A Scaleable Instruction Buffer for a Configurable DSP Core

Christian Panis[1], Michael Bramberger[2], Herbert Grünbacher[1], Jari Nurmi[3]

[1]Carinthian Tech Institute
Europastrasse 4
A-9524 Villach
Austria

[2] Infineon Technologies
Siemensstrasse 2
A-9500 Villach
Austria

[3]Tampere University of Technology
P.O. Box 553
FIN-33101 Tampere
Finland

Abstract:

Increasing system complexity of SOC applications leads to an increasing requirement on powerful embedded DSP processors. To increase the performance of DSP processors the number of parallel-executed instructions has been increased. To program the parallel units VLIW (Very Long Instruction Word) has been introduced. Traditional VLIW architectures feature poor code density and therefore high area consumption caused by the program memory. To overcome this limitation the proposed configurable DSP core supports unaligned program memory, to reduce the size of the program memory port an execution bundle can be mapped onto several fetch bundles. To overcome the memory bandwidth mismatch between fetch and execution bundle an instruction buffer is introduced. Using the instruction buffer during execution of inner loops the power dissipation of the DSP subsystem can be reduced. Cache logic is used to control the entries of the instruction buffer during out-of-order execution. This paper describes the architecture and the implementation of the instruction buffer. The instruction buffer is part of a project for a configurable DSP core.

1. Introduction

Increasing system complexity of SOC applications leads to a strong demand on powerful embedded processors. To increase the performance of embedded processors the number of pipeline stages is increased to reach higher clock frequencies and the number of parallel executed instructions is increased to gain higher system performance. To program the available parallel units VLIW has been introduced [1]. The drawback of traditional VLIW architectures is an increase of the program memory and therefore a poor code density [2]. To overcome this problem available DSP architectures decouple fetch and execution bundle. The size of the fetch bundle (and therefore the size of the program memory port) is equal to the size of the maximum possible execution bundle. xLIW [3], a scaleable long instruction word supports a reduced program memory port size. The size of the fetch bundle is constant; the size of the execution bundle can be different each cycle (depending on application specific requirements). One execution bundle can be mapped onto several fetch bundles. To prevent stall cycles due to an incomplete execution bundle, an instruction buffer is introduced. The entries of the instruction buffer are controlled by cache logic to make use of the advantages of the instruction buffer also during out-of-order execution.

A typical feature of DSP algorithms are loop constructs. Therefore the proposed DSP architecture supports zero-overhead loop instructions. The loop handling like decrementing of the loop counter is handled by hardware and does not require further instructions. Using the instruction buffer during loop handling reduces the number of program memory fetch cycles and therefore reduces power dissipation. The loop is fetched once from memory and then executed from the instruction buffer.

This paper describes the architecture and the implementation of the instruction buffer. The first section is used to introduce VLIW architectures and the xLIW concept. The second part illustrates specific requirements of the proposed DSP core. The third part illustrates the architecture of the instruction buffer and is followed by a section describing implementation details.

2. Motivation

This section is used to briefly introduce the drawback of traditional VLIW architectures concerning code density. Available solutions to overcome this problem are illustrated. At the end of this section xLIW, a scaleable long instruction word is briefly introduced.

2.1. VLIW

Traditional VLIW architectures feature poor code density. Instruction scheduling is done in SW and therefore no hardware support for resolving data and instruction dependencies like scoreboards is available (static scheduling). The fetch bundle (instructions which are fetched in parallel) is fetched from program memory and the instructions are decoded. The size of the fetch bundle is equal to the maximum number of parallel executed instructions. An increasing number of parallel data paths leads to a wide program memory port and poor code density due to missing issuing queues. The data dependency inside the application code leads to a poor usage of the available data paths and for traditional VLIW architectures to a poor code density.

2.2. TI C62xx

A possibility to overcome this problem is to decouple fetch and execution bundle (instructions which are executed in parallel). The C62xx from Texas Instruments enables to map several execution bundles to one fetch bundle. The size of the execution bundle is scaleable and is called VLES (Variable-Length Execution Set). The program memory port is 8 instruction words wide, each instruction 32 bits; this leads to a size of the program memory port of 256 bits. As illustrated in Figure 1 the fetch bundle can consist of several execution bundles, which are executed during consecutive clock cycles. The same figure illustrates a problem related to this implementation. The execution bundle has to fit completely into the size of the fetch bundle (the execution bundles in Figure 1 are marked with n, n+1, …).

