36

Design of Low Power Processor Cores using a Retargetable Tool Flow

36.1 Introduction

Processor cores in SoC design, SoC integration and low power design, Architectural tool support for low-power processor design

36.2 A retargetable tool-flow for designing power-efficient application-specific processors

The CHESS/CHECKERS retargetable tool-suite, Architectural scope, Architectural exploration, Power-conscious architectural design

36.3 Low-power processor architecture design

General characteristics, Instruction-set architecture, Micro-architecture, Methodology

36.4 An ultra-low power DSP for audio coding applications

Background and goals, Architecture, Low power techniques, Results

36.5 Conclusions

Gert Goossens
Target Compiler Technologies

Dirk Lanneer
Target Compiler Technologies

Peter Dytrych
Philips Digital Systems Laboratories

36.1 Introduction

With process geometries shrinking to nanometers, unprecedented levels of silicon integration are now available. This has fuelled the design of complete electronic systems on a single multi-million transistor chip. To master the design complexity of such systems-on-a-chip (SoC), the *re-use of processor cores* has become an important design paradigm. Different types of pre-designed and pre-verified processor cores can be instantiated and connected as building blocks in a heterogeneous chip architecture. However, *power consumption* is becoming a major hurdle in the successful design of future SoCs.

This paper describes a *methodology for designing low power processor cores* in SoCs. Its key component is the ability to quickly and adequately *customise the instruction-set architecture* of the processor core, to match the characteristics of the application. It is demonstrated that this allows for a drastic reduction of the power consumption of the processor, while retaining sufficient design flexibility as offered by a programmable processor. The methodology is supported by a *retargetable tool-suite*, offering *architectural exploration, software development* and *verification* capabilities. The practical applicability of the methodology and tool-suite is demonstrated by the design of an industrial *ultra low power DSP core* for audio coding applications, named COOLFLUX DSP

Processor cores in SoC design

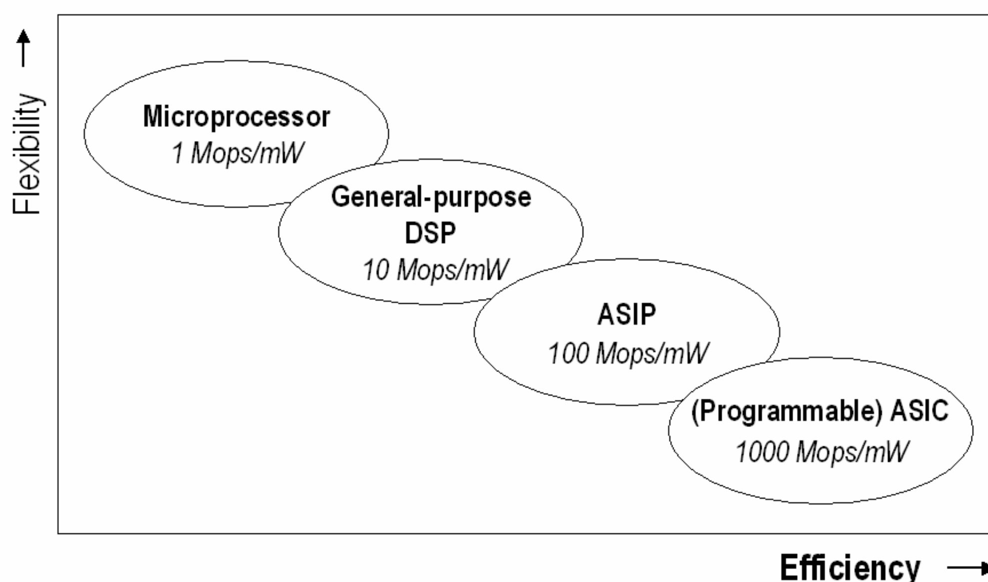

FIGURE 36.1 Classification of processor cores in SoC design

Figure 36.1 shows a classification of processor cores used in SoCs. On the one hand, *general-purpose microprocessor* and *digital signal processor (DSP) cores*, as available from intellectual property (IP) vendors, offer most flexibility. On the other hand, *application-specific cores* are being designed to implement system functions that are critical in terms of computational throughput and power dissipation. Due to their application-specific nature, these cores are often designed in-house, by specialised semiconductor or system companies. Traditionally, these application-specific cores take the form of a fixed-function ASIC block, designed in a hardware description language like Verilog or VHDL. Such an ASIC core consists of a data-path with special-purpose functional units and interconnections. Control flow is typically restricted and can be implemented in a small finite-state machine.

In competitive markets like telecom and consumer electronics, the *flexibility* to take into account rapidly changing functional requirements, and the *efficiency* to cope with high computational throughput and low power dissipation requirements, are both important. New processor cores must combine the best of both worlds. DSP cores are

becoming more application-specific, by extending a general-purpose instruction-set architecture (ISA) with specialised functional units and instructions. This is referred to as *application-specific instruction-set processor (ASIP) cores* [10]. At the same time, new-generation ASIC cores offer a small layer of software programmability on top of a specialised data-path, to allow for limited functional changes that are crucial to extend the lifetime of these cores. This will be referred to *programmable ASIC cores*.

Usually the design of an embedded processor core is significantly influenced by the overall SoC architecture. SOC design addresses the system partitioning, which determines the balance of low power and parallelism, including task parallelism, data parallelism, and instruction-level parallelism (ILP). This defines the broad specification points of the different embedded processor cores in the SoC, which each typically only provide a solution for part of the computational load of the complete system. In this paper we assume that the system partitioning and analysis have been performed upfront, although in practice the procedure is rarely this cleanly decoupled and the optimisation of a processor core architecture is likely to be done in a complete system context.

As an example, two rather different application scenarios could be a digital hearing instrument and an SoC platform for portable consumer audio. The hearing instrument will have very exacting power and area constraints, but can usually be rather application specific and have a small application code base (hundreds of assembly lines), for which an ASIP or programmable ASIC (Figure 36.1) is most adequate. The audio platform will tolerate less demanding power constraints and have a very broad code base (hundreds of thousands of assembly lines), requiring a more general-purpose 24-bit DSP. The audio platform may also be configured as a multi-processor to allow scalability.

SoC integration and low power design

The requirement of low power dissipation is becoming an important, if not *the most important*, motivation for making processors application-specific. While from an area and gate-count perspective, putting together tens or even hundreds of processor cores on a single chip is not a problem anymore, controlling the heat dissipation and the energy consumption of such a chip becomes a major issue.

To control the power characteristics of the SoC, both the overall *system architecture,* the *architectures of the composing processor cores,* and the *circuit-level implementation* are important. This paper primarily focuses on the *processor core architectural level.* The main idea that is explored is that by making the processor architecture application-specific, i.e. by designing an ASIP or a programmable ASIC instead of a general-purpose processor (Figure 36.1), its power consumption can be reduced drastically.

In the authors' opinion, the following observations are cornerstones of low-power architecture design:

(a) Optimising for minimum cycle count is beneficial for power consumption.

The main part of the dynamic power consumption of a circuit is due to the capacitance effect, and is proportional to the average switching frequency (i.e. clock frequency times an activity factor), and to the square of the supply voltage [9]:

$$P = C \times (f_{Clock} \times Act) \times V_{dd}^2$$

Processor architectures that can implement a given software program in a *small number of instruction cycles*, will generally exhibit better power characteristics. Indeed, a lower cycle count will allow for scaling down f_{Clock}, and especially V_{dd} (known as *voltage scaling*).

A low cycle count can be achieved by introducing *specialised functional units* to accelerate the critical functions of the algorithm, and by providing *instruction-level parallelism*. These are key features of *application-specific processor architectures*.

(b) Optimising for reduced memory access is beneficial for power consumption.

A substantial portion of the power consumption in a processor is due to memory accesses. This is both related to the switching activity on data and address busses, and to the loading of word lines in the memories [16]. Processor architectures that can implement a given software program using a *small number of data and program memory accesses*, will generally exhibit better power characteristics.

