Restricted to programme participants

ASAM

Grant agreement no. 100265

Artemis Project

ASAM

Automatic Architecture Synthesis and Application Mapping

D5.1: Specification of architecture-driven code transformations

Due Date of Deliverable	**30st April, 2011**
Completion Date of Deliverable	**May 23, 2011**
Start Date of Project	**1st May, 2010 – Duration 36 Months**
Lead partner for Deliverable	**SH**
Authors	**Menno Lindwer (Silicon Hive), Erkan Diken (TU/e)**

Revision: v0.1

Project co-funded by the Artemis Joint Undertaking Call 2009		
Dissemination Level		
PU	Public	x
PP	Restricted to other program participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Table of Contents

1 Introduction

The ASAM (Automatic Architecture Synthesis and Application Mapping) project aims to develop a system-level design flow that will facilitate the design of complex embedded hardware-software systems. Such systems consist of optimized heterogeneous multi-processor hardware platforms, running efficiently mapped and highly parallel software systems.

The code transformations and encoding primitives, as specified in this document, are leading in terms of defining two significant developments in the ASAM project. They determine:

1. the ways in which the front-end application analysis tooling will express parallelism present in the application
2. the input formalisms for the compilers that map code onto application-specific processors (ASIPs)

The input formalism for the applications being considered in the ASAM project is ANSI-C'89 (or ANSI-C, for short). This language does not support parallelism, meaning that it does not have constructs which can be used to explicitly express parallelism. However, this does not mean that applications encoded in ANSI-C can not contain parallelism. ANSI-C compilers are free to choose the encoding of machine operations and, as long as this does not conflict with the ANSI-C-specified functionality of the application, compilers can choose any order in which to execute the resulting machine operations.

By employing data flow analysis, compilers indeed do determine which operations can be executed in parallel, without affecting the defined functionality. When subsequently generating code for processors which can execute operations in parallel (ILP- or VLIW-style parallelism), such compilers will then utilize this information to construct parallel schedules.

However, this kind of analysis often is not enough to extract and drive all possible multi-ASIP parallelism. Next to ILP-style parallelism, SoCs with multiple Application-specific processors (ASIPs) support many other kinds of parallelism. Even though compilers can extract a certain level of VLIW-style parallelism, higher-level code transformations are needed to allow compilers to fully exploit all kinds of parallelism present in the application. The kinds of parallelism targeted in the ASAM project (and some of their implications for code transformations) are discussed in section 3.

The kinds of code transformations that are identified within the ASAM project are the following:

Abstractions: These are higher-level constructs which describe collections of parallel processing elements. These constructs are chosen such that they represent architectural features in such a way that they facilitate expression of parallelism. For example, intrinsics (which represent multiple RISC operations) are abstractions that facilitate invocation of multi-RISC operations. The abstractions defined in the ASAM project are described in section 4.

Task- and loop-level code transformations: If there were no dependencies between individual operations in an algorithm, then all of these operations could execute completely in parallel. For example, in order to initialize the pixel values in an image buffer, no dependencies exist. All pixel values can be written completely in parallel. On the other hand, when incrementing pixel values, each written value depends on the value first being read and then being incremented. Yet between different pixels, there are no dependencies and all read-increment-write sequences can execute in parallel. Generally, code transformations have three goals: (i) code partitioning according to type of processing, (ii) reduction of dependencies, and (iii) improving regularity. Task-level loop transformations may be based on layering of the code, such as control layer and data plane layer. They may also result in merging of similar activities, in order to facilitate vectorization. Other task-level transformations result in code that can be scheduled more easily, such as rewriting in static single assignment form and inlining. Loop-level transformations are typically meant to reduce dependencies and improve regularity. For example, loop unrolling removes loop dependencies and may improve regularity, thereby increasing the compiler's possibilities to schedule parallel issue slots. Task- and loop-level code transformations are discussed in section 5.

Vectorization: As discussed in section 3, data-level parallelism is expressed in terms of sequences of data items of the same type (e.g. vector elements are 16-bit integers) and the same length (e.g. vector length is 32 elements, or a total of 512 bits). Vector-parallel operation is expressed in terms of operations acting on vectors. Thus, the vector-ADD operation would add all 32 elements of vector A to all 32 elements of vector B. Usually, encoding for vector operations entails certain computational overhead. The data may need to be organized in a specific manner, such that they can be loaded through vector load operations. When dealing with 2D kernels, even vector-organized data may need to be accessed in non-aligned fashion, requiring vector align operations. Loop lengths may not be aligned with integer vector sizes, and the remaining loop iterations may need to be processed on a scalar datapath. The scalar datapath of the processor may be also be required to perform other control tasks. All of these non-vectorized activities decrease the effects of vectorization (Amdahl's law applies). Loop- and task-level code transformations are often required, in order to make efficient use of the vector capabilities of application-specific processors.

Compiler pragmas: Compiler pragmas are a collection of additional constructs that drive compilers to perform certain optimizations or to make use of certain architectural features. For example, loop unrolling and software pipelining are often driven through compiler pragmas. Also compilers may offer different heuristics, in order to trade off speed, power, and register pressure. Such heuristics are also controlled by compiler pragmas.

2 Scope

This current document is intended as a description of the compiler features which need to be supported in order to drive parallel processor systems. It also specifies the code transformations that are needed, in order to express parallelism in ways that compilers can understand.

As such, this document serves as a specification of compiler features and directives which need to be implemented within the ASAM project, in order to support the goal of mapping future parallel applications onto future parallel platforms. The document also specifies which code transformations are to be implemented as part of the application analysis of ASAM's design space exploration loop.

This document does specify the suitability of particular kinds of parallelism, particular constructions, or particular code transformations. This document also does not provide guidelines or trade-offs about when or whether to extend processors with features.

This document also is not a final definition, nor a manual for the use of ASAM-defined compiler extensions and code constructs.

This document is meant to be an initial reference for the designers of the final design methodology and individual tools within the ASAM project, providing them with a specification of the expected interoperability, and I/O behaviour of the tools to be developed.

