Experiences with System Level Design for Consumer ICs

J. van Meerbergen, A. Timmer, J. Leijten, F. Harmsze, M. Strik
Philips Research Labs Eindhoven
meerberg@natlab.research.philips.com

Abstract

The continuing trend towards higher integration densities of ICs makes systems-on-a-chip possible. For well defined application domains "silicon platforms" must be defined which combine efficient implementations with programmability. Platforms are heterogeneous reconfigurable multiprocessor architectures supporting a variety of communication and computation models. As a consequence designers are facing a large architecture space with new possibilities for new architectures. To exploit these opportunities a better understanding of system level architectures is necessary. A first step in this direction is to learn from design exercises. Eventually this may lead towards a system level design method. [1]

1 Introduction

In a .18 micron process (available in 1999) a complete 32 bit microprocessor can be integrated on 1 mm^2. A 1Mbit DRAM has the same size. For logic circuits a density of 6M transistors per mm^2 is possible. According to the SIA roadmap the decrease of feature sizes will continue for more than 10 years to come. At the other hand recent consumer oriented applications easily absorb an exponential increase in processing power. Multimedia applications consist of basic functions such as audio, video and graphics which are combined in a flexible way. Different types of wired and wireless interfaces have to be included. Several real-time signals must be processed in parallel.

The designer has to bridge the gap between the applications and the technology. Large systems which before were implemented on a number of boards must now be integrated on a single chip. At first this might seem a nightmare to the designer, who is not only confronted with technical problems (such as deep sub micron effects influencing performance, timing and power issues), but also with problems such as time to market. At the other hand the design space becomes very large and it is also a challenge to make the right choices and to come up with a real system level design method.

Such a method does not exist today. A first step is based on a true understanding of system level problems. System level design is different from other design problems such as simulation, place and route and synthesis for which generic CAD solutions are studied based on well defined mathematical models. The problem is that system level architecting is less defined and more fuzzy. The goal of this paper is to learn from a real design experiment.

The paper is organised as follows. In section 2 trends in architectures are analysed taking programmable general purpose architectures as a starting point. It will be shown that this leads to some problems. To solve them we need to take the application as a starting point to derive a system level architecture. This will be illustrated with an example in section 3. The example is a multiwindow TV. The problem in discussing examples is that other applications can be totally different. Therefore in section 4 we discuss what can be learned from this experiment and what can be applied to other application domains as well. Finally conclusions are formulated.

2 Trends in architectures

Future systems on silicon will have large design and development cost. Design for integrity and robustness takes a significant effort because of deep submicron effects such as parasitic coupling between components. Noise effects will become relatively more important if the supply voltage drops. For all these reasons it

0-8186-8448-8/98 $10.00 © 1998 IEEE

is expected that future systems will make use of well characterised cores with a large part of the application mapped onto embedded software. Future systems will be software dominated and therefore the software point of view seems to be a good starting point.

Trends in software centric general purpose CPU architectures are characterised by a search for higher levels of instruction level parallelism. First superpipelined architectures were introduced, followed by superscalar architectures. The latest trend is towards VLIW architectures where each PU gets an instruction every clockcycle. The ultimate would be the ultra high performance processor. The SIA roadmap predicts that on chip local clock speeds can be obtained of 1.5 GHz in 2001 up to 10 GHz in 2012.

Although very attractive at first sight, there are also some problems. First of all there is the problem of power dissipation. It is known that programmable architectures can dissipate 2 orders of magnitude more power than architectures which are optimized for power dissipation (see for example [7]). Next these types of load-store architectures usually have a memory bandwidth problem. Finally, from a performance point of view there is the exponential increase in processing power as requested by new applications while the performance offered by general purpose processors exploiting instruction level parallelism is saturating. This is illustrated in [8].

