University of Rhode Island
Dept. of Electrical and Computer Engineering
Kelley Hall
4 East Alumni Ave.
Kingston, RI 02881-0805, USA

Technical Report No. 0797-0001

Overview of the Levo High-ILP Computer
- with Levo-32-chip details -

Augustus K. Uht
Department of Electrical and Computer Engineering
University of Rhode Island

uht@ele.uri.edu
www.ele.uri.edu/∼uht

August 26, 1997

Abstract

This document gives an overview of the microarchitecture of the **Levo** *prototype computer. Both the data and control paths of Levo are described, with the function of all major subsystems given. Levo's operation is outlined. Anticipated absolute dimensions are given, assuming the 32 Processing Element (PE) single-chip version, called Levo-32-chip.*

1 Introduction

In order to significantly improve performance of standard general-purpose processors, such as Intel x86 microprocessors, *Instruction Level Parallelism* (ILP) must be exploited. ILP involves the execution of machine instructions in parallel. It is fine-grain, occurring within an iteration of a loop, for example. In order to enhance ILP, many methods are commonly used[12], such as data dependency reduction and *branch prediction*. In the latter, when a CPU encounters a branch, it uses special hardware that predicts whether the branch will be taken or not taken (its *sign*), based on the branch's prior execution history; execution then proceeds down (just) the predicted path. This is also called *Single Path* (SP) speculative execution; see Figure 1. These basic methods realize an ILP speedup of about a factor of 2 or 3 over strictly sequential execution.

Our goal is to improve upon this by an order of magnitude, and realize ILP speedups of a factor of 20 to 30. Preliminary simulations have demonstrated that this is possible, when our new ideas are used in addition to branch prediction and data dependency reduction.

The two major new ideas are *Minimal Control Dependencies* (MCD)[8] and *Disjoint Eager Execution* (DEE)[11]. The former allows some code statically after a branch to be unconditionally executed before the branch. The latter executes the most likely branches regardless of their position in the tree (see Figure 1).

Levo is a family of prototypes being designed to demonstrate both the functionality of these ideas, as well as the resultant high performance. The major demonstration machine is Levo-32, consisting of 32 Processing

This is a major revision of the similarly-titled document dated February 28, 1997.
This work was supported in part by the Intel Corporation and the NSF under grant No. MIP-9708183.

Elements (PEs), with a target ILP speedup of a factor of 20. We are currently seeking funding for such a large undertaking. This document focuses on a version of Levo that could result if it were fabricated on a single chip, Levo-32-chip.

This document is structured as follows. Section 2 gives a general view of the operation of Levo. Section 3 describes the central control structures, logic, and operation of Levo in detail. Pipelining control and exception handling are reviewed in Section 4. Section 5 contains a brief outline of our strategy for realizing a multi-chip prototype at URI, while Section 6 compares Levo with other approaches, and Section 7 concludes the paper.

2 Levo Basics

The general operation and microarchitecture of Levo are now described. Since Levo is somewhat unusual in its construction, we will provide correlations to more commonly known microarchitectural structures and techniques, especially with reference to the Intel Pentium Pro[4]. In the discussion, the corresponding Pentium Pro element will be indicated in boxed italics: *[italics]*. NOTE that we are NOT equating the correlated structures, just providing familiar points of reference that the reader can build on. We will assume a MIPS R4400 Instruction Set Architecture (ISA) for the sake of simplicity in terms of the operations performed, unless specified otherwise.

Levo is a 32-way superscalar pipelined/superpipelined out-of-order (OOO) computer. By 32-way we mean that up to 32 instructions may be issued to the PEs *[all of: Execution Units, DTLB, Address Generation Unit]* per cycle. 32-wide instruction fetching is not used, since the size of the instruction window (Instruction Queue, or IQ) *[Reservation Station]* is about 256 instructions, and thus *loop capture* may occur frequently, reducing the instruction fetch requirements normally thought to be necessary.

As is common with current computers, Levo is pipelined, and the different stages and timing thereof follow a typical pattern: instructions are fetched, the branches are predicted, the assignment statements are executed, and the results are committed to the ISA register file and main memory once the instructions are no longer speculative. In common with the Pentium Pro, the overall structure can be considered to be formed of a nominally in-order instruction fetch front end, an out-of-order core, and a partially in-order commit section. From this point, the similarity rapidly diminishes.

The Levo Microarchitecture: Levo[11] is a high-ILP prototype computer that will realize DEE with minimal control dependencies; see Figure 2. The State Block (*SBlock*) contains the instructions being executed (and those executed in the recent past) with their corresponding renaming registers. Each renaming register is used for the sink of the corresponding instruction. In the figure, the large (32 by 8) instruction array is for the MainLine path (this corresponds to the branch paths with circled numbers 1, 2, 3 and 5 in Figure 1), while the four smaller arrays (32 by 1) are for four DEE paths (there are two DEE paths in Figure 1: one consists of branch paths 4 and 7, while the other is composed of only branch path 6). The SBlock also contains the status registers and logic necessary to detect the ILP and schedule the instructions for issuance to the Processing Elements.

Normally, the renaming registers provide the source data to the PEs. To keep the electrical loading on the renaming registers low, there are four copies of the SBlock. Each SBlock provides sources for, and issues to, 8 PEs; this allows for the desired total of 32 PEs. Referring to Figure 2, each Processing Element can be issued an instruction from one of 12 possible entries in the IQ: 8 MainLine instructions and 1 instruction from each of the four DEE paths; all of the instructions are in the same row. Hardwiring the allocation of PE resources this way simplifies the logic, keeping the cycle time low. Thus, since each PE corresponds to a unique row, the output of each PE is written to only one logical row (but to all four physical copies of the row, one in each of the four SBlock copies), and all of the PEs may write their results concurrently (since the results are going to different rows of registers). This design allows for the requisite high renaming-register–to–PE read and write bandwidths.