3 Overview of processor architecture features

The five kinds of parallelism, and their implications application encoding are briefly discussed here:

Operation-level pipelining: operations with complex silicon implementations (e.g. multipliers) may be executed in several stages, separated by pipeline registers. This generally results in better balancing of hardware resources and higher attainable overall clock speeds. Generally, even if, for a certain operation, the required clock speed can be achieved with N (N>=0) pipeline stages, introducing N+1 or N+2 pipeline stages may actually significantly decrease the associated chip area. The associated function units take several cycles to execute operations (multi-cycle latency). Yet, they can start a new operation every cycle and can produce a new result every cycle (throughput is 1 operation/cycle). This is only possible when each of the stages of the pipelined operation executes in parallel. As with ILP/VLIW-style parallelism, modern VLIW compilers match the dataflow graph of the application with the constraints of multi-cycle operations, in order to take the additional latency into account. Similarly, some superscalar processors can perform this kind of scheduling automatically. Therefore, this kind of parallelism is not further discussed in this document and this document does not specify code transformations targeting this level of parallelism.

Multi-RISC operations: These kinds of operations are often referred to as *custom operations*. However, this term implies that such operations need to be explicitly addressed using *intrinsics*, which is not always the case. Other terms that are sometimes used are *complex operations* or *CISC* operations. The ASAM project prefers to use the term *multi-RISC operations*, in order avoid the ambiguity and historical debates associated with the other terms. The ASAM project takes a compositional approach when dealing with this kind of parallelism; when searching for ways to increase efficiency of application-specific processing, processor designers may choose to collapse dataflow graphs which combine several relatively simple operations into a single, more complex processor operation. Since this may result in function units with complex/large hardware implementations, additional operation-level pipelining may be needed, as described above. Some compilers apply dataflow graph matching between application code and operation definitions, in order to automatically and optimally determine which (multi-)RISC operations are to be invoked. However, with increasing complexity of the multi-RISC dataflow graphs, chances also increase that compilers can not find matches and will resort to splitting the application into simpler operations, where more efficient operation selection would have been possible. In those cases, the application code needs to be transformed, explicitly introducing *intrinsics*, i.e. direct source-level invocations of multi-RISC operations. Please note that this document does not deal with other aspects of multi-RISC operations, such as their relative silicon or power efficiency or re-usability. This document only describes the code transformations required to apply such operations, assuming that processors support them.

Instruction-level parallelism: This kind of parallelism requires support through either VLIW (Very Long Instruction Word) processors or superscalar processors. Both types of processors contain multiple parallel instruction pipelines in which, during a single cycle, multiple operations can be

started. In the case of a VLIW processor, the instructions themselves encode exactly which operations are being executed in parallel. This implies that the compiler has pre-determined the (partial) ordering and parallelism in the code. Superscalar processors analyze incoming sequential operation streams, detecting parallelism, and starting multiple operations on-the-fly. The advantages of the VLIW approach are that the compiler generally has a broader scope for optimizations and that the processor does not need additional hardware for dynamic operation scheduling. The advantages of the superscalar approach are that instruction words are relatively narrow (reducing instruction and cache memory size) and that code does not need to be re-compiled when the parallelism in the architecture is adapted. In both cases, the data flow analysis determines which operations can be executed in parallel. Also, in both cases, the amount of available parallelism can be increased through certain code transformations, such as loop unrolling.

Vector-, data-, or SIMD-level parallelism: This kind of parallelism is comparable to multi-RISC operations, in the sense that each vector operation is equivalent to a multitude of non-vector operations. However, vector operations typically apply a single non-vector operation to a sequence of data items of the same type (referred to as a *vector*). In vector architectures, the physical data vectors typically all have the same number of elements. Modern applications-specific processors can mix and match vector and scalar processing. For example, an issue slot of a VLIW machine may combine function units that operate on scalar data types with function units that operate on vector data types. The scalar operations in that issue slot could serve e.g. for address calculation when loading and storing vectors. Also, vector issue slots may combine vectors of single-bit flags and vectors of data elements. Thus, compilers for such application-specific processors need to schedule combinations of scalar, vector, and flag operations.

Task-, thread-, or MIMD-level parallelism: In the contenxt of the application-specific processors targeted in the ASAM project, these terms all have the same meaning: when running multiple data plane tasks in parallel, then they run on different application-specific processors. The SoC may also contain control processors, which support multi-threading on a single processor. However, the ASAM project does not target to optimize that kind of processing. Thus, encoding for task-level parallelism in the data plane requires that different parts of the code are targeted at physically different application-specific processors.

4 Abstractions on top of supported compiler platforms

When describing abstractions in the context of parallel processing, these abstractions are actually collections of parallel processes: through a single construct, the code indicates that a set of processes is to be fired in parallel.

Regarding these abstractions, the usage scenarios of the the ASAM project assumes two situations:

- The input is regular straight-line ANSI-C code, which does not contain such abstractions. In this case the application analysis and DSE steps will transform the straight-line code, collecting processes into suitable abstractions.
- The input is existing code, optimized for a certain parallel starting point platform, and already containing abstractions. In this case, the abstractions may have to be converted into abstractions for a new target platform.

The following paragraphs describe the five different kinds of abstractions which the ASAM compilers need to support and which need to be converted or generated by the application analysis and DSE tools.

If one of the compilers targeted in the ASAM project does not directly support an abstraction, then these compilers need to be extended or conversion tools need to be build.

4.1 Intrinsics

Intrinsics are direct representations of processor operations. They resemble inline assembly code. However, each intrinsic has a one-on-one relation with a single processor operation. Intrinsics take regular variables as arguments. Operator overloading (discussed below) may be used to have certain intrinsics suitably mapped onto operation symbols.

Intrinsics are abstractions of parallel processor features, in the sense that application-specific operations often are fairly complex, comprising the pipelined functionality of a collection of parallel RISC-like operations.

4.2 Data memory types and memory addressing

ASAM's application-specific processors may employ different kinds of data memories:

- Scalar memories
- Vector memories
- Vector-addressable memories

Each processor may have any number of such memories, depending on application characteristics. The memory types and widths within one processor do not need to be the same. For example, a processor may have a 32-bit scalar memory, a 32-by-16-bit vector memory, and a 10-bit vector-addressable memory.

ASIPs with multiple data memories typically also have multiple load/store units (LSUs), residing in different VLIW issue slots, in order to benefit from the combined bandwidth of the memories. In particular, an LSU that is connected to a 512-bit vector would be different from an LSU that is connected to a scalar memory, in the sense that the former one would have 512-bit wide data lanes.

