Appears in the proceedings of *Computer-Aided Verification 2000 (CAV '00)*, E.A. Emerson and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 296–311.

Formal Verification of VLIW Microprocessors with Speculative Execution[1]

Miroslav N. Velev

mvelev@ece.cmu.edu
http://www.ece.cmu.edu/~mvelev
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract. This is a study of the formal verification of a VLIW microprocessor that imitates the Intel Itanium [9][12][17] in features such as predicated execution, register remapping, advanced and speculative loads, and branch prediction. The formal verification is done with the Burch and Dill flushing technique [5] by exploiting the properties of Positive Equality [3][4]. The contributions include an extensive use of conservative approximations in abstracting portions of the processor and a framework for decomposition of the Boolean evaluation of the correctness formula. The conservative approximations are applied automatically when abstracting a memory whose forwarding logic is not affected by stalling conditions that preserve the correctness of the memory semantics for the same memory. These techniques allow a reduction of more than a factor of 4 in the CPU time for the formal verification of the most complex processor model examined relative to the monolithic evaluation of the correctness formula for a version of the same processor where conservative approximations are not applied.

1 Introduction

VLIW architectures have been adopted recently by microprocessor design companies in an effort to increase the instruction parallelism and achieve higher instructions-per-cycle counts. The goal of this work is to make the Burch and Dill flushing technique [5] scale efficiently and with a high degree of automation for the formal verification of VLIW processors with speculative execution. The speculative features considered include predicated execution, register remapping, advanced and speculative loads, and branch prediction—all of them found in the Intel Itanium [9][12][17], to be fabricated in the summer of 2000.

The focus of this work is on efficient and automatic scaling that is clearly impossible with theorem-proving approaches, as demonstrated by Sawada and Hunt [16] and Hosabettu *et al.* [11]. The former approach was applied to a superscalar processor and required the proofs of around 4,000 lemmas that could be defined only after months, if not a year, of manual work by an expert. The latter work examined the formal verification of a single-issue pipelined and a dual-issue superscalar processors, each of which was formally verified after a month of manual work, with the complexity of the manual intervention increasing with the complexity of the verified design. Clearly, methods that require months of manual work in order to detect a bug will be impractical for the formal verification of wide VLIW designs with many parallel pipelines and specu-

1. This research was supported by the SRC under contract 99-DC-684

lative features that will be fine-tuned constantly during aggressive time-to-market design cycles.

The most complex VLIW processor examined in this paper has 9 parallel execution pipelines, all the speculative features listed above, including branch prediction and multicycle functional units of arbitrary latency. Its exhaustive binary simulation would require more than 2^{609} sequences of 5 VLIW instructions each. However, the proposed techniques allow that processor to be formally verified in less than 8 hours of CPU time on a 336 MHz SUN4.

2 Background

In this work, the logic of Equality with Uninterpreted Functions and Memories (EUFM) [5] is used in order to define abstract models for both the implementation and the specification processors. The syntax of EUFM includes terms and formulas. A term can be an Uninterpreted Function (UF) applied on a list of argument terms, a domain variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, *term*1, *term*2) will evaluate to *term*1 when *formula* = **true** and to *term*2 when *formula* = **false**. A formula can be an Uninterpreted Predicate (UP) applied on a list of argument terms, a propositional variable, an *ITE* operator selecting between two argument formulas based on a controlling formula, or an equation (equality comparison) of two terms. Formulas can be negated and connected by Boolean connectives. The syntax for terms can be extended to model memories by means of the functions *read* and *write*, where *read* takes 2 argument terms serving as memory and address, respectively, while *write* takes 3 argument terms serving as memory, address, and data. Both functions return a term. Also, they can be viewed as a special class of (partially interpreted) uninterpreted functions in that they are defined to satisfy the forwarding property of the memory semantics, namely that *read*(*write*(*mem*, *aw*, *d*), *ar*) = *ITE*(*ar* = *aw*, *d*, *read*(*mem*, *ar*)), in addition to the property of functional consistency. Versions of *read* and *write* that extend the syntax for formulas can be defined similarly, such that the version of *read* will return a formula and the version of *write* will take a formula as its third argument. Both terms and formulas are called expressions.

UFs and UPs are used to abstract away the implementation details of functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*—the same combinations of values to the inputs of the UF (or UP) produce the same output value. Three possible ways to impose the property of functional consistency of UFs and UPs are Ackermann constraints [1], nested *ITE*s [3][4][19], and "pushing-to-the-leaves" [19].

The correctness criterion is based on the flushing technique [5] and is expressed by an EUFM formula of the form

$$m_{1,0} \wedge m_{2,0} \ldots \wedge m_{n,0} \;\vee\; m_{1,1} \wedge m_{2,1} \ldots \wedge m_{n,1} \;\vee \ldots \vee\; m_{1,k} \wedge m_{2,k} \ldots \wedge m_{n,k}\,, \qquad (1)$$

where n is the number of user-visible state elements in the implementation processor, k is the maximum number of instructions that the processor can fetch in a clock cycle, and $m_{i,j}$, $1 \le i \le n$, $0 \le j \le k$, is an EUFM formula expressing the condition that user-visible state element i is updated by the first j instructions from the ones fetched in a

single clock cycle. (See the electronic version of [19] for a detailed discussion.) The EUFM formulas $m_{1,j}$, $m_{2,j}$, ..., $m_{n,j}$, $0 \le j \le k$, are conjuncted in order to ensure that the user-visible state elements are updated in "sync" by the same number of instructions. The correctness criterion expresses a safety property that the processor completes between 0 and k of the newly fetched k instructions.

In our previous work [19] we developed a completely automatic tool that exploits the properties of Positive Equality [3][4], the e_{ij} encoding [8], and a number of conservative approximations in order to translate the correctness EUFM formula (1) to a propositional formula. The implementation processor is *verified*, i.e., is correct, if the propositional formula obtained from (1) after replacing each $m_{i,j}$ with its corresponding propositional formula $f_{i,j}$, obtained after the translation, evaluates to **true**. This evaluation can be done with either BDDs [2] or SAT-checkers. Our previous research [19] showed that BDDs are unmatched by SAT-checkers in the verification of correct processors. However, we found that SAT-checkers can very quickly generate counterexamples for buggy designs.