Important power savings can be achieved by providing a storage hierarchy. For example, by providing a loop cache, one can avoid excessive program memory fetches for programs containing loop structures. To reduce the required number of *data memory accesses*, the architecture should allow to keep variables as much as possible in *registers* local to functional units. *Program memory accesses* can be reduced among others by designing an application-specific instruction-set with only a *small number of instruction bits*, as well as by using techniques like variable-length encoding and instruction compaction. Again, these are key features of *application-specific processor architectures.*

(c) *Minimalistic architectures are beneficial for power consumption.*

An effective architectural design strategy for low power must be *minimalistic* [6]. By including only those hardware resources that are really needed by the target applications, power consumption can be reduced significantly. Once again, this leads to *application-specific processor architectures.*

(d) *Low power architectural design is holistic.*

An effective architectural design strategy for low power must be *holistic* [6]. To effectively reduce the switching activity and capacitance, all aspects of a processor architecture are important, and only the combination of all elements will result in an overall power-efficient architecture.

Architectural tool support for low-power processor design

Rather than attempting to develop an automatic optimisation tool for low-power architecture design, an *interactive and iterative methodology* is proposed that allows architecture designers to *explore different architectural trade-offs* and obtain *rapid feedback* about the quality of architectural decisions.

This methodology, which takes into account the holistic nature of low-power architecture design (see previous section), is based on a *retargetable tool-suite for processor design* available from Target Compiler Technologies, called CHESS/CHECKERS [2]. Key features of this technology are the following:

- Whereas other architectural design environments are based on a predefined but parameterisable template of a processor architecture [1][3][4][13] one of the key objectives when developing the CHESS/CHECKERS tool-suite was to provide maximum architectural freedom to the designer. In this way the designer can find an optimal balance between architectural flexibility and specialisation, to obtain the best power dissipation characteristics for his/her application.
- CHESS/CHECKERS' architectural exploration capabilities effectively allow to find the architectural sweet-spots that result in low power dissipation.

The retargetable tool-suite must be coupled to a complete *power-aware VLSI design flow*. This allows us to *simulate* rather than to *speculate* about power consumption. Good architectural candidates can be determined in the retargetable tool flow by using CHESS/CHECKERS retargetable C compiler and getting profiling data from the retargetable ISS. These can then be pushed through the VLSI design flow to ensure that a good, low power implementation can actually be achieved. Our experience has been that this process is very revealing and really helps to build efficient processor architectures, taking into account that low-power architecture design is holistic.

The CHESS/CHECKERS tool-suite has been applied successfully to design power-efficient application-specific processor cores for critical applications in wireless and wireline telecommunications, consumer electronics, and medical devices such as hearing aids. The CHESS/CHECKERS tool-suite, and its abilities for low-power architectural exploration, are described in Section 36.2.

Section 36.3 of this paper surveys a number of important architectural optimisations that make part of a holistic strategy for low power architectural design. These optimisations are typically explored in the architectural design phase, using CHESS/CHECKERS.

As an illustration of the methodology, the industrial design of an ultra low power DSP core for audio coding applications is described in Section 36.4. This processor, called COOLFLUX DSP, has been designed by Philips Digital Systems Laboratories [6], with the help of the CHESS/CHECKERS tool-suite.

36.2 A retargetable tool-flow for designing power-efficient application-specific processors

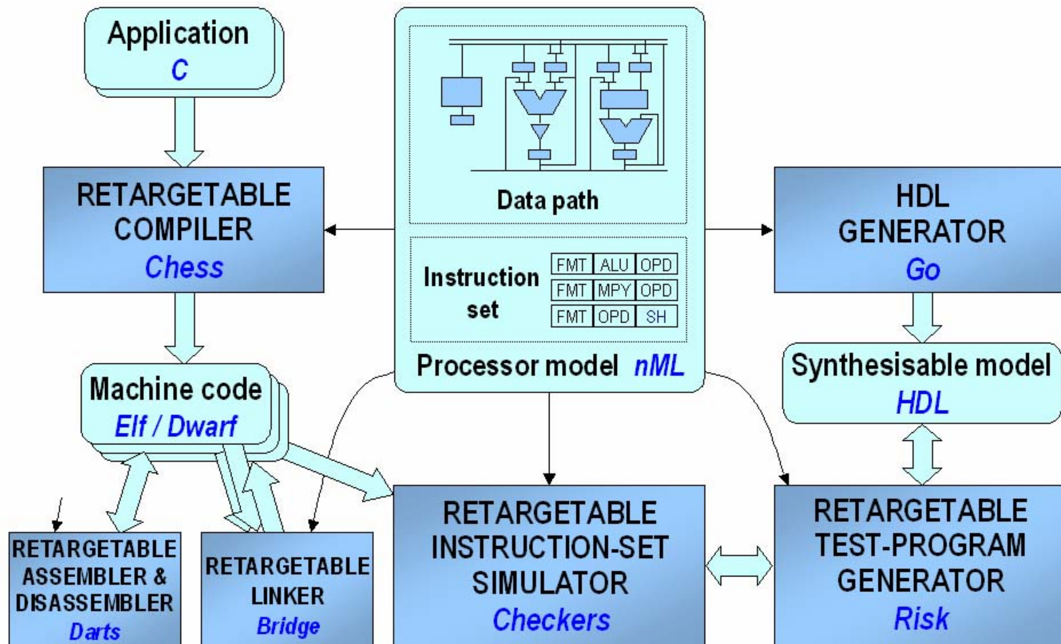

FIGURE 36.2 Outline of CHESS/CHECKERS tool-suite

The CHESS/CHECKERS retargetable tool-suite

CHESS/CHECKERS is a *retargetable tool-suite* that supports the different phases of *designing application-specific processor cores*, *developing application software* for these cores, and *verifying the correctness* of the design. An outline of the CHESS/CHECKERS tool-suite is shown in Figure 36.2. CHESS/CHECKERS consists of the following tools:

- CHESS: a *retargetable C compiler* that translates C source code into machine code for the target processor. Different from conventional compilers such as GCC [12], the CHESS compiler uses graph-based modelling and optimisation techniques [15], to deliver highly optimised code for specialised architectures exhibiting peculiarities such as complex instruction pipelines, heterogeneous register structures, specialised functional units and instruction-level parallelism. CHESS produces machine code in the Elf object file format, with source-level debug information in the Dwarf 2.0 format.
- BRIDGE: a *retargetable linker* that builds executable programs from separately compiled Elf/Dwarf object files and/or libraries.
- DARTS: a *retargetable assembler* and *disassembler* that translates assembly code into binary Elf/Dwarf object files and back. The assembly language syntax is user-defined.
- CHECKERS: a *retargetable instruction-set simulator (ISS) generator* that produces a cycle and bit accurate ISS for the target processor. The ISS can be run in a stand-alone mode or be embedded in a co-simulation environment through an application programming interface (API). CHECKERS comes with a graphical debugger that can connect both to the ISS, as well as to the available processor hardware via a JTAG or debug port for on-chip debugging. Source-level debugging is supported.

- GO: a *hardware description language (HDL) generator* that produces a synthesisable register-transfer level HDL model of the target processor core. Through APIs, users can plug in their own HDL implementations of functional units and of the memory architecture.
- RISK: a *retargetable test-program generator* that allows to quickly generate a large number of assembly-level test-programs for the target processor. These test programs can then be executed both in the ISS and in the HDL model of the processor, to check for consistency of both models.