The ISA registers are stored in the main memory system *[Real Register File, and main memory]*, and their accesses are treated as main memory accesses. This might seem counter-intuitive, that is placing the architectural registers far from the PEs, thereby incurring a time penalty; however, the likelihood is that an instruction's register sources will be supplied by a renaming register, since the instruction window is so

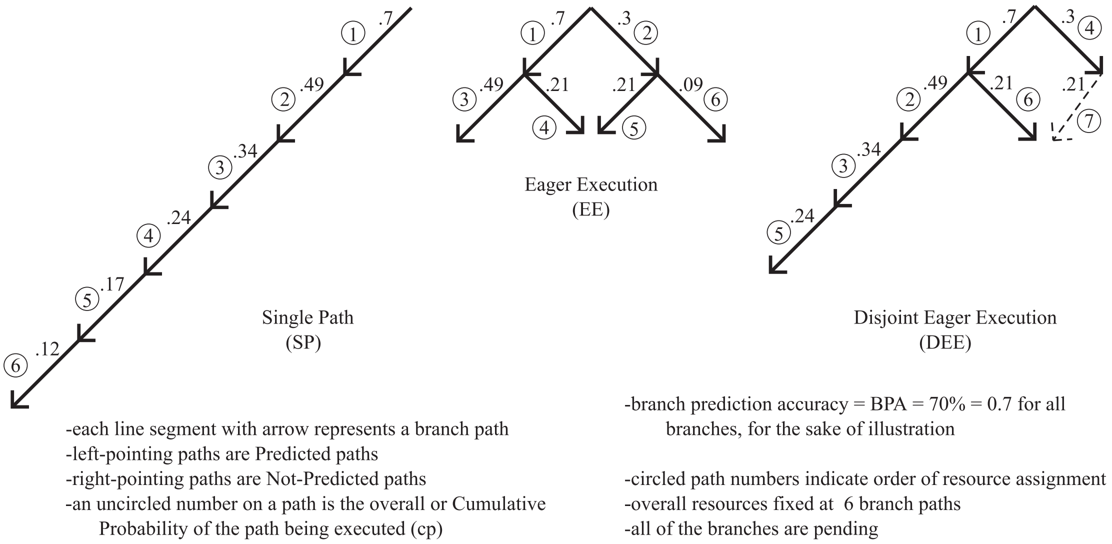

Figure 1: *Comparison of the three speculative execution strategies.* In each strategy, resources are assigned only to the bold paths. Note that the cumulative probabilities of the DEE bold paths are greater than the corresponding ones of both SP and EE; in DEE, the most likely code to be executed is given the resources.

large. Frequently referenced ISA registers that are not written by any instruction in the window will likely be sourced by copies of their values effectively cached in a memory buffer.

The IQ is a primarily static window on the executing code; thus, the window's content is largely independent of how branches execute; this allows the realization of Minimal Control Dependencies. Instruction fetch bandwidth is reduced when *loop capture* occurs (that is, when a combined static/dynamic view of a loop body fits in the IQ, i.e., is less than or equal to $32 * 8 = 256$ instructions long; the combined view allows loops statically greater than 256 instructions long to fit in the IQ). Further, although the DEE paths and Mainline paths contain different paths of execution, normally the DEE paths' static instructions are the same as instructions in the MainLine path, also decreasing instruction fetch bandwidth requirements.

The executions of all of the instructions in a row (MainLine and DEE) compete for the same PE; they are therefore prioritized, in a manner heuristically providing good performance. However, instruction result data that is the same for one or more DEE paths as the MainLine path is copied simultaneously into the different paths (effectively, the DEE paths snoop the MainLine renaming register inputs). If there is a misprediction, and a DEE path is found to be the correct path, the DEE path's renaming registers are copied broadside into the MainLine region's, within a cycle, correcting the execution state. Lastly, the SBlock contains logic to squash incorrectly speculated control-dependent instructions within each of the composite (MainLine or DEE) paths. Execution within each composite path is SP with MCD.

Main memory bandwidth is also kept low by a novel bus system interconnecting the memory buffers in the SBlock copies with the interleaved cache/memory banks. This system eliminates redundant loads and stores, while maintaining a temporally-accurate view of the program state. It has some features similar to the Multiscalar ARB[1], but should be much simpler and have greater performance. Note that the chip boundary is between the caches and the main memory (within the banks shown in the figure); therefore, the memory bus system is entirely on-chip, keeping its speed high. The instruction-fetch system is also entirely on-chip.

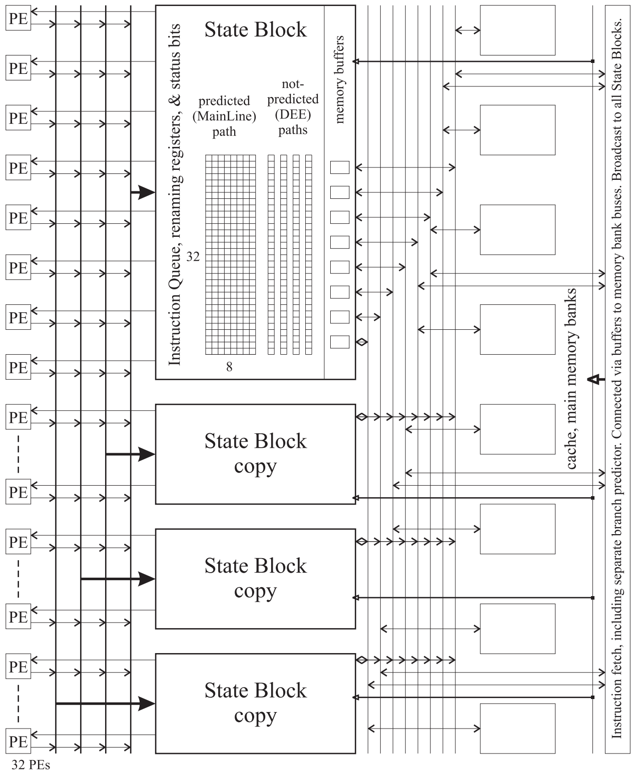

Figure 2: *Block diagram of Levo, the prototype high-ILP computer.*

Illustration of Dependencies and Squashing Relationships: We now describe the types of dependencies in Levo, their squashing effects, and the differences between these and classical approaches.

Consider the following code example:

```
1:  if (z==1) goto 3;
2:      A = b + c;
3:  w = x + y;
4:  d = A + e;
5:  f = g + h;
```

We make the following observations on this example:

– Instruction 2 is in the branch's *domain.* (A branch's domain consists of the instructions between the branch and its target). Instruction 2 is thus control dependent on the branch.

– Instruction 4 is data dependent on instruction 2 through the variable `A`.

Classical control dependencies:

– All instructions after the branch (1.) are control dependent on the branch.
– When the branch mispredicts, all of these instructions (2. on) must be squashed, and possibly re-executed.