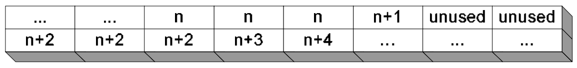

Figure 1: VLES of TI C'62xx

On one side this leads to a wide program memory port (the size of the fetch bundle is equivalent to the size of the largest possible execution bundle). On the other side still unused program memory addresses are available due to the alignment requirements. Does the execution bundle not completely fit into the remaining space of the actual fetch bundle, it has to be shifted into the next fetch bundle. The static scheduling does not allow to change the order of the execution bundle to optimize the consumed program memory space.

2.3. TI C64xx

To overcome the drawback of unused program memory due to alignment restrictions, the C64xx of Texas Instruments allows mapping of parts of the execution bundle into a fetch bundle. The remaining parts of the execution bundle are mapped to the next fetch bundle. For indication of the end of the execution bundle a single bit in the instruction word is used (the same is true for the C62xx architecture).

2.4. SC140

The Starcore SC140 supports a similar concept of decoupling fetch and execution bundle. Instead of using a single bit a prefix is used to indicate the size of the execution bundle. The prefix word also contains information e.g. for predicated execution.

The introduced concepts allow to size the execution bundle to the requirements of the application code and therefore to increase the code density. The concepts still feature a wide program memory port. The size of the port is influenced by the maximum number of instructions, which can be executed in parallel.

2.5. xLIW

The proposed DSP core features a similar concept of decoupling of fetch and execution bundle to increase the code density compared with traditional VLIW architectures. The proposed DSP core allows configuring of the main architectural features, which allows to drive the core architecture into an application specific optimum concerning power dissipation and area consumption.
To reduce the size of the program memory port, without limitations concerning the calculation bandwidth it is possible to map one execution bundle onto several fetch bundles. In average the fetch bandwidth and the required execution bandwidth (driven by the requirements of the application code) has to fit. To compensate a bandwidth mismatch, which can take place in small code sections (e.g. inner loops), an instruction buffer (with n-entries) is introduced. The buffer is filled on one side with fetch bundles of constant width, on the other side execution bundles of variable size are set together.
Using the introduced instruction buffer for reducing power dissipation, loops are executed from the buffer. The instructions of the loop are fetched only once and are executed several times, without additional fetch cycles; the switching activity at the program memory port can be reduced and therefore the power dissipation of the DSP subsystem.

3. Architectural Requirements

This sub-section is used to point out the requirements to the architecture of the instruction buffer.
The requirements are *scalability*, *fully deterministic behaviour* and *power-efficient handling* of loop constructs (which includes also e.g. while loops, which will not be handled as hardware loops)

- Scalability: The proposed DSP core allows scaling of most of the architectural features like the size of the register file, the width and number of the data paths, the instruction coding and the memory bandwidth [4]. This is necessary to drive the DSP core architecture to an application specific optimum in power dissipation and area consumption. For the architecture of the instruction buffer this means that the instruction size and the number of possible entries of the instruction buffer have to be scaleable.
- Deterministic behavior: DSP applications require a deterministic time behavior. The execution time of a certain program has to be constant (worst case timing has to be considered during the definition of the system architecture. Therefore a prediction mechanism does not gain any system performance). For the architecture of the instruction buffer this requirement does not allow any influence on the execution time of the program; independently if the instruction words are already in the buffer or have to be fetched from program memory.
- Power efficient loop handling: The instruction buffer should be used during loop handling to reduce the power dissipation at the program memory port. The proposed DSP architecture supports zero-overhead hardware loops. However, the advantage of the instruction buffer should also be used during e.g. while loops (which are implemented by conditional branch instructions) and branch instructions inside loop bodies. To handle while loops manual control of the buffer entries is necessary. During out-of-order execution the advantage of the instruction buffer has to be available.

Considering these requirements the following section will be used to illustrate the chosen instruction buffer architecture.

4. Instruction Buffer

This section is used to introduce the chosen architecture of the instruction buffer mainly influenced by the requirements introduced in section 3.