A unique feature of the CHESS/CHECKERS tool-suite is its architectural *retargetability,* based on the *nML* processor description language. nML is a high-level language that captures a programmer's model of the target processor [7]. This is the abstraction level commonly found in a programmer's manual of a processor. Using nML, an architecture designer can quickly define the ISA of a processor. After reading the nML description, the different CHESS/CHECKERS tools are automatically targeted to the specified architecture.

```
// Declaration of storage elements and interconnections:
mem DM[1024]<num,addr>;
reg R[4]<num>;
pipe C<num>;
trn A<num>; trn B<num>;
...

// Definition of instruction set (using attributed grammar):
opn my_core (alu_inst | mac_inst | shift_inst);
...

opn alu_inst (op:opcod, x:c2u, val:c16s, y:c2u) {
  action {
    stage EX1:
      A = R[x];
      B = val;
      switch (op) {
      case add : C = add(A, B) @alu;
      case sub : C = sub(A, B) @alu;
      case and : C = and(A, B) @alu;
      case or : C = or(A, B) @alu;
      }
    stage EX2:
      R[y] = C @alu;
  }
  syntax : op " R" y ", R" x ", " val;
  image : "0"::op::x::y::val;
}
```

FIGURE 36.3 Excerpt of an nML processor description

Figure 36.3 shows a part of an nML description of a processor. Structural information about the processor is introduced by declaring its storage elements (memories, registers, pipeline registers) and its interconnections. The instruction-set is defined using an attributed grammar. The grammar breaks down the instruction set into instruction classes (e.g. `alu_inst`, `mac_inst` and `shift_inst` in Figure 36.3). The behaviour of instructions is specified in `action` attributes of the grammar rules, using a register-transfer model. In these register-transfer actions, user-defined primitive functions can be called (e.g. `add()`, `sub()`, `and()` and `or()` in Figure 36.3). To enable instruction-set simulation, the user adds bit-true simulation models for each primitive function. Likewise, to enable hardware generation, the user adds HDL models for each primitive function. Grammar rules also have `syntax` and `image` attributes, defining the assembly language syntax and the binary encoding of the instructions.

Architectural scope

CHESS/CHECKERS supports a wide range of processor architectures. Retargetability is supported within this range. The following parameters indicate the current architectural scope of the CHESS/CHECKERS tools:

- *Data types:* CHESS/CHECKERS can support the built-in data-types of the ANSI C language. In addition, users can also introduce any custom data-type. This is useful for application-specific processors, which often

contain a variety of specialised data-types. CHESS/CHECKERS allows to define application-specific data types as *C++ classes*. The defined classes can then be used in the nML processor description, to specify the data types of the processor's memories, registers and interconnections. The same class definitions can also be used in the source program for the CHESS compiler.

- *Arithmetic functions:* CHESS/CHECKERS can cope with standard arithmetic instructions found in general-purpose processors, as required for compiling ANSI C code. Users can however also define specialised arithmetic instructions in nML, and specify a mapping from the C source code to these instructions using the concept of *intrinsic function calls*.

- *Memories:* CHESS/CHECKERS supports von Neumann and Harvard architectures. The processor may have any number of data memories. Each memory may have one or multiple ports. In case of multiple memories, the user can assign static variables in the C source program to specific memories using a memory qualifier. Several *addressing modes* are supported for data memories. This includes indexed (or offset), direct and indirect addressing – optionally with post-modification of address pointers. Special addressing operations like modulo and bit-reversed addressing are supported through intrinsic function calls.

- *Instruction format:* CHESS/CHECKERS supports a wide range of instruction formats, from *orthogonal* to *highly encoded* formats. An orthogonal format consists of fixed control fields that can be set independently from each other. In an encoded format, the interpretation of the instruction bits as control fields is dependent on the instruction. "Very Long Instruction Word" (VLIW) processors have an orthogonal instruction format. The tools support variable-length instructions, as well as instruction compaction to encode small sequences of instructions in a single instruction word.

- *Registers:* CHESS/CHECKERS supports a wide variety of register structures, ranging from a *homogeneous* structure with a single, general-purpose register-file to a *heterogeneous* structure with special-purpose registers that are dedicated to store operands and results of specific instructions. CHESS/CHECKERS also supports various constraints on the utilisation of registers. For example, one may specify that the selection of multiple operand and/or result registers of an instruction is controlled by a single selection-field in the instruction word. Such *register coupling* constraints often occur in application-specific processors, to save opcode space.

- *Instruction pipeline:* CHESS/CHECKERS supports instruction pipelines of *any depth*. Different instructions do not need to have the same number of pipeline stages. CHESS/CHECKERS also supports multi-cycle instructions, multi-word instructions, and instructions with delay slots. The CHESS compiler can ensure that pipeline hazards, specified in nML, are resolved in the generated code.

- *Control flow:* CHESS/CHECKERS provides support for subroutines and interrupt service routines. Several mechanisms are available to support the concept of a software stack for storage of automatic variables. CHESS/CHECKERS also supports the concepts of hardware do-loop instructions and of mode bits that determine the behaviour of instructions.

Architectural exploration

As explained in the previous section, CHESS/CHECKERS supports a wide architectural design space. Through architectural exploration, a designer can quickly determine a power-efficient architecture for a specific application domain. As a starting point for exploration, the designer will collect the following inputs:

- *Application code* for the critical functions that need to run on the processor. There is a large range of possibilities depending on the nature of the design. At one extreme, only a small number of quite similar algorithms may need to run on an application-specific processor. At the other 'general purpose' extreme one has to consider a large number of potentially highly diverse applications and attempt to optimise from this 'sea of C' to ultimately some fully laid out VLSI design and the corresponding object code for an application. For complete DSP applications, like an MP3 decoder, the code is usually characterised with a *20/80 rule* where 80% of the cycles are spent in 20% of the code. This gives a good spread between unstructured control code and DSP loop kernels.

- *Architectural design constraints* in terms of area, timing and power budgets, as well as time scales and other project-related factors. Other more difficult constraints may be present as well, such as: backward compatibility to an existing processor architecture with a large legacy code base, scalability to allow coverage

of multiple price/performance points, and how application-specific or general-purpose to make the processor architecture. These overall specifications will limit the scope of some architectural choices, such as the number of functional units and the choice of initial ISA.

Based on the above inputs, the designer typically makes an initial proposal of an ISA, which is described in nML. Once this starting point is chosen, a more refined architectural exploration can be performed which will lead to the final design. Note that the initial architecture can be a sub-set or super-set of the finished design, although it is probably more common to start with a sub-set and add features as required. Section 36.3 discusses a number of important architectural choices for a low power processor design.

When using CHESS/CHECKERS, a designer can afford to make many iterations. Of each intermediate architecture, the performance can be evaluated by compiling critical C functions with the CHESS compiler, and simulating and profiling the resulting machine code with the CHECKERS ISS. In the first place, the ISS's profiling capabilities allow to evaluate the cycle and instruction count for the application. However, it is also possible to introduce high-level power models in the ISS and to make a comparative power analysis of different architectures (see next section). Using this feedback from the tools, the designer can compare different ISAs and optimise the architecture in nML to obtain a good match between flexibility, throughput and power characteristics. At some intermediate points, the designer may want to generate synthesisable HDL using the GO HDL generator and enter the VLSI design flow for more accurate measurements.

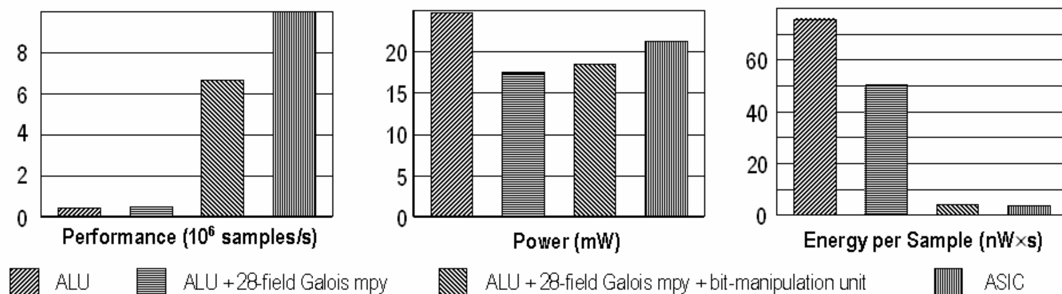

FIGURE 36.4 Performance, power and energy per sample measurements for different processor architectures for a Reed-Solomon encoding function in an ADSL modem chip

Figure 36.4 illustrates the architectural exploration capabilities of the CHESS/CHECKERS tool-suite, during the design of an application-specific processor core for Reed-Solomon encoding, for use in an ADSL modem SoC. The Reed-Solomon encoding algorithm was described in C source code. As a starting point, a simple microprocessor architecture was used, with a single 32-bit ALU. The C code was compiled on the architecture and profiled using the CHESS/CHECKERS tool-suite. The different diagrams show the computational performance, the power consumption, and the energy that is needed to process one data sample. After profiling the machine code for the single-ALU architecture, it was clear that too many cycles and program memory accesses were spent in the calculation of critical functions such as Galois-field multiplications and the bit-manipulation operations in the Reed-Solomon algorithm. These functions were initially implemented in software on the ALU. In a second design iteration, the designer extended the architecture with a dedicated functional unit capable of computing Galois-field multiplications in a single cycle. This resulted in a moderate increase of the computational performance, and in an important reduction of the power consumption. In a third iteration, the designer additionally allocated a dedicated functional unit for bit manipulation. This allowed to offload the ALU significantly, resulting in a major performance improvement and likewise a reduction of the energy needed per data sample.