So there is a need for incorporating domain specific solutions in the overall architecture. Many systems need an external memory. For consumer applications there often exist the constraint of one external SDRAM. This leads to possible solutions as shown in Figure 1. This architecture exploits task

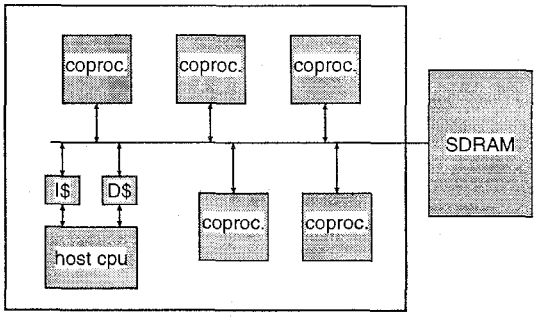

Figure 1: *A single bus architecture template with an embedded CPU core and coprocessors each implementing possibly different functions dependent on the application.*

level parallelism as opposed to instruction level parallelism. It is built around a single bus connected to an external SDRAM memory. A central host CPU processor is connected to the busses via instruction and data caches. Furthermore there are a number of so called *coprocessors* which operate concurrently with the CPU. Each coprocessor is a domain specific processor optimised for a particular (set of) functions. Therefore the architecture can be classified as heterogeneous. Coprocessors can have different levels of granularity which typically reflect the granularity which is present in the application. Coprocessors can have different levels of programmability ranging from programmable embedded DSP cores at one end of the spectrum up to fully hardwired accelerator units at the other end of the spectrum. An important advantage is that reuse is possible of well characterised embedded cores which are present in a library and which are also well supported from a software point of view. Coprocessors can have different implementation styles. For example, the concept allows to use embedded FPGA techniques. While programmable processors are interesting in case arithmetic has to be performed on wide wordlengths (e.g. 32 bits wide) FPGA is interesting for smaller wordlengths. A typical example is a Galois field operation (Figure 2). In software 10 different (32 bit wide) operations are needed. An FPGA implementation only needs 3 exor gates.

```
temp1 = input << 1
if bit(input,7) == 1
      then temp2 := 29
      else temp2  := 0
out = temp1  EXOR  temp2
```

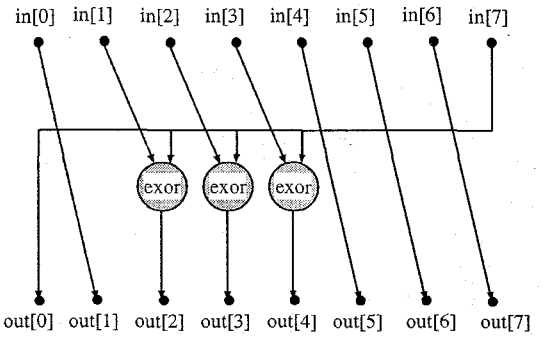

Figure 2: *8 bit wide Galois field operation using 3 exor gates.*

Despite the advantages of this approach there still is a basic limitation if we look at the communication mechanism which is based on a central bus running a standard protocol. The basic idea is that coprocessors communicate with each other via external SDRAM

under control of the CPU. In the SDRAM buffers are allocated under control of the CPU. The CPU is informed when data is available and the real-time kernel (RTK) activates the coprocessors. The central bus introduces a serious bandwidth limitation. Although internally the bus can be made wider (there is a trend towards 64 and even 128 bit wide internal busses) there is a limitation in the number of pins. Furthermore communication between coprocessors, which in principle is internal communication, is implemented via SDRAM. This has a drastic impact on bandwidth and on power dissipation.

Therefore a new approach is needed. What is needed is a true system level architecting approach taking the application domain as a starting point. Characteristic for a system is that *it is composed of a complex set of dissimilar elements that forms an organic whole. The whole is greater in some sense than the sum of the parts. The system has properties that go beyond those of the parts* ([6]). The key issue in system level design is in the communication between the subsystems. Therefore we need a much wider variety of communication mechanisms than the one proposed in the previous architecture.

3 Multiwindow TV

3.1 Requirements analysis

A multiwindow TV application is chosen as a driver to explore system level architecting. Figure 3 shows the different I/O channels. At the input side there are

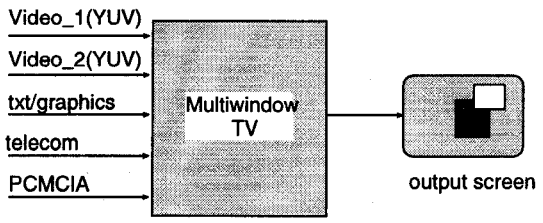

Figure 3: *Interfaces of a multiwindow TV application.*

2 video YUV signals, a graphics input and a telecom interface. An important characteristic is that the 2 video sources are not synchronized. The graphics currently is a 2D character based input to support teletext but in the future 3D pixel based graphics must also be supported. Extra inputs include a telecom port and a PCMCIA interface. This makes it possible to have access to internet services. At the output we have a 100 Hz display device.

