DSPxPlore – Design Space Exploration Methodology for an Embedded DSP Core

Christian Panis
Carinthian Tech Institute
Europastrasse 4
A-9524 Villach, Austria
+43 4242 90500 2124
c.panis@cti.ac.at

Ulrich Hirnschrott
Vienna University of Technology
Argentinierstrasse 8
A-1040 Vienna, Austria
+43 1 58801 58520
uli@complang.tuwien.ac.at

Gunther Laure
Infineon Technologies Austria
Siemensstrasse 2
A-9524 Villach, Austria
+43 4242 305 0
gunnar2@sbox.tu-graz.ac.at

Wolfgang Lazian
Infineon Technologies Austria
Siemensstrasse 2
A-9524 Villach, Austria
+43 4242 305 0
lazian@sbox.tu-graz.ac.at

Jari Nurmi
Tampere University of Technology
P.O.Box 553
FIN-33101 Tampere
+358 3 3115 3884
jari.nurmi@tut.fi

ABSTRACT
High mask and production costs for the newest CMOS silicon technologies increase the pressure to develop hardware platforms useable for different applications or variants of the same application. To provide flexibility for these platforms the need on software programmable embedded processors is increasing. To close the gap concerning consumed silicon area and power dissipation between optimized hardware implementations and software based solutions, it is necessary to adapt the subsystem of the embedded processor to application specific requirements. DSPxPlore can be used to explore the design space of RISC based embedded core architectures. At an early stage of the project the main architectural requirements of the application code can be identified in order to meet the area and power dissipation requirements. During the development process DSPxPlore supports fine-tuning of the subsystem architecture (e.g. modifications of the binary coding of instructions). DSPxPlore is part of a development project for a configurable DSP core.

Keywords
DSPxPlore, Design Space Exploration, embedded DSP

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'04, March 14-17, 2004, Nicosia, Cyprus
Copyright 2004 ACM 1-58113-812-1/03/04...$5.00

1. INTRODUCTION
Decreasing feature size and increasing system complexity enables to map complex system functions onto one die (SoC, System-on-Chip) or into one package (SiP, System in a Package). High mask and production costs for the newest silicon technologies increases the need of platform solutions, enabling to use the same silicon for several applications. Providing flexibility to the platform solutions allowing to realize several applications with the same silicon, embedded software programmable cores can be used. Therefore the importance of embedded processors like microcontrollers, protocol processors and digital signal processors (DSP) is increasing.

One aspect of using dedicated hardware implementations instead of software based solutions is the degree of efficiency in terms of consumed silicon area and power dissipation. To overcome the efficiency drawbacks of software based solutions without loosing the advantage of flexible platform architectures, providers of embedded core architectures provides the possibility to modify their core architectures to application specific requirements [1][2].

Making use of the additional degree of freedom the requirements of the application have to be understood. Quite often the core decisions are done by the most experienced engineers focusing on the aspects „what is already available?" and „what has been already proven in silicon?" to reduce the risk. Different requirements of the applications lead to not optimal solutions concerning consumed silicon area and power consumption by using one core subsystem. In the price-critical consumer IC market this can be crucial for the own market position and the revenues.

This paper introduces DSPxPlore, a design space exploration methodology for an embedded configurable DSP processor. DSPxPlore can be used to understand the requirements of the application code on the processor architectures in an early stage of the project. During the development project DSPxPlore can be

used to fine-tune the chosen architecture. The first part introduces the RISC based DSP core architecture used as basis for DSPxPlore. The introduced methodology is not limited to this architecture. The second part is used to discuss the design space of RISC based DSP core architectures. The influence of configuration parameters concerning consumed silicon area and power dissipation of the core subsystem is illustrated. The third part introduces the DSPxPlore methodology. DSPxPlore is based on an optimizing C-Compiler (about 5 to 10% overhead compared with manual assembly coding) and a cycle-true Instruction Set Simulator (ISS), based on a configurable component framework. A XML-based configuration file contains a description of the chosen core architecture and is used to configure the tool chain and to automatically update the documentation for the DSP core. The last section covers some exploration examples and gives an outlook for future work.

2. ARCHITECTURAL INTRODUCTION

This section is used to give a short introduction of the DSP architecture DSPxPlore has been developed for. The main architectural features and the instruction set have been defined under consideration of low silicon area and power dissipation of the DSP subsystem and to enable the development of an optimizing C-Compiler (about 5-10% overhead compared with manual assembler coding). An example architecture has been chosen for this paper and will be shortly introduced in this section.