For comparison, Figure 36.4 also show the characteristics of a hardwired ASIC core for Reed-Solomon encoding, developed in the same process technology. As can be seen, the ASIC core's characteristics are close to the third alternative designed with the CHESS/CHECKERS tools. This comparison illustrates that the CHESS/CHECKERS tool-suite can span a wide range of processor architectures, from a general-purpose microprocessors to programmable ASICs. The designer can perform a true architectural exploration and get rapid feedback about the quality of the intermediate results.

Power-conscious architectural design

As explained above, CHESS/CHECKERS supports an interactive methodology for architecture design. This approach is based on the assumption that automatically generated architectures can never approach the specialisation of a human designer. Rather than automating the architecture generation phase within a restricted architectural scope, CHESS/CHECKERS relies on the designer's creativity while supporting a wide scope of processor architectures. This section elaborates on how the CHESS/CHECKERS tool-suite can be used to design power-efficient processor architectures.

When defining an architecture, the designer makes the basic decisions that influence the power efficiency of the architecture. Optimisation for power is supported by the tools in the following ways:

- The CHESS compiler primarily aims at optimising the cycle count of the program, with instruction count or code size as the secondary optimisation goal. As explained in the introduction, this generally contributes to low power consumption. With a low cycle count, it is easier to fit a low V_{dd} and f_{clock}, while a low instruction count reduces the power dissipated in program memory accesses.

  Note that the length of a clock cycle is not known a priori, when modelling an architecture in nML. Typically the designer can make an estimate, but this needs to be verified by running the HDL generator GO and performing logic synthesis on the generated description.

- The architectural scope of the CHESS/CHECKERS tools and of the nML language is wide enough so that the designer can experiment with different architectural techniques for low power. These techniques are described in Section 36.3. In particular, the cycle and instruction count can be reduced by exploiting instruction-level parallelism, by bundling multiple functions in a single instruction, by exploiting special purpose registers, and by designing highly encoded instruction sets. The CHESS compiler contains various optimisation phases to make efficient use of these features.

  The tools can give early feedback about cycle count and instruction count. This is mainly obtained through the profiling capability of the ISS. Also the designer can check the effective utilisation of functional units and registers, and strip those that are not frequently used.

- It is possible to have the ISS automatically calculate an approximate power consumption figure when executing a program. Based on the nML processor model, the CHECKERS tool generates an ISS in the form of a C++ source program. The generated model is open enough so that the user can integrate instruction-level power models in the ISS.

  Obviously such power models are library and technology dependent. To construct and tune these models, a basic architecture can be specified in nML and small programs can be run that repeatedly execute specific instructions or instruction sequences, both in the ISS and in the derived HDL model using a tool like Synopsys' POWER ANALYZER. The following experimental observations may serve as guidelines when constructing power models for application-specific DSPs and programmable ASICs:

  - In case of an orthogonal instruction format (see section on CHESS/CHECKERS' architectural scope, above), power models may be constructed per orthogonal sub-class of the instruction word. By adding the power consumption of the orthogonal sub-classes a sufficiently accurate figure for the overall power consumption is obtained.

  - Within an instruction class (e.g. `alu_inst` in Figure 36.3), the specific choice of opcodes (e.g. `add`, `sub`, `and`, and `or` in Figure 36.3) has a dominant effect on the relative power consumed by the instruction. In contrast, the choice of operands or results (e.g. `R[0]` vs. `R[1]` as the source or destination register, and the exact bit-pattern of the immediate constant `val` in Figure 36.3) is much less relevant. Therefore, the choice of opcodes is an important parameter in a power model, while the choice of operands or results may be neglected more easily.

  - Power models are best defined for small *sequences of instructions*, rather than for *individual instructions* [8]. Experiments have shown that power calculations in the ISS based on power models for *pairs of instructions* can be within 30% of the actual power of the circuit (as obtained in a gate-level simulations). However, in case only power models for *individual instructions* are used, the results can differ as much as

80%. To reduce the complexity of the model, the order of the sequence may be neglected (i.e. one may assume that the power consumed by the sequence A|B is the same as for B|A).

- The HDL generator GO contains a number of optimisations that contribute to a power-efficient hardware implementation of the processor core. For example, GO can generate write-enable signals for each register-file, which allows commercially available logic synthesis tools to introduce *clock gating* to reduce power dissipation. Also, GO is able to *latch the inputs of unused functional units*, to prevent toggling of unused logic and hence save power.

36.3 Low-power processor architecture design

This section addresses some common issues covering the design of low power processors and also introduces how such a design may proceed in practice. The general area of low power processor architecture design is potentially a very broad subject and we will limit the discussion to that of embedded, low power, DSP design and specifically focus on the processor core itself. It should be pointed out that the retargetable tool-suite presented in Section 36.2 is not limited to this domain and this choice is driven by our actual design experience with a relatively general purpose audio processor called COOLFLUX DSP, which is described in Section 36.4.

General characteristics

When considering different low power processor core architectures it is worth having some sort of power aware metric with which to compare them. We have used simple metrics which consider some key aspects of a processor core. An example of such a figure of merit for a specific application could be:

$$cost = (m_{app}/m_{max}) \times P \times A$$

where m_{app} is the minimum clock frequency required for the application to achieve real-time operation, m_{max} is the maximum clock frequency the processor core is capable of, P is the power per MHz and A is the complete area, including memory. For this particular metric minimising the cost would be a goal. It is worth noting that this formula contains conflicting factors, so that for example increasing parallelism and hence A is likely to decrease m_{app} due to the ability to exploit ILP, thus in many respects an optimised processor core architecture has to find some good compromises amongst conflicting requirements. This section covers some key aspects regarding the broad architectural choices which have to be made at an early stage.

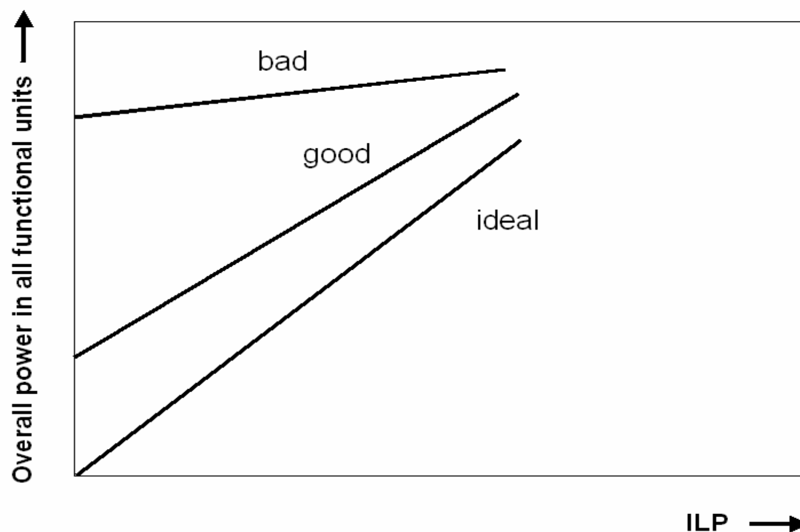

FIGURE 36.5 General relationships between power and ILP used in a processor core (assuming V_{dd} and f_{clock} are kept constant)

- *Parallelism:* Here we have to consider the type and amount of parallelism a single processor core node will support. A low power processor core design should try to approach the ideal power/parallelism characteristic as shown in Figure 36.5. This is based on minimising *control overhead.* The two main types of parallelism exploited here are ILP and data parallelism. ILP is related to the number of operations an instruction can issue to functional units and the pipeline depth of the various functional units. These two factors will define the number of operations in flight at any one time. Given the need for efficient compiler support and the ILP potential of the applications, parallelism needs to also be well balanced in the architectural exploration. We have obtained efficient compilation results for a machine which issued up to 8 RISC like operations per instruction and had 4 pipeline stages. For many DSP applications data parallelism, as supported in a single-instruction multiple-data (SIMD) machine, is a particularly efficient approach to use as the processor corecontrol overhead is further amortised over several sub-word operations. Thus a 32-bit base architecture could also define instructions that perform four 8-bit operations in parallel. This is easily supported by the retargetable tool-suite by the definition of vector data types and operations.