Positive Equality allows the identification of two types of terms in the structure of an EUFM formula—those which appear only in positive equations and are called *p-terms*, and those which can appear in both positive and negative equations and are called *g-terms* (for general terms). A *positive equation* is never negated (or appears under an even number of negations) and is not part of the controlling formula for an *ITE* operator. A *negative equation* appears under an odd number of negations or as part of the controlling formula for an *ITE* operator. The computational efficiency from exploiting Positive Equality is due to a theorem which states that the truth of an EUFM formula under a maximally diverse interpretation of the p-terms implies the truth of the formula under any interpretation. The classification of p-terms vs. g-terms is done before UFs and UPs are eliminated by nested *ITE*s, such that if an UF is classified as a p-term (g-term), the new domain variables generated for its elimination are also considered to be p-terms (g-terms). After the UFs and the UPs are eliminated, a maximally diverse interpretation is one where: the equality comparison of two syntactically identical (i.e., exactly the same) domain variables evaluates to **true**; the equality comparison of a p-term domain variable with a syntactically distinct domain variable evaluates to **false**; and the equality comparison of a g-term domain variable with a syntactically distinct g-term domain variable could evaluate to either **true** or **false** and can be encoded with a dedicated Boolean variable—an e_{ij} variable [8].

In order to fully exploit the benefits of Positive Equality, the designer of an abstract processor model must use a set of suitable abstractions and conservative approximations. For example, an equality comparison of two data operands, as used to determine the condition to take a branch-on-equal instruction, must be abstracted with an UP in both the implementation and the specification, so that the data operand terms will not appear in negated equations but only as arguments to UPs and UFs and hence will be classified as p-terms. Similarly, a Finite State Machine (FSM) model of a memory has to be employed for abstracting the Data Memory in order for the addresses, which are produced by the ALU and also serve as data operands, to be classified as p-terms. In the FSM abstraction of a memory, the present memory state is a term that is stored in a latch. Reads are modeled with an UF f_r that depends on the present memory

state and the address, while producing a term for the read data. Writes are modeled with an UF f_u that depends on the present memory state, the address, and a data term, producing a term for the new memory state, which is to be stored in the latch. The result is that data values produced by the Register File, the ALU, and the Data Memory can be classified as p-terms, while only the register identifiers, whose equations control forwarding and stalling conditions that can be negated, are classified as g-terms.

We will refer to a transformation on the implementation and specification processors as a *conservative approximation* if it omits some properties, making the new processor models more general than the original ones. Note that the same transformation is applied to both the implementation and the specification processors. However, if the more general model of the implementation is verified against the more general model of the specification, so would be the detailed implementation against the detailed specification, whose additional properties were not necessary for the verification.

Proposition 1. *The FSM model of a memory is a conservative approximation of a memory.*

Proof. If a processor is verified with the FSM model of a memory where the update function f_u and the read function f_r are completely arbitrary uninterpreted functions that do not satisfy the forwarding property of the memory semantics, then the processor will be verified for any implementation of f_u and f_r, including $f_u \equiv write$ and $f_r \equiv read$. ❑

3 VLIW Architecture Verified

The goal of this paper is to formally verify the VLIW processor shown in Fig. 1. On every clock cycle, the Fetch Engine produces a packet of 9 instructions that are already matched with one of 9 functional units: 4 integer (Int FU), 2 floating-point (FP FU), and 3 branch-address (BA FU). There are no data dependencies among the instructions in a packet, as guaranteed by the compiler. Any number of instructions in a packet can be valid, i.e., can have the potential to modify user-visible state. Data values are stored in 4 register files—Integer (Int), Floating-Point (FP), Branch-Address (BA), and Predicate (Pred). A location in the Predicate Register File contains a single bit of data. Every instruction is predicated with a qualifying predicate register, such that the result of the instruction is written to user-visible state only if the qualifying predicate register has a value of 1. The predication is done at compile time.

A Current Frame Marker register (CFM) is used to remap the register identifiers for accessing the Integer, Floating-Point, and Predicate Register Files. The CFM can be modified by every instruction in a packet. Two of the Integer Functional Units can generate addresses for accessing the Data Memory that is used for storing both integer and floating-point values. An Advanced Load Address Table (ALAT) is used as hardware support for advanced and speculative loads—a compile-time speculation. Every data address accessed by an advanced/speculative load is stored in the ALAT and is evicted from there by subsequent store instructions overwriting the same address. Special instructions check if an address, that was accessed by an advanced/speculative load, is still in the ALAT. If not, these instructions perform a branch to code where the load will be repeated non-speculatively together with any computations that have been

performed on the incorrect speculative data. Both the CFM and the ALAT are user-visible state elements. A Branch Predictor supplies the Fetch Engine with one prediction on every clock cycle.

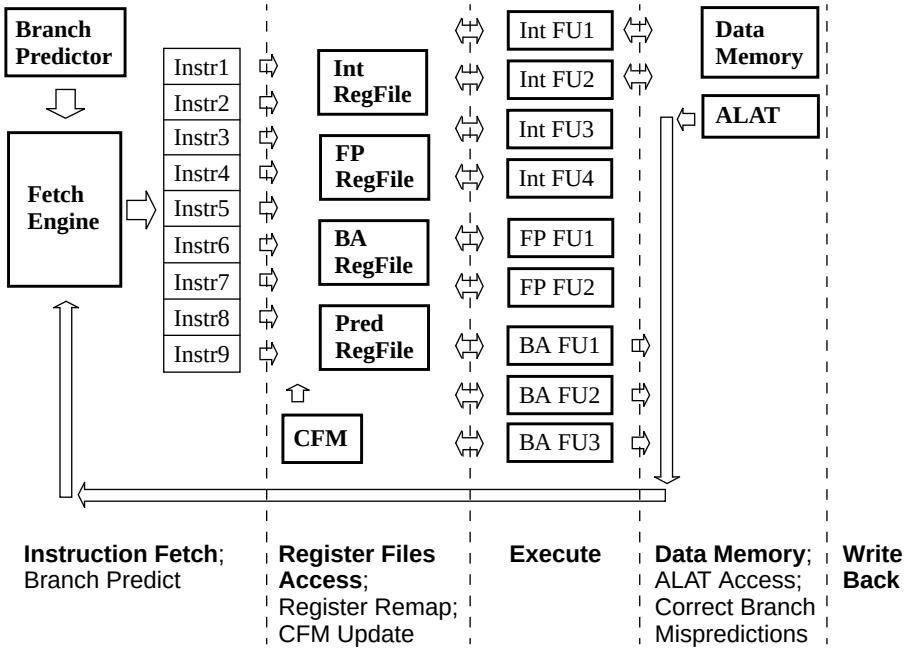

Fig. 1. Block diagram of the VLIW architecture that is formally verified.