4.1. Program Memory Fetch

The instruction buffer is part of the fetch stage of the pipeline. To fulfil the requirement concerning deterministic behaviour the execution of the program has to use the same number of cycles independent whether the instructions are already inside the buffer or not. Therefore at the beginning of the fetch stage, in parallel to the access to program memory, the content of the instruction buffer will be compared with the required instruction words. Assuming the data has been already fetched before and is already available inside the instruction buffer; the fetch cycle to the program memory is suspended. If there is more than one clock cycle reserved for the program memory fetch that is true for all of them. Therefore instructions already available inside the instruction buffer do not reduce the number of pipeline stages, but reduces the power dissipation at the program memory port.

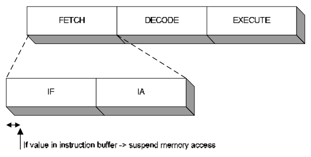

Figure 2: Program Fetch Decision

In Figure 2 the fetch stage of the pipeline is illustrated. In the beginning of the first cycle the decision of suspending the memory access is done.

4.2. Buffer Structure

In Figure 3 an example for the instruction buffer is illustrated (the fill operation of the buffer). A fill pointer is filling empty cells. Each cell has additional control bits. The executed bits (*E*) are used to indicate, whether a fetched instruction word has already been used for execution and therefore can be overwritten with new entries. The valid bits (*V*) are used to indicate valid entries inside the instruction buffer. Therefore no initialization of the buffer entries is necessary. Assuming linear program flow the buffer can be built up as a circular buffer.

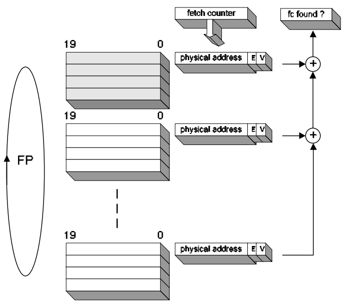

Figure 3: Instruction Buffer Write

If branch instructions interrupt the program flow, already fetched instructions cannot be used any more. That is why cache logic is used to control the entries of the instruction buffer.

4.3. Cache Logic

For the cache logic a *set-associative* approach has been chosen as illustrated in Figure 4. The advantage compared with a *fully-associative* approach is that the cache directory and the cache data memory are getting the address in parallel [5]. In the fully-associative approach the cache data memory is getting the address sequentially (which increases the critical path in timing).
The tag-address is used to differentiate the different *pages* of the memory addresses. One possible problem can get the *cache trashing*. Cache trashing takes place when a frequently used location is replaced by another frequently used location. The problem of cache trashing can be reduced by using a *n-way set associative* cache.

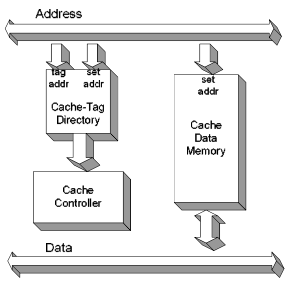

Figure 4: Set Associative Cache

5. Implementation

This section is used to illustrate the implementation aspects of the instruction buffer. For regular structures a full-custom implementation has significant advantage in power dissipation, area consumption and reachable frequency (which is quite important due to the critical timing of the program memory access). To obtain the requirement of scalability the DPG (Data Path Generator) of RWTH Aachen was used to develop the full custom parts of the instruction buffer [6].

5.1. Partitioning

In Figure 5 the set associative cache can be split into a CDM (cache data memory) and into a CAD (cache address directory).

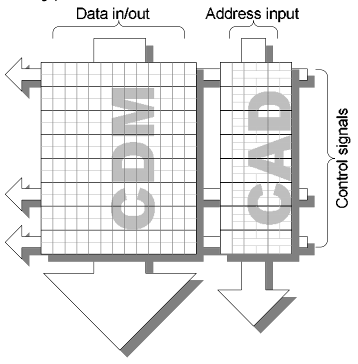

Figure 5: CDM and CAD

Both are well suited to be implemented in full-custom due to the regular architecture. The EBM (executed bit management also including the handling of the valid bits) is part of the control logic and therefore implemented in VHDL.
The CDM is used to store the instructions inside the instruction buffer, which is organized in cache-lines. The

number of cache lines is scaleable, as also the number of bits of each cache line. This enables the implementation of instruction buffer variants with a different number of entries and also the size of the instruction words can be changed. Each block in Figure 3 (consisting of 4 entries, the first one is marked with grey colour) is equal to one cache line inside the instruction buffer.

