RC 20505 (90881) 7/18/96
Computer Sciences/Mathematics

IBM Research Report

Dynamic translation of tree-instructions into VLIWs

Jaime H. Moreno
jmoreno@watson.ibm.com

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden • Austin • China • Haifa • Tokyo • T.J. Watson • Zurich

Dynamic translation of tree-instructions into VLIWs

Jaime H. Moreno
jmoreno@watson.ibm.com

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

We describe an approach and its implementation for the dynamic translation of programs represented as sequences of *tree-instructions* into *very-long instruction words (VLIWs)* suitable for execution on wide-issue processors. Tree-instructions are characterized by not reflecting the organization of any processor implementation, whereas VLIWs incorporate implementation-specific features. Tree-instructions are dynamically translated , during instruction cache reloading/accessing, to fit processor implementations constraints. Such a translation is performed without the need for checking dependencies between primitive instructions because tree-instructions consist of operations executable simultaneously. This approach allows executing the same object-code and delivers scalable performance on implementations with varying resources, including superscalar and VLIW-based ones, thus solving one of the most critical objections to the adoption of the VLIW paradigm in general-purpose architectures.

1. Introduction

An important focus of activity regarding computer architectures/micro-architectures in recent years has been the exploitation of instruction-level parallelism (ILP), that is, the ability to simultaneously execute several operations [1-2]. Processors capable of exploiting ILP contain multiple functional units, fetch several instructions per cycle from the instruction cache, and in a given cycle may dispatch multiple operations for execution. Such processors are referred to as *superscalar*, to make the distinction with *scalar* processors which dispatch at most one instruction per cycle [1]. Moreover, when the set of operations issued simultaneously is large (say larger than eight), the implementations are also referred to as *wide-issue superscalar processors*.

Depending on the machine organization, the multiple simultaneous operations executed by a superscalar processor may be an *in-order* or an *out-of-order* sequence of instructions. In the case of an in-order sequence, the corresponding processor has some mechanism to determine the set of adjacent operations that are executable simultaneously, and issues them in the order in which they appear in the program. In contrast, an out-of-order processor has hardware resources for determining the dependencies among successive operations as well as the order in which those operations are executed; that ordering may be different from the order in which instructions appear in the program. Most existing modern (high-performance) processor implementations belong to the out-of-order class [3], though the current highest-frequency ones belong to the in-order type [4].

The complexity arising from hardware reordering for wide-issue machines, as well as recent advances on compiler technology, have been advocating the adoption of in-order wide-issue architectures coupled with aggressive compiler optimization techniques to group independent instructions executable in parallel [2]. Aggressive in-order wide-issue execution leads to *very-long instruction word (VLIW)* processors, wherein scheduling of instructions is done statically by the compiler/programmer [2]; the resulting programs consist of "packets" of independent instructions which can be issued simultaneously for execution without the need for checking dependencies among them.

VLIW originated as an extension of microcode-based techniques [5-6], though the concept has changed drastically since its inception. VLIW implementations have been recently described as "a natural successor to RISC," because complexity is moved from the hardware into the compiler. As stated in [7], "the objective of VLIW is to eliminate the complicated instruction scheduling and parallel dispatch that occurs in most modern microprocessors. In theory, a VLIW processor should be faster and less expensive than a comparable RISC processor."

In spite of the attractiveness of its potential design simplicity, VLIW-based implementations have been perceived as suffering from important limitations when compared to out-of-order issue implementations, such as the need for a powerful compiler, larger object-code size arising from the aggressive scheduling performed by the compiler, larger memory bandwidth and register-file bandwidth, lock-step execution of VLIWs, inability to cope with operations with varying latency, and -perhaps most importantly- binary incompatibility across implementations with varying number of functional units [1] [7] [8].

This report describes a new approach towards achieving scalable object-code compatibility in a processor architecture; the same object-code can be executed in scalar, superscalar and VLIW-based implementations, with wider-issue ones delivering better performance. Programs are represented as *implementation-independent tree-instructions** (i.e., without reflecting the organization of the processor where they are executed), which convey fine-grain parallelism in a straight-forward form. Implementation-dependent features are incorporated while a program is being executed, as part of the instruction cache (I-cache) reloading/accessing (but could also be integrated at other levels of the memory hierarchy). As a result, the simplicity in instruction dispatch logic that is characteristic of VLIW processors is preserved.

We first review the problem of object-code compatibility in wide-issue processors, then summarize the relevant features of the proposed program representation (tree-instructions) and the approach to translate such tree-instructions into VLIWs, and describe an implementation of the translation mechanism which has been verified through a behavioral VHDL model. Performance implications of the proposed approach are given in [11], and a description of an associated simulation environment for its evaluation is given in [12].

2. Object-code compatibility in wide-issue processors

Modern computer architectures exhibit object-code compatibility, that is, the capability to execute the same code on all implementations of the architecture. Although members of a given processor family may differ in their resources and capabilities for dispatching primitive instructions, and object-code may be tuned for the resources available in a specific implementation, all processors can execute the same object code.

VLIW object-code such as that used in [13-15] is explicitly dependent on the features of the implementation (i.e., the number, type and location of the functional units). This has been a conscious decision, under the assumptions that the implementations would be simpler, and the compiler could better exploit the processor resources if it had good knowledge of the corresponding features and limitations. In other words, the separation among *architecture* and *implementation* that has been common practice in processor design for scalar/superscalar implementations was abandoned in those VLIW implementations, in order to simplify and exploit better the hardware by the compiler/programmer.

* The tree-instructions described here are an extension to those used in [9-10], which have fixed size and are intended for parallel execution on a specific VLIW processor (with fixed issue-width). Those tree-instructions and their primitives match exactly the computing capabilities of the processor, including suitable alignment with the processor resources. Consequently, they do not allow executing the same program on processors with different issue-width nor in scalar implementations.

The drawback of VLIW as described above is that it sacrifices object-code compatibility among VLIW-based implementations and with scalar/superscalar implementations. The VLIW object-code explicitly indicates the detailed structure of one specific implementation which is different from others, and/or the VLIW object-code uses features that might not exist in other implementations. The cost associated with porting and maintaining object-code across implementations has limited the acceptability of the VLIW paradigm as part of a processor architecture [1] [7] [8].

Two classes of solutions have been pursued to solve the VLIW object-code compatibility problem: software-based and hardware-based techniques. Software-based approaches (excluding source code recompiling due to its inherent limitations) rely on binary-to-binary (object-code) translation, either with or without hardware support [16-19]; some of these actually correspond to translating object-code among different architectures, but they can be applied to different implementations of the same architecture, including VLIW-based ones.

Of particular relevance is the scheme described in [19], which addresses the translation of code from one VLIW implementation to another by performing object-code rescheduling at page-fault time. This is a promising general technique, capable of coping with any implementation constraint; its limitations are the overhead introduced at page-fault time and the added complexity required to manage rescheduled and non-rescheduled pages. In contrast to earlier solutions, the scheme assumes that the object-code is already an explicit representation of instruction-level parallelism, and is able to exploit that representation accordingly.

A hardware-based solution for VLIW-based processors was proposed in [20], wherein the concept of *delayed split-issue* together with dynamic scheduling hardware are introduced; these characteristics allow using the interlocking and score-boarding techniques known for dynamic scheduling in superscalar processors. In fact, the primary objective of that paper is "to show that dynamic scheduling is as viable for VLIW processors as with more conventional ones." As a result, the object-code generated for one implementation of a VLIW processor family can be executed on an implementation with fewer functional units and/or different latencies for the operations.

Another related class of hardware-based solutions consists of using sequential code as the compatible representation and performing dynamic parallelizing. For example, the *fill-unit* approach originally proposed in [21] and extended in [22] extracts VLIWs from the sequential execution of a program, so that successive executions of the same instructions are performed using the parallel rather than the original sequential version. However, this type of approaches is limited due to the small "window" of instructions analyzed, and the complexity associated to run-time parallelizing.

Since there have been very few proposals on representing a VLIW program without depending on the specific aspects of an implementation, it appears that exposing the details of a VLIW implementation to the compiler/programmer also *requires* that such details be reflected in the object code. We argue that this is not necessary, in spite of the perception to the contrary; in fact, the approach described in the next section provides a solution to this problem.

3. A processor architecture based on tree-instructions

We now describe the features of a processor architecture capable of solving the object-code compatibility problem across implementations of widely varying issue-width, without requiring checking dependencies among primitive operations. In this architecture, a program consists of a sequence of *tree-instructions*, or simply *trees,* each of which corresponds to an *unlimited multiway branch* with multiple branch targets and an *unlimited set of primitive operations* (see Figure 1). All operations and branches are independent and executable in parallel, as a single very-long instruction word. The multiway branch is associated with the internal nodes of the tree, whereas the operations are associated with the arcs. The multiway branch is the result of a set of *binary tests on condition registers*: the left outgoing arc from a tree node corresponds to the false outcome of the associated test, and the right outgoing arc corresponds to its true outcome.

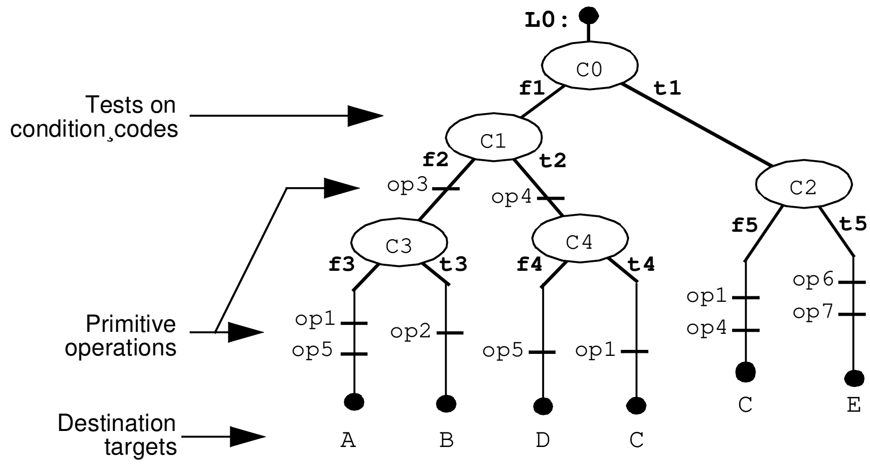

Figure 1: A tree-instruction

Based¸on the evaluation of the multiway branch, a single path within a tree-instruction is selected at execution time as the *taken path* (a tree-path starts from the root of the tree and ends in a branch target). Operations on the taken path are executed to completion, and their results placed in the corresponding target registers or storage locations. In contrast, operations not on the taken path of the multiway-branch are inhibited from committing their results to storage or registers. Such operations do not produce any effect on the architectural state of the processor.

Constraints in tree-paths

The primitive operations in a tree¸are subject to *sequential constraints for each path of the tree.* That is, the final result from executing a tree must be the same as if all primitive operations in the taken path are executed one at a time (sequentially). Consequently, a primitive operation *cannot* use a resource which has been set by a previous operation in the same tree-path (a read¸after write conflict); Figure 2 illustrates legal and illegal tree-instructions. If this constraint is not fulfilled within a tree-instruction, the results from primitive operations having an operand set by a previous operation in the same tree-path are undefined.