The Predicate Register File can be updated with predicate values computed by each of the 4 integer and 2 floating-point functional units. A predicate result depends on the instruction Opcode, the integer or floating-point data operands, respectively, and the value of a source predicate register. The Predicate Register File contents can be overwritten entirely or partially with the value of a data operand, as determined by an instruction Opcode for those 6 functional units. Note that each predicate value consists of a single bit, so that the contents of the Predicate Register File is the concatenation of these bits for all predicate register identifiers. The Predicate Register File contents can be moved to a destination integer register by each of the 4 integer functional units.

Integer values from the Integer Register File can be converted to floating-point values and written to a destination in the Floating-Point Register File by each of the two floating-point functional units. These functional units can similarly convert a floating-point value to an integer one, which gets written to a destination in the Integer Register File. The floating-point functional units also perform floating-point computations on 2 floating-point operands. The 2 floating-point ALUs, the Instruction Memory, and the Data Memory can each have a multicycle and possibly data-dependent latency for producing a result.

The value of a register in the Branch-Address Register File can be transferred to a destination in the Integer Register File by each of the 4 integer functional units. Fur-

thermore, the integer functional units can perform computations for which one of the operands is supplied by the Branch-Address Register File, the other by the Integer Register File, with the result being written to a destination in the Branch-Address Register File. The values in that register file are used by the 3 branch-address functional units for computing of branch target addresses, which also depend on the PC of the VLIW instruction packet and the instruction Opcode for the corresponding functional unit. A branch is taken if its qualifying predicate evaluates to 1.

The VLIW processor that is formally verified has 5 pipeline stages (see Fig. 1): Fetch, Register-Files-Access, Execution, Data-Memory-Access, and Write-Back. The Fetch Engine is implemented as a Program Counter (PC) that accesses a read-only Instruction Memory in order to produce a VLIW packet of 9 instructions. Forwarding is employed in the Execution stage in order to avoid read-after-write hazards for data values read from each of the 4 register files by supplying the latest data to be written to the corresponding source register by the instructions in the Data-Memory-Access and Write-Back stages. However, forwarding is not possible for integer and floating-point values loaded from the Data Memory and used by an instruction in the next packet. In such cases, the hazards are avoided by stalling the entire packet of the dependent instruction(s) in the Register-Files-Access stage and inserting a bubble packet in the Execution stage. Testing a qualifying predicate register for being 1 is done in the Execution stage after forwarding the most recent update for that predicate register.

The updates of the CFM are done speculatively in the Register-Files-Access stage in order not to delay the execution of the instructions in subsequent packets. However, the original value of the CFM is carried down the pipeline together with the packet that modified it. Should that packet be squashed in a later pipeline stage, then its modification of the CFM should not have been done and the original CFM value is restored.

Accounting for all control bits that affect the updating of user-visible state and for all possible forwarding and stalling conditions, an exhaustive binary simulation need consider 2^{609} sequences of 5 VLIW instruction packets each, based on the Burch and Dill flushing technique. This number increases significantly when also considering possible mispredictions or correct predictions of branches, and single-/multi- cycle latencies of the multicycle functional units and memories.

4 Discussion

The above VLIW architecture imitates the Intel Itanium [9][17] in that it has the same numbers and types of execution pipelines, as well as features such as predicated execution, register remapping, advanced and speculative loads, and branch prediction. A further similarity is that the Instruction Memory, the two Floating-Point ALUs, and the Data Memory can each take single or multiple cycles in order to produce their results. The processor also has the same register files as the Itanium, as well as the same capabilities to move data between them. However, two differences should be mentioned.

First, the Register-Files-Access stage is distributed across 3 pipeline stages in the Intel Itanium [9]. While these 3 stages are very simple, their modeling as separate stages resulted in an order of magnitude increase in the CPU time for the formal verification. This was due to an increase in the number of e_{ij} Boolean variables encoding equalities (see Sect. 2) between the then much larger numbers of source and destina-

tion register identifiers for accessing the Integer and Floating-Point Register Files. However, the single Register-Files-Access stage can be viewed as an unpipelined implementation of these 3 stages in the Itanium. It will be the focus of our future research to prove the correctness of a superpipelining transformation that splits that single stage into 3 stages. Alternatively, one can prove the correctness of unpipelining the 3 stages into a single stage. Unpipelining has been previously used for merging stages in a pipelined processor [13] in order to reduce the formal verification complexity, although that was possible only after extensive manual intervention for defining induction hypotheses correlating the signals in the two design versions.

Second, the Itanium Fetch Engine also occupies 3 pipeline stages, the last of which dispatches instructions to the 9 functional units. However, the Fetch stage of the architecture in Fig. 1 can be viewed as an abstraction of a 3-stage Fetch Engine in that it has the same communication mechanism with the Execution Engine (the last 4 stages in Fig. 1) as the Fetch Engine in the Itanium. Namely, the Fetch stage will provide the Execution Engine with the same group of inputs on the next clock cycle if it is stalled by the Execution Engine in the present clock cycle. The refinement of the single Fetch stage into a detailed model of the Itanium Fetch Engine will be the focus of our future research.

