RED: A Reconfigurable Datapath

Fernando Rincón, José M. Moya, Juan Carlos López
Universidad de Castilla-La Mancha
Departamento de Informática
{frincon,fmoya,lopez}@inf-cr.uclm.es

Abstract

The popularity that certain applications such as mobile telephony or mp3 audio reproduction have gained lately, has made it possible the increase in the number and complexity of embedded systems. Depending on the tradeoffs performance/flexibility/cost, several architectures that range from ASIPs to full-custom computing machines, based on reconfigurable logic, are being used. Here, what we propose, is a new architecture based on a standard microprocessor whose functionally will be enhanced by a reconfigurable datapath, acting as a customized coprocessor. This architecture not only offers a reasonable tradeoff between cost and performance, but also can be easily exploited through the use of a HW/SW compiler.

1. Introduction

The increase in complexity and requierements of current embedded systems has led to the appearance of alternative architectures to standard microprocessors and DSPs, that were being used until now.
These alternatives differ in their adaptability, their performance and their cost. Intermediate solutions are normally based in the use of reconfigurable logic, however the way reconfigurability is provided can be rather different.
This paper presents a novel architecture, RED, based on a standard processor plus some reconfigurable logic, in the form of a pipelined datapath, that can execute several complex operations of more than two operands at a time. RED would then be used for those operations of an application that are critical in time, while the rest are being executed by the processor. This task can be easily performed by a compiler based on FLECOS methodoly (see section 5), avoiding the classical problem of hardware/software partitioning.
Section 2 presents an overview of some of the architectures that are currently used for embedded systems, and justifies our proposed alternative. Section 3 describes RED architecture. Section 4 describes its fuctionality, while section 5 deals with the problem of designing applications based on RED. Finally, section 6 presents some conclusions and current state of the work.

2. Types of Architectures

We could fix the following three parameters to identify the characteristics of several of the architectures currently used for embedded systems: flexibility of the system, cost and performance. Depending on them we could make the following 5 groups.

1. ASIPs
2. processor + DSP
3. reconfigurable processors
4. processor + FPGA
5. Full Custom Computing Machines

	Flexibility	**Cost**	**E.O.U**[1]	**Perform.**
1	none	low	high	med./high
2	low	medium	low	med./high
3	medium	med./high	med.	med./high
4	medium	med./low	high	med.
5	v. high	high	low	high

Table 1: Types of Architectures

With respect to the relationship between flexibility and cost, as it can be seen on table 1, the first and the last are the two opposite solutions. The use of processors specifically designed for certain kind of applications, such as DSPs for example, is the cheapest alternative, although, at the same time, the least flexible, where flexibility would be the adaptability of the architecture to the applications it should support. On the other side, custom computing machines are based normally on some kind of reconfigurable logic, thus allowing to tune their architecture to each application executed over them. However their cost is enormous

[1] Ease of Use

compared to the previous case, in part for their generality, in part for the inclusion of reconfigurable logic. These solutions also differ in the performance they can achieve at the execution of certain applications. Since FCCMs can adapt their architecture they can be even one order of magnitude quicker than ASIPs.

But between these solutions, new architectures have been proposed that try to reach the performance needed for most common embedded systems, without paying a high cost for it. We have identified three of these solutions in table 1. They all have in common the use of a more less standard microprocessor and additional hardware, in the form of a DSP or reconfigurable logic. The standard processor is cheap, and deals with control tasks of the application, or those not critical in time. The additional hardware is used for critical tasks, or those that don't fit very well in the microprocessor (because of special data widths for example).

Successful implementations of applications using a processor and a DSP have been reported, such as [dsp98]. Here the DSP provides the processor with an extended set of instructions for digital signal processing, at the extra cost of a DSP, which is not ver high. The main disadvantage of this approach is the lack of a good design methodology. It must be decided at the very first stages of the design process which parts of the application will be performed in each processor. Then the code must be developed separately. Synchronization between both processors must be provided manually. An additional problem is that depending how autonomous the DSP is from the microprocessor, the latter is most of the time dedicated to supervise the DSP, which prevents it from performing other tasks.

The rest of architectures are all based on the use of reconfigurable logic. However, they differ in how this reconfigurability is used.

Xilinx proposes in its ARC architecture [Dav01] the use of an FPGA to hold an configurable processor. The core of the processor is custom-made, including only the necessary instructions for a certain application from a set of possible ones. This doesn't requires nothing but a standard FPGA, but that also means that a high prize is paid for the hardware related to control sequencing and many other tasks of a microprocessor that don't take any benefit from being implemented over an FPGA.