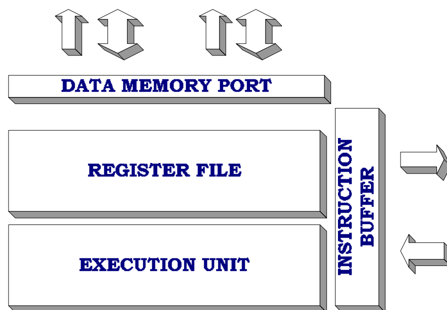

Figure 1: Core Overview

The proposed DSP core features a modified Dual-Harvard load-store architecture (an overview is illustrated in Figure 1) [3]. An independent data bus connects the program memory with the DSP core, an instruction buffer is used to execute loop constructs power efficient [4]. Data and program memory are featuring different address spaces [5]. The bit width of the ports in Figure 1 is scaleable, which allows application specific adaptation of memory bandwidth.

The core is featuring a RISC like 3-phase pipeline, instruction *fetch*, *decode* and *execute*. The three phases can be split over several clock cycles. The example architecture illustrated in Figure 2 is using five clock cycles for the three pipeline phases.

The *instruction fetch* phase is split over a fetch and an align clock cycle, the decode stage takes one clock cycle, the execution phase is split over two clock cycles (EX1, EX2). Splitting of a pipeline phase over several clock cycles enables to reach higher clock frequencies. But additional pipeline stages in the fetch phase increases the number of branch delays, additional clock cycles for the execution phase leads to increased load-in-use and define-in-use dependencies [6]. Therefore deeper pipeline structures can lead to a decreased overall system performance due to data and control dependencies in the application code.

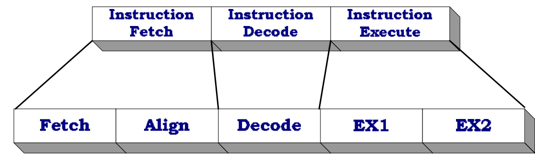

Figure 2: Pipeline

The instructions are divided into three operation classes: load/store instructions, used to transfer data between the data memory and the register file, arithmetic/logic instructions performing calculations on register values, and branch instructions influencing the program flow. Each instruction consists of one or two instruction words. The size of the native instruction word for the example architecture is 20 bit; the optional second word is used for long immediate values and offsets (parallel word as in Figure 3).

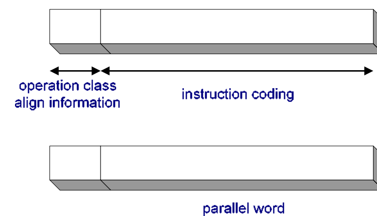

Figure 3: instruction coding

All arithmetic instructions support 3 operands, which prevents data copy functions between different registers of the register file. All features of the DSP core are coded inside the instruction set; no mode bits are used to increase code density. The drawbacks of using mode bits are limitations during instruction scheduling when moving instructions between different mode sections [7]. As illustrated in Figure 3 the first three bits of the instruction words are used for assigning the operation class and the alignment information.

Figure 4: parallelism

The number of possible parallel executed instructions is scaleable. The example architecture enables the execution of up to five instructions in parallel. It is possible to execute two load/store, two arithmetic and one branch instruction in parallel (illustrated in Figure 4). The chosen programming model is VLIW (Very long instruction Word), which implies static scheduling (data and control dependencies are analyzed and resolved in software). The drawback of traditional VLIW architectures featuring low code density is solved by xLIW (a scalable long instruction word) [8]. xLIW is based on VLES (Variable long execution set) and additionally supports a decreased program memory port. For this purpose also the already mentioned instruction buffer is used [9].

The example architecture supports two busses to data memory. Therefore two independent AGUs (address generation unit) are available. Each of the AGU can make use of each of the address registers (no banked address register). If two parallel generated addresses access the same physical memory block, the core hardware automatically detects the hazard and serializes the memory operations. Data memory operations exceeding the physical size of the memory port are realized as consecutive memory operations at the same data bus.