On the other hand, the VLIW architecture verified is comparable to, if not more complex than, the StarCore [10] by Motorola and Lucent. The StarCore is a VLIW processor that also has a 5-stage pipeline, consisting of exactly the same stages. It supports predicated execution. However, it has 6 instructions per VLIW packet, does not implement register remapping, does not support floating-point computations (so that it does not have a Floating-Point Register File), does not execute advanced and speculative loads, and does not have branch prediction.

5 Exploiting Conservative Approximations

The CFM and the ALAT were abstracted with finite state machines (FSMs), similar to the Data Memory (see Sect. 2). In the case of the CFM, shown in Fig. 2, the first instruction in a packet modifies the present state via an UF that depends on the CFM present state and the Opcode of the instruction. Each subsequent instruction similarly modifies the latest CFM state obtained after the updates by the preceding instructions in the packet. The final modified CFM state is written to the CFM latch only if the instruction packet is valid. In the case of the ALAT, each valid load or store instruction modifies the present state. The modification is done by an UF that depends on the present state of the ALAT, the instruction Opcode, and the address for the Data Memory access. In this way, an arbitrary update is modeled, including the actual update that takes place only if the Opcode designates an advanced/speculative load or a store instruction. The special check instructions, that check whether the address of an advanced/speculative load has been evicted from the ALAT, are modeled with an UP that depends on the present ALAT state, the instruction Opcode, and the checked address. That UP produces a Boolean signal indicating whether to take a branch to an address provided in the instruction encoding in order to redo the speculative computations done with possibly incorrect data. Because the UFs used in the abstraction of the CFM and the ALAT do not model any actual properties of these units, the FSM models

of the CFM and the ALAT are conservative approximations (see Proposition 1). Note that the ALAT is part of the user-visible state and is used in the non-pipelined specification as it provides support for static compile-time speculation—advanced and speculative loads—as opposed to dynamic run-time speculation that is done only in the implementation processor.

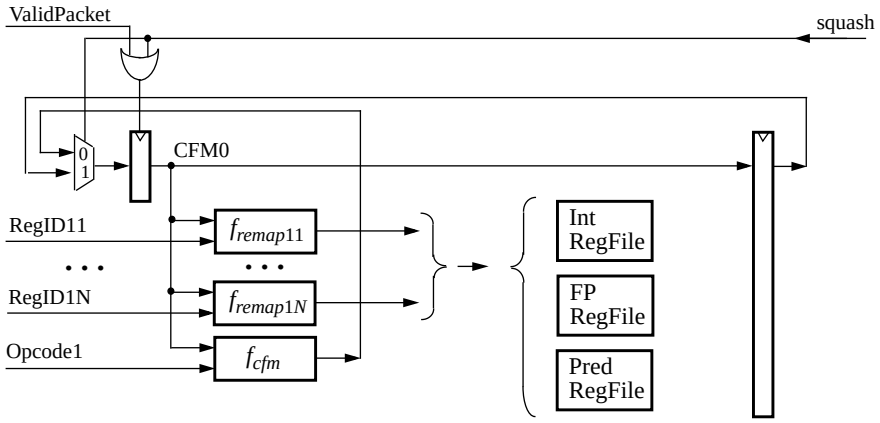

Fig. 2. Abstraction of the CFM. Uninterpreted functions $f_{remap11}$, ..., $f_{remap1N}$ abstract the functional units that remap register ids provided in an instruction encoding into new register ids to be used for accessing a register file. The updates of the CFM are abstracted with UF f_{cfm}. The original value CFM0 is restored if the packet is squashed in a later stage. The remapping and updating UFs are shown for only one instruction. Register remapping is a compile-time optimization.

The Predicate Register File was also abstracted with an FSM, as shown in Fig. 3. The latest updated state of the entire Predicate Register File is available in the Execution stage, because the VLIW architecture is defined to be able to transfer the entire contents of the Predicate Register File to a destination in the Integer Register File.

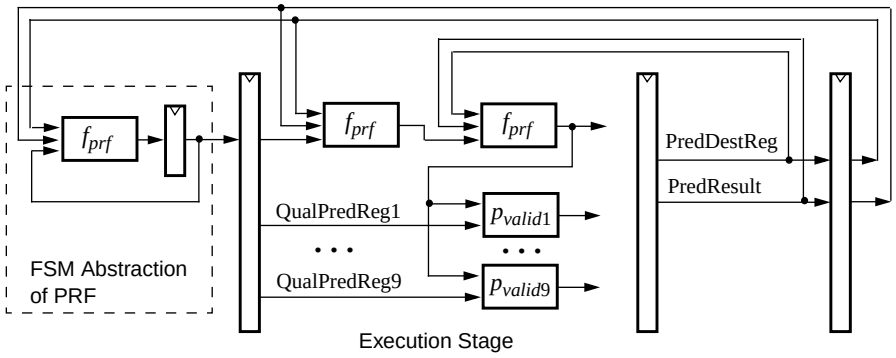

Fig. 3. Abstraction of the Predicate Register File (PRF). Uninterpreted function f_{prf} abstracts the updating of the PRF in both the FSM model of this register file and its forwarding logic. Each predicate result in flight gets reflected on the latest PRF state by one application of f_{prf}. Uninterpreted predicates p_{valid1}, ..., p_{valid9} abstract the testing of a qualifying predicate register for being 1.

Each level of forwarding logic for the Predicate Register File is abstracted with an application of the same UF that is used to update the state of that register file in its FSM abstraction. The value of a qualifying predicate register identifier is tested for being **true**/**false** by means of an UP that depends on the latest Predicate Register File state and the qualifying predicate register identifier. Thus, an arbitrary predicate is modeled, including the actual multiplexor selecting the predicate bit at the location specified by the qualifying predicate register identifier. The instruction Opcode is included as an extra input to that UP in order to achieve partial functional non-consistency [18] of the UP across different instructions. Including extra inputs is a conservative approximation, because if the processor is verified with the more general UP (UF), it would be correct for any implementation of that UP (UF), including the original one where the output does not depend on the extra inputs.