Altera has recently released the Excalibur series [exc00]. They consist in a standard core, such an ARM or a MIPS, integrated with an FPGA in the same chip. Here the reconfigurable logic is only used for those parts of the circuit that have to perform a critical task, while the rest are executed by the microprocessor. The main problem of this approach is again hardware/software partitioning. Even the knowledge of a hardware designer is needed for the design of hardware parts that will be held by the FPGA, while in ARC a custom compiler is derived automatically once the set of instructions has been decided.

Other alternatives, that make a more sophisticated use of reconfigurable logic exist, although all of them are still under research. One is GARP, from Berkeley university [HW97], and is based on a MIPS core plus a reconfigurable array organized in a set of contexts, one holding a different circuit, but only one active at a certain time.

Another one is Chimaera [HFHK97], from Northwestern University. Chimaera is a reconfigurable functional unit that is able to provide a set of simultaneous hardware operators, that can be replaced dinamically, during the execution of the application. In both architectures the reconfigurable part is tightly coupled to the microprocessor, sharing, for example, the same data cache to reduce memory bottlenecks. Both are also based on the use of a compiler for developing applications, although they work at a different level of granularity. In GARP the compiler uses VLIW techniques for finding the inner loops that will be fully synthesized in hardware, and stored in one of the contexts. Chimaera compiler works at the level of operations, trying to identify those that don't fit in standard processors, or that can be packed and executed in parallel.

The third one is Piperench [GSM+99], from Carnegie Mellon University. PipeRench is a dynamically reconfigurable datapath that allow the implementation of pipelined operations that are not constrained by the physical number of stages in the datapath.

RED, that shares some features of the last three alternatives, is not an FPGA, but a reconfigurable coprocessor. Since it will work at the level of operation it isn't necessary such a general architecture. It is a datapath much like PipeRench, although the granularity of the stages, the use of local memory and the reconfiguration schema is rather different. Reconfiguration is based, as in Garp, on the use of several switchable With respect to Chimaera, RED also provides simultaneous execution of instructions, but it is achieved through the use of a pipeline whose stages can replace their functionality at each clock cycle.

3. RED Architecture

RED has been conceived with the following ideas in mind:

- Provide HW acceleration to critical operations but avoiding the problem of HW/SW partitioning, as it will be explained in section 4. This implies working at the level of operations or system calls, which is a lower degree of granularity compared to classical HW/SW codesign approaches.

- Provide a flexible architecture that allows the implementation of operations at a variable level of complexity.
- Achieve a high percentage of utilization of reconfigurable hardware.
- Include a data cache to reduce the overhead produced when accessing to data memory.
- Reduce the overhead caused by communication between the main processor and the coprocessor, reducing the number of orders, but without increasing instruction width.

Let's see what the implications of each of the above points are. Since we are interested in working at the level of operations, the reconfigurable architecture can be organized like a datapath, where information has a directional flow, from registers that contain operators to other intermediates, and finally to the ones that will hold the result, being processed as it travels from one register to another. This schema doesn't need the amount of generality that normal FPGAs provide, and that have a great impact in the density, speed and cost of such devices. For example, as a result of this directional approach a much simpler interconnection architecture is needed.

Flexibility can be easily provided by splitting operations into stages, so each operation can use as many as are necessary. Even at each stage new operands could be incorporated to the partial result sent by the previous one.

Pipelining comes as a natural extension to the above point. Since operations are splitted into stages, the termination of each one can consist in a set of registers, so every stage is independent from previous one. Pipelining is the normal solution to increase the throughput of the system with a low additional cost, so instead of letting the user design in a pipelined fashion, RED is itself a pipeline.

Another extension provided by RED is dynamic reconfiguration. It's achieved through the use of several contexts, that hold the configuration bits for every stage in the datapath. Context-switching mechanism was first introduced by [DeH96] in its DPGA (Dynamically Programmable Gate Array). DPGA had several memory layers, each containing the programming bits for a different configuration (one active at a time), that made possible to change functionality of the circuit in a single cycle, by just choosing a different context. Applied to RED, every stage of the pipeline has its own set of contexts, so each one holds a certain part of one operator, thus having as many complete operators as the number of contexts in the pipeline. This makes possible not only initiating a new operation every cycle clock (since it is a pipeline), but also switching to another context to change the operator, thus providing some kind of pipelined paralelism.

Top-level architecture

Figure 1 shows one possible implementation of a coprocessor with a datapath with all the above characteristics. The stages, each containing a configurable combinational part plus a set of registers to hold the result, can be easily identified. Not all of them must be equal. For example, the first one receives three inputs, while the second only the output from the first. The third again receives two more inputs comming from a external register file. This organization allows the implementation of complex operators that can deal with many operands in the same instruction.