Thus, it is important to distinguish which data is placed in which memory, because the data types must match. Thus, external agents (programmable or not) should be programmed to exchange data directly with the local vector memories of the image processing ASIP.

Additionally, if multiple memories of the same type are present in the ASIP, it is still important to explicitly identify which data resides in which memory. One could assume that compilers would automatically and optimally select the memory in which to place data, depending on the *proximity* of that memory to the function units on which the data should be processed . (In this case, *proximity* is loosely defined as the number of operations needed to transport the data item from its original memory location into the function unit on which the data will be processed.) However, typically the situation is reversed. Compilers expect the code to indicate where the data is stored and, when selecting issue slots for the processing of data items, compilers select those issue slots that are closest (i.e. having smallest *proximity*) to the data item's memory.

Obviously, the variable may be an array, of multiple items of type T.

When developing, analyzing, and generating code for vector ASIPs, it is important to take into account the specific data types which the ASIP supports and alignments which the ASIP supports. This is specifically the case for ASIPs with vector capabilities, and is accomplished by designing memory-based structures such that have:

- Size of individual data elements aligned with vector element size. This may mean that input and output data are cropped, saturated, or extended to exactly fit a particular vector element size (an ASIP may have multiple element sizes). This may also mean that the input data is reduced below the vector element size, in order to allow scaling of the data elements, as required by the algorithm.
- Multiple such vector-element-sized data items packed in such a way that sequences of such elements correspond to vectors that can be processed in parallel.
- Packed vectors of element-sized data items are placed in vector memories, through allocation of variables of (arrays of) vectors. Please note that the allocation of vectors of data in vector memories must be aligned to vector size.

Generally, besides the standard ANSI-C data types, the compilers in the ASAM project need to support the following ASIP-specific scalar data types:

```
__int<n> r; // <n>-bits wide non-signed integer
__sint<n> i; // <n>-bits wide signed integer
__uint<n> u; // <n>-bits wide unsigned integer
```

Please note that it is necessary to take into account that only certain memories can be used to map values of specific types onto. An attribute needs to be used in those cases where the ASIP's default memory would not support the specific data type.

4.3 Buffering and synchronization

In order to achieve optimal parallelism between multiple data processing elements, data is often organized such that these processing elements have their input, output, and intermediary data located in physically different memories. However, the processing elements do need to share buffers, in order to communicate their results. Often, this means that some form of double buffering is applied, both on input and output data.

In order to communicate buffer addresses, there need to be map files, indicating the addresses at which variables are located. These map files are header files which are included in all code within the system (*program*`.map.h` for an application called "program"). After having included such a map file, the code running on another processing element can access the data through a pointer which can be calculated, based on the definitions in the header file.

We notice that the address is constructed by taking the base address of a particular slave interface of the target processor. Since system-level memory maps are not necessarily seen the same way by all masters, a translation takes place to find the particular base address of the target slave, as seen by the particular master interface of the local ASIP. Subsequently, since ASIPs may have multiple local memories accessible through the same slave interface, the remote offset through the slave interface to the target memory should be retrieved, and added to the slave base address. Then, the offset of the addressed variable, within that memory, needs to be added.

Since both the input and output buffers should be considered as shared between multiple data processing elements, the producer and consumer applications addressing these input and output buffers need to synchronize. The ASAM project uses blocking send and receive commands to achieve synchronization between producer and consumer blocks. The send and receive commands access FIFOs with limited depth. When sending a token into a full FIFO, the sending block is stalled, until the receiver reads a token. Similarly, when a block tries to receive a token from an empty FIFO, the receiver is blocked until the sender writes a token into the FIFO.

When scheduling code containing external buffer data and synchronization primitives, compilers act in a conservative way, assuming that these buffers contain volatile data. This then means that compilers are severely restricted in the way in which they can exploit parallelism between input, output, and intermediary data processing.

4.4 Vector operations

Generally, one can identify four different types of vector operations:

- Element-wise operations
- Cloning operations
- Intra-operations
- Permutation operations

These operations are further defined in the sub-sections below.

Each ASIP may contain its own specific instruction set with different (vector) operations. However, generally each vector operation can be regarded as belonging to one of the above types. The naming of the

vector operations shown below is only indicative. The designer of the ASIP is free to choose the names of the operations. Even if the name of the operation is vec_add, this does not necessarily mean that the operation performs a vectorized addition.

Generally, above operations indeed are parallel processing constructs. Even the intra-operatoins, in which intermediary results typically depend on their predecessors, can be executed (partially) in parallel. Another interesting operation is the element-wise multiplexer. It takes a vector of (single-bit) flags and two vectors of data elements. For each data element, depending on the setting of the associated flag, an element from either input vector is chosen to be copied to the output (at the same location).

The ASAM project does not assume that the compiler will automatically infer vector operations. The application analysis and DSE steps will introduce the abstract operation vector intrinsics or use the overloading mechanism to express vector parallelism.

4.4.1 Element-wise vector operations

Element-wise operations, such as element-wise addition combine all elements of the input vectors with vector elements at the same locations in the other input vectors.

4.4.2 Cloning operations

These are operations in which a vector input and a single scalar input are combined, i.e. each vector element is combined with the same scalar input. The scalar input is first copied as many times as there are vector elements, producing a vector of clones of the scalar input and then the actual operation takes place on the input vector and the clone vector.

4.4.3 Intra-vector operations

Intra-operations combine all or several of the elements of vectors with other elements of the same vector. For example, the *intra-sum* operation adds all vector elements together, producing a scalar result.

4.4.4 Permutation operations

Permutation operations either shuffle vector elements within the same input vector or between multiple input vectors. Many permutation operations could be considered and it is not possible to give a general template.

5 Task- and loop-level code transformations

In the following subsections, loop transformations, to be used in the ASAM project, are explained in brief and illustrated with an example code. By doing that, it is aimed to clarify the transformations and their benefits for exploiting parallelization and data locality.

Loop transformations can be used for different purposes. For example, there are transformations which aim to expose the concurrency in the application, e.g by unrolling, to increase the data availability, e.g by tiling, or to change the the order in which iteration space is traversed, e.g. loop interchange. Classification of loop transformations is provided at the end of this subsection. In general, the transformation aim:

1. to reduce the dependencies
2. to partition the code according to a type of processing
3. to improve regularity

This subsection continues with explanation of various loop transformations and their roles. A simple code example is given to illustrate the explanation of each transformation. Also some comments are provided about the loop transformations related to the ASIP capabilities.