Minimal Control Dependencies:

– All of the instructions after the branch's domain are control independent of the branch. This includes instructions 3-5.
– Since instruction 4 is dependent on instruction 2, which in turn is control dependent on the branch, then instruction 4 is *transitively dependent*[a] on the branch.
– Neither instruction 3 or 5 is dependent on any instruction in the branch's domain, therefore neither is transitively dependent on the branch.
– If the branch mispredicts, only instruction 4 must be squashed, since only it is transitively dependent on the branch.
– Before the sink `d` of instruction 4 can be committed to memory, all of the instruction's transitive *dependor* branches (just instruction 1, here) must have been completely resolved, such that no more mispredictions of these branches are possible.

[a] In prior publications, we referred to transitive dependencies as *total* dependencies; the new term is more descriptive.

Thus, mispredictions have less of a negative effect in Levo than in classical computers. Levo is the first machine to realize minimal control dependencies.

Levo Operation: The basic operation of Levo consists of the following steps, each happening every cycle. Many of the steps' actions take place concurrently.

1. load the IQ *buffer* with static instructions to be executed; while loading, compute the data dependencies amongst the instructions; compute the instructions' control dependencies on *forward branches* (target statically after the branch); and compute the instructions' transitive dependencies on both the branches and the loads;
2. if all of the instructions in column 1 have unconditionally executed (their transitive dependor branches having been finally correctly predicted, and their dependor loads unconditionally executed or resolved), retire column 1 by writing as-yet-unsaved instruction sinks to their corresponding memory buffers, which are in turn sent to the caches; shift the IQ and all state matrices left by 1 column, and shift the contents of the (full) IQ buffer into the rightmost column of the IQ;
3. using the pre-computed data dependencies, and the execution history of the instructions, compute which instructions have their sources available and are free to execute (from a data-flow perspective);
4. couple the source data from the renaming registers for up to 32 instructions to up to 32 *Processing Elements* (PE) (one instruction per PE), and begin execution of the ready instructions;
5. predict the sign of unexecuted branches (taken or not taken), and conditionally execute them that way; a taken forward branch disables the execution of instructions in its domain; a taken backwards branch enables a new iteration of the instructions in its domain;
6. if there is a free DEE path, assign it to the earliest unresolved branch in the IQ, and execute down the not-predicted path of the branch, within the DEE path;
7. resolve predicted but unexecuted branches (determine the final sign); mispredicted branches cause transitively dependent instructions on the branch to be squashed, possibly causing them to be re-executed; a mispredicted DEE'd branch causes the not-predicted path's (DEE path's) state to be loaded into the main execution section (MainLine path) of the processor; the DEE path is then freed for reassignment to another branch;
8. memory loads are performed as soon as their memory addresses have been computed;

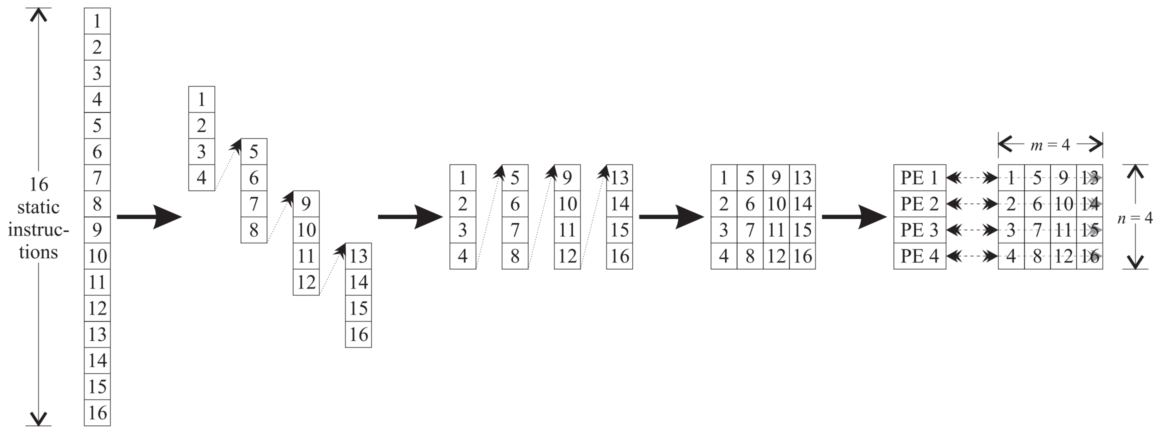

Figure 3: *Instruction Queue (IQ) "folding".* Each box represents a static machine instruction. The numbers in the boxes indicate the nominal dynamic execution order of these instructions. For the sake of illustration, $n = m = 4$.

9. memory addresses of loads are compared to addresses of prior memory stores in column 1; if they are equal, the loads and their transitively dependent instructions are squashed and possibly re-executed;
10. update the instructions' execution history; load the PEs' outputs into the renaming registers in parallel;
11. each memory buffer (one per PE) competes with the other buffers for the memory busses to load or store their contents from or to the ISA register file or main memory (cache);
12. repeat, goto (1).

3 Detailed Description of Levo Elements and Operation

Basic Architecture: For clarity, the initial description of the architecture will not include that needed for the disjoint eagerly executed code (*DEE* region). All of the concurrency and dependency structures are built around the IQ. In Levo-32-chip, the IQ is 256 instructions long. It is a *quasi-static window* (see Section 3.5) on the executing code; it nominally holds static instructions, i.e., instructions in the order as they appear in memory, which does not vary with branch executions. There are 32 PE's. The IQ is folded like an accordion so as to form 8 columns ($m = 8$) of 32 instructions ($n = 32$ rows) each; see Figure 3 for an illustration assuming $n = m = 4$. The instructions along a row share the same PE. This simplifies the construction and control, and speeds up execution. A more detailed picture of the Levo data path is shown in Figure 6-a); for illustration, the number of SBlocks is equal to the number of PEs (n).

3.1 Storage Structures

Unless stated otherwise, all of the structures are in the CPU. Also, unless stated otherwise, there are four copies of each of the following structures, one copy per SBlock.

For the sake of clarity, all of the structures defined below are assumed to be as large as necessary, as is, to cover all $nm = 256$ instructions in the IQ. In reality, this is prohibitive for the dependency matrices, and actually modifications of the methods are used, such that the matrices are only about 32 instructions on a side. These new methods will be published at a later date.