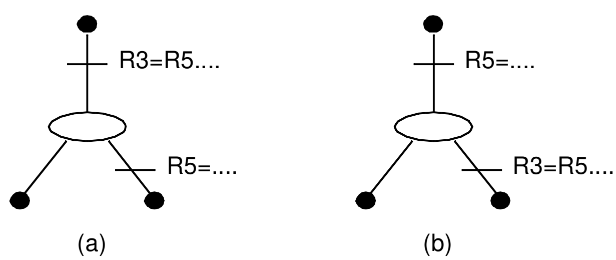

Figure 2: Legal (a) and illegal (b)¸tree-instructions

Memory representation of a tree-instruction

Each tree-instruction is represented in main storage as a *contiguous sequence of tree-chunks* (or simply *chunks),* with each chunk containing the primitive instructions. A chunk is the minimum unit of program specification, and the size of a tree-instruction is a multiple of the size of a chunk. For example, a chunk could correspond just to a single memory word and each primitive operation would be encoded in one memory word; alternatively, a chunk could correspond to several memory words (for example, a quad-word) and several operations could be encoded in such a chunk.*

Each tree-instruction is represented in main storage as a *contiguous sequence of primitive instructions* which is obtained from the *depth-first traversal* of the tree. For example, assuming one-word tree-chunks, the code sequence in Figure 3 represents the 6-way tree-instruction depicted in Figure 1. This representation allows describing arbitrarily complex tree-instructions, without any explicit reference to the resources required in an implementation for their execution.

```
L0: skip C0,t1          t4: op1
f1: skip C1,t2              branch C
f2: op3                 t1: skip C2,t5
    skip C3,t3          f5: op1
f3: op1                     op4
    op5                     branch C
    branch A            t5: op6
t3: op2                     op7
    branch B                branch E
t2: op4
    skip C4,t4          L1: ...
f4: op5
    branch D
```

Figure 3: Representation of a tree-instruction in memory

Testing a condition register is performed with a *skip conditional* instruction, which corresponds to a flow-control operation within the tree, and which indicates where the tree-path corresponding to the true outcome of the test continues in storage; as a result, a skip conditional instruction is a branch with a (short) positive displacement. On the other hand, the branch targets of a tree are represented as *unconditional branch instructions*, which specify the next tree to be executed when the corresponding path of the current tree is selected.

The *end of a tree-instruction* is delimited by a primitive instruction that follows an unconditional branch which is not reachable by any skip conditional instruction within the tree. In the tree-instruction above, for example, the end of the tree is detected at label `L1` because such a label does not appear in any of the skip instructions. This property is used to detect the boundaries of a tree without having to mark such boundaries explicitly.

The program representation illustrated in Figure 3 is directly executable by a scalar/superscalar processor, which can use conventional hardware dependence-checking and scheduling resources to determine which operations can be executed simultaneously. In the case of a wide-issue (a VLIW-based) processor, all oper-

* The chunk size determines trade-offs in memory space and decoding complexity, so its choice its part of the design of the architecture. Furthermore, the architecture could restrict the possible contents of a chunk so that implementation requirements are reduced, thus providing an adequate balance among architecture features and implementation constraints. For example, a chunk could be limited to contain at most two floating-point operations, or there could be fewer operations per chunk than the number of words (say three operations in a four word chunk), or there could be at most three skip instructions (four paths) per chunk, and so on.

ations in the tree can be executed simultaneously; thus, a processor need only detect the end of the tree-instruction without requiring performing dependency analysis among the primitive operations.

Note that each primitive instruction within the tree-instruction in Figure 3 may correspond to the starting point of a different tree. As a result, branching into a tree-instruction leads to the execution of a tree which is a subset of a larger tree. For example, branching into the instruction at label `f2` in Figure 3 leads to the execution of the two-way tree instruction ending at `branch B`. The boundaries of this tree are detected in the same way as described above.

Additional aspects in the representation of tree-instructions are as follows:

- A tree-instruction is stored starting at a chunk boundary (*k* words).
- The right outgoing arc from any internal node of a tree-instruction (the target of any skip instruction) is stored starting at a chunk boundary.
- Unconditional branch instructions appear only as the last operation within a chunk, so there is at most one unconditional branch per chunk.
- As necessary, a tree-instruction is expanded by inserting as many no-op operations as required to achieve the alignment specified in the items above.

Note that these aspects are relevant only for chunks larger than one word. As an example, Figure 4 illustrates the representation of the tree-instruction depicted in Figure 1 in the case of two-words chunks (chunks are separated by horizontal lines). Note that two no-op instructions were added (denoted in uppercase), to comply with the alignment requirements.

```
            L0: skip C0,t1        f4: op5
            f1: skip C1,t2            branch D
            ---------------       ---------------
            f2: op3               t4: op1
                skip C3,t3            branch C
            ---------------       ---------------
            f3: op1               t1: skip C2,t5
                op5               f5: op1
            ---------------       ---------------
added  -->      NOP                   op4           <--  chunk
                branch A              branch C
            ---------------       ---------------
            t3: op2               t5: op6
                branch B              op7
            ---------------       ---------------
            t2: op4                   NOP
                skip C4,t4            branch E
            ---------------       ---------------
                                  L1: ...
```

Figure 4: Representation of a tree-instruction with two-word chunks

Execution of a tree-instruction in a processor with few resources

Sequential constraints in each tree-path make possible *binary compatibility among different implementations* of this architecture, each with varying degrees of parallel execution capabilities, by allowing the decomposition of a large tree into subtrees which are executed in different cycles. If the size of a tree-instruction exceeds the resources available in a processor implementation (such as branching degree, number of fixed-point or floating-point operations), then the tree-instruction is dynamically decomposed *(pruned)* to fit the resources. The resulting subtrees have the same general structure as the original trees (that is, a multiway-branch tree with operations in the tree-paths), but their size is limited. Subtrees are executed in successive cycles, unless the taken path is completely contained within the first subtree.

As an example of the pruning step, consider the execution of the tree-instruction depicted in Figure 1 by a processor capable of executing at most a 4-way branch. Since the tree-instruction contains a 6-way branch, it must be pruned. In this case, pruning takes place at the first two skip instructions, as depicted in Figure 5. As illustrated, the original tree-instruction is decomposed into three smaller subtrees, the first one containing a 4-way branch (targets T1,T2,A,B), and the other two containing 2-way branches (targets D,C and C,E, respectively). A similar decomposition occurs whenever a tree-instruction exceeds any other type of resource (such as arithmetic-logic or load/store units).

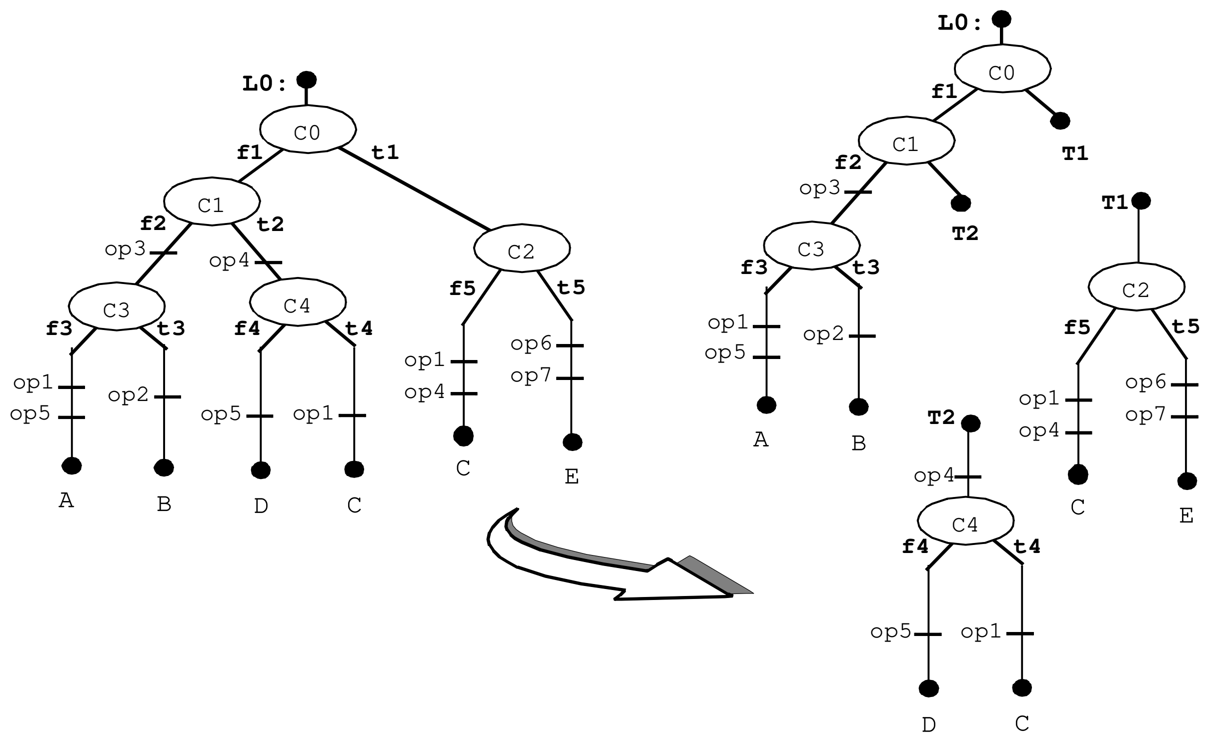

Figure 5: Pruning a tree-instruction

Performance implications of pruning

One objective of this architecture is providing scalable performance at the object-code level; that is, the same object-code should deliver better performance in larger implementations. For this purpose, the architecture assumes that a compiler/programmer generates tree-instructions targeted for the *largest implementation* available, or even larger. That is, the object-code may contain as much instruction-level parallelism as is possible to extract from a program, including speculative and conditional instructions, while introducing minor¸degradation with respect to code recompiled¸for an specific¸implementation.

To support the scalable performance objective, execution paths are assigned to tree paths in left-to-right order, according to their probability of being taken: the most probable path is the left-most one whereas the least probable path is the right-most one. As a result, all operations in the most probable path appear¸in adjacent locations in memory (from `L0` up to `branch A` in Figure 3). Operations are placed on paths other than the left-most one to take advantage of available resources, if they are assumed available.

For implementations with fewer resources than those assumed available when a program was generated, the pruning step automatically eliminates some of the conditional operations (the right-most parts of the trees), if they exist. Consequently, pruning reduces the performance of a program when the number of operations in the taken path exceeds the resources available, or when the taken path is not the most probable one. On the other hand, as long as the taken path is fully contained within the first subtree, no performance degradation occurs because the remaining subtrees are not accessed at all.

However, if a program is generated assuming the availability of p functional units but is executed on a processor implementation with $q > p$ units, then there is neither performance gain nor performance degradation with respect to execution on an implementation with exactly p units.

4. Translation of tree-instructions into VLIWs

As it has already been stated, tree-instructions are dynamically translated into a VLIW form. This is accomplished in a two-step process, as follows:

Pruning step. At instruction cache (I-cache) reload time, trees are decomposed into subtrees whose maximum size fits the capabilities of the processor, as described in the previous section. Information extracted from the trees is added in the I-cache, identifying the starting and ending position of subtrees within each cache line, the number and type of operations per subtree, and so on.

Alignment step. At I-cache access time, the subtree starting at the specified memory address is extracted from the corresponding I-cache line and is aligned to match the resources of the processor. The alignment is based on the extra information encoded in the same I-cache line.

Consequently, VLIW-based implementations of this architecture are different from classical ones, because the instruction cache does not contain VLIWs explicitly; instead, the predecoded information added in the I-cache is used to format the VLIWs as they are fetched from the cache and to decode them in the processor. On the other hand, these implementations fetch exactly as many primitive instructions as the maximum that the processor is capable of executing at once (i.e., a VLIW). If the subtree to be executed contains fewer operations than those fetched from the I-cache, then the alignment/decoding step disables those operations beyond the end of the subtree (as if they were a path that is never taken).