In below sub sections, wherever references are made to pragmas that are supported by the ASAM compilers, please refer to section 7.

5.1 Loop Normalization

Loop Normalization makes the dependence testing process as simple as possible and simplifies the application of many other loop transformations. Many compilers normalize the loops to run from a lower bound of 1 or 0 to some upper bound with a stride of 1. In the example code below, the index variable is set to the initial zero value, and accordingly, the rest of loop is adjusted. These kinds of preliminary transformations are explicitly listed at the end of this subsection. Almost all compiler front-ends support this kind of preliminary transformations.

DO I=2, N
  sum += A[I-2]
ENDO
	DO I=0, N-2
	  sum += A[I]
	ENDO

5.2 Loop folding / Loop (Software) Pipelining / Kernel Scheduling

Sortware pipelining is a technique which schedules an entire loop at a time to take full advantage of the parallelism across iterations. Basically, loop pipelining allows a new iteration of a loop to be started before the current iteration has finished. The amount of overlap is determined by the itiation interval (II). The schedule is such that it can be initiated every II instructions.

L1: a = *p++
 b = a + 1;
 *q++ = b;
 if(--n != 0) goto L1
	Loop pipelining with II=1
	
	a = *p++;
	a = *p++; b = a + 1;
	a = *p++; b = a + 1; *q++ = b; if(--n != 0) .

<table>
<tr>
<td>Without loop pipelining scheduling might result in:

L1: a = *p++
b = a + 1;
*q++ = b; if(--n != 0) goto L1

(4 operations are scheduled in 3 cycles, 4/3 ops/cycle)</td>
<td>a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
b = a + 1; *q++ = b;
*q++ = b;

The compiler generates the following code:

n=n-2
a = *p++;
a = *p++; b = a + 1;
L1: a = *p++; b = a + 1; *q++ = b; if(--n != 0) goto L1
b = a + 1; *q++ = b;
*q++ = b;

During the kernel of the loop 4 operations are executed per cycle, 4/1=4 ops/cycle)</td>
</tr>
</table>

The compilers used in the ASAM project support automatic software pipelining, which can be enabled by pragma `pipelining`. Loops eligible to software pipelining should consist of one basic block. In practice this means that it is not allowed to place if-then-elses, other loops, or function calls in the to-be-pipelined loop body. Neither are break and continue statements allowed. The ASAM compilers contain pragmas which automatically transform such control code into constructs that can be pipelined, e.g. `break-conversion`, `if-conversion`, and `uroll`. Also, where applicable, functions are inlined (see sub section 5.3) in order to remove function calls from loops that need to be pipelined.

5.3 Function inlining

The first objective of introducing functions in source code is to structure the code as a means of increasing maintainability and re-use. However, by invoking functions performance is reduced (relative to just flattening the code and repeating the function bodies at the locations where they are called). Performance is not only reduced by the overhead of the function invocation code itself (i.e. retention of return address, jump operation), but also by having to save the caller function's state and, upon return, restoring that state. In addition, the ASAM compilers can not apply many of the block-level and super block level optimizations across function calls, such as software pipelining and hoisting.

Thus, whenever function bodies are small or their invocation locations are very scarce (i.e. few invocation instances in the application source code), it makes sense to actually copy the function body at the invocation location. This is what is being accomplished through function inlining. The ASAM compilers

support this optimization, as follows, as is illustrated in below table.

inline int f(int a, int b) { return a+b; }
 	
int invoke(int x, int y) {
  return f(x,y) + f(x+1,y+1);
}
	Int invoke(int x, int y) {
	  return (x+y) + ((x+1)+(y+1));
	}

5.4 Loop unrolling

Loop Unrolling is a process of replication of the loop body, usually the innermost loop of a nest. Moreover, the index expression corresponding to each of the unrolled iterations is propagated to the statements in each instance of the loop body, as shown in the code example.

DO I = 1, N
  A(I) = B(I) + C(I)
ENDDO
	Unrolling factor(k)=4
	
	DO I = 1, N, k
	  A(I) = B(I) + C(I)
	  A(I+1) = B(I+1) + C(I+1)
	  A(I+2) = B(I+2) + C(I+2)
	  A(I+3) = B(I+3) + C(I+3)
	ENDDO
	
	Vectorization:
	
	DO I = 1, N, k
	  A(I:I+k) = B(I:I+k) + C(I:I+k)
	ENDDO

Loop unrolling favors ILP by enabling the compiler to map each unrolled statement to a different issue slot or functional unit, and execute them simultaneously. Dependencies between the statements and hardware resources limit the degree of parallelism. And also, (assuming no limiting factor due to the dependencies and hardware support) SIMD-vector operation can be replaced with unrolled statements. In addition, unrolling decreases the iteration control overhead. However, there are also some drawbacks of loop unrolling:

- If unroll factor is not divisor of trip count (iteration count), remainder loop is required to be added.
- If trip count is not known at compile time, the trip count is required to be checked at runtime.
- The code size increases which may result in a higher instruction cache miss rate. Thus, it is a compromise decision to select the loop unrolling factor that results in the highest performance improvement, yet does not expand the code too much.

- A global determination of the the optimal unroll factor is difficult. (However, ASIP customization process deals with portions of the whole application code that is assigned to a single ASIP; hence determination of unroll factor seems to be somewhat easier).

The compilers used in the ASAM project support automatic loop unrolling, which can be enabled by pragmas `unroll` and `force-unroll`.

5.5 Loop Fusion / Combining / Jamming / Merging

These are processes of merging two or more loops into one nested loop, as shown in the following code example. It enables to exploit the parallel execution (e.g. ILP) of the statements as long as no data dependencies exist between the merged statement, and loop bounds are identical. Moreover, loop merging can improve spatial and temporal data locality.