Program State: The sinks or outputs or results of instruction execution are held in the *Shadow Sink* (SSI) renaming register matrix *[Reorder Buffer, both ROB and MOB]*, which is also n by m. The ISA register or main memory addresses of the sinks are held in the *Instruction Sink ADdress* (ISAD) matrix *[Register Alias Table]*, also n by m. An appended bit to the ISAD address field distinguishes main memory addresses from ISA addresses. Each element of both the SSI and ISAD matrices is 32-bits wide. For each PE, the SSI has two primary outputs: the source data (operands) of the instruction to be issued to the PE in the current cycle. There is special logic that gates each output to a PE, called *Sink ENable* or SEN logic *[Reservation Station]*. The SSI is not accessed like a standard memory having an address input. Rather, the SEN logic has n by m output lines, each of which is a signal gating the corresponding SSI element to an SSI output bus; only 0 or 1 of these signals is asserted at the same time. Since there are nominally two sources to an instruction, there is one set of SEN logic per source, or SSI output. When an instruction is issued to a PE, a tag is sent to the PE indicating which column the result of the instruction should be written to when returned to the SSI after instruction execution. Inside the PE, the tag is shifted with the computation in the PE's pipeline. Whenever column 1 of the IQ is retired, shifting the other columns left by 1 column, the tag is updated to refer to the new column location of the destination.

The secondary output of the SSI is used when storing data to register or main memory. In order to keep things simple, writes are only made from the first column, and only when the corresponding instruction and all of its earlier instructions have executed unconditionally (they are no longer speculative). A similar output of the ISAD supplies the sink address. The SSI/ISAD (data/address) pair is sent to the *memory buffer* of the row, and from there to external storage. The ISAD output is also used to send the main or register memory read address to the memory buffer *[Reorder Buffer, both ROB and MOB]* (and system) during either a memory load instruction execution, or when getting an architectural register datum that is not in the SSI. Loads can execute from any column, and may be speculative.

The SSI has 32 inputs, which normally come from all of the PE's. The row i PE's output goes only to row i of all copies of the SSI's. The column a datum is written to is determined by the tag sent back to the SSI with the datum by the PE. Each input may also be driven by the memory buffer when the buffer returns a fetched datum.

Execution State: It is necessary for the hardware to keep track of which instructions have already been executed in order to determine which to issue in the current cycle. This state is held in several bit matrices, each n by m, and thus each bit corresponds to a distinct instruction (and SSI sink, etc.).

A *Really Executed* (RE) bit is set if an instruction is finishing execution in the current cycle. In other words, an instruction's RE bit is set if the instruction has actually been executed and has produced data (assignment instruction) or been predicted (branch).

The *Virtually Executed* (VE) bits are set by taken forward branches to indicate that instructions between the branch and its target (in the branch's *domain*) have been branched around, and should not be really executed.

Instruction Issued (II) bits are set when the corresponding instruction is issued to a PE for execution. Both II and RE bits are needed since some instructions take more than one cycle to execute. Branches do not use these bits since they are always executed in one cycle.

The main memory load and store instructions use the *Address Valid* (AV) bits to indicate that a PE has computed the effective address of the instruction, and that it is in ISAD. Note that the normal execution of a memory access (load or store) is composed of two steps: computation of the effective address, including virtual-to-physical address translation (this all happens in the PE), and the actual movement of the data[11].

Branch-Specific State: Several bit-matrices are maintained to keep track of the state of resolution and execution of branches. *Assignment Statement* (AS) indicates that an instruction is an assignment statement. *Branch PRedicted* (BPR) bits are set when a branch has been predicted. *Branch EXecution Sign* (BEXS) bits are set (asserted) when a branch has been predicted or executed as "taken". *Branch Prediction Type or sign* (BPT) bits indicate the sign of the latest prediction of the corresponding branch. It may be possible, although not necessarily desirable, to store some of this state in the SSI matrix.

Dependency Matrices: The SEN logic uses the execution state to help determine which instructions to issue in a cycle, and to link the appropriate SSI contents to instruction inputs (use as operands). However, to do this, the SEN logic also needs to know all of the possible inputs to an instruction. These are those instructions that the pending instruction is *data dependent* on. Although there are three possible types of data dependencies, Levo only needs to enforce one of them, the *true* or *flow* or *Read-After-Write* (RAW) dependencies; this is since the SSI realizes *renaming*, i.e., allowing multiple copies of data to exist at the same time, eliminating two types of data dependencies.

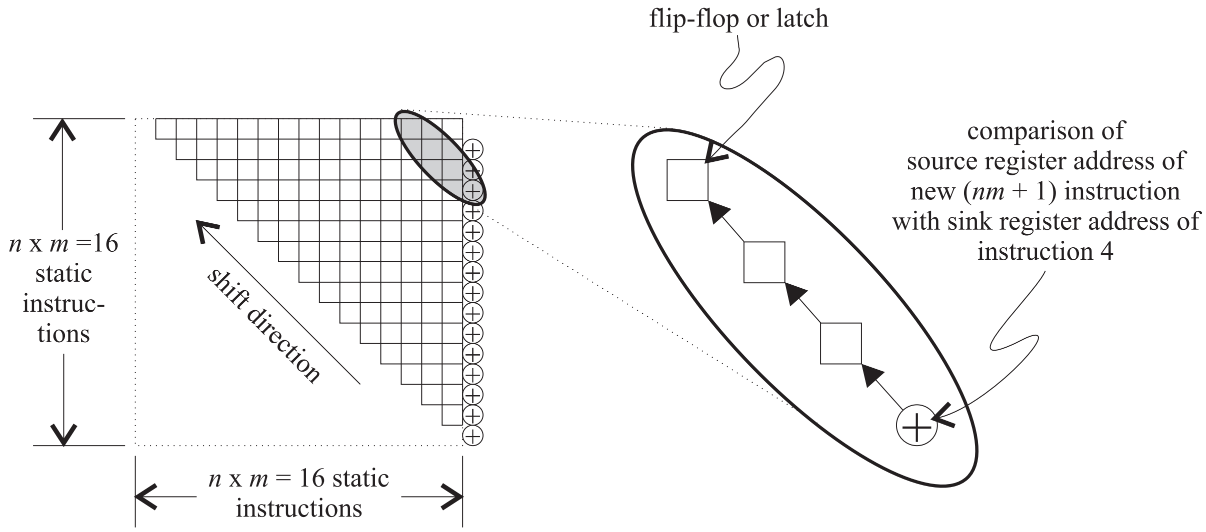

Figure 4: *Loading of Data Dependency (DD) matrices.* For this example, $n = m = 4$ (normally $n = 32$, and $m = 8$).