A high-level description of the algorithm used for pruning is given by the pseudo-code depicted in Figure 6. Basically, the procedure consists of fetching a block from main memory and inspecting each primitive instruction in the block. For each instruction, it is added into the current subtree if there are resources available of the corresponding type; otherwise, the tree-instruction is pruned, completing the current subtree. If no pruning occurs, the current subtree is completed when the number of unconditional branch instructions matches the number of paths as determined by the number of skip instructions (each skip instruction adds a path).

```
fetch memory block;
foreach (instruction) loop
 increment_count_type(instruction);
 if    (num_skips  > max_skips)  then prune();
 elsif (num_classA > max_classA) then prune();
 ...
 elsif (num_branches = num_paths) then emit_subtree();
 else
    insert(instruction);
 end if;
```

Figure 6: Pruning algorithm

Figure 7 illustrates the outcome of the pruning step for the tree-instruction listed in Figure 3, assuming a processor capable of executing a 4-way branch and 4 primitive operations per VLIW, with single-word chunks; for simplicity, this example assumes a large I-cache line. As depicted in Figure 7, the original tree is decomposed into four subtrees, indicated by the heavy lines in the figure. Subtree labeled `L0` contains four branches and three¸operations; subtree¸labeled `T3` consists of just one operation and¸a branch; subtree `T2` specifies two branches, one skip instruction and three operations; and subtree labeled `T1` contains one skip, two branches, and four basic operations. Additional bits (shown below the instructions in Figure 7) specify the boundaries of subtrees within the I-cache line, as well as the location of primitive operations, pruning points, and branches within the lines.

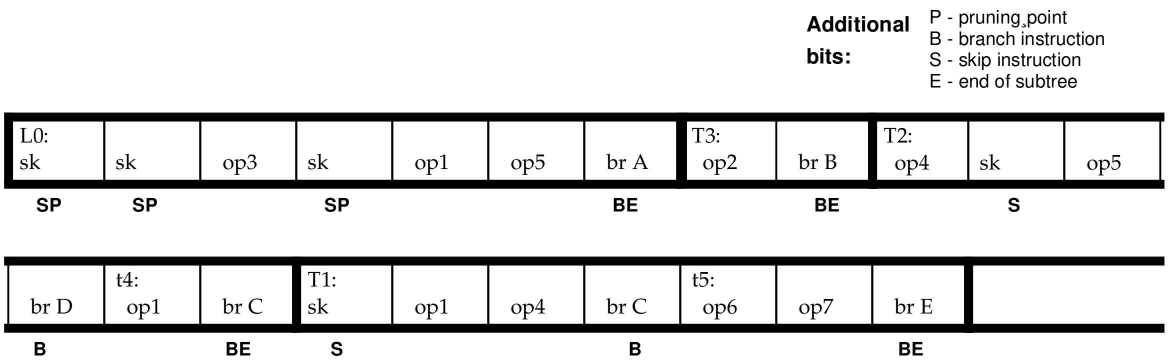

Figure 7: Subtrees in instruction cache

The decomposition of a tree-instruction as outlined above assumes that the total number of operations (including skip and branches) is the same in the original tree-instruction as well as in the resulting subtrees. In other words, the decomposition has serialized the execution of the tree but has not changed the total number¸of operations required (and therefore has not changed¸the number of memory words needed to represent the decomposed version). The underlying assumption is that primitive operations appearing on one arc of the tree appear together in a subtree, so that pruning is performed at skip instructions. Consequently, the starting memory address of each subtree can be extracted from the address specified in the corresponding skip instruction.

However, pruning might be required in the middle of a tree arc (at instructions other than skips). For example, let us assume that the tree-instruction in Figure 1 must be decomposed so that it can be executed by a processor that accepts a maximum of eight primitive operations per VLIW (other than skips and branches). In such a case, as depicted in Figure 8, pruning is performed

- at the skip instruction whose target is labelled `t5;` and¸
- right after the instruction labelled `f5` (say, at instruction labelled `f5b).`

Instructions `t5` and `f5b` become the first operation of the corresponding subtrees. Since `t5` is the target of a skip instruction, pruning at this point is done as described¸earlier. In contrast, pruning at `f5b` is achieved by adding an *implicit unconditional branch instruction* whose target address is the *next sequential memory address*, effectively splitting the arc into two arcs. Since the decomposition is carried out at I-cache reload time, the space required to encode this unconditional branch is needed only inside the I-cache. Furthermore, since the target destination is the next sequential memory address, this branch can be encoded with just a single bit.

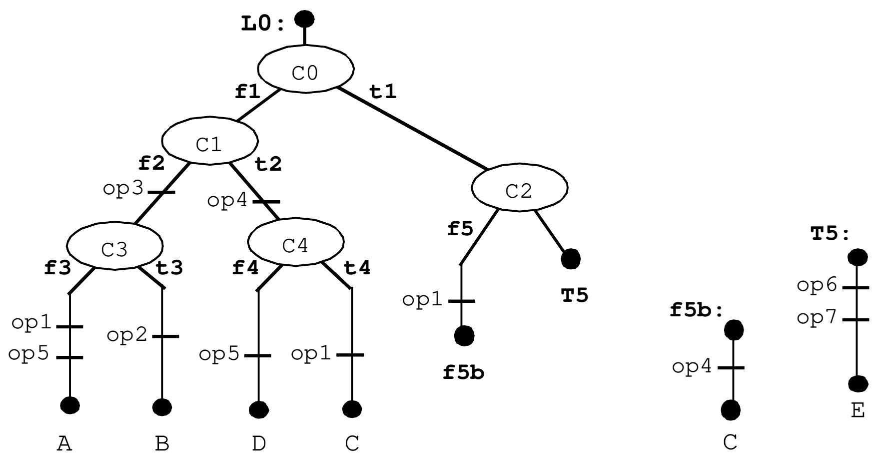

Figure 8: Pruning in the middle of an arc

Processor organization

Figure 9 depicts the overall organization of a VLIW-based implementation of this architecture. Tree-instructions are fetched from main memory and fed into a *pruning unit,* which converts them into subtrees whose requirements fit the resources of the specific processor; the subtrees resulting from the pruning step are stored into the *instruction cache.* A subtree fetched from the instruction cache is aligned by the *fetch unit* and placed in the *VLIW Register*, which feeds the *multiway branch unit* and multiple *fixed-point* and *floating-point units.* The branch unit generates the address for the next tree-instruction, whereas the fixed-point and floating-point units perform the operations and interact with register files and storage.

VLIW execution

The execution of a VLIW contained in the VLIW register consists of the following tasks:

1. evaluating the multiway branch specified in the VLIW and selecting the path taken;
2. executing all operations contained in the VLIW;
3. dispatching to the instruction cache the address of the next VLIW to be executed; and
4. completing all operations in the taken path of the multiway branch, discarding the effects of operations in other paths.

The first two tasks listed above, namely evaluating the multiway branch and executing all the operations in the VLIW, are carried out simultaneously. The evaluation of the multiway branch generates two results:

- the address of the next VLIW to be executed; and
- an *execution mask (EM)* identifying the operations within the VLIW which appear in the taken path, that is, indicating which operations should complete their execution by placing their results in the corresponding destinations, and which operations should be aborted (their results discarded). This mask specifies one bit per operation, which is used to activate the completion of the corresponding instruction.
  As an example, Figure 10 depicts the execution masks for each path of the tree-instruction shown in Figure 3, assuming VLIWs with 24 words and no pruning (i.e., a processor capable of executing the entire tree at once).

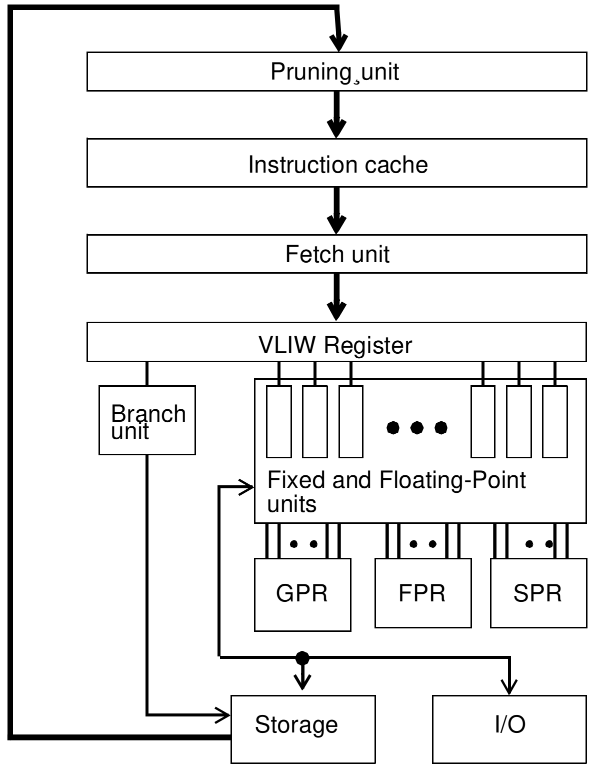

Figure 9: VLIW-based processor organization

Table 1:

Path	Execution mask
L0-A	111111100000000000000000
L0-B	111100011000000000000000
L0-D	110000000111100000000000
L0-C	110000000110011000000000
L0-C	100000000000000111100000

Figure 10: Execution masks for paths in Figure 3

5. Implementation of the translation mechanism

We now address the implementation of the mechanism for dynamically translating tree-instructions into VLIWs. For the purposes of illustration, let us consider an architecture with the following features:

- the chunk size (*k*) is four¸memory words (a quadword, 128 bits); and
- each memory word contains a single primitive instruction (i.e., a chunk contains four¸instructions).

Under these conditions, the tree-instruction depicted in Figure 1 is represented in main memory as shown in Figure 11.

```
      L0: skip C0,t1          t4: op1
      f1: skip C1,t2              NOP
      f2: op3                     NOP
          skip C3,t3              branch C
      ____________________    ____________________
      f3: op1                 t1: skip C2,t5
          op5                 f5: op1                  <--- chunk
added --> NOP                     op4
          branch A                branch C
      ____________________    ____________________
      t3: op2                 t5: op6
          NOP                     op7
          NOP                     NOP
          branch B                branch E
      ____________________    ____________________
      t2: op4                 L1: ...
          skip C4,t4
      f4: op5
          branch D
      ____________________
```

Figure 11: Representation of a tree-instruction with four-word chunks

Furthermore, for simplicity in the description, let us consider a processor implementation with the following characteristics:

- each chunk is allocated entirely to a single subtree, so pruning may introduce at most one implicit branch per chunk (at the last primitive in the chunk);
- each chunk may contain at most three skip instructions (a four-way branch);
- each slot in a chunk may be assigned any operation (other than an unconditional branch, which may only be the last operation in the chunk); and
- the processor resources are a 4-way branch unit, 16 functional units, and seven chunks per VLIW.

Other values for the features of architecture and processor implementation are possible, leading to tradeoffs which are inferred from the description given here.

We describe an implementation of the translation mechanism which processes a complete chunk at a time.* The translation is performed as three steps in the path from main memory to the processor, as depicted in Figure 12, assuming two levels of instruction cache (I2 and I1, with I1 being closer to the processor):

1. A *pruning unit* fetches tree-instructions from main memory, transforms them into subtrees which can be executed with the resources available in the processor, and stores the resulting subtrees into the I2-cache.
2. A *reload unit* fetches subtrees from the I2-cache memory, and places a predecoded form of those subtrees in the first-level instruction cache (the I1-cache). The predecoded subtrees include

* A simpler implementation of pruning is a sequential one, wherein one primitive instruction is examined at a time and the actions indicated in Figure 6 are performed. The detection of the end of a tree, as well as the pruning points, is supported by a stack of tree-paths: each skip instruction pushes the current path into the stack, and each unconditional branch instruction pops the previous uncompleted path from the stack. The end of the tree-instruction is detected when attempting to pop from an empty stack, whereas pruning points are associated to those paths left in the stack when a processor resource is exhausted.