The Branch-Address Register File was abstracted automatically. Originally, it was implemented as a regular memory in the Register-Files-Access stage with regular forwarding logic in the Execution stage. Then, the following transformations were applied automatically by the evaluation tool on the EUFM correctness formula, starting from the leaves of the formula:

$$read(m, a) \rightarrow f_r(m, a) \tag{2}$$

$$write(m, a, d) \rightarrow f_u(m, a, d) \tag{3}$$

$$ITE(e \wedge (ar = aw), d, f_r(m, ar)) \rightarrow f_r(ITE(e, f_u(m, aw, d), m), ar) \tag{4}$$

Transformations (2) and (3) are the same as those used in the abstraction of the Data Memory, described in Sect. 2. Transformation (4) occurs in the cases when one level of forwarding logic is used to update the data read from address *ar* of the previous state *m* for the memory, where function *read* is already abstracted with UF f_r. Accounting for the forwarding property of the memory semantics that was satisfied before function *read* was abstracted with UF f_r, the left handside of (4) is equivalent to a read from address *ar* of the state of memory *m* after a write to address *aw* with data *d* is done under the condition that formula *e* is **true**. On the right handside of (4), functions *read* and *write* are again abstracted with f_r and f_u after accounting for the forwarding property. Multiple levels of forwarding are abstracted by recursive applications of (4), starting from the leaves of the correctness formula. Uninterpreted functions f_u and f_r can be automatically made unique for every memory, where a memory is identified by a unique domain variable serving as the memory argument at the leaves of a term that represents memory state. Hence, f_u and f_r will no longer be functionally consistent across memories—yet another conservative approximation.

After all memories are automatically abstracted as presented above, the tool checks if an address term for an abstracted memory is used in a negated equation outside the abstracted memories, i.e., is a g-term. If so, then the abstractions for all such memories are undone. Hence, abstraction is performed automatically only for a memory whose addresses are p-terms outside the memory and the forwarding logic for it. From Proposition 1, it follows that such an abstraction is a conservative approximation. The fact that address terms of abstracted memories are used only as p-terms outside the abstracted memories avoids false negatives that might result when a (negated) equation of two address terms will imply that a write to one of the addresses will (not)

affect a read from the other address in the equation when that read is performed later—a property that is lost in the abstraction with UFs. In the examined architecture, the Integer and Floating-Point Register Files end up having g-term addresses after abstraction because of the stalling logic that enforces a load interlock avoiding the data hazard when a load provides data for an immediately following dependent instruction. Hence, only the Branch-Address Register File, that is not affected by load interlocks, is abstracted automatically. The Data Memory was abstracted manually with an FSM in order to define its UFs for reading and updating to have the Opcode as an extra input in order to model byte-level memory accesses, as well as to achieve partial functional non-consistency. However, if the Opcode is not used as an extra input and the Data Memory is defined as a regular memory, then the algorithm will automatically achieve conservative approximation of the Data Memory by applying transformations (2) and (3) only.

Note that the abstraction of the Branch-Address and Predicate Register Files helps avoid the introduction of e_{ij} Boolean variables that would have been required otherwise in order to encode the equality comparisons of branch-address and predicate register identifiers, respectively, as would have been needed by unabstracted memories and forwarding logic.

An UF was used as a "translation box" for the new PC value before it is written to the PC latch, similar to our previous work [20]. That UF models an arbitrary modification of the new PC values. Therefore, this transformation is a conservative approximation in that if a processor is verified with a translation box, it would be correct for any implementation of that UF, including the identity function that simply connects the input to the output—the case before inserting the translation box. However, that UF results in common subexpression substitution, reducing the complexity of the PC values, and hence of the final equality comparisons for the state of the PC as used in the correctness formula.

6 Decomposing the Computation of the Correctness Criterion

The problem that was encountered in formally verifying variants of the VLIW architecture presented in Sect. 3 is that the monolithic evaluation of the propositional formula obtained from (1) does not scale well with increasing the design complexity. The solution is to propose a framework for decomposing the monolithic evaluation into a set of simpler evaluations, each of which depends on a subset of the propositional formulas $f_{i,j}$ obtained after the translation of the EUFM formulas $m_{i,j}$ in (1) to propositional logic. Furthermore, these simpler evaluations are not dependent on each other, so that they can be performed in parallel. Some terminology first.

A *literal* is an instance of a Boolean variable or its complement. Any Boolean function that depends on n Boolean variables can be expresses as a sum of products (disjunction of conjuncts) of n literals, called the *minterms* of the function. Equivalently, a Boolean function can be interpreted as the set of its minterms. Then, the disjunction and the conjunction of two Boolean functions are the union and intersection of their minterm sets, respectively. Stating that $a \subseteq b$, where a and b are two Boolean functions, is equivalent to say that the set of minterms of b includes the minterms of a, while $a \subset b$ means that b includes extra minterms in addition to the minterms of a. We

will say that the k+1 Boolean functions, b_0, b_1, ..., b_k, *form a base*, if they cover the entire Boolean space, i.e., $b_0 \vee b_1 \vee ... \vee b_k \equiv$ **true**, and are pair-wise disjoint, i.e., $b_i \wedge b_j \equiv$ **false** for all $i \neq j$, $0 \leq i, j \leq k$. We will refer to functions b_i, $0 \leq i \leq k$, that form a base as *base functions*. The definition of base functions is identical to that of disjoint window functions in Partitioned-ROBDDs [14].

Proposition 2. *If the Boolean functions b_0, b_1, ..., b_k form a base, and $b_0 \wedge a_0 \vee b_1 \wedge a_1 \vee ... \vee b_i \wedge a_i \vee ... \vee b_k \wedge a_k \equiv$ **true**, where a_i, $0 \leq i \leq k$, are Boolean functions, then $b_i \subseteq a_i$ for all i, $0 \leq i \leq k$.*