DO I=1, N
  A(I) = B(I) + K
ENDDO
 	
DO I=1, N
  D(I) = C(I) + K
ENDDO
	DO I=1, N
	  A(I) = B(I) + K
	  D(I) = C(I) + K
	ENDDO

5.6 Unroll-and-Jam

Unroll-and-Jam is a process of unrolling outerloops and fusing the new copies of the inner loops as shown in the example code. It increases the size of the loop body and hence improves the available parallelism (e.g ILP). It can also improve the data locality. Moreover, the inner-most loop can be a candidate for vectorization. In the code example, the k value is chosen to be compatible with the vector length (VL).

DO I=0, N
  DO J=0, M
  A[I][J] = B[J][I]
  ENDDO
ENDO
	DO I=0, N, 2
	  DO J=0, M
	  A[I][J] = B[J][I]
	  A[I+1][J] = B[J][I+1]
	  ENDDO
	ENDO
	
	Vectorization:
	
	DO I=0, N, 2
	  DO J=0, M, k
	  A[I][J:j+k] = B[J:J+k][I]
	  A[I+1][J:j+k] = B[J:j+k][I+1]
	  ENDDO
	ENDO

5.7 Loop Fission / Distribution

Loop Fission / Distribution is the inverse of the loop fusion. Basically loop fission splits a single loop into two or more loops with the same loop bounds, but distributing the statements of the original loop between the bodies of the newly created loops. Loop fission usually enhances the coarse-grained parallelism. Here, the term *coarse-grained parallelism* refers to task-level/subprogram alike parallelism. On the other hand, loop fission is also used to enhance fine-grained parallelism by mean of vectorization which is explained further in detail in the vectorization subsection.

DO I=1, N A(I) = B(I) + K D(I) = C(I) + K ENDDO	DO I=1, N A(I) = B(I) + K ENDDO DO I=1, N D(I) = C(I) + K ENDDO

5.8 Loop Blocking / Tiling / Partitioning

These transformations convert the iteration space of the loop nest by structuring the execution of the loop into blocks / tiles of iterations of the original loop. The outer loop determines which of the blocks of iterations the inner loop executes.

DO i = 0 . . .M DO j = 0 . . . N DO k = 0 . . . K C(i,j) += A(i,k) * B(k,j)	DO ii = 0 . . . M by B1 DO jj = 0 . . . N by B2 DO kk = 0 . . . K by B3 DO i = ii . . . ii + B1 DO j = jj . . . jj + B2 DO k = kk . . . kk + B3 C(i,j) += A(i,k) * B(k,j)

After loop tiling, the compiler can distribute the computation of the outer loop across the various issue slots, in order to have concurrent invocation of the inner loops. This concurrency requires adequate memory support by distribution of the data across the local memories of the computational units (as discussed in section 4.2). Tiling can enhance data parallelism both at the macro and micro levels. It can support the applicability of ILP and vectorization.

5.9 Strip-mining

Strip-mining is a special case of loop tiling where the strip-mining is applied to a single nested loop, creating an inner loop responsible for the computation on each strip, and an outer loop to control or traverse the various strips. The strip length (iteration count) is usually selected to be compatible with vector length (VL) in order to facilitate vectorization.

DO I=1, N A(I) = A(I) + B(I)	DO I=1, N, s DO J=I, MIN(I+s-1, N)

ENDDO	A(J) = A(J) + B(J) ENDDO ENDDO Vectorization: DO I=1, N, s A(I:I+s-1) = A(I:I+s-1) + B(I:I+s-1) ENDDO

5.10 Loop Interchange / Permutation / Reordering

These three transformations involving loop-interchange, loop-reversal and loop skewing mainly change the order of iteration space. As can be seen from the code example, loop interchange switches the nesting order of loops by moving the index variables. The advantages of the loop-interchange are twofold: First of all, loop-interchange improves the memory accesses by changing the row-major and column-major ordering of the memory accesses. For example, the innermost loop should index the right-most array index expression in case of row-major storage like in C. Secondly, loop interchange can expose additional parallelism. If an inner-loop does not carry any dependencies (in the example code, the flow dependencies between the inner-loops are moved to the between outer-loop, leaving the inner-loop dependence-free), this loop can be executed in parallel, for instance by vectorization.

DO I=1, N DO J=1, M A(I, J+1) = A(I, J) + B ENDDO ENDDO	DO J=1, M DO I=1, N A(I, J+1) = A(I, J) + B ENDDO ENDDO Vectorizable: DO J=1, M A(I:N, J+1) = A(I:N, J) + B ENDDO

5.11 Loop Reversal

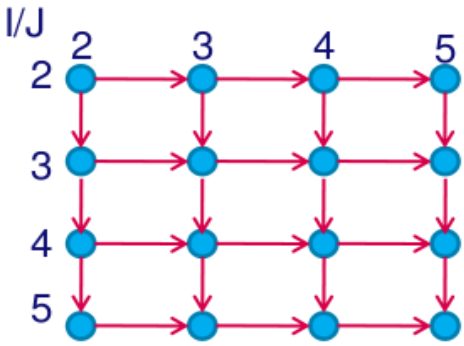
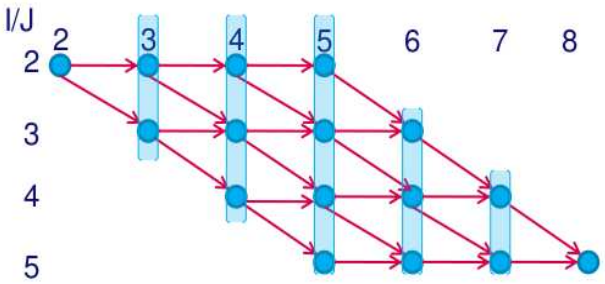
Loop Reversal is another transformation which changes the iteration order of the loops. It executes iterations in a loop in reverse order. Loop reversal is usually applied to enable other transformations.

Also, loop reversal may be needed, in order to apply auto-decrement-test-and-jump operations, which many ASIPs support.

DO I=0, N
  sum += A[I]
ENDO
	DO I=N, 0
	  sum += A[I]
	ENDO

5.12 Loop Skewing / Bumping

Loop Skewing / Bumping is another transformation which changes the iteration space. In this way, iteration space is reshaped in order to uncover existing parallelism. As shown in the example, the innermost loop is skewed by changing the iteration space in order to make parallel patterns visible.