Tasks can have different characteristics. Typical examples of event driven tasks are set control, the modem function, user interaction etc... Event driven tasks require a relatively low processing power and memory bandwidth. Hard real-time tasks have deadlines that must be met under all circumstances, for example video processing. Soft real-time tasks have deadlines that can be missed without compromising the integrity of the system ([1]). Examples are generation of graphics, scaling and format conversion of text/graphics etc... real-time tasks are often specified in a task graph. Figure 4 shows a typical example. There are two video streams, a main stream and

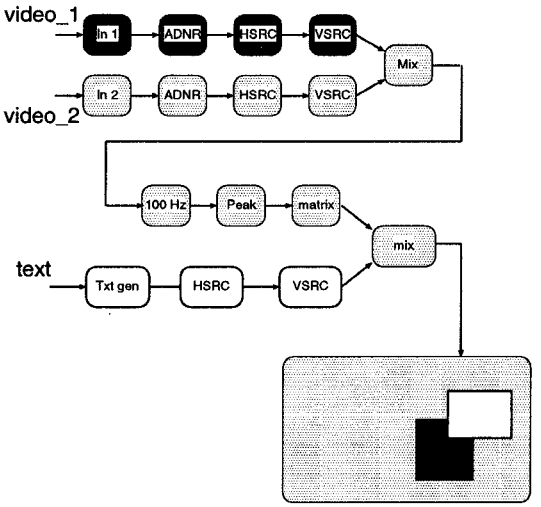

Figure 4: *An example of a task graph showing a basic application mode with two video streams and a text/graphics stream.*

a secondary stream which is shown as a smaller PIP picture on the display (picture in picture). After noise reduction (NR) zooming and scaling of the images is done using horizontal and vertical sample rate converters (HSRC and VSRC). After merging the two video streams a computation intensive 100 Hz up-conversion is performed, followed by peaking (sharpness and color improvement) and matrix conversion. On top of that there is a third teletext/graphics window which has soft real-time constraints.

An important remark is that figure 4 is only one example of a task graph. In total there are many such graphs and new ones are still under definition. *The true specification is an API at the platform layer* listing the basic functions that must be supported. These functions are represented in the task graph as nodes. The order in which these functions appear can be different in the different flow graphs. The architecture

must support "plug and play" with those functions. This way a well defined form of flexibility is specified. A second important remark is related to *quality trade-off*. In case only one video stream is processed there is a need to switch to the highest quality. In case two or more streams have to be processed it is acceptable that the quality of each individual stream is lower.

The total estimated processing performance is about tens of Gops and the total bandwidth between functions is tens of Gb/s which is an order of magnitude more than the architecture in figure 1 can handle. Therefore we have to keep most of the traffic between functions on chip. But this is not always possible. Synchronization between video streams requires buffering as large as several video fields and must therefore be performed via background memory. So the mixing functions are implemented as shown in figure 5. The connections to the external memory

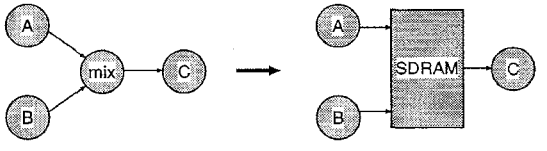

Figure 5: *Mixing and synchronization of images is done via SDRAM memory.*

are modeled as extra inputs and outputs to the flow graphs. In other words the flow graphs are cut at the place of the mixing functions. Therefore the application can be represented as a set of decoupled subgraphs as shown in figure 6. The subgraphs process different streams which are not synchronized. Since the same function can occur in different subgraphs, resource sharing across subgraphs is needed. This requires a solution which provides some run-time flexibility.

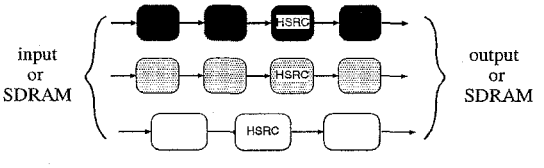

Figure 6: *The application can be represented as a set of decoupled subgraphs.*

3.2 Architecture template

The tasks can be divided into two categories with a periodic and random nature respectively. Most signal processing tasks are periodic. This is obvious for the hard real-time tasks. For periodic tasks we have to design for throughput, for random tasks we want to concentrate on the latency. Since these requirements are different the system has been split in three subsystems with the arbiter and the external SDRAM as the third subsystem. This is shown in Figure 7. Soft real-time task can be modeled as periodic or as random tasks dependent on the circumstances.