Proof. The proposition is proved by contradiction. Let $b_j \not\subseteq a_j$ for some j, $0 \leq j \leq k$. Then a_j does not contain at least one minterm of b_j, so that $b_j \wedge a_j \subset b_j$ since the conjunction of two Boolean functions can be interpreted as the intersection of their minterm sets. Furthermore, since $b_i \wedge a_i \subseteq b_i$ for $i \neq j$, $0 \leq i \leq k$, it follows that $b_0 \wedge a_0 \vee b_1 \wedge a_1 \vee ... \vee b_j \wedge a_j \vee ... \vee b_k \wedge a_k \subseteq b_0 \vee b_1 \vee ... \vee b_j \wedge a_j \vee ... \vee b_k \subset b_0 \vee b_1 \vee ... \vee b_j \vee ... \vee b_k \equiv$ **true**. Hence, $b_0 \wedge a_0 \vee b_1 \wedge a_1 \vee ... \vee b_i \wedge a_i \vee ... \vee b_k \wedge a_k \not\equiv$ **true**, which is a contradiction to the assumption in the proposition. ❑

Proposition 3. *If the Boolean functions b_0, b_1, ..., b_k form a base, and $b_0 \vee ... \vee b_{i\text{-}1} \vee a_i \vee b_{i+1} \vee ... \vee b_k \equiv$ **true**, $0 \leq i \leq k$, (i.e., replacing one b_i with a Boolean function a_i does not change the value of the disjunction) then $b_i \subseteq a_i$.*

Proof. The proposition is proved by contradiction. Let $b_i \not\subseteq a_i$. Then a_i does not contain at least one minterm of b_i and since that minterm is not contained in any other b_j, $j \neq i$, $0 \leq j \leq k$, as the base functions are pair-wise disjoint, it follows that $b_0 \vee ... \vee b_{i\text{-}1} \vee a_i \vee b_{i+1} \vee ... \vee b_k \subset b_0 \vee ... \vee b_{i\text{-}1} \vee b_i \vee b_{i+1} \vee ... \vee b_k \equiv$ **true**. Therefore, $b_0 \vee ... \vee b_{i\text{-}1} \vee a_i \vee b_{i+1} \vee ... \vee b_k \not\equiv$ **true**, which is a contradiction to the assumption in the proposition. ❑

Proposition 4. *If the Boolean functions b_0, b_1, ..., b_k form a base, and $b_i \subseteq a_i$, for some i, $0 \leq i \leq k$, where a_i is a Boolean function, then $b_0 \vee ... \vee b_{i\text{-}1} \vee b_i \wedge a_i \vee b_{i+1} \vee ... \vee b_k \equiv$ **true**.*

Proof. If $b_i \subseteq a_i$ then $b_i \wedge a_i \equiv b_i$. Therefore, $b_0 \vee ... \vee b_{i\text{-}1} \vee b_i \wedge a_i \vee b_{i+1} \vee ... \vee b_k \equiv b_0 \vee ... \vee b_{i\text{-}1} \vee b_i \vee b_{i+1} \vee ... \vee b_k \equiv$ **true**. ❑

Propositions 2 – 4 allow us to decompose the evaluation of the propositional formula for the correctness criterion. First, we have to find a set of Boolean functions b_0, b_1, ..., b_k that form a base, where each b_i is selected as the conjunction of a subset of the Boolean functions $f_{1,i}$, $f_{2,i}$, ..., $f_{n,i}$, which are conjuncted together in the propositional translation of (1). Then, for each $f_{j,i}$, $1 \leq j \leq n$, not used in forming b_i, we have to prove that its set of minterms is a superset of the minterms of b_i by using either Proposition 2 or Proposition 3. Then, by Proposition 4, we can compose the results, proving that the monolithic Boolean formula for the processor correctness evaluates to **true** without actually evaluating it.

As an example, let's consider a pipelined processor that can fetch either 0 or 1 new instructions on every clock cycle and that has 3 user-visible state elements—PC, Data Memory (DM), and Register File (RF). In order for the processor to be correct,

we will have to prove that a Boolean formula of the form $PC_0 \wedge RF_0 \wedge DM_0 \vee PC_1 \wedge RF_1 \wedge DM_1$ evaluates to **true**. However, instead of evaluating the monolithic Boolean formula, we can prove that PC_0 and PC_1 form a base, i.e., $PC_0 \vee PC_1 \equiv$ **true** and $PC_0 \wedge PC_1 \equiv$ **false**. Additionally, we can prove that, for example:

1. $PC_0 \wedge DM_0 \vee PC_1 \wedge DM_1 \equiv$ **true**, which implies that $PC_0 \subseteq DM_0$ and $PC_1 \subseteq DM_1$ by Proposition 2;
2. $RF_0 \vee PC_1 \equiv$ **true**, which implies that $PC_0 \subseteq RF_0$ by Proposition 3; and,
3. $PC_0 \vee RF_1 \equiv$ **true**, which implies that $PC_1 \subseteq RF_1$ by Proposition 3.

Then, by Proposition 4, it follows that:

$$PC_0 \wedge RF_0 \wedge DM_0 \vee PC_1 \wedge RF_1 \wedge DM_1 \equiv \textbf{true.}$$

If computing resources are available, we can run a redundant set of computations in parallel, i.e., using multiple sets of Boolean functions as bases, and exploiting both propositions 2 and 3. Note that Proposition 2 can be applied in a version where only one of the base functions b_i, $0 \leq i \leq k$, is conjuncted with a Boolean function a_i by setting $a_j =$ **true** for all $j \neq i$, $0 \leq j \leq k$. Also note that both propositions 2 and 3 hold when a_i, $0 \leq i \leq k$, is the conjunction of several Boolean functions. Furthermore, we can use both BDDs and SAT-checkers. All computations that are still running can be stopped as soon as enough containment properties have been proved, so that they can be composed by Proposition 4 in order to imply that the monolithic Boolean formula for the correctness criterion will evaluate to **true**. Additionally, the simpler Boolean computations will speed up the generation of counterexamples for buggy designs.

Note that the proposed decomposition of the Boolean evaluation of the correctness formula does not require much expertise and can be done automatically. Namely, the Boolean functions that form a base can be selected automatically by choosing such $f_{i,j}$ from the propositional translation of (1) that have the smallest number of Boolean variables before their BDD is built or, alternatively, the smallest number of nodes in their EUFM representation. The base functions serve as automatic case-splitting expressions. In his work, Burch had to manually identify 28 case-splitting expressions [6]. He also had to manually decompose the commutative diagram for the correctness criterion into three diagrams. That decomposition required the intervention of an expert user, and was sufficiently subtle to warrant publication of its correctness proof as a separate paper [21].