All common DSP address modes like memory direct, register direct and register indirect addressing are supported. The auto in-/decrement address operation supports pre- and post address calculation and an efficient stack frame addressing. The size of the modulo buffer is programmable; the start address of the buffer has to be aligned. This is a compromise between hardware effort and supported features.

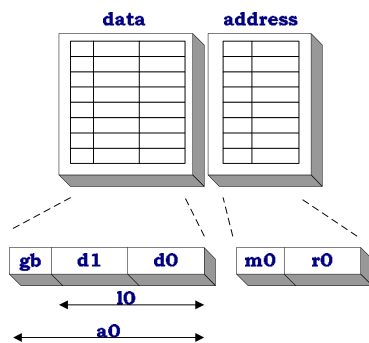

Figure 5: register files

Load-store architecture implies that all operands for the arithmetic instructions reside in registers. Therefore the register file has an important role. The structure of the register file and the size and the number of registers is configurable; for the example architecture a register file as in Figure 5 is used. It is split into two parts, a data register file, and an address register file.

a0	gb	d1	d0	l0
a1	gb	d3	d2	l1
a2	gb	d5	d4	l2
a3	gb	d7	d6	l3
a4	gb	d9	d8	l4
a5	gb	d11	d10	l5
a6	gb	d13	d12	l6
a7	gb	d15	d14	l7

Figure 6: data register file

The data register file as in Figure 6 consists of 8 accumulators, 8 long registers or 16 data registers. Two consecutive data registers can be addressed as a long register. A long register including guard bits (for higher precision calculation) can be addressed as accumulator. The size of the operands can be modified application specific. The registers inside the register file are orthogonal, which means that none of them is assigned to a certain instruction. The drawback of an orthogonal register file is the crossbar to enable mapping of the read and write ports to each of the registers.

3. DESIGN SPACE FOR RISC BASED DSP ARCHITECTURES

This section is used to introduce the available design space for RISC based DSP subsystems with influence on area consumption, power dissipation and overall system performance. The example architecture is used to illustrate the main architectural features. The influence of some of the parameters is illustrated by first exploration results.

3.1 Register File

The register file in load-store architectures has a central role. All arithmetic instructions are fetching their operands from the register file and store their results into the register file. Therefore the number of supported registers of the register file influences the performance parameters of the DSP subsystem.

Supporting less register reduces the necessary core area but can lead to additional spill code. Spill code is added if no registers are available to store a result. In this case register file content has to be stored to data memory to free register resources. If any of the spilled data is needed again, it has to be reloaded from memory. The added spill code increases the demand on program memory and therefore decreases the code density of the application code. Further it increases execution time and therefore decreases system performance.

Supporting a larger register file with more entries increases the core area and again has influence on code density. More entries require more coding space to address the register entries – especially considering the orthogonal requirement to enable the development of an optimizing C-Compiler banking registers or supporting registers for special functions is not possible.

a0	gb	d1	d0	l0
a1	gb	d3	d2	l1
a2	gb	d5	d4	l2
a3	gb	d7	d6	l3
a4	gb	d9	d8	l4
a5	gb	d11	d10	l5
a6	gb	d13	d12	l6
a7	gb	d15	d14	l7

a0	d3	d2	d1	d0	l0
a1	d7	d6	d5	d4	l1
a2	gb		d9	d8	l2
a3	gb		d11	d10	l3
a4	gb			d12	l4
a5	gb			d13	l5
a6	gb			d14	l6
a7	gb			d15	l7

Figure 7: register file (64-bit accu)

It is possible to change the structure of the register file. Figure 7 is used to illustrate an example for a 64-bit data register file (e.g. used for a 64-bit/quad MAC architecture). The register file on the left side of Figure 7 has a similar structure as the register file in Figure 5; instead of using guard bits the accumulator supports 64-bit. The number of addressable data registers has not been doubled; the necessary coding space for the additional data registers has influence on the code density. If an application code requires the use of more than 16 data registers to reduce the spill code a register file like in Figure 7 can support up to 32 data register. The same register file on the right side of Figure 7 has a different structure. Eight of the data registers are mapped onto the first two accumulator registers, the remaining eight are split onto the next six accumulator registers.

3.2 Data paths

Increasing the number of data paths and parallel executed instructions increase the maximum possible calculation power of core architectures. Providing the possibility to execute several instructions in parallel requires the availability of operands. Therefore a balanced relation between memory bandwidth, number of independent load/store instructions and the number of arithmetic data paths characterize the possible performance of core architectures.

