High Speed DSP Data Paths. *IEEE Journal of Solid State Circuits*, vol 27, no 12, Dec. 92, 1895-1904.

34 Leijten, J., van Meerbergen, J., Timmer, A., Jess, J. (1997) PROPHID: A heterogeneous multiprocessor architecture for multimedia. *IEEE Proc. ICCD 97*, Austin, 164 -169.

35 Leijten, J., van Meerbergen, J., Timmer, A., Jess, J. (1998) Stream Communication between Real-Time Tasks in a High-Performance Multiprocessor. *IEEE Proc. DATE 98*, Paris, 125-131.

36 Lee, E. and Parks, Th. (1995) Data Flow Process Networks. *Proceedings of the IEEE*, May 95, 773-799.

37 Gerndt, R. and Ernst, R. (1997) An Event-Driven Multi-Threading Architecture for Embedded Systems. *IEEE Proc. Workshop on Hardware/Software Co-Design (Codes) 97*, Braunschweig, 29-33.

38 MC68332 Reference Manual. Motorola, 1991.

39 Hauck, S. (1998) The Roles of FPGA's in Reprogrammable Systems. *Proceedings of the IEEE*, vol 86, no 4, Apr. 98, 615-638.

40 Rabaey, J. (1997) Reconfigurable Computing: The Solution to Low Power Programmable DSP. *IEEE Proc. ICASSP*, Munich, April 1997.

41 Abnous, A., Seno, K., Ichikawa, Y., Wan, M., Rabaey, J. (1998) Evaluation of a Low-Power Reconfigurable DSP Architecture. *Proc. Reconfigurable Architectures Workshop*, Orlando, March 1998.

42 DeHon, A. (1996) *Reconfigurable Architectures for General-Purpose Computing*. PhD Thesis, MIT, 1996.

43 Wolf, W. *Co-Synthesis Algorithms*. This Book.

44 Kim, K., Karri, R., Potkonjak, M. (1997) Synthesis of application specific programmable processors. *Proc. DAC 97*, Anaheim, 353-358.

45 Potkonjak, M. and Wolf, W. (1995) Cost Optimization in ASIC Implementation of Periodic Hard-Real Time Systems using Behavioral Synthesis Techniques. *Proc. ICCAD 95*, San Jose, 446 - 451.

46 Kalavade, A. and Subrahmanyam, P. (1997) Hardware/Software Partitioning for Multi-Function Systems. *Proc. ICCAD 97*, 516-521.

47 Ziegenbein, D., Richter, K., Ernst, R., Teich, J., Thiele, L. (1998) Representation of Process Mode Correlation for Scheduling. To appear: *IEEE Proc. ICCAD 98.*

48 Wilberg, J. and Camposano, R. (1996) VLIW Processor Codesign for Video Processing. *Journal on Design Automation for Embedded Systems*, vol 2, 79-119.

49 Athanas, P.M. and Silverman, H.F. (1993) Processor Reconfiguration Through Instruction Set Metamorphosis. *IEEE Trans. on Computers*, March 93, 11-18.

50 Kozuch, M. and Wolfe, A. (1994) Compression of Embedded System Programs. *Proc. ICCD 94*, 270-277.

51 Mirinda, M., Catthoor, F., Janssen, M., De Man, H. (1996) ADOPT: Efficient Hardware Address Generation in Distributed Memory Architectures. *Proc. ISSS 96*, 20-25.

52 Tomiyana, H. and Yasuura, H. (1996) Optimal Code Placement of Embedded Software for Instruction Caches. *Proc. ED&TC 96*, 96-101.

53 Schmit, H. and Thomas, D. (1995) Address Generation for Memories Containing Multiple Arrays. *Proc. ICCAD 95*, 510-514.

54 Ismail, T.D., Abid, M., Jerraya, A. (1994) COSMOS: A Codesign approach for communicating systems. *Proc. Workshop on Hardware/Software Codesign (Codes) 94*, 17-24.

55 Van Rompaey, K., Verkest, D., Bolsens, J., De Man, H. (1996) CoWare: A Design Environment for Heterogeneous Hardware/Software Systems. *Proc. EURODAC 96.*