DO I=2, 5
 DO J=2, 5
  A(I, J) = A(I-1, J) + A(I, J-1)
 ENDDO
ENDD
O
 
	DO I=2, 5
	 DO J=I + 2, I + 5
	  A(I, J-2) = A(I-1, J-2) + A(I, J-3)
	 ENDDO
	ENDDO
	 

5.13 Privatization

Privatization is applied in order to ensure that a variable within the loop is used only in the same iteration in which it is assigned. Such variables are replicated across the iterations. In this way, the loop-carried dependencies are eliminated. The following two transformations (scalar expansion, array and scalar renaming) are variants of privatization.

DO I=1, N
  T = A(I)
  A(I) = B(I)
  B(I) = T
ENDDO
	DO I=1, N
	  PRIVATE T
	  T = A(I)
	  A(I) = B(I)
	  B(I) = T
	ENDDO

5.14 Scalar Expansion

Scalar Expansion breaks the inter-loop dependencies by expanding a scalar into an array. As it can

be seen from the code example, variable T causes some inter-loop dependencies between each iteration of the loop. In order to break this kind of dependency, the variable that causes the dependency can be expanded to an array. Scalar expansion enhances the fine-grained parallelism by breaking dependencies and easing application of other transformations. Since a variable is expanded to an array, scalar expansion requires extra memory and more complex addressing.

DO I=1, N
  T = A(I)
  A(I) = B(I)
  B(I) = T
ENDDO
	DO I=1, N
	  T$(I) = A(I)
	  A(I) = B(I)
	  B(I) = T$(I)
	ENDDO
	
	T=T$(N)

5.15 Scalar and Array Renaming

These transformations eliminate loop-independent anti- and output dependencies. The reason for such dependencies is due to the reuse of memory locations. Renaming basically introduces another variable to break the reuse. In the following code example, T is replaced by T1 and T2, that results in elimination of output and anti dependencies inside the loop. The same approach is also applicable to the arrays. In this way, the elimination of the dependencies allows to improve the fine-grained parallelism such as ILP and vectorization. Scalar renaming leads to an insignificant cost, because introduced variables are usually allocated in register files; while array renaming can cause a significant cost due to extra memory usage and additional array addressing operations.

DO I=1, N
  T = A(I) + B(I)
  C(I) = T + T
  T = D(I) – B(I)
  A(I+1) = T * T
ENDDO
	DO I=1, N
	  T1 = A(I) + B(I)
	  C(I) = T1+ T1
	  T 2= D(I) – B(I)
	  A(I+1) = T2 * T2
	ENDDO

5.16 Loop Unswitching

Loop Unswitching moves a conditional statement from inside to outside loop by duplicating the loop's body. In this way, conditional statement (branch instruction) is removed from the loop body that can enhance parallel loop execution.

DO I=1, N
  X(I) = X(I) + Y(I)
  IF(W)
  Y(I)=0
ENDDO
	IF(W)
	DO I=1, N
	  X(I) = X(I) + Y(I)
	  Y(I)=0
	ENDDO
	ELSE
	DO I=1, N
	  X(I) = X(I) + Y(I)

	ENDDO

5.17 Index-set Splitting

A sequential loop with dependence is transformed into two independent parallel loops. A careful selection of split point is required. For instance, in the following example, there are dependencies (flow, anti) between the statements when the index variable hits 50 and 51. So, it is preferred to split the loops at that point which results in two independent loops.

DO I = 1, 100 A(101 - I) = A(I) ENDDO	DO I = 1, 50 A(101 - I) = A(I) ENDDO DO I = 51, 100 A(101 - I) = A(I) ENDDO

5.18 Loop Peeling / Splitting

Loop Peeling / Splitting removes the "special case" iterations from a loop (e.g. removing the first/last iteration of loop into a separate code). For instance, in the following example, the computation on each iteration uses the value of A(1) computed in the first iteration. The loop-carried dependence can be converted into a loop-independent dependence by peeling the first iteration into the loop prologue. Moreover, loop-peeling is also used in order to facilitate vectorization (section 6).

DO I=1, N A(I)= A(I) + A(1) ENDDO	A(1)= A(1) + A(1) DO I=2, N A(I)= A(I) + A(1) ENDDO

5.19 Summary of frequently used loop transformations

Table 5 summarizes some frequently used loop transformations w.r.t their effect on enhancing the fine-, coarse-grained parallelism, and resource pressure on the register files, FUs and local memory.

	Parallelism		Resource pressure at micro- level		
Loop Transformations	fine-grained	coarse-grained	Register Files	FUs	Local Memory
Loop Unrolling	⇧	N	⇧	⇧	⇧

Software Pipelining	⇧	N	⇧	⇧	⇧
Loop Distribution	⇩	⇧	⇩	⇩	⇩
Loop Fusion	⇧	⇩	⇧	⇧	⇧
Loop Tiling	⇧	⇧	⇧	⇧	⇧
N: Neutral					

Table 1: Loop transformations w.r.t their effect on enhancing fine-, coarse-grained parallelism, and resource pressure.

6 Vectorization

Section 3 provides an overview of ASIP architecture features which the ASAM aims to address in its design space exploration phases. One of the most important architectural features, in terms of processing efficiency, is SIMD or vector parallelism, as explained in that section. Subsequently, sub section 4.4 gives a taxonomy of the different vector operations which can be found in such ASIPs. Section 4 also discusses how these operations are being addressed in the source code.

Some partial information about loop transformations for vectorization is already given in the code section 5. Usage of **loop unrolling**, **unroll-and-jam**, **loop interchange**, **strip-mining** for vectorization is explained there with an example code. In this sub section additional information is provided with a special focus on the dependence and alignment issues.

The first example below demonstrates a loop with a single statement. A single- statement loop that carries no dependence can be directly vectorized. This is shown in the right column of the table.

DO I=1, N X(I) = X(I) + C ENDDO	X(1:N) = X(1:N) + C

If a loop carries any loop-carried and/or loop-independent dependency, then appropriate transformations are needed to make it vectorizable. For example, the following code is not vectorizable due to the loop-carried dependency. On each iteration, the sequential version uses a value of X that is computed on the previous iteration.

DO I=1, N X(I+1) = X(I) + C ENDDO	X(2:N+1) = X(1:N) + C (Not vectorizable !)

6.1 Loop distribution & statement reordering

In the following example, the loop carries a loop-carried dependency through S1 to S2, because it stores into A on one iteration and loads from A on the next one. However, the loop distribution operation enables the loop to be vectorized as it can be seen from the example.