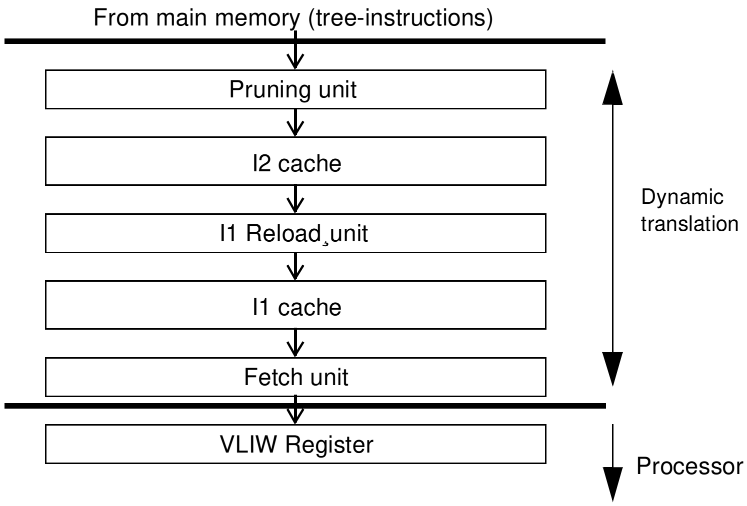

Figure 12: Translation mechanism

information which simplifies the final fetching, alignment, decoding and execution of fixed-length VLIWs.

3. A *fetch unit* extracts subtrees from the I1-cache, formats them as a fixed-length VLIWs, and places them in the VLIW register in the processor.

We first describe the format of subtrees and VLIWs at the different places in the translation process, and then the units which generate the corresponding representations.

5.1 Representation of subtrees

Representation in I2-cache

The format used to represent subtrees in the I2-cache memory is depicted in Figure 13; it consists of

- the chunks fetched from main memory;
- a *k*-bit field per chunk (PR) indicating if skip instructions within the chunk are pruning points;
- a single-bit field per chunk (IB) indicating if an implicit branch has been inserted at the end of the chunk; and
- a single-bit field per chunk (ES) indicating the end of a subtree.

Note that each chunk is augmented with the fields listed above, so the size of the new chunks is larger. Consequently, this representation requires extra bits per chunk in the I2-cache.*

As an example, Figure 14 depicts the representation in I2-cache for the tree-instruction in Figure 11 pruned according to the resources of the processor being considered (in particular, 4-way branch). The notation in Figure 14 is as follows: Ki denotes a skip instruction, Pi denotes an operation, and branches are indicated by the label of their corresponding targets; instructions are identified by a sequence number placed above them. Based on the resource constraints, the PR field of the first chunk has the left-most two bits set to 1 indicating that instructions 1 and 2 are pruning points, and all other PR bits are set to 0. There are no implicit branches inserted in this case. ES fields set to 1 indicate that the original tree-instructions has been decomposed into three subtrees.

* Alternatively, the representation of tree-instructions in main memory could reserve some bits per chunk which would be available for the augmented representation, at the cost of a more restricted encoding space for operations within the chunks.

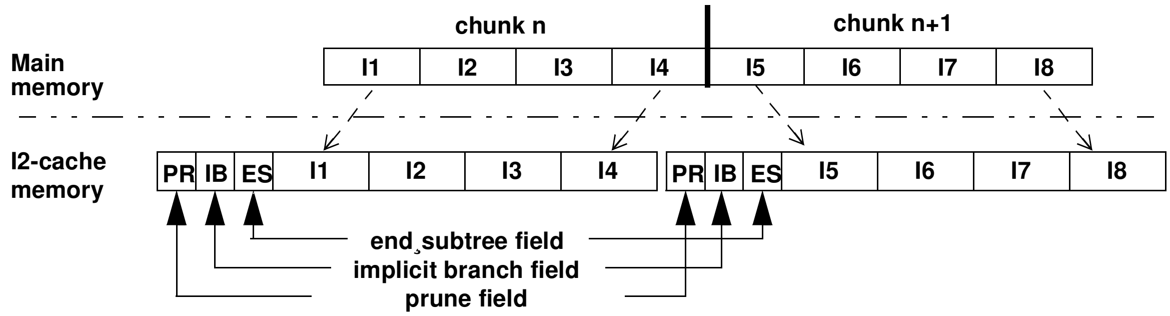

Figure 13: Format of subtrees in I2-cache

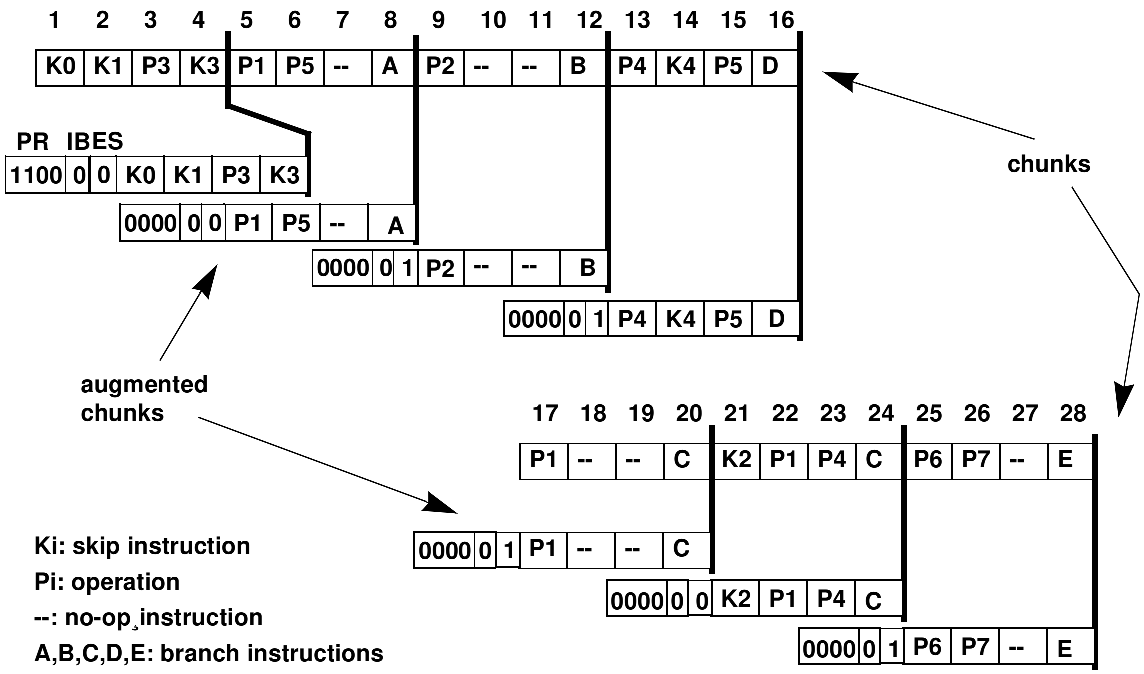

Figure 14: Representation of tree-instruction in I2-cache

Representation in I1-cache

The format of subtrees in the first-level instruction cache is depicted in Figure 15; it consists of

- the augmented chunks fetched from the I2-cache;
- a *c*-bit field per chunk (BC)¸containing the conditions tested by the skip instructions in the chunk;
- a *k*-bit field per chunk (SK) indicating the position of skip instructions within the chunk;
- a single-bit field¸per chunk (BR)¸indicating whether¸the last operation in a chunk is an unconditional branch; and
- a *t*-bit field per chunk (BT)¸containing the target of the unconditional branch or the target of the implicit branch in the chunk, if any.*

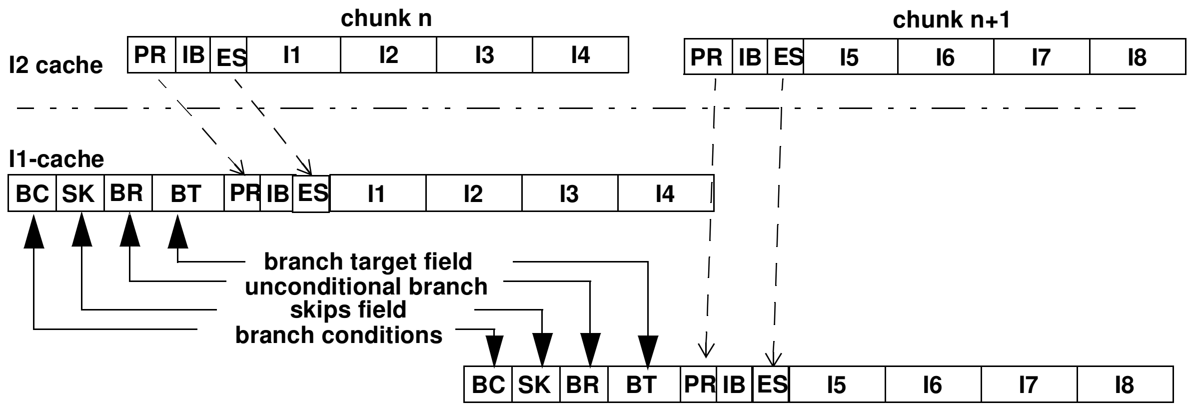

Figure 15: Format of subtrees in I1-cache

As an example, Figure 16¸depicts the representation in the I1-cache of the subtrees shown in Figure 14. For each chunk, fields are shown with their corresponding values. The notation in Figure 16 is as follows: Ci denotes a condition tested by a skip instruction, and Ti denotes the target of a branch instruction.¸

Representation of very-long instruction words

The length of the VLIW register¸and its fields is determined¸by the number¸of operations which the processor can execute in parallel. The format of the VLIW register is depicted in Figure 17; it consists of fields indicating

- the conditions tested by the multiway branch (BC);
- the position of the skip instructions within the VLIW (SK);
- one or more addresses of branch targets referenced by the VLIW (BT);
- the position of explicit and implicit branches within the VLIW (BR, IB);
- the skip instructions which are pruning points (PR);
- the ending chunk of the subtree¸(ES); and
- the operations composing the VLIW.

Fields in the VLIW register are composed by the concatenation of the fields from the different chunks fetched from the I1-cache.

For the processor implementation being considered here (7 chunks, 4-way branch), the VLIW register is 7 chunks long; Figure 18 depicts partial VLIW register contents corresponding to the subtrees in Figure 16. In this case, field BT contains up to two branch targets for each VLIW, which encode the first two targets referenced¸by the VLIW. The first subtree¸is three chunks long, and contains two explicit branches and¸two pruning points. The remaining two subtrees are only two chunks long, with two explicit branches each. In the figure,¸the VLIW register has some contents not specified; as it will be shown later, the VLIW register is loaded with adjacent chunks from the I1-cache, even if they belong to different subtrees.

5.2 Implementation

We now focus on the description of the mechanisms and apparatus to convert among each of the corresponding representations.

* Fields BC and BT contain redundant information (their contents are part of the instructions in the chunk). For simplicity in the description, we assume these fields are replicated; however, this is not required, because the corresponding values can be extracted from the instructions in the chunk.