Table 1: ILP

Tjaden and Flynn	31 library programs			1,2-3,2	**1,9**
Kuck et.al.	20 Fortran programs			1,2-17	**4**
Riesemann, Foster	7 Fortran/ assembler	1,2-3	**1,8**	1,4-1,6	**1,6**
Jouppy	8 modulo2 programs	1,6-2,2	**1,9**	2,4-3,3	**2,8**
Lam, Wilson	6SPECmarks+4others	1,5-2,8	**2,1**	2-293	

Additional influence comes from the application program executed on the core architecture. Control and data dependencies can lead to a low usage of the provided core resources. In Table 1 some examples for ILP (instruction level parallelism) can be found. The benchmark examples are based on general purpose code (column 3,4) as also scientific code (column 5,6). The average ILP in these examples is about two to three instructions.

Traditional algorithms executed on DSP cores are filtering operations. Filter algorithms are characterized by an inner loop, where a significant amount of execution time is spent. These inner loops (considering software pipelining) can make efficient use of parallel provided resources. Therefore the ILP for this kind of algorithms is higher than that for general purpose code. The MAC (multiply and accumulate) instruction is typical used for e.g. FIR filter algorithms. Therefore the performance of DSP cores is measured in the number of provided MAC instructions per second and in the number of clock cycles needed for execution (considering the define-in-use dependency).

Changing the number and kind of data paths has influence on the core hardware. If the changes in the data path structure have influence on the instruction set (by adding or removing instructions) the code density is influenced. Changes of the data path structure have influence on the execution bundle. Therefore after changing the data path structure, it is necessary to verify if the average relation between the size of the fetch and execution bundle is still balanced and that the memory bandwidth still fits to the data path structure.

3.3 Memory bandwidth

The memory bandwidth is closely related to the *data path* parameter. Providing a lot of parallelism with insufficient memory bandwidth is resulting in bad usage of available core resources. The size of the memory ports has influence on power dissipation and consumed silicon area of a DSP subsystem.

Data memory port: Today most of the commercial available DSP cores are supporting two independent data memory busses. Supporting additional busses increases the flexibility of data transfer and several algorithms e.g. FFT algorithms can make use of it. But the drawback of more memory ports is the hardware effort for additional AGUs (Address Generation Unit) and the wiring effort to the memory sub system.

Program memory port: For most of the commercial available DSP cores, the size of the program memory port is equal to the maximum number of parallel executed instructions. Similar as for the data memory port, the wiring is influencing area and power consumption. One possibility to decouple the size of the program memory port with the provided parallelism of the execution unit is the usage of an instruction buffer, as mentioned in section 2.

3.4 Instruction size/encoding

The instruction set describes the functionality supported by the core architecture. The mapping of the instruction set to binary instruction words has significant influence on the area consumption of the core sub system, because the memory used to store the instructions is dominating the area consumption.

In Figure 8 an example for different mappings of the same instruction set to two different instruction layouts is illustrated. In the right example, the instruction set has been mapped using instructions with a native size of 16-bit, using 32 bit for the remaining instructions, which cannot be mapped to the native instructions set like three operand arithmetic instructions. For the example of the left column a native instruction word size of 20 bit is used, allowing to map all instructions into the native instruction word size. The second word is only used for long immediate values and offsets. Considering a certain algorithm (e.g. some control code as in Figure 8) the smaller native instruction word size is providing a lower overall code effort. This can be different for another code example, which e.g. requires three operand instructions, coded more efficient in the longer native instruction word.

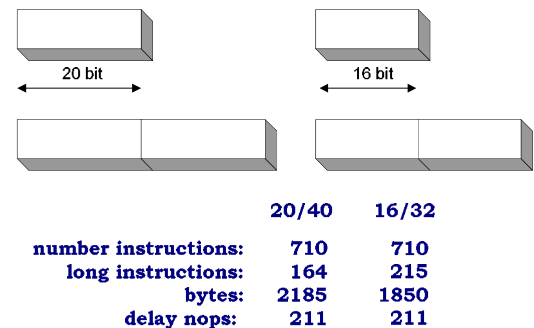

Figure 8: example for instruction set mapping

The binary coding is influencing the switching activity at the program memory port and therefore the mapping of the instruction set to a certain binary coding has influence on the power dissipation of the DSP subsystem. More often used instructions can be coded more efficiently resulting in an increased code density. Also reordering of instructions inside the

same execution bundle can be performed in order to decrease power dissipation at the program memory bus [10][11][12].