The data dependencies are held in two upper-diagonal square bit matrices ($DD^{z=0 \text{ or } 1}$), one matrix per instruction source. The matrices are nm by nm in size. A bit at row i and column j of a DD matrix z is set if instruction j is flow dependent on instruction i, through source number z.

The data dependencies represented in the DD matrices are only the architectural register dependencies amongst the instructions, not the main memory data dependencies. Although register addresses are much shorter than main memory addresses, 5 bits versus 32, computing the dependencies at run-time could still potentially require excessive hardware ($1/2 \cdot [nm]^2$ comparators). Therefore, since the dependencies are static, i.e., they do not change during program execution, they are computed *relatively statically* with nm comparators at the time the instructions are loaded into the IQ. Considering one source, this is done by comparing the source address of the instruction about to be loaded into the IQ with the sink addresses of all of the *nm-1* prior instructions already held in the IQ, and then shifting the $nm - 1$ bits thus determined diagonally into the DD matrix (towards the upper-left corner) when the new instruction is actually loaded into the IQ buffer; see Figure 4.

Control Dependencies: Only the forward-branch control dependencies are computed and saved in Levo. They are computed relatively statically and shifted diagonally into the *Forward Branch Domain* FBD matrix at instruction load time, as with the data dependencies. FBD is effectively upper-diagonal, and nm by nm, as are the DD matrices. The actual dependency computations are a bit more complex, however, and are described elsewhere[8].

Transitive Dependencies: In Levo instructions are issued and executed only respecting their data dependencies, i.e., in a data flow manner. However, recall that when a branch mispredicts, its *Transitive Dependent* (TD) instructions must be squashed (and possibly re-executed). A branch's transitive dependent instructions are those either directly or indirectly control or data dependent on the branch. An instruction is indirectly dependent on a branch if it is dependent on a third instruction which in turn is dependent on the branch; the potential depth of the indirection is actually limited only by the size of the IQ. Thus, a branch's transitive dependent instructions are those whose results may change (they might not even be executed) if the branch mispredicts.

Transitive dependencies are also used with speculative loads. All instructions directly or indirectly dependent on the sink of a load are transitively dependent on the load. If the load is mis-speculated, then all instructions transitively dependent on the load must be squashed, and possibly re-executed.

The TD bit matrix is similar to the DD and FBD matrices, being upper-diagonal and nm by nm in size. However, the setting of its bits is more difficult, and will be covered later in a separate document. Briefly, my student Wei Tan has devised a way of computing transitive dependencies quickly, within a cycle, with little hardware.

Other State: There is one *Processing Element Busy* (PEB) bit per PE (per row). It is set when a PE cannot accept another issued instruction. It will never be set, if the PE is fully pipelined (one result per cycle).

3.2 Other Logic

The detailed equations for the SEN logic and FBD setting logic appear in [10] and [8], respectively. The rest of the detailed equations for the Levo logic will be published at a later date.

Prediction Logic: The *CORrectly EXecuted* (COREX) logic is used to generate *OK EXecuted* (OKEX) signals which indicate when instructions have been correctly (finally) executed, i.e., when prior dependor branches have been completely resolved and executed. This logic is used by instructions in the first column to enable the writing of their sinks to the memory system.

The *INCORrectly EXecuted* (INCOREX) logic is activated by branch mispredictions to generate *RE-EXecute* (REEX) signals which squash dependent instructions, by clearing their RE bits. The TD matrix is used in REEX calculations.

Fault Tolerance: Levo is to be fault tolerant; if a fault is detected in it via scan logic testing after a system crash, the containing row is configured out of the system, and execution of programs begins again. The necessary modifications for this capability consist primarily of altering parts of the DD-loading logic. Details will be published later.

Interrupts and Exceptions: Handling interrupts and exceptions, such as page faults, is challenging with ILP machines. The method for handling them is given in Section 4.2. Basically, all instructions before an exception must have finally executed (RE or VE asserted, and no dependor mispredicted branches) before the exception is handled; the prior instruction requirement includes other exceptions, so only the earliest exception in the dynamic instruction execution order will be handled. Later exceptions will be handled after the CPU has resumed execution, and of course, only if the exception arises again.

Branch Predictors: There is one branch predictor per row. The type of predictor is variable. The initial plan is to use a Pap type of two-level adaptive predictor[13] having small dimensions.

There is a separate predictor for the instruction fetch logic. Its type has not yet been determined. Note that since there is only one of these, it can be fairly large.

Main Memory Access Dependencies: In Levo loads always execute (assuming the address has been calculated), regardless of the execution state of prior stores, i.e., they may be speculative[2, 3]. Therefore, the memory addresses of loads must at some point be compared to those of prior stores, to ensure that a load is not dependent on a prior store. In Levo load addresses are compared to prior store addresses in the first column every cycle; if there is a match, the matching loads and their transitive dependent instructions are squashed and re-executed; this forces such loads to use the data generated by the matching store(s).

The comparisons are performed by n comparators per row. These comparators need not necessarily be 32-bits in length; probably only 8-bits need be compared, since the comparison is only used for squashing determination, not data linking. There will be false positive matches, but it is felt that their number and ill effects on performance will be slight. (Only 1 out of 2^8 matches, at most, will be false positives, and they will result in a time penalty of only a cycle or two for the particular load involved.)

3.3 Disjoint Eager Execution

One of the major reasons for Levo's high ILP is its use of *Disjoint Eager Execution* (DEE). This is the execution down both paths of a branch (eager execution) at different times (disjoint), of selected (likely

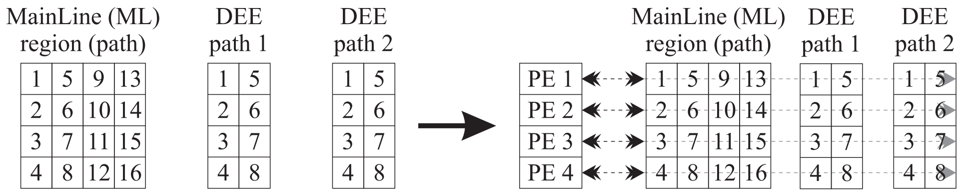

Figure 5: *Addition of Disjoint Eager Execution (DEE) to Instruction Queue.* For the sake of illustration, $n = m = 4$.

to be executed) branches. Realizing it requires the following modifications to what has been previously discussed.