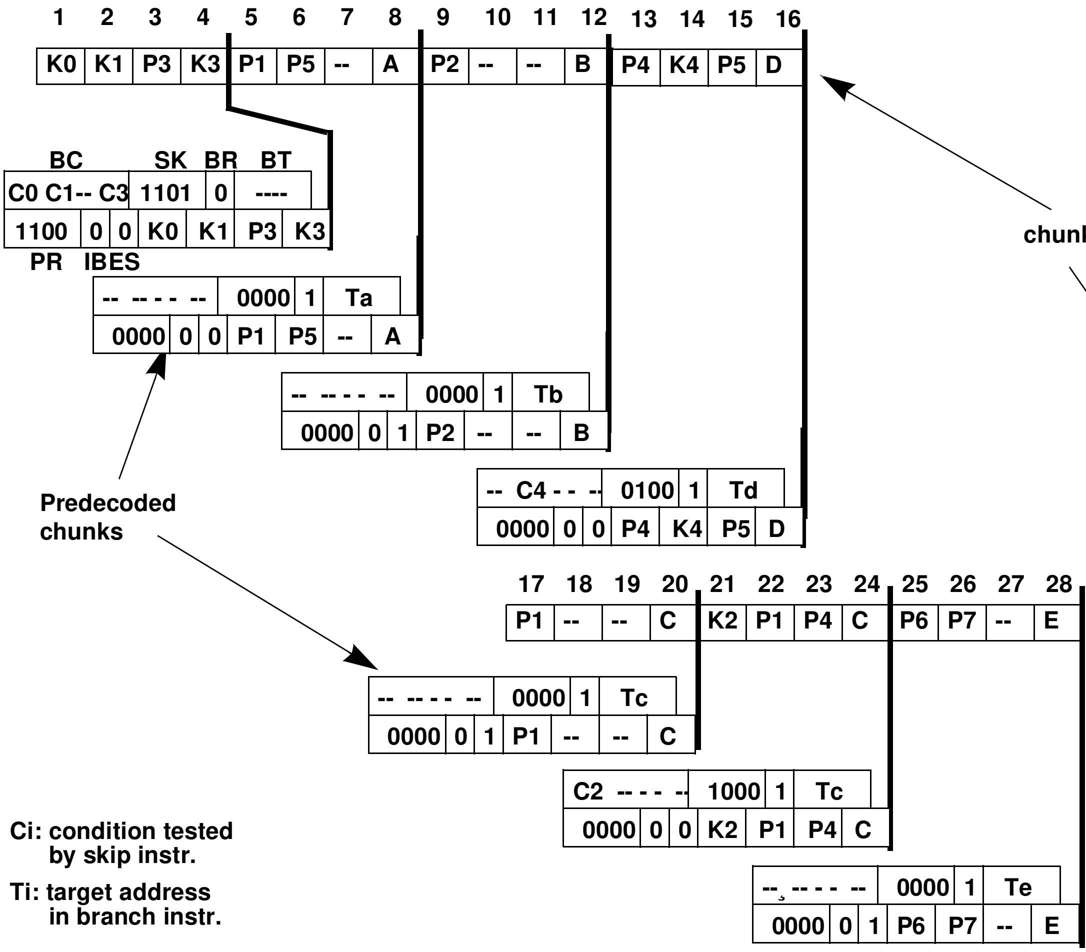

Figure 16: Representation of tree-instruction in I1-cache

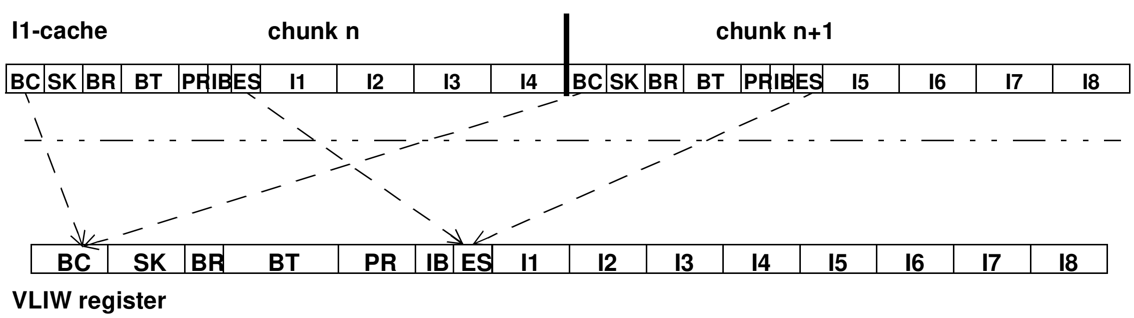

Figure 17: Format of very-long instruction word

Pruning unit

The pruning unit examines one chunk at a time, checking for availability of the resources required by the chunk; this checking is accomplished by counting the number of chunks in the tree, the number of operations, and the number of paths in the multiway branch.

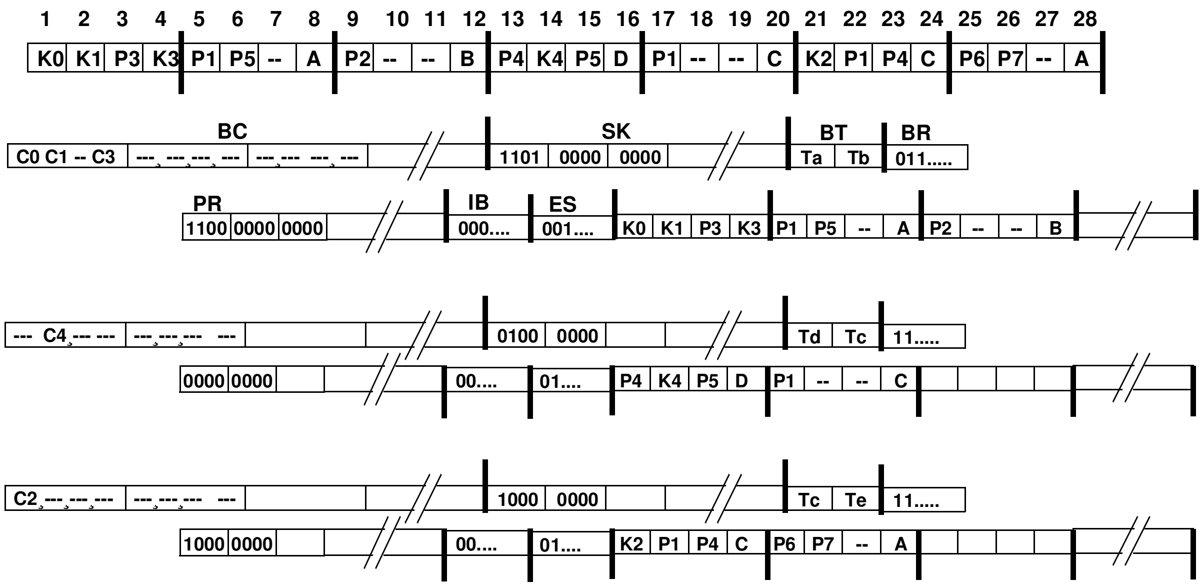

Figure 18: Representation of tree-instruction in VLIW register

Figure 19¸illustrates the mechanism used by the pruning unit, and the fields generated for the first subtree obtained from the tree depicted in Figure 14. An algorithm for this mechanism is given as a behavioral VHDL description in the Appendix. The usage of resources is updated according to the chunk requirements; when any of the counts exceeds its corresponding limit, the unit prunes the tree-instruction by

- setting an implicit branch at the end of the chunk where the resource limit is exceeded or the previous chunk, if that instruction is not a branch;* and¸
- marking each skip instruction whose target has not yet been found by setting to 1 the corresponding bit in field PR.

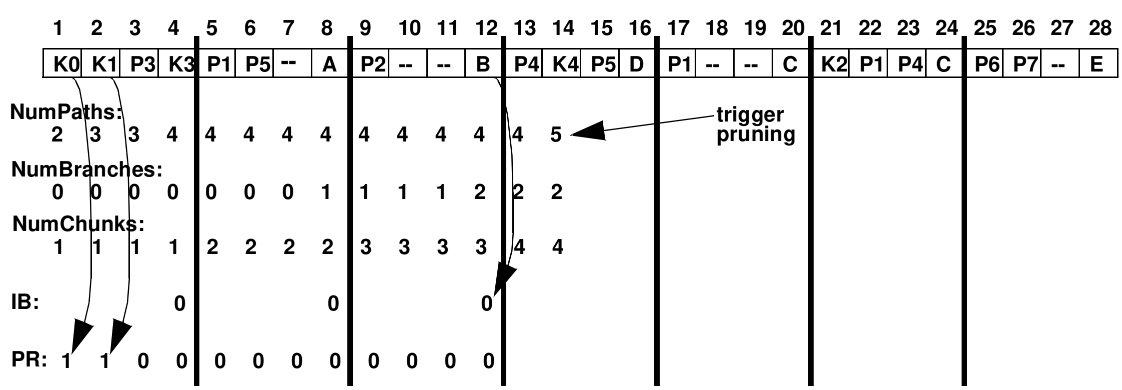

Figure 19: Pruning unit¸mechanism

* The implicit branch is set in the current chunk when pruning is triggered by the last instruction within the chunk; otherwise, the implicit branch is set in the previous chunk.

In the case of the example shown in Figure 19, pruning is triggered at instruction 14 because this skip instruction implies at least five paths in the tree. Since instruction 12 is a branch, no implicit branch is inserted at the end of the previous chunk. Two of the four paths in the first subtree have already been completed because two branch instructions have been processed; consequently, the targets of the left-most two skips (instructions 1 and 2) have not yet been found, so the tree-instruction is pruned at those points by setting the corresponding PR bits to 1.

I1-cache reload unit

The structure of the I1-cache reload unit is described by the block diagram depicted in Figure 20.* This unit performs the following tasks for each chunk:

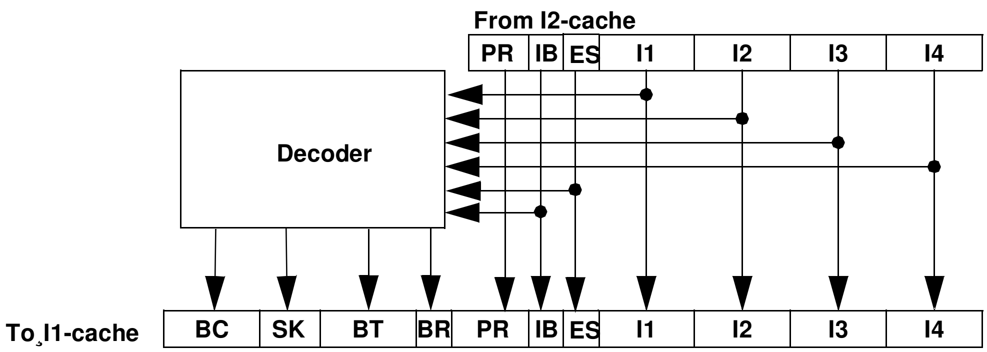

Figure 20: Block diagram of I1-cache reload unit

- determine which instructions within the chunk are skips, set the corresponding bits in field SK, and group the conditions tested by such skip instructions into field BC; and
- determine whether the last instruction in the chunk contains a branch, either explicit or implicit, copy the target address of the branch into field BT, and set bit BR if the branch is explicit.

Fetch unit

Figure 21 illustrates the mechanism used by the fetch unit. This unit fetches subtrees from the I1-cache, and aligns fields from the different chunks so that related fields are placed together at suitable points in the VLIW register. Aligning the fields from the different chunks consists of connecting the fields properly, as inferred from Figure 21. The fields generated as part of the dynamic translation process can be stored in banks other than those containing the associated instructions but are accessed at the same time, as depicted in Figure 21; in this way, aligning is limited only to the region of the processor containing those banks.