7 Modeling Multicycle Functional Units and Branch Prediction

Multicycle functional units are abstracted with "place holders" [20], where an UF abstracts the function of the unit, while a new Boolean variable is produced by a generator of arbitrary values on every clock cycle in order to express the completion of the multicycle computation in the present cycle. Under this scheme, however, a multicycle computation will never finish under the assignments where the Boolean variables expressing completion evaluate to **false**. This is avoided by forcing the completion signal to **true** during the flushing of the processor, based on the observation [6] that the logic of the processor can be modified during flushing as all it does is completing the instructions in flight. A mistake in this modification can only result in a false negative.

The correct semantics of the multicycle functional unit is modeled in the non-pipelined specification only by the UF used to abstract the functionality. The place holder for a multicycle memory is defined similarly, except that a regular memory or an FSM abstraction of a memory is used instead of the UF.

The Branch Predictor in the implementation processor is abstracted with a generator of arbitrary values [20], producing arbitrary predictions for both the taken/not-taken direction (represented with a new Boolean variable) and the target (a new domain variable) of a branch. What is verified is that if the implementation processor updates speculatively the PC according to a branch prediction made in an early stage of the pipeline and the prediction is incorrect as determined when the actual direction and target become available, then the processor has mechanisms to correct the misprediction. The non-pipelined specification does not include a Branch Predictor, which is not part of the user-visible state and is irrelevant for defining the correct instruction semantics. Note that if an implementation processor is verified with completely arbitrary predictions for the direction and target of a branch, that processor will be correct for any actual implementation of the Branch Predictor.

8 Experimental Results

The base benchmark, **9×VLIW**, implements all of the features of the VLIW architecture presented in Sect. 3 except for branch prediction and multicycle functional units. Conservative approximations were used for modeling the Predicate Register File, the CFM, and the ALAT (see Sect. 5). The Branch-Address Register File was abstracted automatically, as described in Sect. 5. In order to measure the benefit of using conservative approximations, 9×VLIW was formally verified with unabstracted Branch-Address Register File that was left as a regular memory with regular forwarding logic—**9×VLIW.uBARF**. The base benchmark was extended with branch prediction in order to create **9×VLIW-BP**, which was then extended with multicycle functional units—the Instruction Memory, the 2 floating-point ALUs, and the Data Memory—in order to build **9×VLIW-BP-MC**. These models were verified against the same non-pipelined VLIW specification, whose semantics was defined in terms of VLIW instruction packets. Because of that, there was no need to impose constraints that the instructions in a packet do not have data dependencies between each other.

The VLIW implementation processors were described in 3,200 – 3,700 lines of our HDL that supports the constructs of the logic of EUFM, while the non-pipelined VLIW specification processor was described in 730 lines of the same HDL. This HDL is very similar to Verilog, so that the abstract description of the implementation processor can be viewed as a level in a hierarchical Verilog description of the processor. At that level, functional units and memories are left as modules (black boxes), while the control logic that glues these modules together is described entirely in terms of logic gates, equality comparators, and multiplexors (the constructs of EUFM). Then, based on information provided in the description and identifying whether a module is either a single-cycle or multicycle functional unit or memory, each of the modules can be automatically replaced by an UF, or an abstract memory, or a place holder for a multicycle functional unit or memory. Similarly, modules like the ALAT (see Sect. 5) can be automatically replaced with an FSM-based abstraction. Hence, the abstract description of

the implementation processor can be generated automatically.

The results are presented in Table 1. The experiments were performed on a 336 MHz Sun4 with 1.2 GB of memory. The Colorado University BDD package [7] and the sifting BDD variable reordering heuristic [15] were used to evaluate the final propositional formulas. Burch's controlled flushing [6] was employed for all of the designs.

Processor	Correctness Computation		BDD Variables			Max. BDD Nodes $\times 10^6$	Memory [MB]	CPU Time [h]	CPU Time Ratio
			e_{ij}	Other	Total				
9×VLIW.uBARF	**monolithic**		1,816	236	2,052	4.8	122	13.68	2.6
	decomposed	sufficient	1,176	174	1,350	2.9	77	5.28	
		max.	1,194	170	1,364	3.3	90		
9×VLIW	**monolithic**		1,468	236	1,704	1.6	53	6.26	3.58
	decomposed	sufficient	1,176	174	1,350	1.2	45	1.75	
		max.	1,194	170	1,364	1.9	57		
9×VLIW-BP	**monolithic**		1,886	246	2,132	1.8	59	9.45	3.72
	decomposed	sufficient	1,191	175	1,366	1.3	43	2.54	
		max.	1,209	175	1,384	9.0	200		
9×VLIW-BP-MC	**monolithic**		2,356	259	2,615	6.8	163	31.57	3.97
	decomposed	sufficient	1,469	214	1,683	2.5	74	7.96	
		max.	1,469	214	1,683	4.2	97		

Table 1. BDD-based Boolean evaluation of the correctness formula. The BDD and memory statistics for the decomposed computations are reported as: "sufficient"—the maximum in each category across the computations that were sufficient to prove that the monolithic Boolean formula is a tautology; "max."—the maximum in each category across all computations run. The CPU time reported for the decomposed computations is the minimum that was required for proving a sufficient set of Boolean containment properties that would imply the monolithic Boolean correctness formula is a tautology.

Decomposition of the Boolean evaluation of the correctness formula accelerated almost 4 times the verification of the most complex benchmark, 9×VLIW-BP-MC. Furthermore, the CPU time ratio would have been much higher than 3.97 if it were calculated relative to the CPU time for the monolithic verification of a version of that processor where conservative approximations are not used to abstract the Branch-Address and Predicate Register Files and their forwarding logic. Indeed, the CPU time ratio for the monolithic verification of 9×VLIW.uBARF (where only the Branch-Address Register File is not abstracted) vs. the CPU time for the decomposed verification of 9×VLIW is 7.86. Note that with faster new computers, the CPU times for the formal verification will decrease, while the CPU time ratio of monolithic vs. decomposed computation can be expected to stay relatively constant for the same benchmarks.