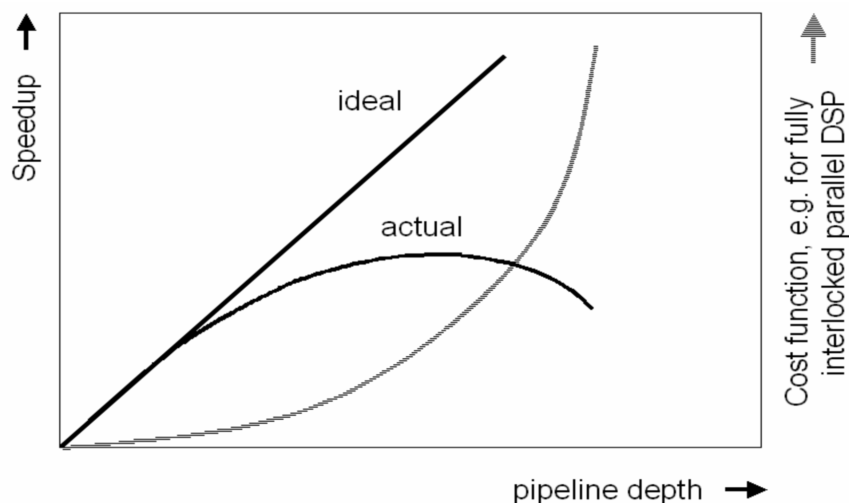

FIGURE 36.6 Speedup and cost as a function of pipeline depth

- *Pipeline structure:* This is a key aspect of low power processor core design and many factors need to balance well here. We spent a lot of architectural exploration time in this area when designing our audio DSP (see Section 36.4). We have found that relatively short simple pipelines (i.e. 3 to 5 stages) with limited interlocks and bypassing have given us good results, with minimal design and verification effort. Instruction issue policy is strictly in order. Generally pipeline depth has a quickly diminishing improvement on performance. At the same time, cost factors such as design and verification effort tend to increase rapidly, as is illustrated in Figure 36.6. As an example of many of the factors concerned, a search for the best pipeline depth would have to consider the following elements: the speed up possible for the system clock (this gives a power advantage from the larger potential voltage scaling with its quadratic power reduction), reduced cycle efficiency due to extra delay slots (or the need for extra bypasses), the potential de-glitching effects of pipeline registers, the extra power cost of the clock tree, the pipeline schedule and length of control transfers. A final comment on pipeline structure is that the CHESS compiler is good at resolving static pipeline scheduling issues, thus the onus is on providing a rather exposed pipeline and allowing a minimal hardware solution which is good for low power.

- *Register file structure:* The register file of a processor core can be homogeneous (a central structure) or may be fully heterogeneous (distributed) at the two extremes. For low power DSP applications, a distributed register file is desirable as this reduces the number of register ports and the sizes of the various files as well as leveraging locality of storage. The costs associated with a single central register file serving many functional units over many (probably bypassed) pipeline stages is usually prohibitive for a low power machine. However with distributed register files, particularly within the main data path, comes the problem of efficient register

allocation and scheduling. Our experience here is that CHESS is particularly well positioned to solve this problem and has allowed us to use this low power distributed register file feature in our designs.

- *Memory architecture:* Memory access can now be over 50% of the power budget of an embedded DSP and is likely to rise as geometry shrinks and applications grow. There are several issues to address here, covering aspects of memory hierarchy and also that of separate memory data spaces. The second issue is a standard feature of most DSP designs and will not be elaborated here. Most low power machines should exploit a data memory hierarchy of some sort which may span main storage, local tightly coupled memory and register files [16]. Here intelligent pre-fetch techniques can have significant advantages over traditional caches, particularly for applications such as video which combine statically known addressing patterns and large data objects.

  For program memory, a small cache, or at least a loop buffer, will reduce overall program memory power consumption for the 20/80 code encountered in DSP applications, as will any techniques used to reduce program memory size. Compact instruction encoding should be sought to reduce the program memory's size. In addition, we have used compression of the program memory contents coupled with the good code density already produced by the CHESS compiler to aggressively reduce the program memory footprints.

- Memory *addressing:* For DSP applications there are a well established set of extended addressing modes, usually operating as a post modify on the relevant pointer register. These greatly contribute to reducing the cycle count required for the application, and hence indirectly to low power. Several tradeoffs are possible here which balance the number of addressing modes against the complexity of the addressing units. The likely address modes include facilities for cyclic addressing as well as bit reversed addressing for FFTs and other butterfly based computations. To support a C compiler and to maintain efficient data structures that minimise use of data memory, good support of a software stack is also desirable. Thanks to the software stack concept, both the data memory size and the cycle count can be reduced, which contributes to low power. Stack support can be provided in a number of ways. We have typically used a fully indexed stack with a dedicated stack pointer.

- *ISA:* DSP designs encompass a wide variety of ISA design styles, from orthogonal to highly encoded. For low power applications we have favoured highly encoded ISA design styles. This approach minimises program memory size whilst providing enough parallel instruction classes for the ILP extracting CHESS compiler to operate efficiently. Although we provided symmetry across the various parallel views of the machine, parallel operations were only introduced for the common forms of parallelism in DSP applications, like multiply-accumulate (MAC) based inner loop kernels. This has also allowed us to have a rather asymmetric data path where most of the functionality is in the primary ALU, thus allowing a relatively compact design. Some of these issues are further expanded in the next section.

Instruction-set architecture

A low power processor has a carefully optimised ISA, which attempts to make a fine balance amongst code size, encoding, instruction decoder complexity and compiler efficiency in scheduling ILP. This is quite a difficult balance to achieve due to the requirement to maintain enough parallelism in much of the ISA to keep compilation efficient, whilst maintaining a short instruction word and hence small program memory footprint.

Because DSP code is characterised by the 20/80 rule there is a need to support rather diverse requirements and the ISA of the processor core can be thought of as having several distinct facets, in the form of instruction classes, that implement various styles of computation. So for example in our audio DSP design there is a relatively non-parallel 'RISC like' facet as well as one that codes for maximum parallelism in DSP kernels.

The design of the ISA has repercussions throughout the processor core design and as far as producing a low power instruction decoder is concerned it is important to use regular formats where possible to minimise the amount of field extraction multiplexing that is needed. Also in order to keep good pipeline timing balance, certain encoding styles can be used which allow the fast production of time critical control signals and a spread of distributed decoding function across pipeline stages.

Another issue we addressed aggressively is the potential inefficiencies due to flow control instructions, such as branches. As control flow instructions can occur up to about once every 5 instructions in general compiled C code,

it is very important to maintain high cycle efficiency here. A number of techniques can be used that minimise the power consumption of the processor core. We typically provide a good mixture of flow control instructions with and without exposed delay slots. This allows the CHESS compiler to make good code selection choices based on whether delay slots can be scheduled efficiently. We also provide zero overhead hardware looping to maintain high efficiency within inner loop constructs. Again the compiler handles this automatically and forms software based loops when the hardware loop stack is fully utilised. Another feature we use is conditional execution of instructions, which eliminate use of a branch construct and have no exposed delay slots.