DO I=1, N (S1) A(I+1) = B(I) + C (S2) D(I) = A(I) + E ENDDO	DO I=1, N (S1) A(I+1) = B(I) + C ENDDO DO I=1, N (S2) D(I) = A(I) + E ENDDO

	Vectorization: (S1) A(2:N+1) = B(1:N) + C (S2) D(1:N) = A(1:N) + E

The above example demonstrates the forward dependency relation from S1 to S2. If the dependency relation is backward, vectorization also may take place; but it very much depends on the fact if the loop body carries any loop-independent dependencies. In the following example, the dependency relation is backward. However, since there are no loop-independent dependencies between statements, statements can be freely interchanged to have exactly the same statement ordering as the original code presented in the previous example.

DO I=1, N (S2) D(I) = A(I) + E (S1) A(I+1) = B(I) + C ENDDO	Vectorization: (S1) A(2:N+1) = B(1:N) + C (S2) D(1:N) = A(1:N) + E

However, it is not the case for the following example. If there is a backward carried dependence and a loop-independent dependence (through B) between the statements, computations can not be vectorized because the interchange above is illegal.

DO I=1, N (S1) B(I) = A(I) + E (S2) A(I+1) = B(I) + C ENDDO	Not vectorizable.

If the data in the memory is not aligned (locally disjoint), the required data have to be explicitly packet to the registers before execution of a SIMD instruction. Usually many architectures have a support for alignment and packing; however using them is costly in terms of performance (possibly also area and power consumption).

6.2 Loop versioning

This code transformation is applicable if the alignment of the code cannot be assured at a compile-time. Loop versioning introduces extra code in order to check alignment at run-time. On the other hand, the new code increases the code size and introduces overhead of the alignment check.

DO, I=0, N, 1
  A[I] = B[I] + C[I]
ENDDO
	If(A & B & C) is aligned { //SIMD version
	  DO, I=0, N, 4
	  A(I:I+4) = B(I:I+4) + C(I:I+4)
	  ENDDO
	else { //standard version

	DO, I=0, N, 1
	A[I] = B[I] + C[I]
	ENDDO
	}

6.3 Loop peeling for alignment support

These transformations are applied when the loop does not directly start at an alignment boundary, loop peeling is applied to ensure the correct alignment of the data accesses. Those iterations causing the misalignment are peeled off the original loop and build a separate loop.

Vectorization is usually performed in the innermost loop. However, sometimes it is beneficial to perform vectorization on the outer loops. Outer loop vectorization has traditionally been performed by **interchanging** an outer-loop with the innermost loop, followed by vectorizing it at the innermost position. A more direct **unroll-and-jam** approach can be used to vectorize an outer-loop without involving loop interchange, which can be especially suitable for SIMD architectures.

6.4 Classification of loop transformations

Table 2 classifies the previously mentioned loop transformations.

Fine-grained Parallelism	
Transformation Type	Transformations
Preliminary Transformations (to ease the data dependency test and other loop transformations, almost all compiler front-ends support these transformations)	loop normalization, constant propagation, dead code elimination, induction variable substitution, if-conversion etc.
Transformations to Break Dependencies	privatization, scalar and array renaming, scalar expansion, loop unswitching, index-set splitting, loop peeling.
Transformations for Vectorization	loop unrolling, strip-mining, loop distribution, loop interchange, loop versioning, loop peeling, unroll-and-jam, loop tiling.
Transformation for ILP	loop unrolling, software pipelining, loop fusion, unroll-and-jam, loop tiling.
Transformation for Locality	loop fusion, unroll-and-jam, loop tiling, loop interchange.
Coarse-grained parallelism	

loop distribution, loop tiling (coarse grained data parallelism)

Table 2: Classification of loop transformations

7 Compiler pragmas

After loop transformation, the significance of all the loops is known, meaning that the application analysis and DSE tools have produced statistics indicating for each loop whether it is a hot loop, complexity of the loop, loop dependencies, etc.
At that point, it is important to provide this knowledge to the compiler, in order to direct its efforts. This is done through compiler pragmas. For example, simple (i.e. short) hot loops (executed very often) would typically benefit from loop unrolling, which is driven through such a pragma.

The pragmas for ASAM-compliant compilers are identified as follows:

```
#pragma asam <pragma>
```

Pragmas have a specific scope. The scope may cover the whole source file (*module*), a *function*, or a compound/(basic) *block*. In some cases, this depends on the pragma itself. In other cases, it depends on the location of the pragma in the source code. The defined scope of a pragma may limit the locations where they can be placed. The following example shows a few uses and dangers in the pragma scope mechanism:

```
#pragma asam cse // common sub-expression elim. at module scope
int array[12]; // Without this declaration, the above cse pragma
               // would be interpreted as having function scope.
#pragma asam break-conversion // has function scope
void foo(void) {
  // start block 1
  int i=0;
# pragma asam latency=1 // block scope for blocks 1 AND 3!
  for( ; i<10; i++ ) {
    // start of block 2
#   pragma asam pipelining=1 // scope is only block 2.
    Array[i]=i;
  }
  // start of block 3
  array[i++] = 200;
}
```

8 Conclusions

The transformations presented in this section comprise a substantial part of the transformations studied in the literature. Explanation of the aforementioned transformations take [Kenn2001] as a reference resource. The references that are pointed out in that book are also used as a secondary resources, such as ones listed in [Bann1993] and [Bann1994].

It is obvious that characteristics of the application code drive the decision of kind and parameter of transformations to be applied in order to enhance the parallelism of the application. The selected transformation(s), which are appropriate for the given application, among the aforementioned transformations should enable the application of the following three tasks. First of all, these transformations assumed to be sufficient enough to explore and exploit the parallelism space in order to enhance the application parallelism.

Secondly, the transformations that are selected and applied to the application code should provide the necessary information to configure or construct the ASIP architecture for the parallelized code version.

Moreover, the selected transformations should generate the necessary information to HiveCC compiler for efficient mapping and scheduling of the parallelized application software onto parallel architecture.

Therefore, it is believed that the presented transformations are sufficient enough to handle all possible combinations of application features and processor architecture features. Also, it is expected that the presented compiler features are sufficient to support all reported transformations.