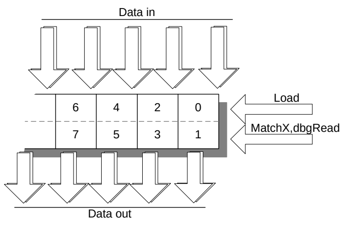

Figure 6: Cache Line

To prevent an unsymmetrical height to width ratio, two successive bits are placed one upon the other as illustrated in Figure 6. The data are fed vertically and the control (like *load*, *MatchX* or debug control signals) of the cells is done horizontally. With the implementation example in Figure 6 the granularity of the size of the instruction words is limited to two bits. If the number of bits per instruction word is odd one cell keeps unused.

The CAD is used to store the address of the instructions of the instruction buffer. Each of the address bits is compared with five entries (four data ports and one debug port, illustrated in Figure 7). If the compared entry matches to one of them then the related MatchX signal is set. The MatchX lines are cascaded to reduce the timing critical path. In this implementation example four MatchX lines are grouped.

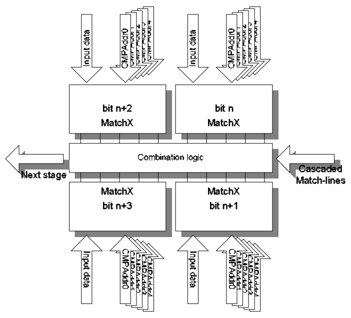

Figure 7: CAD Architecture

5.2. Results

The instruction buffer has been implemented in a 0.13µm CMOS technology with a supply voltage of 1.2V (1.5V is the nominal supply voltage; the lower one has been chosen to reduce power dissipation).

cache lines	area	*MatchX*	data present time
16	0,069 mm²	920 ps	1.2 ns
32	0,138 mm²	920 ps	1.3 ns
64	0,276 mm²	920 ps	1.4 ns
128	0,552 mm²	920 ps	1.6 ns

Table 1: Implementation Results

In Table 1 implementation results for certain configurations are illustrated. Due to the chosen architecture the timing for the MatchX line is independent of the number of cache lines. The data present time (access time) is increasing with the number of cache lines due to the increasing capacity of the data output lines.

In Figure 8 the layout of one cache configuration is displayed (16 cache lines, 80 data bit, 14 address bit).

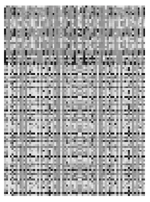

Figure 8: Layout

The upper part contains the CAD, followed by the driver of the MatchX signals. The lower part in Figure 8 contains the CDM.

6. Conclusion

The scaleable instruction buffer as proposed in this paper can be used to reduce the width of the program memory port without limitations to the calculational bandwidth. Inner loops in typical DSP application code can be handled power efficient due to a reduced number of program memory fetch cycles. The scalable implementation of the buffer architecture enables application specific adaptations to minimize the consumed silicon area. The introduced cache logic to control the buffer entries allows making use of the advantages of the instruction buffer during out-of-order execution. The chosen architecture allows minimizing the worst-case execution time as required by real-time constraints of DSP algorithms. The scaleable instruction buffer is part of a project for a configurable DSP core.

7. Acknowledgement

The work has been supported by RWTH Aachen and by the EC with the project SOC-Mobinet (IST-2000-30094).

Literature

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

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

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

[4] C.Panis, G.Laure, W.Lazian, A.Krall, H.Grünbacher, J.Nurmi "DSPxPlore – Design Space Exploration for a Configurable DSP Core", *GSPx 2003*, Dallas, Texas, USA, 2003.

[5] J.Handy, "The Cache Memory Book", Academic Press, 1998

[6] O.Weiss, M.Gansen, T.G. Noll, "A flexible Datapath Generator for Physical Oriented Design", *Proceedings of the ESSCIRC 2001*, Villach, 18.-20. September 2001, pp. 408-411