As a general rule it is best to try to avoid the explicit coding of no-operation (NOP) instructions. This is partly aided by the options that have been provided on flow control instructions, but we have also added a form of NOP compression to some of our designs, which reduces our program memory size by up to 25% for typical compiled applications in our audio DSP. This saving has a significant impact on power and area: we measure an average 25.3 bits/instruction for typical compiled code for our COOLFLUX DSP, while the instruction width is 32 bits.

Micro-architecture

A low power micro-architecture should attempt to minimise control overhead whilst keeping the main data path as efficient as possible, within the bounds of the technology used. This means that pipeline interlocks and bypass networks should be used only when necessary. It will also be useful to run candidate designs right through placement and routing to ensure that cell row utilisation during chip layout can be maintained as this will reduce area and hence the capacitance and power associated with many nets whilst improving timing.

From a clocking perspective a single edge clocked synchronous design can achieve a better timing balance and clock tree efficiency than designs using both clock edges. We have always tended to carefully limit the number of overall registers in a design in order to keep control of the clock tree size and its significant power consumption (up to 40% of processor core power for semi-custom VLSI design flows). A low power design will use the standard techniques of micro-clock gating and operand isolation that are now available with many synthesis tools. The GO HDL generator in the CHESS/CHECKERS tool-suite is capable of selectively enabling these techniques in the generated HDL design. These are standard techniques and will not be further discussed here.

The pipeline design should already be designed so that good timing balance is achievable and when implementing the micro-architecture this goal must be furthered through the VLSI design flow. Usually for critical sections this will mean attention at RTL source, synthesis and back end of the flow. For the memory sub-systems this is particularly important as many critical timing paths are likely to be present here. A common solution to this problem is to use a write back buffer which schedules writes to memory only when free access slots are available. This technique allows a full clock cycle to be allowed for memory access.

The control signals from the instruction pipeline should be held until they are actually needed by a functional unit. From a power perspective it is detrimental to toggle, for example, multiplier input selection lines unless an instruction specifies a valid multiplier operation [14]. If possible the instruction decoder itself should be designed to minimise internal toggling, this may be achieved by using a distributed design, for example by instruction class.

Many of the final micro-architectural optimisations are made when the VLSI flow is exercised. This is particularly true for issues like unnecessarily toggling control logic and optimisation of critical timing paths.

Methodology

We have used a design methodology which attempts to provide as much information as possible to the processor core system architect, so that fine design tradeoffs can be made using real simulation data. So we have established a full VLSI design flow as well as having the retargetable compiler tool-suite in place at a very early stage in the project. Thus RTL design has proceeded in parallel with processor core architectural exploration and this has allowed very useful insights to be had. These activities have also tended to bond the team together through having access to a common design database.

Most architectural exploration is done within the retargetable tool-flow environment by compiling a suite of applications with CHESS and performing profiling with the CHECKERS ISS. This loop will include changes to the nML processor description, typically things like ISA, pipeline schedules and internal computational resources, but also optimisation of the application source code is explored here. Good architectural candidates are pushed through synthesis, test insertion and occasionally full layout to ensure that there are no problems with the realisation of the complete processor core. We are particularly interested in maintaining high efficiency through physical design.

This VLSI design flow is power aware and we use the accurate gate-level power simulator DIESEL, which was developed by Philips. This simulator is driven by actual simulation vectors and typically reaches accuracy within 10% of final silicon for the technologies we use. We had experimented with RTL power estimators before but have kept with the gate level simulator as we typically need early area and timing figures anyway. Once the flow is scripted it is easy to get some accurate figures in an overnight run. This design flow is also complete in that full layouts are produced and we use fully extracted (HYPEREXTRACT 3D) parasitic data when producing final area, timing and power figures.

The specification of a design has to be carefully managed. We have tended to take a minimalist approach where additional features are only added if they demonstrate a significant performance/cost benefit. Another factor in this process has been to lock the specification once confidence has been established, any changes beyond this are handled by a strict change request procedure. A factor we have particularly avoided has been ‘creeping’ specifications.

We have been fortunate in being able to build a small effective team of like minded engineers run by a single system architect who makes any final design related decisions. Of particular significance has been the synergy between engineers at Philips and Target Compiler Technologies and the close co-operation that was achieved. Perhaps a small tightly knit team is reflected in a compact power optimal processor core design?

36.4 An ultra-low power DSP for audio coding applications

This section will illustrate some of the principles outlined before by considering a few aspects of the actual design of a low power C programmable audio DSP, named COOLFLUX DSP, developed within Philips PDSL.

Background and goals

The COOLFLUX DSP audio core was designed with two main objectives in mind: first, the need for very low power consumption and secondly, the need to be efficiently programmable with the C high-level language. The extra productivity advantages of programming in C outweighed the minor power increase for efficiently providing C language support in the processor core.

The project was actually also a first introduction into the co-design of the ISA and compiler using a retargetable compiler. In many ways the question of how low power a general purpose high-level language programmable DSP could be made was being addressed, e.g. did we have to include application-specific features (in this case for MP3 coding) or programming restrictions to meet our power goals?

The motivation behind this is that power consumption continues to be a very important issue in applications. This is due to the increase in portable products on the one hand and their higher processor core needs on the other hand. The portable MP3 players that are appearing in the market are a good example. The processor core throughput requirements for these devices increase rapidly. The application algorithms become increasingly computationally demanding and also a larger number of functions needs to be executed. At the same time product lifetimes are decreasing and this puts increasing pressure on product development cycle times, particularly the software component of these projects.

These factors and a large code base of 24/56-bit fixed-point applications were the starting specification points for the COOLFLUX DSP. An MP3 decoder program was used as the main driver application because this gave a good mix of unstructured ‘control like’ code as well as some tight DSP kernels with high ILP potential.