Levo-32-chip's implementation of DEE will allow for up to four additional paths of execution to be active at the same time as the original or *MainLine* (ML) path; see Figure 5. However, as is seen in the figure, each DEE path is shorter than the ML path. This saves hardware, and is justified by the fact that any code held in the end of a longer DEE path would be very unlikely to be used. Storage and logic for the additional DEE program state and execution state must be added to the hardware. However, since DEE instructions are the same as those in the ML path, the original dependency matrices can be used for the DEE paths as well. Further, the DEE paths use the same branch predictions as the ML path, so that this hardware is also not copied. Lastly, as is seen in both Figures 5 and 6-b), the DEE paths share the PE's with the ML path.

In operation, the ML state is copied into the DEE state, for all instructions dynamically before (above) the DEE (eagerly executed) branch, as the state is generated, by having the DEE path pick the SSI data off of the corresponding PE result bus, at the same time that the data are being sent to the ML path. The DEE branch instance in the DEE path is executed oppositely to the way it was predicted in the ML path. Within a DEE path, code executes as before.

If a DEE branch was correctly predicted, the ML path stays as is, and the corresponding DEE path is ignored, and given no further priority for execution resources. The time penalty here is 0 cycles. If the DEE branch was mispredicted, then the corresponding DEE path state must be copied into the ML path, and the ML state dynamically later than the DEE path end and transitive dependent on the DEE branch cleared. The state copying is accomplished by placing the specific DEE state on return busses and reading them into the ML path state hardware in parallel. The time penalty for one or more DEE branch mispredictions, determined in the same cycle, is zero or one cycle.

3.4 The Levo Memory System

A key challenge of designing a memory system for Levo is to supply its necessarily large bandwidth requirements. The Levo-32-chip memory system architecture consists of one small (say 2 to 8 word) buffer per IQ row all of which are connected via an on-chip crossbar (realized with busses) to 8 interleaved combination cache-memory units. All load and store requests on a row are sent to the row's buffer. All of the buffers and the first level caches snoop the crossbar busses. With this method, a store from one instruction is sent to its cache via a bus; a load from another instruction in a (typically) different row from the same address can pick the store's data off of the bus on its way to the cache, eliminating a memory load, and reducing the load's latency. Or, a load placed on the bus can be satisfied by a store datum having the same address, held in a different buffer, also eliminating a memory access. With such methods, memory accesses are minimized (within a window's worth of instructions). The memory system is described in detail, with logic, in [9].

3.5 Detailed Operation of Levo

Instruction Fetching: The basic operation of the fetch mechanism is as follows. There is one additional column of 32 instructions called the *IQ buffer* into which the fetch unit loads new instructions. The buffer

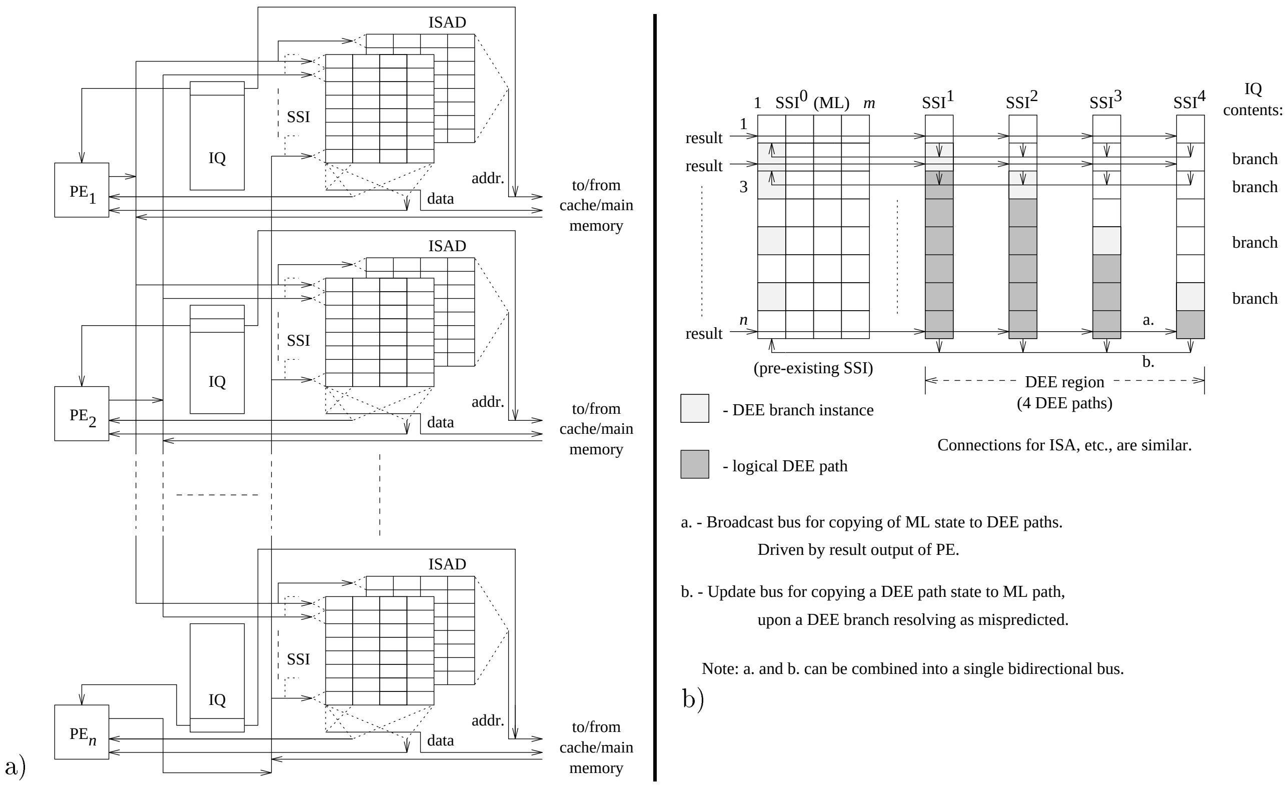

Figure 6: a) *Levo central CPU data path, without DEE paths.* For illustration, one SBlock is shown per PE; Levo-32-chip will have one SBlock per 8 PEs. b) *DEE additions and modifications to Levo.* These figures assume $n = 8$, $m = 4$, and 4 DEE paths, versus $n = 32$, $m = 8$, and 4 DEE paths in Levo-32-chip.

is parallel to the 8 MainLine columns, and is logically to their right. Once the buffer is full, and the first (leftmost, or earliest) column of the MainLine section retires (committing its sinks to the memory section), the contents of the buffer is shifted broadside into the rightmost column of the MainLine section, all 32 instructions at once.