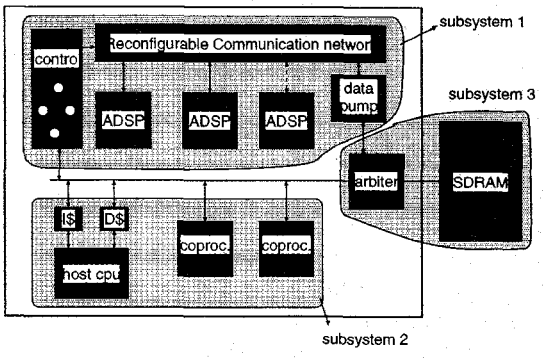

Figure 7: *Target architecture with 3 subsystems. The template allows any number of ADSPs and coprocessors, each executing different functions dependent on the application.*

Since it was clear from the beginning that the bandwidth to the SDRAM is a critical resource special attention was paid to the interface. To maximize the effective bandwidth a basic access action takes 1 bus time slot which corresponds to 16 clockcycles (burst of 8 and 2 banks) and fetches 512 bits. The interface runs at a local clock of 96 MHz. Appropriate arbitration mechanisms can be found in the literature ([3]). The time axis is divided in bus service cycles. Each bus service cycle consists of N (e.g. 64) bus time slots. A number of these times slots (Q=<N) are reserved for periodic streams. As long as sufficient time slots are available priority is given to random requests. In [3] it has been shown that this algorithm minimizes the latency for random requests while guaranteeing the throughput for periodic requests. The number Q can be changed dependent on the application.

Next the other subsystems will be discussed in more detail (Figure 7). The random tasks are mapped on subsystem 2 which is a classical architecture with an embedded CPU similar to Figure 1. The periodic tasks are mapped on subsystem 1. The basic periodic functions are implemented in Application Domain Specific Processors (ADSPs) which are communicating with each other via a reconfigurable network. Typical examples of ADSPs include HSRC, VSRC, 100Hz conversion, peaking etc... ADSPs can span a

wide range of programmability from hardwired dedicated functions, over weakly programmable processors up to programmable ones, including embedded FPGAs (=programmability in hardware). The last category is especially interesting in case we want to add functions after the design is finished. ADSP are similar to so called satellite processors in [7]. In contrast to coprocessors ADSPs operate autonomously.

The underlying communication model is dynamic dataflow. Processors are isolated from the communication network via buffers. Processors are active when data is available and are blocked when inputbuffers are empty or outputbuffers are full. This way the data processing is separated from the communication.

The communication makes use of labeled data packets. The role of the reconfigurable network is to ensure the correct transmission of data packets from the producing to the consuming processors. The network is a TST network (time-space-time network, [9]) controlled by the task graph which is loaded in the controller. The controller can guarantee communication bandwidth between inputs and outputs of the network. More details can be found in [5].

Another important component is the datapump. It takes care of buffering and communication between the stream based subsystem and the SDRAM. It also takes care of the arbitration between periodic requests. Therefore it plays the role of a data cache for the subsystem. Memory inputs and outputs (see figure 6) are mapped onto this component.

4 Discussion

Since different applications can have different characteristics there will exist many different system level architectures. This leads to the question of what can be learned from actual design exercises that is applicable to other application domains as well. This generalisation phase is an important step towards a system level design method ([2]). In this discussion we also take into account the experiences with the design of a low cost MPEG2 encoder for consumer applications ([4]. Two aspects are discussed in this section. First the resulting architecture template is discussed, followed by a discussion of the process of *how* we came to this architecture.

4.1 Architecture template

Although the content of the ADSPs and the coprocessors is application dependent, some characteristics of the template are more generally applicable.

First the resulting architecture template is *heterogeneous* from different points of view. The ADSPs and coprocessors are implementing different functions at different levels of granularity. They span a wide range of programmability from hardwired solutions at one end of the spectrum, over weakly programmable cores to fully programmable DSP or CPU embedded cores. By making the right trade-off for each application domain solutions can be obtained which combine flexibility with efficiency from area or power dissipation point of view. Finally the resulting architecture template is also heterogeneous because it combines different communication principles.