Architecture

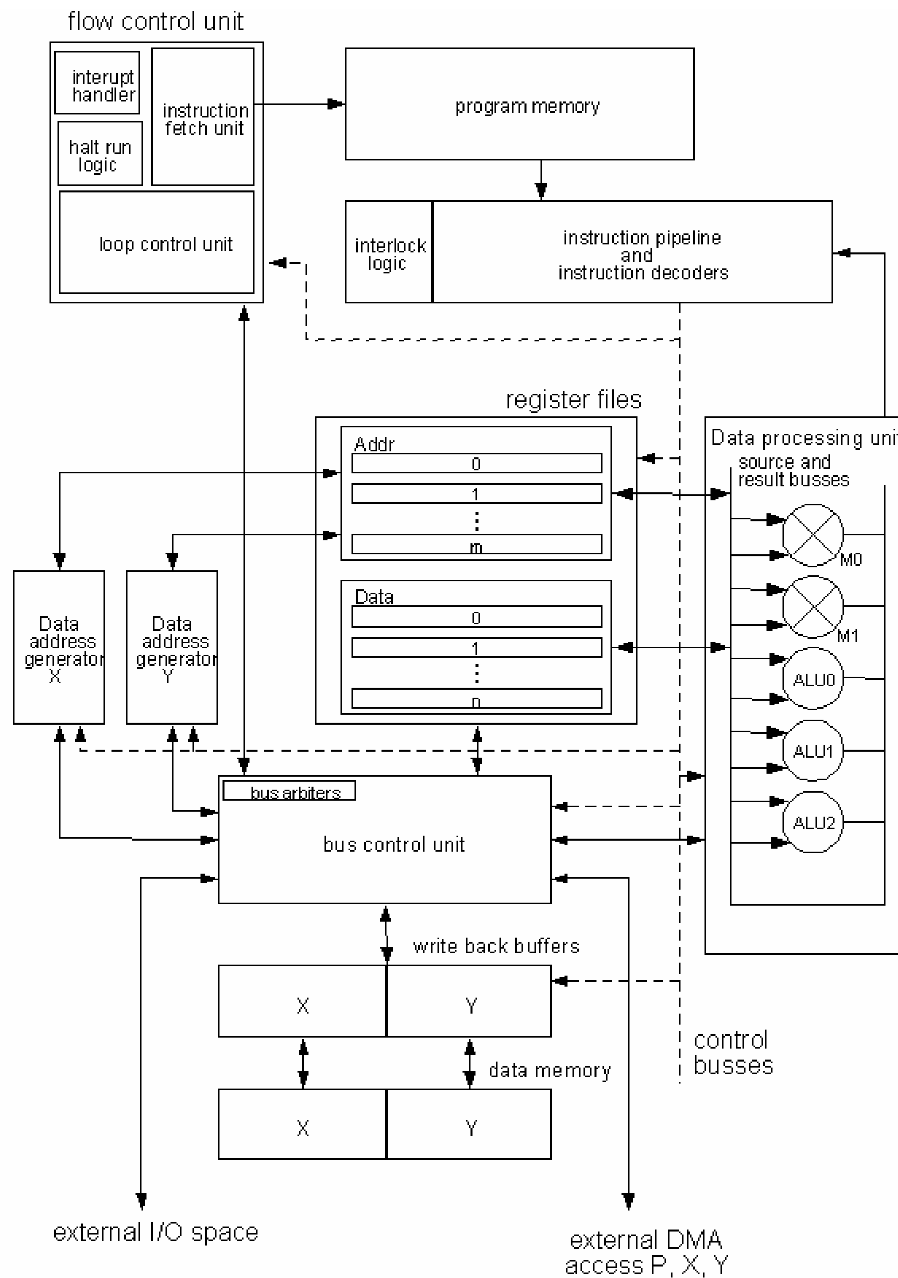

FIGURE 36.7 Block diagram of COOLFLUX DSP audio core

The COOLFLUX DSP architecture is shown in Figure 36.7. It is a dual multiply accumulate (MAC), dual Harvard machine capable of sustaining two MACs, two memory operations and two pointer updates per instruction making it highly cycle efficient for computationally intensive DSP applications when scheduled by CHESS. Due to the unique ISA design, the COOLFLUX DSP is also highly efficient at supporting unstructured 'control type' code as well making it very well balanced for complete embedded applications in which 20/80 code mix is typical.

Finally much attention has been paid to efficiently supporting the operations needed by ANSI C making the CoolFlux DSP an excellent compiler target whilst maintaining very low power consumption through the careful realisation of the underlying micro-architecture and also the entire design flow.

The data path consists of an X and a Y data processing side (Figure 36.7). The two sides are highly asymmetric. The main arithmetic components that are available are two multipliers, two full ALUs and two rounding and saturation units.

The X multiplier is coupled to a pre-adder, hence the third small ALU. This ALU (ALU0) can perform a non-saturating addition or a subtraction of two data path registers. The result of this operation is then multiplied by another data path register. The X multiplier performs all useful combinations of signed and unsigned multiplication. This allows efficient C type support as well as enabling higher precision arithmetic if needed. The Y multiplier is a lot simpler than the one on the X side. There is no pre-adder function available and only a signed/signed multiplication is possible.

The X ALU is the main ALU in the processor core. It has full 56-bit precision as well as efficient support for C `long` and `int` types. The operations supported are quite extensive and include DSP specific functions such as absolute and maximum. There is also division support and full four-quadrant division is efficiently supported from C. The condition code flags are also primarily generated by the X ALU. The Y ALU is much simpler and also has 56-bit precision. It only supports a sub-set of the X ALU operations. The main data path has four distributed register banks: the X, Y registers and the A and B accumulators.

Moves within the machine are supported by the bus control unit. This unit has been designed as a central bus switch used to exchange the contents within and between the X and Y busses as well as supporting move operations on each side. Considerable design effort was spent on this unit to ensure it was power efficient as well as meeting the stringent timing requirements needed.

Besides the main data path unit, there is also an address generation unit. Two separate units are available for the X and Y memories respectively. These units contain the address registers, the modulo protected and indexing ALUs as well as the bit reverse addressing logic.

The processor core is controlled by the flow control unit. This fetches the instruction stream, decodes it and schedules the pipelines by issuing control signals to the entire machine. Interrupts are also processed here and we have implemented a low latency, vectored interrupt system. This system guarantees interrupt response within a limited number of cycles thus easing the buffering needs of external devices. We support fully interruptible hardware loops, even if they only contain one instruction.

A very important unit is the loop control unit, which provides zero-overhead hardware looping. A maximum of four nested loops is supported, although this can be set as a parameter. This loop control unit is very efficient, both in software overhead and in real hardware parameters such as area and power.

Low power techniques

The processor core uses all of the standard techniques for low power design, mainly based around micro-clock gating, operand isolation and general unnecessary toggle reduction techniques. These will not be elaborated on. We also use a technology library, including SRAM, which allows use of aggressive voltage scaling to below 1V.

A pipeline structure was developed that allowed use of a single edge clock whilst giving memory access a full clock cycle. This will tend to maximise execution clock rate thus allowing most scope for voltage scaling which gives approximately quadratic power reduction. The pipeline structure is also finely balanced with the rest of the micro-architecture design resulting in minimal and simple interlocks, minimal bypassing and minimal length control transfers amongst pipeline segments.

The pipeline also leverages the ISA structure by utilising a small number of distributed instruction decoders as opposed to one large one, thus reducing unnecessary logic toggling. Also some instruction encoding techniques were used to attempt to minimise unneeded logic toggling [17]. Finally, instructions are included which can put the core into various sleep modes including a deep sleep where the clock can be de-activated.

The processor core uses distributed register files, local to their respective computational resources, to reduce power consumption. The ISA also includes some coupling mechanisms that attempt to reduce the need for copies

amongst register files. Another factor is that the pipeline structure of the processor core was designed so as to minimise the need for bypassing mechanisms. These advantages all add up when compared to using a single multi-ported central register file. For a machine with the parallelism that is available within our processor core this is a large advantage in both power and maximum clock frequency.

Significant area (cost) and power consumption is now evident in the memory sub-systems of processor cores. Several techniques have been used to reduce both the size and power consumption of these memories. Globally all memory spaces are made up of smaller physical segments such that only one is active in any space at any cycle.

Data memory utilisation is optimised by allowing use of efficient data structures that are supported by powerful addressing modes within the processor core. Both cyclic and bit reversed addressing are supported as well as other common DSP addressing modes. Particularly efficient stack support is provided allowing efficient linkage and local storage for functions. These techniques ensure that data memory requirements are minimised as well as execution cycles with the provision of efficient addressing modes. On the architectural front a good balance has been sought between memory sub-system costs and the ability to provide sufficient memory bandwidth for the parallel computational units.

To increase the efficiency of the program memory, innovative techniques for the code size reduction have been included, on top of the very efficient code that is generated by the CHESS C-compiler anyway. E.g. an efficient method for code compression has been included. A major trade-off in designing the ISA was the width of the instruction word against the amount of encoding used whilst still leaving enough degrees of freedom for the compiler to perform code selection well. An ISA with key areas that were orthogonal was developed. This ISA was enhanced with a form of NOP compression which used very little control logic. This has been leveraged by the use of scheduling techniques within the compiler, which favour the generation of instruction sequences, which allow maximal compression to be used. We have measured savings between 20% and 25% in code size for typical applications.

The processor core supports extensive I/O facilities allowing easy, efficient interfacing to other systems. Particularly, interrupts are implemented in a complete and robust way. The interrupt system is characterised by very low latency so that even single instruction hardware loops are fully interruptible. This allows a minimal amount of specific buffering to be implemented, thus keeping system costs low. This flexible I/O system and the ability of the retargetable environment to support intrinsic functions also means that more application-specific accelerators can be easily added to increase the system power efficiency if needed.