3.5 Instruction buffer size

The instruction buffer mentioned in section 2 is not available in each core, but shall be mentioned for the core architecture introduced in section 2. For this core the instruction buffer is used to compensate the memory bandwidth mismatch between fetch and execution bundle and also to execute loop constructs power efficient by reducing the number of memory accesses. To make use of this feature, the size of the instruction buffer has to be scalable to adapt the instruction buffer to application code specific requirements. Power efficient loop handling can only be achieved, if the loop body fits into the buffer. Therefore the chosen size of the instruction buffer has influence onto the power dissipation of the core subsystem. On the other side providing a buffer with many entries leads to a significant increase on core area.

3.6 Pipeline stages

Increasing the number of pipeline stages allows increasing the reachable core frequency. Higher core frequencies lead to increased power dissipation due to the need of a higher supply voltage and an increased switching activity [13].

Increasing the number of pipeline stages also increases the core complexity, because additional hardware circuits like bypass are getting necessary to reduce the increased dependency between instructions of different pipeline stages [14][15][16].

Increasing the number of pipeline stages can even lead to a decrease of system performance due to control and data dependencies. Therefore a balanced pipeline structure considering dependencies of the application code and physical aspects of technology are important to obtain a good cost ratio between area consumption, power dissipation and system performance. Classifying core subsystems by MIPS, MOPs or MMACs or any other similar parameter is misleading: for an embedded core the core performance has to be classified, how efficient an application code can make use of the available core resources.

Increasing the number of pipeline stages for the fetch phase of the pipeline relaxes the timing at the program memory but increases the number of branch delays. Additional hardware circuits have to be introduced to compensate the unused branch delays [17][18]. Predicated execution can help to reduce the number of branch delays by reducing the number of conditional branch instructions [19].

Adding pipeline stages to speed up the execution phase and to relax the timing at the data memory ports leads to an increased define-in-use and load-in-use dependency. Bypass logic can be used to reduce the dependencies but again by increasing core complexity.

3.7 Summary

This section has been used to briefly introduce the architectural features of RISC based core architectures (with focus on DSP cores) which are significant influencing the area consumption and power dissipation of the core subsystem. None of these parameters can be considered isolated; changing one of them influences several others. There is not a single shot solution satisfying the requirements of all applications efficient. The application code executed on a core architecture make a certain core configuration efficient. To understand the requirements of an application code, the following section is used to introduce a design space exploration methodology for RISC based core subsystems.

4. EXPLORATION METHODOLOGY

The DSP core architecture introduced in section 2 allows adapting the architectural features introduced in section 3. Providing a configurable DSP core architecture to meet application specific requirements enables to reduce area consumption and power dissipation. To find the optimal core architecture (optimal for one application) it is important to understand the application specific requirements.

For this purpose DSPxPlore is introduced. DSPxPlore can be used to analyze the influence of certain core subsystem configurations on the system parameter core area, power dissipation and overall system performance. During the product development process DSPxPlore supports a fine tuning of the core subsystem. The exploration methodology is based on an optimizing C-Compiler and a configurable ISS (instruction set simulator).

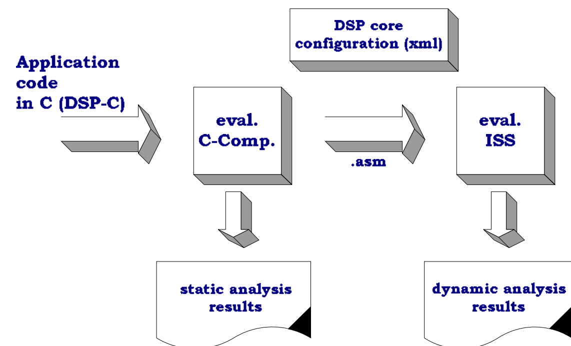

Figure 9: DSPxPlore Overview

In Figure 9 an overview of the exploration methodology is illustrated. An optimizing C- compiler is used to generate static analysis results. A cycle true Instruction Set Simulator (ISS) is used for evaluation of dynamic results. Both results together can be used to analyze the application specific requirements to the core subsystem. The chosen core configuration is located in an XML-based configuration file, which is used by both tools.