5.3 VLIW execution

As stated earlier, the execution of a VLIW is decomposed into concurrent tasks, including the generation by the Branch Unit of the execution mask (EM) and the next VLIW address. These elements are computed with the apparatus depicted in Figure 22. Fields in the VLIW register produced during the translation process are used by the *Mask Generator* to create an execution mask for each path in the VLIW being executed.

* If instruction caches are organized as multiple parallel banks, then multiple chunks can be fetched from the I2-cache and stored in the I1-cache at the same time. In such a case, the reload unit contains multiple copies of the logic depicted in Figure 21 (one for each chunk fetched in parallel), and there can be one BT field per chunk or the BT fields can be combined into one by selecting the most probable branch target among the different chunks.

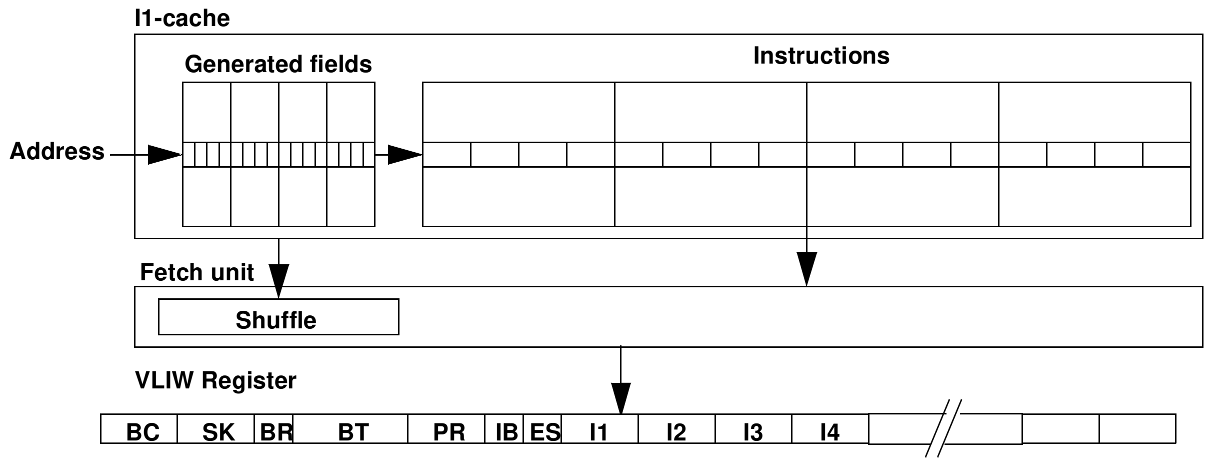

Figure 21: I1-cache structure and fetch unit

The *Branch Evaluator* determines which path is taken, and selects the corresponding EM from those generated by the Mask Generator. Furthermore, the Branch Evaluator selects the address of the next VLIW to be executed.

In practice, the address of the next VLIW can be predicted based on the target addresses contained in the current VLIW, thus saving time in the instruction cache access; this option, which appears in Figure 22, requires fetching a new VLIW whenever the predicted one is not the correct one.

The Branch Evaluator is quite simple: it compares the contents of the Condition Register in the processor with the conditions specified in the BC field. Such a comparison leads to identifying the taken path, and¸to the selection of the corresponding EM and next VLIW address.

The mechanism implemented by the Mask Generator is illustrated in Figure 23; a corresponding algorithm is given as a behavioral VHDL described in the Appendix. Basically, this module uses fields SK, BR, IB, and PR to determine the chunks corresponding to each path in the multiway branch. Note that the same chunk may be part of more than one path. EMs are associated left-to-right with paths in the multiway branch; consequently, EM1 corresponds to the left-most path, EM2 to the next one, and so on.

EM1 has bits set to 1 starting from¸the left end¸of the VLIW, up to the first branch instruction; this path corresponds to the "fall-through" path. Consequently, generating EM1 requires finding the position of the first branch instruction in the VLIW, which is determined from field BR.

The remaining EMs have the following characteristics:

- bits after the position of the last branch instruction already identified and up to the position of the next branch instruction are set to 1; they represent the part of the VLIW corresponding to the target of the skip instruction being processed;
- bits after the position of the skip instruction being processed¸and¸up to the position of the last branch instruction already identified are set to 0; they represent the part of the VLIW corresponding to paths already processed; and
- bits from¸the left end¸and up to the position of the skip instruction selected above are set the same as the last EM generated; they correspond to the portion of the VLIW which is common to this path and the previous one.

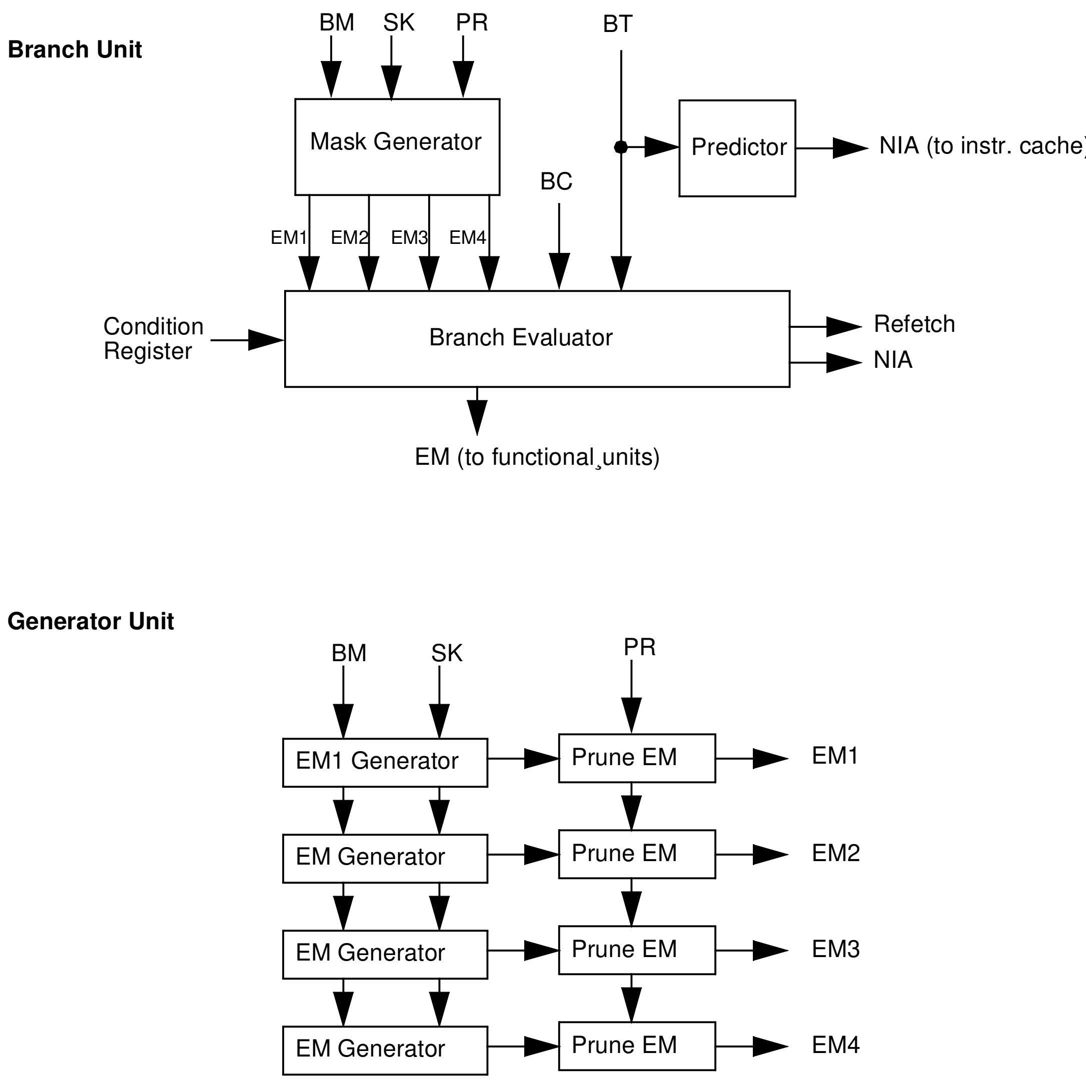

Figure 22: Four-way branch unit

EMs generated in this way do not yet take into account whether there has been pruning. The effect of pruning is incorporated by recognizing that a path which has been pruned ends at the pruning point. Therefore, all bits in the EM after the position of a bit set to 1 in PR must be set to 0. The relationship among bits set to 1 in PR and EMs is in reverse order; that is, the left-most bit set in PR corresponds to the last EM, the next PR bit set corresponds to the EM before the last one, and so on.

Figure 23 illustrates the EMs generated for the first subtree in Figure 18. Since the processor is capable of executing seven chunks, the Fetch Unit loads seven chunks into the VLIW register; since the original tree-instruction has been pruned, these seven chunks correspond to different subtrees. The EM generation unit performs the functionality represented by the behavioral VHDL description given in the Appendix, first generating EMs as if all¸chunks correspond to a single subtree. Then, the information in field¸PR is used¸to cancel EM bits set to 1 beyond the pruning points. The resulting EMs contain bits set to 1 only for instructions 1 to 14, which corresponds to those in the first VLIW.

EMs for the other VLIWs are generated¸in the same manner when those VLIWs are selected¸for execution.

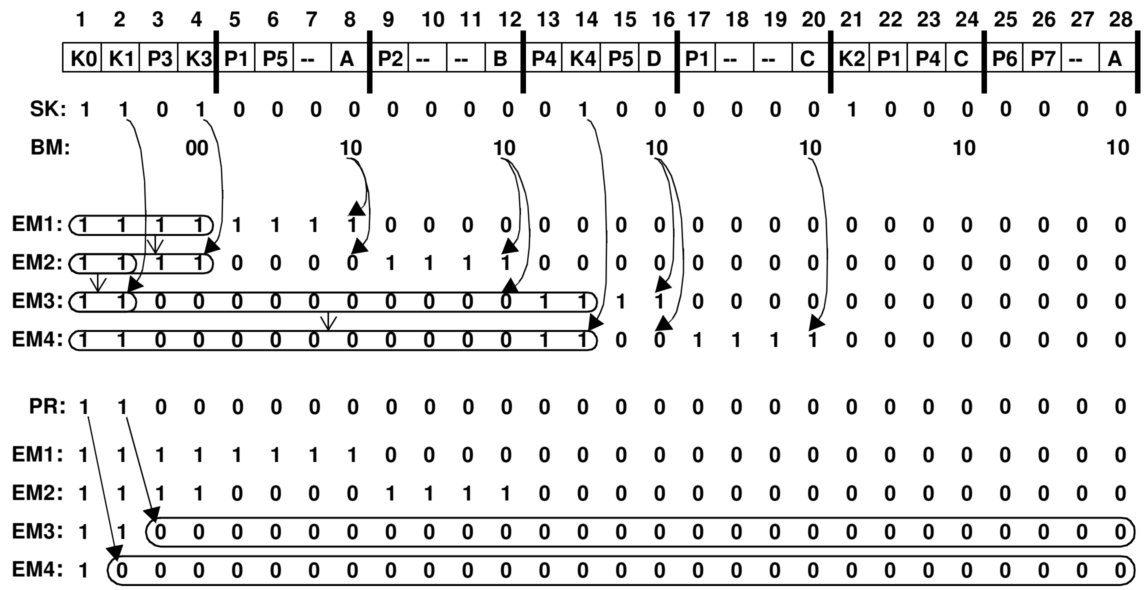

Figure 23: EM generation unit mechanism

6. Concluding remarks

We have presented a mechanism for the dynamic translation of tree-instructions into very-long instruction words suitable for execution on wide-issue processors. Tree-instructions are characterized by not reflecting the organization of any processor implementation, whereas VLIWs incorporate implementation-specific features. Translation is performed during instruction cache reloading/accessing, without the need for checking dependencies between primitive operations because the tree-instructions consist of operations executable simultaneously.