The IQ buffer itself is loaded from the memory system, appearing as a “memory buffer” to it. It shares connections to the unified caches with the data memory buffers connected to the core’s IQ rows. In this way self-modifying code is readily handled. The IQ buffer keeps a record not only of the instructions it holds, but also those held in the MainLine section proper, with their accompanying *time tags*. These tags indicate the relative time of instructions, and are formed by the concatenation of an instruction’s IQ column number (or *column tag*) with its row number. A write to a code address from a data memory buffer will appear on one of the memory buses, which is snooped by the IQ buffer. If the data write is to an instruction in the IQ, and the time tag of the write is less than that of the instruction, then the IQ is flushed from that point on, and reloaded from there.

IQ loading is slightly different when loop capture occurs. To get a rough idea for how this works, assume for now that the loop body completely fills the MainLine region. If the loop-forming backwards branch at the end of the region executes or is predicted true, then when the first column is retired, its contents are shifted broadside into the last column, column 8, of the region; this operation is like a circular shift register. At the same time, the execution status of the “new” column 8 is cleared. Thus, the beginning of the loop re-executes, but nominally after the end of the prior iteration. With loop capture, little or no instruction loading takes place, reducing memory bandwidth requirements; the execution in the OOO core is also enhanced.

Levo’s instruction fetch mechanism uses a separate branch predictor from the OOO core’s predictors. We have not yet decided on a particular predictor to use. It may also be the case that information from the core will be used to aid the fetcher’s predictor. Branch predictor state for loading branches is copied to the IQ from a Branch Target Buffer *[Branch Target Buffer]*, which acts as a cache for all of Levo’s predictors.

IQ loading is also slightly different when a branch is about to be loaded into the buffer. Normally, the code that is loaded into the buffer is in the static order; in other words, the code after the branch (i.e., if it were not taken) is always loaded after the branch, ignoring how the branch might execute. However, if the domain of the branch (typically a forward branch) is large (of the same order as the total size of the IQ), indicated by a large target offset address, there is the potential that much of the IQ might be wasted, and performance will suffer; for example, if the branch executes predominantly taken, then not many instructions will be available in the queue to extract concurrency from. In such cases, a *quasi-static*, or combined dynamic/static, order is used as follows. The fetch unit’s branch predictor is used to estimate if the branch will likely be taken; if so, then after the branch is loaded, the IQ is loaded with instructions starting from the branch’s target. Subroutines can be loaded inline using this methodology, handling calls within loops well. Thus, the IQ is efficiently used, and can handle loops statically larger than the size of the IQ very well.

There is another possible source of inefficiency, at the other extreme, with backwards branches whose domain size is much smaller than the size of the IQ. In this case it is desirable to unroll the loop. We anticipate that some of this may be done by the operating system, at program run-time or before. However, much of the unrolling, and possibly all of it, can be done automatically by the fetch unit.

Dynamic (hardware-based) unrolling works as follows. When a branch is about to be loaded, if it is a forward branch, nothing is done (except as mentioned previously); if it is a backwards branch, then its target offset is compared to the size of IQ; if it is less than half as long (less than 128 instructions), then unrolling occurs (and possibly more than once). The trick is to convert the backwards branch to a forward branch with zero offset, and complement the sign of its original branch condition. Once loaded, this has the effect of creating a forward branch from the branch to the new instance of the original backwards branch (the two have the same instruction address, causing the FBD-loading hardware to create a domain of the proper shape). Now, during execution, when the new condition of the forward branch is false, this branch is not taken, and a new iteration of the loop is enabled; if the condition is true, the new iteration is disabled, the backwards branch executes false, and the loop terminates. Loop capture can also be used with dynamic unrolling for further gains.

There are other details which will be published at a later date.

Main Steps in a Cycle, OOO Core: The following steps are performed every machine cycle. Steps with the same number are performed simultaneously. Refer to the timing diagram in Figure 7. The numbers of the items correspond roughly to the different thirds of a machine cycle. Note that "AE" refers to all of the execution state in an SBlock, e.g., RE and VE matrices.

<u>Start of cycle:</u>

1 & 2. Perform SEN calculations. Instructions are determined that should issue this cycle.
NOTE: the following steps are applied to each row or PE.
Issue instruction to PE, along with the instruction's column tag, and set the PE's busy bit.

3. For each issued instruction, gate operands to PE, using the SEN outputs.

1 & 2. The last stage of computation in a PE, initiated in a prior cycle, takes place.

3. Receive new SSI value from PE. Latch into column of SSI indicated by the value's column tag. Result is implicitly bypassed (forwarded) to an instruction using it, and issued this cycle. *(NOTE: this is the best that could be achieved. It is likely that bypassing will take place elsewhere in a production chip.)*

3. Update AE-like matrices, esp. RE, VE and II.

1. Predict one branch per row.

2. Compute the necessary modifications to the RE and VE matrices arising from the execution of the branch as predicted. If there is a free DEE path, allocate the path to the branch, and enable its execution.

1. Perform a fast computation on the PE to determine the sign of an executing branch initiated in the prior cycle. Determine if the branch was mispredicted.

2. Perform the $REEX$ calculations to determine those instructions that need to be squashed (if there was a misprediction).

2 & 3. Update the branch predictor. (This may take less time.)

2. If a DEE branch was found to have been mispredicted in this cycle, route the DEE SSI outputs to the ML SSI inputs.

3. Inputs to SSI stabilize.

1 & 2. If the instruction initiated in the prior cycle is a Load or Store, and its address has not yet been computed, compute its address, and then perform a virtual address translation on the computed address, using a TLB (Translation Lookaside Buffer) in the PE. If there is a TLB miss, generate an exception.

3. Address stabilizes at the ISA inputs.

3. The address is sent to the memory system, if the instruction is a Load; if it is a Store, it is sent to the memory system if both its data is ready and the instruction is in the first column.

1. The Load address from an address calculation in a prior cycle is (partially) compared with all of the first column Store addresses.

2. $REEX$-like calculations are made to determine the instructions transitively dependent on a matching Load; these instructions are squashed.

1 & 2. A Load of a prior cycle is satisfied by the memory system during this time.

3. The data returned from the memory system stabilizes at the inputs to the SSI.

The rising edge of the clock occurs during the latter part of time 3, saving the (revised) SSI, ISA and AE-like data and state.