Secondly the template does not impose any restrictions on the implementation style and is *open for reuse.* For example, embedded FPGAs are an interesting option. ADSPs can be designed using synthesis tools such as for example PHIDEO. Buffers can easily be designed as an interface between the strict periodicity as required by PHIDEO [10] and the dynamic stream model of the template. The only extension is that it must be possible to hold the processor to implement the data driven concept. The same holds for importing existing blocks from other designs.

4.2 Process of architecting

In the process of architecting a number of heuristics were applied which are generally applicable. In system level design it is important to identify bottlenecks as soon as possible. In this example the bottleneck is the throughput for internal communication between ADSP processors as well as for external communication between ADSPs and the SDRAM. This has lead to a *hierarchical approach* to the memory architecture. At the top level we have the SDRAM, at the subsystem level we have the datapump and at the level of the ADSP processors we use local Fifo buffers. The same hierarchy is reflected in the arbitration scheme. At the top level we arbitrate between periodic and random request coming from different subsystems. The datapump arbitrates between periodic request and at the level of the ADSP processors we arbitrate between the different tasks belonging to different subgraphs.

This hierarchical approach has lead to the definition of subsystems with *well defined interfaces.* The complexity often comes from the interfaces, more in particular from timing aspects related to interfaces. Therefore these aspects should not come as an afterthought but should be taken into account from the beginning. In the example above the throughput was guaranteed by the architecture. Simulations were used to verify, for example, the latency not the throughput.

The architecture also has an important *impact on the mapping.* For example in the stream oriented subsystem the major tasks are a time assignment for the TST network and the checking of constraints with respect to the throughput. In the context of the architecture these two tasks are well defined and can be solved easily.

An important remark is related to *programmability.* In the example we first tried to understand what kind of programmability was really needed. This was made part of the *specification* which is a list of functions at the platform API layer. This form of specification is dynamic, i.e. it is changing during the design. The final specification is often the result of an interactive process with the customer. Future design methods have to take *late spec changes* into account.

5 Conclusions

The definition of a system level architecture is important for a number of reasons. First of all, it is needed in order to manage system level complexity. Special attention must be paid to interfaces, more in particular to timing aspects of interfaces. Second it is needed for scalable solutions. In many cases a design is part of a whole family of designs which are planned on a roadmap. Furthermore a good architecture makes transfer easier and documentation better readable. Finally exercises in architecting help to understand what the real problems are and are therefore a first step towards the development of a system level design method.

Acknowledgement
The authors want to thank Medea for sponsoring this work in project AT-403 (SMT).

References

[1] A. Burns, "Scheduling hard real-time systems: a review", Software Engineering Journal, pp. 116-128, May 1991.

[2] H. De Man, "Education for the deep submicron age: Business as usual ?", Proceedings of the 34th Design Automation Conference, 1997.

[3] Hosseini-Khayat and A. Bovopoulos, "A simple and efficient bus management scheme that supports continuous streams", ACM Transactions on Computer Systems, 13, no. 2, pp. 122-140, 1995.

[4] R.P. Kleihorst, A. van der Werf, W.H.A. Bruls, W.F.J. Verhaegh and E. Waterlander, "MPEG2 Video Encoding in Consumer Electronics", Journal of VLSI Signal Processing, 17, pp. 241-253, 1997.

[5] J. Leijten, J. van Meerbergen, A. Timmer, J. Jess, "Stream communication between real-time tasks in a high-performance multiprocessor", Proceedings DATE Conference, 1998.

[6] E. Rechtin, "Systems Architecting: Creating and building complex systems", Prentice Hall, 1991, ISBN 0-13-880345-5. pp. 68-72, Edinburgh, March 1990.

[7] J. Rabaey, A. Abnous, Y. Ichikawa, K. Seno, M. Wan, "Heterogeneous Reconfigurable Systems", in Signal Processing Systems, pp. 24-34, 1997.

[8] H. Sasaki, "Multimedia complex on a chip", Proceedings ISSCC, pp. 16-17, 1996.

[9] M. Schwartz, "Telecommunication networks: Protocols, Modeling and Analysis", Addison-Wesley, Reading.

[10] W. Verhaegh, P. Lippens, E. Aarts, J. van Meerbergen, "Multidimensional Periodic Scheduling: A Solution Approach", Proceedings of the European Design and Test Conference, pp. 468-474, 1997.