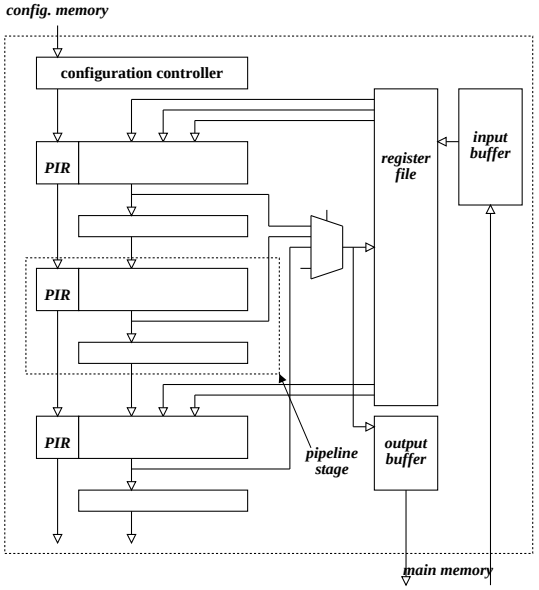

Figure 1: RED Architecture

Pipeline Stages

Figure 2 shows the insides of one of the stages. The combinational part is an array of programmable purely combinational cells. The configuration bits of these cells are stored in local memory planes (contexts), each providing a different functionality. The contexts are connected to combinational logic through some kind of multiplexor, so it is possible to switch between them just modifying the value of a control signal. As everything is combinational inside the programmable array, the switching can take place from one cycle to another. This gives the stage the possibility of performing a different computation at each cycle clock. Of course, only between those stored in the contexts.

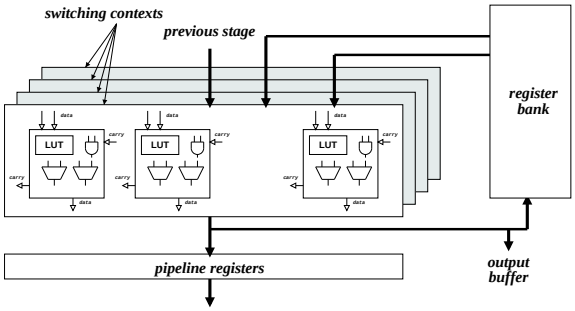

Figure 2: Pipeline Stage

The combinational array receives several inputs that come from previous stages and from the local data cache. The inputs are then processed depending on the active context, and delivered to the output, which is the sequential part of the stage. This mechanism allows the implementation of operations with more than one operand, taking more than one clock cycle (one per stage), but in just one instruction.
The secuential block is composed by a battery of registers. They hold the output generated by the combinational plane, and isolate the stage from the next one. The secuential block it's not reconfigurable.

Local Storage

To reduce memory overhead, RED has been provided with a register file and a input buffer. The input buffer will hold data temporally until the target register from the register file is empty. If the register file is large enough it will store data that is accessed more than once, saving some CPU cycles and avoiding unnecessary memory accesses, that are normally the bottleneck of the system. In the same way, there's an output buffer that stores the data that is not going to be reused, while the bus or the CPU is busy to send it back to memory.

Configuration Controller

The configuration controller is the responsible for sequencing the instructions into the pipeline. Instructions are extracted from a configuration memory (local to RED) and stored into PIRs (pipeline instruction registers). Every instruction holds the context number of the operator to execute and the registers that will be used as operators. The contents of the PIRs are shifted down to the pipeline as the execution goes own so each stage has access to the whole instruction.

4. RED execution

Now that the architecture of the datapath has been described, let's see how it works. RED can hold as many operations as the number of different contexts. These operations must be designed in a pipelined fashion, and can have a variable number of stages. The combinational part of the stage will be translated into a configuration context, that will program the combinational array to perform the corresponding part of the operation and will route the result to the sequential block of the stage. For every stage, then, each context will hold one part of an operation an the routing to store the result.
Let's suppose that just before the application starts we will have preloaded all the contexts for all the stages. Stage 1 could then start executing the first part of operation **A**. At the next clock cycle, operation **A** will continue its execution, this time, at stage number 2, taking the result from stage 1 and having the possibility to add new operands. Meanwhile, at stage 1 we have 2 possibilities. We could start another copy of operation **A** again, this time with new operands, or we could switch to a different context and perform the first stage of a different operation, **B**.
Summarizing, it's not only possible to start the execution of a new operation each clock cycle, but even the operation doesn't have to be the same. This way it's possible to have in the pipe more than one different operation executing simultaneously, although each one would be in a different stage of execution.
One additional feature of RED is its ability to chain operations. A group of operations can be combined into a single RED instruction. However all RED instructions are dataflow ones, since its just a datapath. Everything related to control is done by the main processor. What we gain through chained operations is to reduce the number of interactions between RED and the main processor for a single (but complex) task. This has an important consequence, since RED and the processor must be synchronized to guarantee the validity of the program being executed, it's only possible to chain operations that are kept into the same basic block of the program (see section 4). It isn't correct to create instructions with operations that belong to different control branches of the program.