4.1 Static analysis

To obtain reasonable accurate results for static analysis it is necessary to use a C-Compiler that generates near-optimal assembly code (compared to manually optimized code). If the quality of the C-Compiler is poor, the generated results can be misleading and architectural decisions can lead to a suboptimal solution. The C-Compiler for the core architecture introduced in section 2 provides an accuracy of about 5-10% overhead compared with manual coding. Some of the generated static evaluation results are

4.1.1 code size

The memory of a DSP subsystem is dominating the silicon area consumption. Therefore a high code density reduces area consumption. An example for the parameter *code size* is illustrated in Figure 10. The number of instructions necessary to port the application code to the chosen core architecture is counted and the required long instructions are summed up. The chosen instruction word length is normalized to bytes to have a comparable value. The example architecture is using a 20-bit native instruction word and therefore the number of counted instructions have to be multiplied by 2,5 to get the code effort in bytes. Instructions with long words are counting double.

number instructions:	827
long instructions:	70
bytes:	2242,5
delay nops:	42

Figure 10: code size analysis

4.1.2 parallelism

The analysis result *parallelism* gives an indication of the usage of the provided core resources. Data and control dependencies in the application code restrict the execution of parallel instructions and leads to a poor use of the available processor resources. The example in Figure 11 illustrates the dependency problem (on the left side a summary, on the right side more in detail).

bundles with 1 instruction(s):	255
bundles with 2 instruction(s):	105
bundles with 3 instruction(s):	52
bundles with 4 instruction(s):	5
bundles with 5 instruction(s):	0

Nop (incl. delay fill nops)	80	19.2 %
MemX.....................	82	19.7 %
...........MemY................	8	1.9 %
MemX.MemY................	10	2.4 %
......................ALU1...........	48	11.5 %
MemX............ALU1...........	22	5.3 %
......................ALU1.ALU2.......	15	3.6 %
MemX............ALU1.ALU2.......	4	1.0 %
MemX.MemY.ALU1.ALU2.......	2	0.5 %
....BrUnit	37	8.9 %
MemX..............................BrUnit	26	6.2 %
MemX.MemY...................BrUnit	1	0.2 %
......................ALU1..........BrUnit	32	7.7 %
MemX............ALU1..........BrUnit	18	4.3 %
......................ALU1.ALU2.BrUnit	29	6.9 %
MemX............ALU1.ALU2.BrUnit	2	0.5 %
...........MemY.ALU1.ALU2.BrUnit	1	0.2 %

Figure 11: bundle assignment

Only a few execution bundles can make use of the parallel units (the example architecture provides to execute five instructions in parallel). DSP architectures like the C62x from Texas Instruments will not have a higher usage of the core resources, even if its relative performance (calculated as number of possible parallel instructions multiplied with the reachable clock frequency) provides higher numbers [20].

4.1.3 instruction histogram

The *instruction histogram* analysis result provides a list of the used instructions and their static occurrence inside the application code. This result can be used to optimize the instruction set during fine tuning of the core subsystem (e.g. optimized coding of frequently used instructions).

4.1.4 immediate values

The size of the immediate values can be analyzed already during the static process. This gives an indication about the needed coding space inside the instruction set. These results (similar as in Figure 8) can be used to choose the optimal size for the native instruction word.

4.1.5 delay slots

The number of delay slots can significantly influence the overall system performance of a DSP subsystem. Delay slots are caused by branch instructions or function calls. Increasing the number of pipeline cycles during the fetch phase results in more delay slots. Some of the delay slots can be filled with useful instructions, the others are lost cycles.

4.2 Dynamic analysis

To weight the static analysis results, dynamic analysis are necessary. A cycle true instruction set simulator (ISS) is used to obtain the results. xSIM, the ISS used for the core introduced in section 2 is based on a configurable component framework. An XML based configuration file is used to define the chosen core configuration. Some of the dynamic results are

4.2.1 program memory fetch

The fetch of instructions from program memory significantly influences the power dissipation of the DSP subsystem. Therefore reducing the switching at the program memory port can be used to reduce power dissipation. The number of fetch cycles from program memory is analyzed, the fetch frequency of the different fetch bundles is counted and the alignment analysis for loop and branch constructs is considered.