Although decomposition always reduced the number of BDD variables—with up to 35%, compared to the monolithic evaluation—some of the simpler computations required almost 5 times as many BDD nodes and more than 3 times as much memory

as the monolithic computation. This was due to differences in the structures of the evaluated Boolean formulas, some of which proved to be difficult for the sifting dynamic BDD variable reordering heuristic. However, the computations that were sufficient to prove that the monolithic formula is a tautology always required fewer resources than the monolithic computation.

The examined processors had 8 user-visible state elements: PC; 4 register files—Integer, Floating-Point, Predicate, and Branch-Address; CFM; ALAT; and Data Memory. The Boolean formulas that form a base were selected to be the ones from the equality comparisons for the state of the CFM, which has a relatively simple updating logic (see Sect. 5). Proving that these functions form a base took between 6 seconds in the case of 9×VLIW and 54 seconds in the case of 9×VLIW-BP-MC, while the number of BDD nodes varied between 10,622 and 52,372, and the number of BDD variables was between 255 and 411.

The relatively simple Boolean formulas formed from the equality comparisons for the state of the ALAT were used in order to perturb the structure of the evaluated Boolean formulas. Specifically, instead of proving only that $CFM_i \subseteq f_i$ for some i (i.e., the base function CFM_i is contained in f_i), where f_i is the Boolean formula formed from the equality comparison for the state of some user-visible state element, a simultaneous proof was run for $CFM_i \subseteq f_i \wedge ALAT_i$. This strategy resulted in additional Boolean formulas with slightly different structure, without a significant increase in the number of Boolean variables. Note that the number of Boolean variables in a formula is not indicative of the time that it would take to build the BDD for the formula. Sometimes the structural variations helped the sifting heuristic for dynamic BDD variable reordering to speed up the BDD-based evaluation. Applied only for Proposition 2, this strategy reduced the formal verification time for 9×VLIW-BP-MC with around half an hour—from 8.44 hours, that would have been required otherwise, to 7.96 hours.

Decompositions based on Proposition 2 usually required less time to prove the same containment properties, compared to decompositions with Proposition 3. However, Proposition 2 alone would have required 9.64 hours in order to prove a sufficient set of containment properties for the most complex model, 9×VLIW-BP-MC. Given sufficient computing resources, a winning approach will be to run a large set of computations in parallel, exploiting different structural variations in the Boolean formulas in order to achieve a performance gain for the BDD variable reordering heuristic.

9 Conclusions

A VLIW microprocessor was formally verified. Its Execution Engine imitates that of the Intel Itanium [9][16], while its Fetch Engine is simpler. The modeled features are comparable to, if not more complex than, those of the StarCore [10] microprocessor by Motorola and Lucent. Efficient formal verification was possible after an extensive use of conservative approximations—some of them applied automatically—in defining the abstract implementation and specification processors, in addition to exploiting a decomposed Boolean evaluation of the correctness formula.

Acknowledgments

The author thanks his advisor, Prof. Randy Bryant, for comments on the paper.

References

[1] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.

[2] R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," ACM Computing Surveys, Vol. 24, No. 3 (September 1992), pp. 293-318.

[3] R.E. Bryant, S. German, and M.N. Velev, "Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions,"[2] *Computer-Aided Verification (CAV'99)*, N. Halbwachs and D. Peled, eds., LNCS 1633, Springer-Verlag, June 1999, pp. 470-482.

[4] R.E. Bryant, S. German, and M.N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic,"[2] Technical Report CMU-CS-99-115, Carnegie Mellon University, 1999.

[5] J.R. Burch, and D.L. Dill, "Automated Verification of Pipelined Microprocessor Control," *Computer-Aided Verification* (*CAV'94)*, D.L. Dill, *ed.*, LNCS 818, Springer-Verlag, June 1994, pp. 68-80.

[6] J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *33rd Design Automation Conference (DAC'96)*, June 1996, pp. 552-557.

[7] CUDD-2.3.0, `http://vlsi.colorado.edu/~fabio`.

[8] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 244-255.

[9] L. Gwennap, "Merced Shows Innovative Design," Microprocessor Report, Vol. 13, No. 13 (October 6, 1999), pp. 1-10.

[10] T.R. Halfhill, "StarCore Reveals Its First DSP," Microprocessor Report, Vol. 13, No. 6 (May 10, 1999), pp. 13-16.

[11] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Decomposing the Proof of Correctness of Pipelined Microprocessors," *Computer-Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 122-134.

[12] *IA-64 Application Developer's Architecture Guide*,[3] Intel Corporation, May 1999.

[13] J.R. Levitt, *Formal Verification Techniques for Digital Systems*, Ph.D. Thesis, Department of Electrical Engineering, Stanford University, December 1998.

[14] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli, "Partitioned ROBDDs—A Compact, Canonical and Efficient Manipulable Representation for Boolean Functions," *International Conference on Computer-Aided Design (ICCAD'96)*, November 1996, pp. 547-554.

[15] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *International Conference on Computer-Aided Design (ICCAD'93)*, November 1993, pp. 42-47.

[16] J. Sawada, and W.A. Hunt, Jr., "Processor Verification with Precise Exceptions and Speculative Execution," *Computer-Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 135-146.

[17] H. Sharangpani, "Intel Itanium Processor Michroarchitecture Overview,"[3] *Microprocessor Forum*, October 1999.

[18] M.N. Velev, and R.E. Bryant, "Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors,"[2] *36th Design Automation Conference (DAC '99)*, June 1999, pp. 397-401.

[19] M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic,"[2] *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre and T. Kropf, eds., LNCS 1703, Springer-Verlag, September 1999, pp. 37-53.

[20] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction,"[2] *37th Design Automation Conference (DAC '00)*, June 2000.

[21] P.J. Windley, and J.R. Burch, "Mechanically Checking a Lemma Used in an Automatic Verification Tool," *Formal Methods in Computer-Aided Design (FMCAD'96)*, M. Srivas and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November 1996, pp. 362-376.

2. Available from: `http://www.ece.cmu.edu/~mvelev`
3. Available from: `http://developer.intel.com/design/ia-64/architecture.htm`