The processor core is implemented in VHDL at RTL level and a power aware semi-custom ASIC VLSI design flow is used. The requirements for low power are carefully considered in all stages of the VLSI design flow as well, including the coding, synthesis and layout areas.

Results

The COOLFLUX DSP design has been initially realised in Philips' 0.18µ CMOS technology. All of the figures given are for simulation data; the timing, area and power results are using fully extracted parasitic 3D data from the layout database and not wire load estimates.

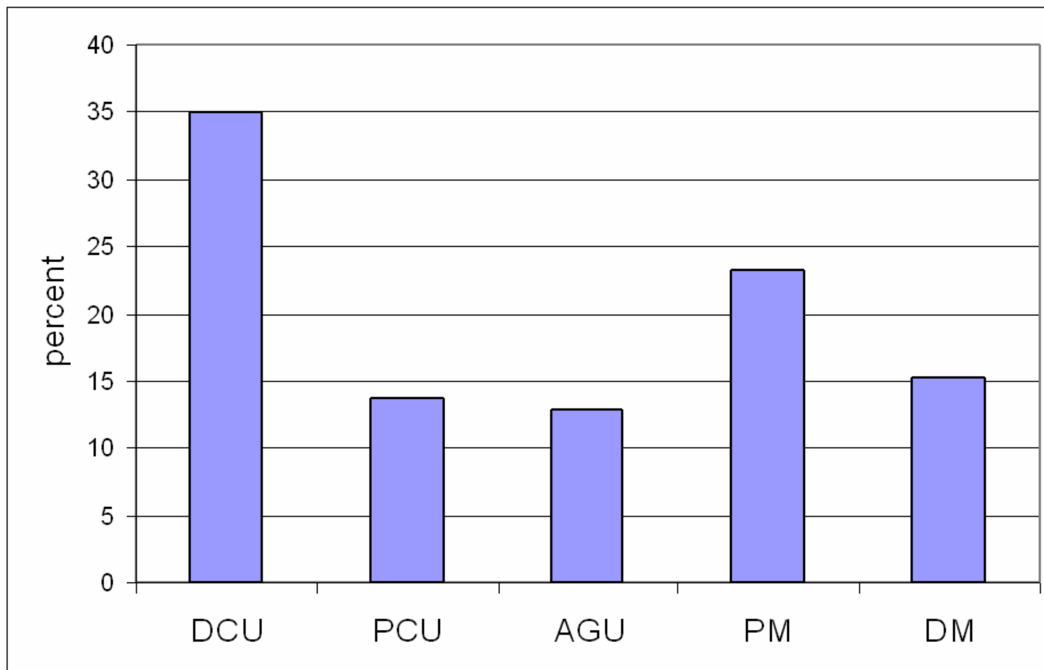

FIGURE 36.8 COOLFLUX DSP power breakdown

The power breakdown of the processor core is shown in Figure 36.8. It demonstrates a low control overhead of approximately 14% in the processor control unit (PCU) logic and power that is dominated by the data path components (DCU). The overall control overhead with program memory fetches included is some 37% of the total power consumption. The memories are represented by the program and data memories (PM, DM) and the address calculation units are in the AGU module.

Table 36.1 shows some figures obtained for the COOLFLUX DSP for an MP3 decoder application and some general core area and timing parameters.

MP3 decode computation load needed	14.9 MIPS (128Kbps, 44.1KHz stereo material)
Program memory size	4.6 Kwords × 32 bits
Data memory size	10 Kwords × 24 bits (total)
MP3 core power consumption	1 mW (Philips 0.18u standard cell low-leakage CMOS, $V_{dd} = 0.9$V, $f_{clock} = 15$ MHz)
Maximum clock frequency	135 MHz (worst-case commercial)
Core size	Approximately 45 kGate (NAND2 equivalents)

TABLE 36.1 COOLFLUX DSP performance figures for MP3 decoder

36.5 Conclusions

In this paper, a methodology has been described for the design of low power programmable processor cores in SoCs. The main idea that is explored is to customise the instruction-set architecture of the core. By making the

core application-specific, efficient architectures with minimal overhead, running at lower clock frequencies and exploiting lower supply voltages, can be obtained whilst retaining much of the flexibility and programmability of a general purpose processor.

The key infrastructure for making this methodology work in an industrial environment, is a retargetable tool-flow that allows for quick and thorough exploration of the architectural design space as well as for efficient software development starting from C source code. In this paper, the CHESS/CHECKERS tool-suite from Target Compiler Technologies has been introduced for this purpose. Also, a survey has been given of various architectural techniques that are beneficial for designing low-power processor cores. Most of the presented architectural optimisations are within the architectural solution space of the CHESS/CHECKERS tool-suite.

The effectiveness of the methodology has been demonstrated through the design, by Philips, of an ultra low power processor core for audio coding applications, called COOLFLUX DSP. As an example, this core can run MP3 decoding from compiled C code in less than 15 MIPS, which results in a processor core power consumption of about 1 mW in 0.18u standard cell technology.

Acknowledgement – The authors wish to thank their colleagues at Easics N.V. for sharing some of the data on low-power design experiments.

References

[1] "ARCtangent-A4 Core - A Technical Summary", ARC International Inc., http://www.arc.com, 2003.

[2] "CHESS/CHECKERS: a retargetable tool-suite for embedded processors", Technical white paper, Target Compiler Technologies, http://www.retarget.com, January 2002.

[3] "Xtensa Architecture and Performance", Tensilica Inc., http://www.tensilica.com, Sept. 2002.

[4] R. Camposano, J. Wilberg, "Embedded System Design", in: Design Automation for Embedded Systems, vol. 1, no. 1-2, pp. 5-50, Kluwer Academic Publishers, Jan. 1996.

[5] A. Chandrakasan, R. Brodersen, "Low Power Digital CMOS Design", Kluwer Academic Press, 1996.

[6] P. Dytrych, M. Adé, J. Coninx, J. David, P. Vandebroek, "The design of a very low power MP3 decoder accelerator", DSP Valley Annual Research and Technology Symposium, Leuven, Oct. 2002.

[7] A. Fauth, J. Van Praet, M. Freericks, "Describing instructions set processors using nML", Proc. European Design and Test Conf., pp. 503-507, March 1995.

[8] M.T-C. Lee, V. Tiwari, S. Malik, M. Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software", IEEE Tr. VLSI Systems, Vol. 5, no. 1, pp. 123-133, March 1997.

[9] J.M. Rabaey, M. Pedram, "Low power design methodologies", Kluwer Academic Publishers, 1996.

[10] J. Sato and M. Imai and T. Hakata and A. Alomary and N. Hikichi, "An Integrated Design Environment for Application Specific Integrated Processor", Proc. Int. Conf. Computer Design, pp. 414-417, Oct.1991.

[11] D. Singh, J. Rabaey, M. Pedram, F. Catthoor, S. Raigopal, N. Seghal, T. Mozdzen, "Power conscious CAD tools and methodologies: a perspective", Proceedings of the IEEE, Special Issue on Low Power Electronics, Vol. 83, No. 4, 1995.

[12] R.M. Stallman, "Gnu Compiler Collection Internals", http://gcc.gnu.org, Dec. 2002.

[13] J. Sato, A. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, M. Imai, "PEAS-I: a hardware/software codesign system for ASIP development", in: IEICE Trans. on Fundamentals, Vol. E77-A, No. 3, March 1994.

[14] C. Su, C. Tsui, A. Despain, "Low power architecture design and compilation techniques for high performance processors", Proceedings of the IEEE COMPCON, February 1994.

[15] J. Van Praet, D. Lanneer, W. Geurts, G. Goossens, "Processor modelling and code selection for retargetable compilation", ACM Tr. Design Automation of Electronic Systems, Vol. 6, no. 3, pp. 277-307, July 2001.

[16] F. Catthoor, “Custom memory management methodology: exploration of memory organisation for embedded multimedia system design”, Kluwer Academic Publishers, 1998.

[17] S. Woo, J. Yoon, J. Kim, “Low power instruction encoding techniques”, School of Computer Science and Engineering, Seoul National University, undated.