4.2.2 unused program memory

The DSP core introduced in section 2 features an instruction buffer to overcome the bandwidth mismatch between fetch and maximum execution bundle size and to execute loop constructs power efficient. Especially during breaks in the program flow already fetched program data are not executed. This parameter is used to analyze which code sections have been fetched but not executed and can be used to reduce the switching at the program memory port.

4.2.3 execution count per bundle

Counting the execution frequency of each execution bundle can be used to identify *hot spots* and to optimize the HW/SW

partitioning (e.g. deciding which parts can be more efficient implemented in hardware). Together with the static *parallelism* analysis the provided parallelism can be classified and the number of data paths adjusted to the requirements of the application code. The results can be visualized by *x*SIM, which easies the interpretation of the results.

4.2.4 execution count per instruction

The list of used instructions generated during static analysis is extended by the execution count of instructions. With this information the instruction set and the binary coding can be optimized, increasing code density and decreasing switching activity at the program memory port. Frequently executed instructions can be coded more efficient, not used instructions even removed.

4.2.5 stall cycles

During execution of application code stall cycles can take place. During the stall cycles the core is not contributing to the system performance of the DSP subsystem. This can be caused e.g. by simultaneous memory access to the same physical memory block or by missing program data, due to an empty instruction buffer (e.g. at not aligned branch targets). This information can be used to identify possible reasons and to modify the core architecture and the application code to prevent useless stall cycles.

5. RESULTS

This section is used to illustrate some of the results using DSPxPlore. The set of benchmark examples consists of traditional DSP functions like FFT and code examples of the area of cryptology but also of control code examples e.g. framing algorithms.

For the results in Table 2 the size of the register file has been modified. The number in the first column is equal to the number of supported accumulator register.

Table 2: register size evaluation

#regs	bundles	inst.	delay nops	code size
4	24008	35284	2473	47263
8	17041	26544	2722	31810
16	14507	23046	2826	26497

Increasing the number of registers relaxes the register pressure for register allocation, resulting in a decreased code effort. In the second row the number of available registers has been increased from four to eight leading to a reduced code effort of about 50%. Doubling the register file again from eight to 16 accumulator registers increases the code density only by about additional 17%. The algorithm examples cannot make use of the additional registers. The results in Table 2 do not include the influence on the coding space due to the increased number of registers. The coding used for the comparison supports the medium sized register file; considering also the difference in the coding space will reduce the absolute distance between the result values.

Table 3: parallelism analysis

model	bundles	inst.	delay nops	code size
0 1M-1A-1B	21070	27962	2694	33441
1 2M-1A-1B	20783	28022	2714	33441
2 1M-2A-1B	18194	27728	2862	33196
3 2M-2A-1B	17872	27871	2866	33421
4 2M-3A-1B	17347	27890	2958	33558

The first column in Table 3 indicates the number of parallel executed instructions: The M for load/store instructions, A for arithmetic/logic instructions and B for branch instructions. As expected adding more units in parallel decreases the number of necessary execution bundles (column three in Table 3). Data and control dependencies reduce the effect of further added units. One remark concerning the increase of the number of branch delay NOPs: During compilation, the C-Compiler has been configured to execute the instructions as early as possible. Providing more parallelism leads to shorter branch distances and therefore fewer instructions are available to fill delay slots. The number of necessary NOP instructions for filling delay slot is increasing.

Table 4: model 0, branch delay slots

branch delay	bundles	inst.	delay nops	code size
2	21051	27943	2694	33422
3	22544	29438	2686	34868
4	24118	31004	2664	36432

For the results in Table 4 (model 0) and Table 5 (model 4) the same core models as for the results of Table 3 are used. The parameter on the left side is the number of branch delays. Additional branch delays caused by further clock cycles used for the fetch phase of the pipeline e.g. to relax the timing at the program memory port, leads to an increased number of instructions and therefore to a decreased code density (e.g. due to additional NOP instructions to fill delay slots).

Table 5: model 4, branch delay slots

branch delay	bundles	inst.	delay nops	code size
2	17324	27867	2959	33537
3	18897	29451	2966	35121
4	20524	31055	2947	36725

Comparing the results for model 0 and model 4, as expected the number of necessary execution bundles is decreasing. The configuration for the results of Table 4 supports to execute only

three instructions in parallel, the configuration used in Table 5 up to six instructions in parallel.