9 References

[ASAM6.1] ASAM deliverable 6.1 describes a number of industrial applications, which can serve to validate all three use cases.

[Beko2004] M. Bekooij, *Constraint Driven Operation Assignment for Retargetable VLIW Compilers*, ISBN 90-74445-80-8, Eindhoven University of Technology, 2004, pp. 17-28

[Buyu2005] B.Buyukkurt, Zhi Guo, Walid A. Najjar, *Compiler Optimization for Configurable Accelerators*, ODES'05, 2005, Carnegie Mellon U.

[Kenn2001] Ken Kennedy and John R. Allen. 2001, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

[Bann1993] Utpal Banerjee, *Loop transformations for restructuring compilers - the foundations*. Kluwer 1993: I-XVIII, 1-305

[Bann1994] Utpal Banerjee, *Loop parallelization*. Kluwer 1994: I-XVIII, 1-174

10 Glossary and Terminology

ACN	Argument Connection Network: the HiveLogic IP module instantiated within the processor's datapath to provide a sparsely connected communications facility running from Register file outputs to Issue slot inputs
AHB	AMBA High-performance Bus: bus fabric and interface introduced by ARM Ltd.
AMBA	Advanced Microcontroller Bus Architecture: bus fabric and interface introduced by ARM Ltd.
ANSI-C	Americal National Standards Institute version of the 'C' programming language.
API	Application Programmers Interface
ARM	ARM Ltd.: semiconductor intellectual property provider
ARU	Arithmetic Unit
ASIP	Application-specific Instruction-set Processor
AXI	Advanced eXtensible Interface: third generation of ARM Ltd's bus interface specification
C++	Programming language, based on the 'C' language. SystemC is built on top of C++.
CIO	Silicon Hive's proprietary MMIO interface protocol
CoreBrowser	HiveCC tool providing graphical view of HiveLogic, HiveFlex, and HiveGo processors
CoreIO	Input/output and memory module of HiveLogic processor
DDR	Double DataRate SDRAM
DLP	Data-Level Parallelism: refers to kind of processing where a single operation on a processor handles vectors of processors simultaneously
DMA	Direct Memory Access: In this document, DMA refers to a particular hardwired system block which can autonomously transfer (blocks of) data.
DSP	Digital Signal Processor: may either refer to the kind of software specifically targeted at processing digital signals, or to the type of processors specifically meant to run that software
EDA	Electronic Design Automation
ELF	Executable and Linkable Format for object files and binaries generated by HiveCC
ESL	Electronic System Level: term to qualify tools which deal with SoC-level issues
eVC	e Verification Component: referring to verification IP expressed in the Cadence 'e' language
FU	Function Unit
GeneSys	HiveLogic tool for instantiating and interconnecting IP blocks (either HiveLogic or customer-supplied)
HDL	Hardware Description Language
HiveDB	HiveCC debugger tool
HiveCC	Silicon Hive's Software Development Kit, including ANSI-C compiler
HiveIDE	HiveCC Integrated Development Enironment
HiveLogic	Silicon Hive's configurable parallel processing flow
HiveRT	HRT: Hive Run-Time, application programmers interface for driving and communicating with HiveLogic processors
HRT	HiveRT
HSD	Hive System Description: language for describing multi-core systems
IDE	Integrated Development Environment
ILP	Instruction-Level Parallelism: refers to the kind of instruction processing where a single instruction contains multiple operations. Also refers to the measure of the average number of operations executed in parallel on a VLIW machine, throughout (part of) and application.
IS	Issue Slot
ISP	Image Signal Processor: referring to the HiveFlex 2xxx family of processors
JTAG	Joint Test Action Group; typically describing a serial interface standard allowing test access to SoCs.
Kgate	Kilo-gate, measure of chip logic complexity and area

LSU	Load/Store Unit: Function unit specifically meant for exchanging data between VLIW datapath and memory
MAC	Multipy-Accumulate: the combined operation of multiplying and adding, also refers to the multiply-accumulate function unit within a processor
MIMD	Multiple Instruction Multiple Data: referring to multi-processors which can execute multiple independent parallel operation streams on multiple independent data streams
MMIO	Memory-mapped Input/Output
NCSim	Cadence Incisive toolsuite, meant for verification of ASICs, in particular containing the ncsim unified simulation engine for Verilog, VHDL, and SystemC
OCP	Open Core Protocol: bus interface introduced by the OCP-IP Association
OLP	Operation-Level Parallelism: single operations performing multiple tasks simultaneously which, on a RISC processor, would have taken multiple operations
PC	Program Counter
RPC	Remote Procedure Call
RISC	Reduced Instruction-Set Computer
RF	Register File
RSN	Result Select Network: the HiveLogic IP module instantiated within the processor's datapath to provide a sparsely connected communications facility running from Issue slot outputs to Register file inputs
RTL	Register Transfer Level
SAD	Sum of Absolute Differences
SoC	System-on-Chip
SR	Status Register
SRAM	Static Random Access Memory
SW	Software
System	Top-level collection of hardware components in HSD, often roughly corresponding to the functionality of an SoC. This specifically does not refer to application-level systems consisting of constellations of different processing kernels, such as often associated with MatLab descriptions and Kahn process networks, but rather the hardware on which such systems could be mapped.
Sub-system	Collection of hardware components in HSD, corresponding to some intermediary hierarchy level
SystemC	A set of C++ classes and macros which provide an event-driven simulation kernel
Tcl	Tool Command Language: scripting language used by many EDA tools to automate processes using those EDA tools
TIM	Silicon Hive's proprietary processor description language
UART	Universal Asynchronous Receiver Transmitter
VCS	Synopsys' multicore-enabled functional verification solution (HDL simulator)
Verilog	Hardware description language, used to model electronic systems
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
VLIW	Very Long Instruction Word: generally used to refer to processors which can execute multiple independent operations in parallel.
VLSI	Very Large Scale Integration
VSP	Video Signal Processor: referring to the HiveFlex VSP2xxx family of processors
VSS	Video Sub System: referring to HiveGo VSS3xxx SoC sub-systems
SDK	Software Development Kit
TLP	Thread-Level Parallelism: In Silicon Hive terminology, this refers to the kind of processing where multiple processors within one SoC operate simultaneously.
X86	Abbreviation of the line of Intel processors (and associated instruction-set architectures) which started with the 8086