The translation requires detecting the end of tree-instructions, as well as the decomposition of tree-instructions into subtrees. These tasks are accomplished in a straight-forward manner, as a result of the way in which tree-instructions are represented in memory.

The mechanism has been illustrated assuming a particular processor architecture and organization; the implementation of the translation relies on two levels of instruction cache memory. The application to other processor architectures and organizations can be inferred from this one, as also other implementations of the translation mechanism.

The use of tree-instructions and the dynamic translation approach described here allow executing the same object-code on implementations with varying resources, including superscalar and VLIW-based ones, and delivers scalable performance across the different implementations. These properties represent a solution to one of the most critical objections to the adoption of the VLIW paradigm in general-purpose architectures.

6.1 References

[1] J. Hennessy, D. Patterson, *Computer architecture - a quantitative approach,* 2nd. ed.,¸Morgan Kaufmann Publishers, Inc., 1996.

[2] B. Rau, J. Fisher, “Instruction-level parallel processing: history, overview, and perspective,” *The Journal of Supercomputing,* No. 7, pp.9-50, Kluwer Academic Publishers, 1993.

[3] MicroDesign Resources, *Microprocessor Report*, current year issues.

[4] Alpha microprocessors

[5] J.A. Fisher, *The optimization of horizontal microcode within and beyond basic blocks,* Ph.D. Thesis, New York University, NY, 1979.

[6] J.A. Fisher, “Very long instruction word architectures and the ELI-52,” in *Proceedings of 10th Annual International Symposium on Computer Architecture,* pp.140-150, 1983.

[7] L. Gwennap, “VLIW: The wave of the future?,” *Microprocessor Report,* February 14, 1994.

[8] J.S. O’Donell, “Superscalar vs. VLIW,” *Computer¸Architecture News*, Vol. 23, pp. 26-28, March 1995.

[9] K. Ebcioglu, “Some design ideas for a VLIW architecture for sequential natured software,” in *Parallel Processing (Proceedings of IFIP WG 10.3, Working Conference on Parallel Processing),* M. Cosnard et al. (editors), pp. 3-21, 1988.

[10] S-M. Moon, K. Ebcioglu, “An efficient resource-constrained global scheduling technique for¸superscalar and VLIW processors,” *Proceedings of 25th Annual International Symposium on Microarchitecture (MICRO-25),* pp.25-71, December 1992.

[11] J.H. Moreno et al., “Object-code compatible representation of VLIW programs,” submitted to *29th Annual International Symposium on Microarchitecture (MICRO-29)*, 1996.

[12] J.H. Moreno et al., “Simulation/evaluation approach for a VLIW processor,” submitted to *IBM Journal of Research and Development,* 1996.

[13] J.A. Fisher, “Very long instruction word architectures and the ELI-52,” in *Proceedings of 10th Annual International Symposium on Computer Architecture,* pp.140-150, 1983.

[14] R.P. Colwell, R.P. Nix, J.J. O’Donnell, D.B. Papworth and P.K. Rodman, “A VLIW architecture for a trace scheduling compiler,” *IEEE Transactions on Computers,* Vol. C-37, No. 8, pp. 967-979, 1988.

[15] G.R.¸Beck,¸D.W.L.¸Yen and T.L. Anderson, “The Cydra 5¸mini-supercomputer: architecture and implementation,” *The Journal of Supercomputing,* Vol. 7, No. 1/2, pp.143-180, 1993.

[16] R.L. Sites, A. Chernoff, M.B. Kerk, M.P. Marks, and S.G. Robinson, “Binary translation,” *Communications of the ACM*, Vol. 36, pp. 69-81, February 1993.

[17] P. Koch, “Emulating the 68040 in the PowerPC¸Macintosh,” in *Proceedings Microprocessor Forum*, October 1994.

[18] G. Silberman and K. Ebcioglu, “An architectural framework for supporting heterogeneous instruction-set architectures,” *IEEE Computer*, Vol. 26, pp. 39-56, June 1993.

[19] T. Conte, S. Sathaye, “Dynamic rescheduling: a technique for object-code compatibility in VLIW architectures,” in *Proceedings of 28th Annual International Symposium on Microarchitecture (MICRO-28),* 1995.

[20] B.R. Rau, “Dynamic scheduling techniques for¸VLIW processors,” Technical Report HPL-93-52, Computer Research Center, Hewlett-Packard Company, June 1993.

[21] S. Melvin, M. Shebanow, and Y. Patt, “Hardware support for large atomic units in dynamically scheduled¸machines,” in *Proceedings of¸21st Annual International Symposium on Microarchitecture (MICRO-21)*, pp. 60-66, December 1988.

[22] M. Franklin and M. Smotherman, “A fill-unit approach to multiple-instruction issue,” in *Proceedings of 27th Annual International Symposium on Microarchitecture¸(MICRO-27)*, pp. 162-171, December 1994.

7. Appendix - Behavioral VHDL description of units

7.1 VHDL package used by the units

```
-- ------------------------------------------------------------------------
-- PACKAGE
-- ------------------------------------------------------------------------
Package GenEM_pkg is
   Constant ChunkSize: Natural:= 4;
   Constant MaxChunks: Natural:= 7;
   Constant MaxPaths : Natural:= 4;
   Constant MaxBTs   : Natural:= 2;
   Constant VLIWlength: Natural:= MaxChunks*ChunkSize;

   Type¸  StringT    Is Array(Natural Range <>) of Character;
   Type¸  OpCodeT    Is (nop,skp,brn,iop,fop,lop);
   SubType  OpFieldT   Is StringT(0 To 4);
   Type¸  InsT¸     Is Record
¸¸   OpCode : OpCodeT;
¸¸   OpField: OpFieldT;
¸¸End Record;

   Type¸  ChunkT     Is Array(0 To¸ChunkSize-1) Of InsT;
   Type¸  BlockT     Is Array(Natural Range <>) Of ChunkT;
   SubType  PRfieldT   Is Bit_Vector(0 to ChunkSize-1);
   Type¸  AugChunkT  Is Record
¸¸   PRfield : PRFieldT;
¸¸   IBfield : Bit¸   ;
¸¸   Endfield: Bit¸    ;
¸¸   Instrs  : ChunkT  ;
¸¸End Record;

   Type¸  SubTreeT   Is Array(Natural Range <>) Of AugChunkT;
   SubType  Str2T¸     Is StringT(0 to 1);
   Type¸  CCfieldT   Is Array(0 To¸ChunkSize-1) Of Str2T;
   Type¸  ExtChunkT  Is Record
¸¸   CCfield : CCfieldT;
¸¸   SKfield : PRfieldT;
¸¸   BTfield : Str2T   ;
¸¸   BRfield : Bit¸   ;
¸¸   PRfield : PRFieldT;
¸¸   IBfield : Bit¸   ;
¸¸   Endfield: Bit¸    ;
¸¸   Instrs  : ChunkT  ;
¸¸End Record;

   Type¸  ExtSubTreeT Is Array(Natural Range <>)¸of ExtChunkT;
   Type¸  vBCFieldT   Is Array(Natural Range <>) of Str2T;
   Type¸  vBTFieldT   Is Array(Natural Range <>) of Str2T;
   Type¸  vInstrsT    Is Array(Natural Range <>) of InsT ;
   Type¸  VLIWT¸Is Record
¸¸   BCfield : vBCFieldT (0 to VLIWLength-1);
¸¸   SKfield : Bit_Vector(0 to VLIWLength-1);
¸¸   BTfield : vBTfieldT (0 to MaxBTs-1¸);
¸¸   BRfield : Bit_Vector(0 to MaxChunks-1¸);
¸¸   PRfield : Bit_Vector(0 to VLIWLength-1);
¸¸   IBfield : Bit_Vector(0 to MaxChunks-1¸);
¸¸   Endfield: Bit_Vector(0 to MaxChunks-1 );
¸¸   Instrs  : vInstrsT  (0 to VLIWlength-1);
¸¸End Record;
¸SubType emT  Is Bit_Vector(0 To¸VLIWlength-1);
¸Type¸ emsT Is Array(0¸To MaxPaths-1)¸of emT;
End GenEM_pkg;
```

```
-- -------------------------------------------------------------------------
-- PACKAGE BODY
-- -------------------------------------------------------------------------
Package Body GenEM_pkg is
End GenEM_pkg;
-- -------------------------------------------------------------------------
```