6. OUTLOOK

DSPxPlore is used to understand the requirements of the application code on the core architecture, to identify hot spots and to optimize the HW/SW partitioning. DSPxPlore is still an expert system. For interpretation of the generated results and the related modifications of the core architecture a deep understanding of the core architecture, the configuration parameter and the influence of the chosen configuration onto silicon area and power consumption is necessary. In the next development phase it will be possible to get easier understandable feedback from DSPxPlore. This enables the system architect optimizing his core subsystem for application specific requirements and to gets hints for further optimizations.

7. SUMMARY

DSPxPlore is a design space exploration methodology for RISC based embedded cores. Analyzing application specific requirements in an early stage of the project enables to modify the core subsystem and therefore to obtain low silicon area consumption and low power dissipation. During the design process DSPxPlore can be used for fine tuning of the core subsystem e.g. optimization of the binary coding to reduce power dissipation. With the application specific optimized core subsystems it is possible to reduce the gap between a dedicated hardware implementation and a core based solution providing the flexibility of software programmability. DSPxPlore is part of a project for a configurable DSP core.

8. ACKNOWLEDGMENTS

The work has been supported by the Christian Doppler Lab for C*ompilation Techniques for Embedded Processors* and by the EC with the project SOC-Mobinet (IST-2000-30094).

9. REFERENCES

[1] www.arc.com

[2] www.tensilica.com

[3] Hennessy, J. L., Patterson, D. A., Computer Architecture. A Quantitative Approach, Morgan Kaufmann Publishers, San Mateo CA, 1996.

[4] Panis, C., Bramberger, M., Grünbacher, H., and Nurmi, J., A Scaleable Instruction Buffer for a Configurable DSP Core, ESSCIRC 2003, Lissabon, Portugal, 2003.

[5] Lapsley, P., Bier, J., Shoham, A., and Lee, E.A., DSP Processor Fundamentals, Architectures and Features, IEEE Press, New York, 1997.

[6] Sima, D., Fountain, T., and Kacsuk, P., Advanced Computer Architectures: A Design Space Approach, Addison Wesley Publishing Company, Harlow, 1997.

[7] Morgan, R., Building an Optimizing Compiler, Digital Press, 1998.

[8] Panis, C., Leitner, R., Grünbacher, H., and Nurmi, J., xLIW – a Scaleable Long Instruction Word, ISCAS 2003, Bangkok, Thailand, 2003.

[9] Panis, C., Leitner, R., Grünbacher, H., and Nurmi, J., Align Unit for a Configurable DSP Core, CSS 2003, Cancun, Mexico, 2003.

[10] Hirnschrott, U., and Krall, A., VLIW Operation Refinement for Reducing Energy Consumption, Proceedings of International Symposium on System-on-Chip '03, Tampere, 2003.

[11] Shin, D., Kim, J., and Chang, N., An Operation Rearrangement Technique for Power Optimization in (VLIW) Instruction Fetch, ACM, Munich, 2001.

[12] Choi, K., and Chatterjee, A., Efficient Instruction-Level Optimization Methodology for Low-Power Embedded Systems, Proceedings of International Symposium on System Synthesis ISSS 01, 2001.

[13] Chandrakasan, A., Sheng, S., and Brodersen, R., Low-Power (CMOS) Digital Design, Design. JSSC, Nr.4, 1992.

[14] Smith J.E., A study of branch prediction strategies, in Proc. 8th ASCA, pp.135-48, 1981.

[15] Albert D. and Avnon D., Architecture of the Pentium Microprocessors, IEEE Micro, Juni 1993.

[16] Heinrich J., MIPS1000 Microprocessor Users Manual Alpha Draft 11.Oct, Mips Technologies Inc., Mountain View. Ca, 1994

[17] Motorola Inc., Power PC620 RISC Microprocessor Technical Summary, MPC 620/D, Motorola Inc., 1994

[18] Lee J.K.F. and Smith A.J., Branch prediction strategies and branch target buffer design, Computer 17(1), pp.6-22, 1984.

[19] Pnevmatikos D.N. and Soshi G.S., Guarded Execution and branch prediction in dynamic ILP processors, In Proc. 21. ISCA, pp. 120-9, 1994.

[20] Texas Instruments, TMS320C6000 CPU and Instruction Set Reference Guide, Texas Instruments, 10.2000.