5. Compiling for RED

RED architecture was conceived to be easily integrable into FLECOS methodology [MMDL00]. In fact, FLECOS isn't only a design methodology, but also a set of tools to develop complex heterogeneous systems. It's main goal is reduce the productivity gap that exists nowadays when designing such kind of systems. To achieve it, specification, architecture and optimization criteria must be clearly separated 3, thus maximizing reuse in all these three aspects, and allowing groups working on each of them to perform in parallel. These three items converge in a compiler,

that will generate the code corresponding to the specification, for a certain architecture and with the optimization criteria that has been fixed for the design.

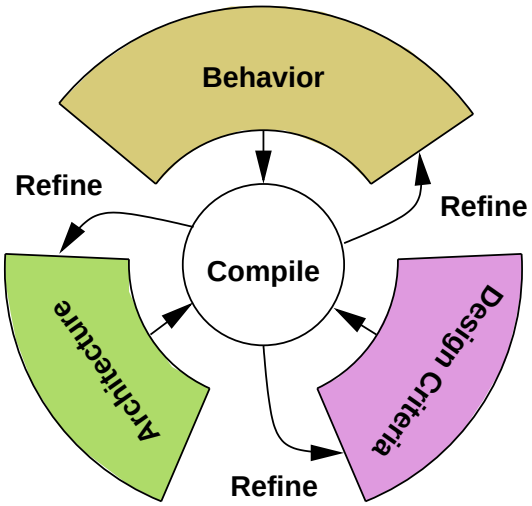

Figure 3: FLECOS

The integration of RED into FLECOS is almost immediate. Part of the work must be done at the architecture side, since it's neccesary to define the machine description (the description of its architecture) for the datapath. Also, the set of operators that RED will be able to provide to a certain kind of application, must be implemented by a hardware designer.
Next, the compiler must be slightly adapted. Since not all operators used by an application can be stored in RED (it will depend on the number of contexts) , it must be decided a mechanism to select only those that are really interesting, probably based on a profiling of the application and an estimation of the data traffic between RED and the main processor.
The compiler could be also modified to detect automatically from the specification which are the operations that can be executed by RED. Otherwise they will have to be indicated explicitly by the specification designer in the form of system calls.
In important characteristic of RED is the possibility of chaining the execution of several operations in the datapath into just one instruction. Compilers normally divide the code to compile into what they are called basic blocks. A basic block is just a set of instructions that are always executed sequentially, since they don't contain jumps. It's possible, therefore, to determine at compile-time which is going to be the sequence of operations to be executed by RED in each one of the basic blocks. So this sequence can be coded into a single instruction for the coprocessor, which has mainly three advantages:

- The main processor doesn't have to keep watching RED constantly, but only send junst one instruction per basic block.
- It's not necessary to provide any synchronization mechanism at run-time between the processor and RED, because although the latter will execute its own program in parallel, sychronization was resolved at compile-time.
- Instructions are custom generated for each basic block, but always using the same set of operators, previously stores in RED contexts. In other kind of architectures, accelerating each basic block would mean designing a custom circuit for each of them, and therefore having to reload each one at run-time.

6. Conclusions

A novel reconfigurable architecture for high-performance embedded systems has been outlined. It provides a good degree of flexibility, since it's possible to design complex specialized operators.
The cost of a system on chip based on this architecture could be similar to those implementing an FPGA. RED uses several memory contexts, however it's less general and has a much simpler interconnection architecture than normal FPGAs, which is an important percentage of the final cost. Finally, although it must be tested, the degree of utilization of the reconfigurable hardware can be very high, because of pipelining and the use of several contexts.
RED architecture is currently being prototyped, and will be soon integrated into FLECOS evironment.

*

References

[Dav01] Henry Davis. Conventional dsp or configurable microcontroller: which way to go? *EDN Europe*, pages 26–32, january 2001.

[DeH96] Andre DeHon. *Reconfigurable Architectures for General Purpose Computing*. PhD thesis, MIT, 545 Technology Sq., Cambridge MA 02139, September 1996.

[dsp98] Dsp directory (16-bit). advanced risc machines. piccolo. EDN Magazine [on line], September 1998.

[exc00] Excalibur backgrounder. Altera white paper, june 2000.

[GSM+99] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Reed Taylor, and R. Laufer. Piperench: a coprocessor for streaming multimedia acceleration. *ISCA*, 1999.

[HFHK97] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The chimaera reconfigurable functional unit. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.

[HW97] R. Hauser and J. Wawrzynek. Garp: A mips processor with a reconfigurable coprocessor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 1997.

[MMDL00] J.M. Moya, F. Moya, S. Domnguez, and J.C. Lpez. Multi-language specification of heterogeneous systems. *Forum on Design Languages (FDL'2000)*, September 2000.