7.2 Pruning unit¸

```
Use Work.GenEM_pkg.all;
-- -------------------------------------------------------------------------
-- Entity
-- -------------------------------------------------------------------------
Entity¸Pruner is
   Port¸ (Block_in : In  BlockT(0 To MaxChunks-1);
¸¸Block_out: Out SubTreeT(0 To MaxChunks-1));
End Pruner ;
-- -------------------------------------------------------------------------

-- -------------------------------------------------------------------------
-- Architecture
-- -------------------------------------------------------------------------
Architecture Basic of Pruner¸is
Begin
   Process (Block_in)
-- -------------------------------------------------------------------------
-- Types, Constants, Variables
-- -------------------------------------------------------------------------
   Constant Delay¸     : Time:= 1 ns;
   Variable CntBranches: Natural;
   Variable CntPaths   : Natural:= 1;
   Variable Chunk¸     : ChunkT;
   Variable Subtree    : SubTreeT(0 To MaxChunks-1)¸   ;
   Variable IB¸     : Bit_Vector(0 To MaxChunks-1);
   Variable Done¸     : Boolean;
   Variable lastEnd    : Natural;
-- -------------------------------------------------------------------------
-- Procedures and functions
-- -------------------------------------------------------------------------
-- Procedure PrunePaths(ChunkIndex,InstrIndex: Natural)
--
-- Prune subtree due multiway branching requirements
-- Arguments:
--¸ ChunkIndex: chunk triggering¸pruning
--¸ InstrIndex: instr. triggering pruning (in chunk)
-- -------------------------------------------------------------------------
   Procedure PrunePaths(ChunkIndex,InstrIndex: Natural)
   Is
¸ Variable NumPRs: Natural;
¸ Variable k¸: Natural;
¸ Variable ChIndex : Natural:= ChunkIndex;
   Begin
¸ If (InstrIndex /= ChunkSize -1)¸Then
¸   ChIndex:= ChunkIndex-1;¸ -- prune at end of previous chunk
¸ End If;
¸
¸ Subtree(ChIndex).Endfield:= '1';
¸ lastEnd:= ChIndex;
¸ If (Subtree(ChIndex).Instrs(ChunkSize-1).OpCode /= brn) Then
¸   Subtree(ChIndex).IBfield:= '1';
¸ End If;
¸ NumPRs:= MaxPaths - CntBranches;
¸ k:= 0;
¸ For i in 0 To ChIndex Loop
```

```
¸   For j in 0 to ChunkSize-1 Loop
¸     If (Subtree(i).Instrs(j).OpCode = skp) Then
¸¸Subtree(i).PRfield(j):= '1';
¸¸k:= k+1;
¸¸If (k¸= NumPRs)¸Then Return; End If;
¸     End If;
¸   End Loop;
¸ End Loop;
  End PrunePaths;
-- -------------------------------------------------------------------------

-- -------------------------------------------------------------------------
-- Procedure PruneEnd(ChunkIndex: Natural)
--
-- Prune subtree due to end of block
-- Arguments
--¸ ChunkIndex: first chunk in subtree
-- -------------------------------------------------------------------------
  Procedure PruneEnd(ChunkIndex: Natural)
  Is
¸ Variable NumPRs: Natural;
¸ Variable k¸: Natural;
  Begin
¸ Subtree(MaxChunks-1).Endfield:= '1';
¸ If (Subtree(MaxChunks-1).Instrs(ChunkSize-1).OpCode /= brn)¸Then
¸   Subtree(MaxChunks-1).IBfield:= '1';
¸ End If;
¸ NumPRs:= CntPaths - CntBranches;
¸ k:= 0;
¸ For i in ChunkIndex To MaxChunks-1 Loop
¸   For j in 0 to ChunkSize-1 Loop
¸     If (Subtree(i).Instrs(j).OpCode = skp) Then
¸¸Subtree(i).PRfield(j):= '1';
¸¸k:= k+1;
¸¸If (k¸= NumPRs)¸Then Return; End If;
¸     End If;
¸   End Loop;
¸ End Loop;
  End PruneEnd;
-- -------------------------------------------------------------------------

-- -------------------------------------------------------------------------
-- Main Body
-- -------------------------------------------------------------------------
  Begin
¸   Done¸  := False; lastEnd¸:= 0;
¸   CntPaths := 1    ; CntBranches:= 0;
¸   For i in 0 To MaxChunks-1 Loop
¸     -- default values for chunk
¸     Subtree(i).PRfield := (others => '0');
¸     Subtree(i).IBfield := '0';
¸     Subtree(i).Endfield:= '0';
¸     Subtree(i).Instrs  := Block_in(i);
¸     For j in 0 To ChunkSize-1 Loop¸¸   -- check resources
¸       Case Subtree(i).Instrs(j).OpCode Is¸¸-- required by chunk
¸¸ When skp   => CntPaths   := CntPaths+1;
¸¸ When brn => CntBranches:= CntBranches+1;
¸¸ When others => Null;
¸       End Case;
¸       If (CntPaths > MaxPaths)¸Then
¸¸ PrunePaths(i,j);¸¸     --¸prune if branching exceeded
¸¸ Done:=¸True;
¸¸ Exit;
¸       Elsif (CntBranches = CntPaths) Then
¸¸ Subtree(i).Endfield:= '1';
```

```
¸¸ lastEnd:= i;
¸¸ Done:=¸True;
¸¸ Exit;
¸       End If;
¸     End Loop;
¸     If (Done) Then
¸       Done¸  := False;
¸       CntPaths := 1; CntBranches:= 0;
¸     End If;
¸   End Loop;
¸   If (Subtree(MaxChunks-1).Endfield = '0') Then
¸     PruneEnd(lastEnd+1);¸¸    -- prune if¸end of block
¸   End If;
¸-- deliver results
¸   Block_out <= Subtree After Delay;
   End Process;
End Basic;
```

7.3 I1-cache reload unit

```
Use Work.GenEM_pkg.all;
-- ---------------------------------------------------------------------------
-- Entity
-- ---------------------------------------------------------------------------
Entity¸Reload is
   Port¸ (SubTree_in : In  SubTreeT(0 To MaxChunks-1);
¸¸SubTree_out: Out¸ExtSubTreeT(0 To MaxChunks-1));
End Reload ;

-- ---------------------------------------------------------------------------
-- Architecture
-- ---------------------------------------------------------------------------
Architecture Basic of Reload¸is
Begin
   Process (SubTree_in)
-- ---------------------------------------------------------------------------
-- Types, Constants, Variables
-- ---------------------------------------------------------------------------
   Constant Delay¸ : Time :=¸1 ns;
   Variable Subtree: ExtSubTreeT(0 To MaxChunks-1);
   Variable tempOpField: OpFieldT;
   Variable temp¸     : Str2T;
-- ---------------------------------------------------------------------------
-- Main Body
-- ---------------------------------------------------------------------------
   Begin
¸ If (SubTree_in'Event) Then
¸   For i in 0 To MaxChunks-1 Loop
¸     Subtree(i).CCfield:= (others => "  ");     -- default values
¸     Subtree(i).SKfield:= (others => '0' );
¸     Subtree(i).BTfield:= "  ";
¸     Subtree(i).BRfield:= '0' ;
¸     Subtree(i).PRfield:= Subtree_in(i).PRfield;
¸     Subtree(i).IBfield:= Subtree_in(i).IBfield;
¸     Subtree(i).Endfield:= Subtree_in(i).Endfield;
¸     Subtree(i).Instrs := Subtree_in(i).Instrs¸;
¸     For j in 0 to ChunkSize-1 Loop
¸       Case Subtree(i).Instrs(j).Opcode Is
¸¸ When skp   =>
¸¸   Subtree(i).SKfield(j):= '1';
¸¸   Subtree(i).CCfield(j):= Subtree(i).Instrs(j).OpField(0 to 1);
¸¸ When brn =>
¸¸   Subtree(i).BRfield¸ := '1';
¸¸   Subtree(i).BTfield¸ := Subtree(i).Instrs(j).OpField(0¸to 1);
```

```
¸¸ When others => Null;
¸       End Case;
¸     End Loop;
¸   End Loop;
   -- deliver results
¸   Subtree_out <= Subtree After Delay;
¸ End If;
   End Process;
End Basic;
```

7.4 Alignment unit

```
Use Work.GenEM_pkg.all;
-- ------------------------------------------------------------------------
-- Entity
-- ------------------------------------------------------------------------
Entity¸Fetch is
   Port¸ (ExtSubTree_in: In  ExtSubTreeT(0 To MaxChunks-1);
¸¸VLIW_out     : Out VLIWT);
End Fetch ;

-- ------------------------------------------------------------------------
-- Architecture
-- ------------------------------------------------------------------------
Architecture Basic of Fetch is
Begin
   Process (ExtSubTree_in)
-- ------------------------------------------------------------------------
-- Types, Constants, Variables
-- ------------------------------------------------------------------------
   Constant Delay¸ : Time :=¸1 ns;
   Variable VLIW¸ : VLIWT;
   Variable BTcount: Natural;
   Variable k¸ : Natural;
-- ------------------------------------------------------------------------
-- Main Body
-- ------------------------------------------------------------------------
   Begin
¸   For i in 0 To MaxChunks-1 Loop
¸     For j in 0 To ChunkSize-1 Loop
¸       k:= i*ChunkSize+j;
¸       VLIW.BCfield(k):= ExtSubTree_in(i).CCfield(j);
¸       VLIW.SKfield(k):= ExtSubTree_in(i).SKfield(j);
¸       VLIW.PRfield(k):= ExtSubTree_in(i).PRfield(j);
¸       VLIW.Instrs (k):= ExtSubTree_in(i).Instrs(j)¸;
¸     End Loop;
¸     VLIW.BRfield(i):= ExtSubTree_in(i).BRfield;
¸     VLIW.IBfield(i):= ExtSubTree_in(i).IBfield;
¸     VLIW.Endfield(i):= ExtSubTree_in(i).Endfield;
¸   End Loop;
¸   BTcount:= 0;
¸   For i in 0 to MaxChunks-1 Loop
¸     If (ExtSubTree_in(i).BTfield /= "  ") Then
¸       VLIW.BTfield(BTcount):=¸ExtSubTree_in(i).BTfield;
¸       BTcount:= BTcount+1;
¸       Exit When (BTcount = 2);
¸     End If;
¸   End Loop;
   -- deliver results
¸   VLIW_out <= VLIW after 1 ns;
   End Process;
End Basic;
```

7.5 Branch unit

```
Use Work.GenEM_pkg.all;
-- -------------------------------------------------------------------------
-- Entity
-- -------------------------------------------------------------------------
Entity¸BranchUnit is
   Port¸ (VLIW_in: In  VLIWT     ;
¸¸EM_out¸: Out emsT      ;
¸¸Clk¸: In  Bit¸     );
End BranchUnit¸;

-- -------------------------------------------------------------------------
-- Architecture
-- -------------------------------------------------------------------------
Architecture Basic of BranchUnit is
Begin
   Process (Clk)
-- -------------------------------------------------------------------------
-- Types, Constants, Variables
-- -------------------------------------------------------------------------
   Constant Delay¸: Time := 1 ns;
   Variable zeroSK: PRFieldT;
   Variable EMs: emsT;
   Variable SK : Bit_Vector(0 To VLIWlength-1);
   Variable lastBR, lastSK, nextPR: Natural;
   Variable PathsCount: Natural;
   Variable endVLIW: Natural;
-- -------------------------------------------------------------------------
-- Main Body
-- -------------------------------------------------------------------------
   Begin
¸ If (Clk = '1')¸Then
¸   For i in 0 To MaxPaths-1 Loop¸¸¸-- default values
¸     EMs(i):= (others =>¸'0');
¸   End Loop;
¸   SK:= VLIW_in.SKfield;
¸   For i in 0 to MaxChunks-1 Loop¸     --¸find last chunk¸in VLIW
¸     endVLIW:= i;
¸     Exit When (VLIW_in.Endfield(i) = '1');
¸   End Loop;
¸   PathsCount:= 1;
¸   For i in 0 to endVLIW*ChunkSize-1 Loop     --¸count number of¸paths
¸     If (SK(i) =¸'1') Then
¸       PathsCount:= PathsCount+1;
¸     End If;
¸   End Loop;
¸   For i in 0 to endVLIW¸Loop¸¸     --¸generate EM1
¸     EMs(0)(i*ChunkSize to (i+1)*ChunkSize-1):= (others => '1');
¸     lastBR:= i;
¸     Exit When ((VLIW_in.BRfield(i) OR¸VLIW_in.IBfield(i)) = '1');
¸   End Loop;
¸   For i in 1 To PathsCount-1 Loop¸     --¸generate EM2 to¸EMn
¸¸¸  ¸¸¸¸-- part after last branch
¸     For j in lastBR+1 To MaxChunks-1 Loop
¸       EMs(i)(j*ChunkSize to (j+1)*ChunkSize-1):= (others =>¸'1');
¸       lastBR:= j;
¸       Exit When ((VLIW_in.BRfield(j)¸OR VLIW_in.IBfield(j)) = '1');
¸     End Loop;
¸¸¸  ¸¸¸¸-- part in last branch
¸     For j in lastBR-1 Downto 0 Loop
¸       EMs(i)(j*ChunkSize to (j+1)*ChunkSize-1):= (others =>¸'0');
¸       lastSK:= j;
¸       Exit When (SK(j*ChunkSize to (j+1)*ChunkSize-1) /= zeroSK);
```

```
      End Loop;
        -- find last skip
      For j in (lastSK+1)*ChunkSize-1 Downto lastSK*ChunkSize Loop
        lastSK:= j;
        Exit When (SK(j) = '1');
      End Loop;
        -- common part to last skip
      EMs(i)(0 to lastSK):= EMs(i-1)(0 to lastSK);
    End Loop;
        -- prune EMs
    nextPR:= PathsCount-1;
    For i in 0 to endVLIW*ChunkSize-1 Loop
      If (VLIW_in.PRfield(i) = '1') Then
        EMs(nextPR)(i+1 To VLIWlength-1):= (others => '0');
        nextPR:= nextPR-1;
      End If;
    End Loop;
        -- deliver results
    EM_out <= EMs After Delay;
  End If;
   End Process;
End Basic;
```