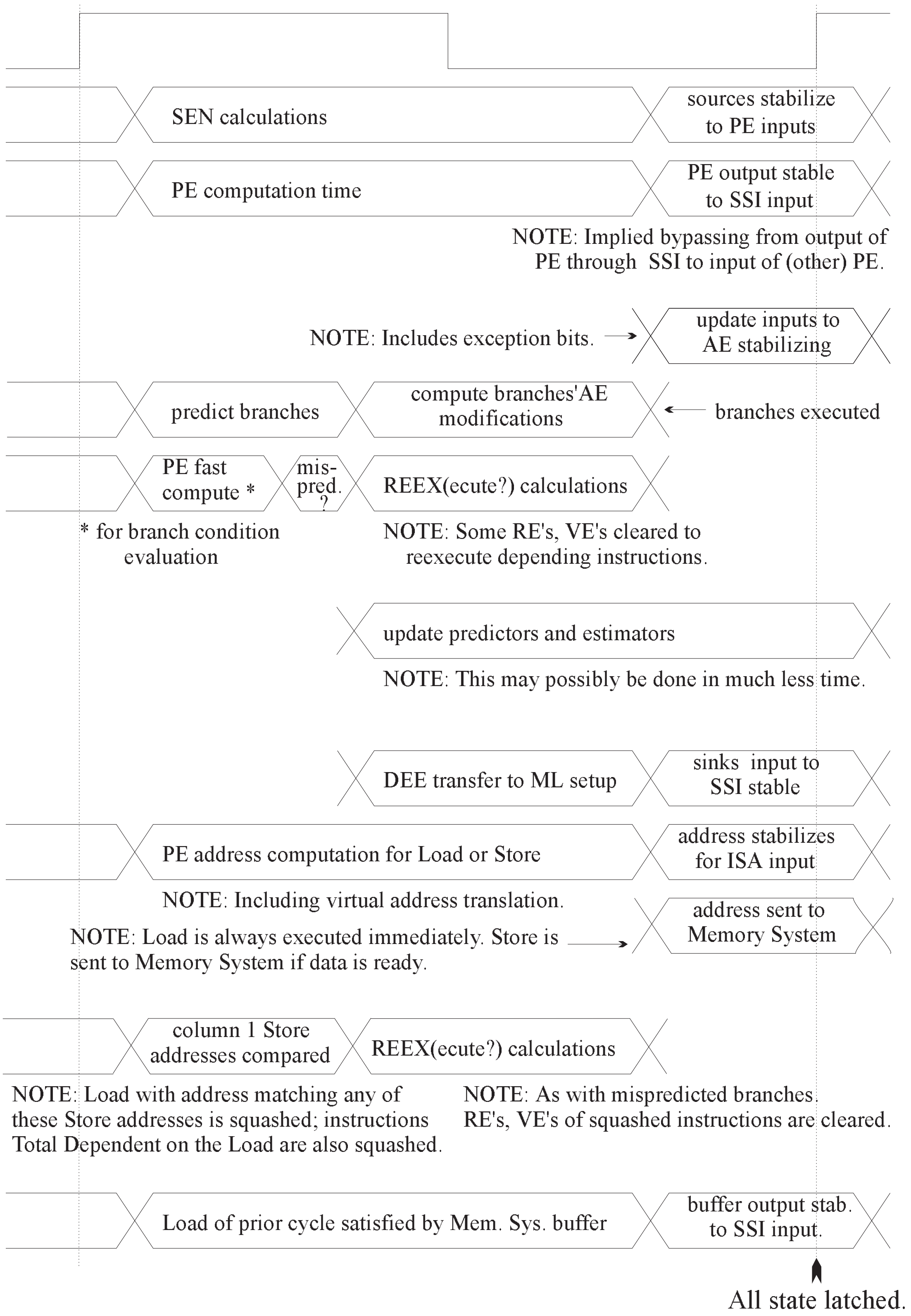

Figure 7: *Levo machine cycle timing.*

In-order Committing: The instruction sinks (results) of column 1 are sent to the memory system when the column is shifted left and retired. These sinks include ISA registers as well as memory locations. The memory system minimizes the number and ordering of the accesses. See [9] for details of that part of Levo.

The OOO core's branches' branch predictor state updates the central BTB's state when column 1 retires, as well. Thus, the usual desired caching effect of the BTB is kept in Levo.

4 Other Aspects of the Levo Implementation

4.1 Pipelining Control and Tags

The basic data path of an individual pipelined Processing Element (PE) is shown in Figure 8. The key attributes of this pipeline are as follows:

- to begin the computation, the input data must be latched into the first pipeline register; there is no combinational logic before the register's inputs;
- there is no output register; the combinational logic of the last stage flows to the instruction's sink register in the SSI, where it is latched and potentially forwarded to a later instruction;
- there is a *column* or *destination tag* associated with each pipeline register holding intermediate results of the computation; it indicates which column of the SSI the result should be written to; the destination SSI row is always the same, being the one associated with the PE.

As indicated in the figure caption, the tag is modified to reflect the latest position of the result's destination column. This column changes when the AE-like matrices are shifted left (a column is retired), effectively moving the destination column to the left.

4.2 Exceptions and Interrupt Handling

Exceptions and interrupts are handled *precisely* in Levo. In other words, the paused program is stopped such that there is one value of the Program Counter for which all of the program dynamically before the PC has executed, and all after the PC is unexecuted. Further, for exceptions, the PC points to the exception-causing instruction during the pause period.

Exceptions (directly or indirectly program-generated events) are handled somewhat differently from interrupts, and thus are treated separately below. Common examples of exceptions are: software traps, floating point overflows, and page faults. In some sense, interrupts can be viewed as *externally-generated* exceptions. The main difference is that while one dynamic instruction is associated with (causes) an exception, there is no such association with an interrupt; thus, the method of handling an interrupt is less constrained and complex (as far as the concurrency hardware is concerned).

4.2.1 Exception Handling

There are a few important issues that must be addressed by the Levo exception-handling system:

- the exception must be handled precisely, as described above;
- multiple exceptions occurring in any order must be handled correctly, in particular:
- an exception E_2 occurring later with respect to wall clock time than another exception E_1, but occurring *earlier* in the dynamic instruction stream (E_2 is before E_1; see Figure 9) must be handled before E_1.

The latter issue must be handled that way for two reasons: the handling of E_2 may eliminate E_1, and the complexity must be kept manageable.

The solution is illustrated by the example given in Figure 9. The figure shows the contents of the Really Executed (RE) matrix at four different times: a., b., c., and d., during the handling of an exception. (The Virtually Executed [VE] matrix may also have instructions executed; it is treated similarly to RE.)

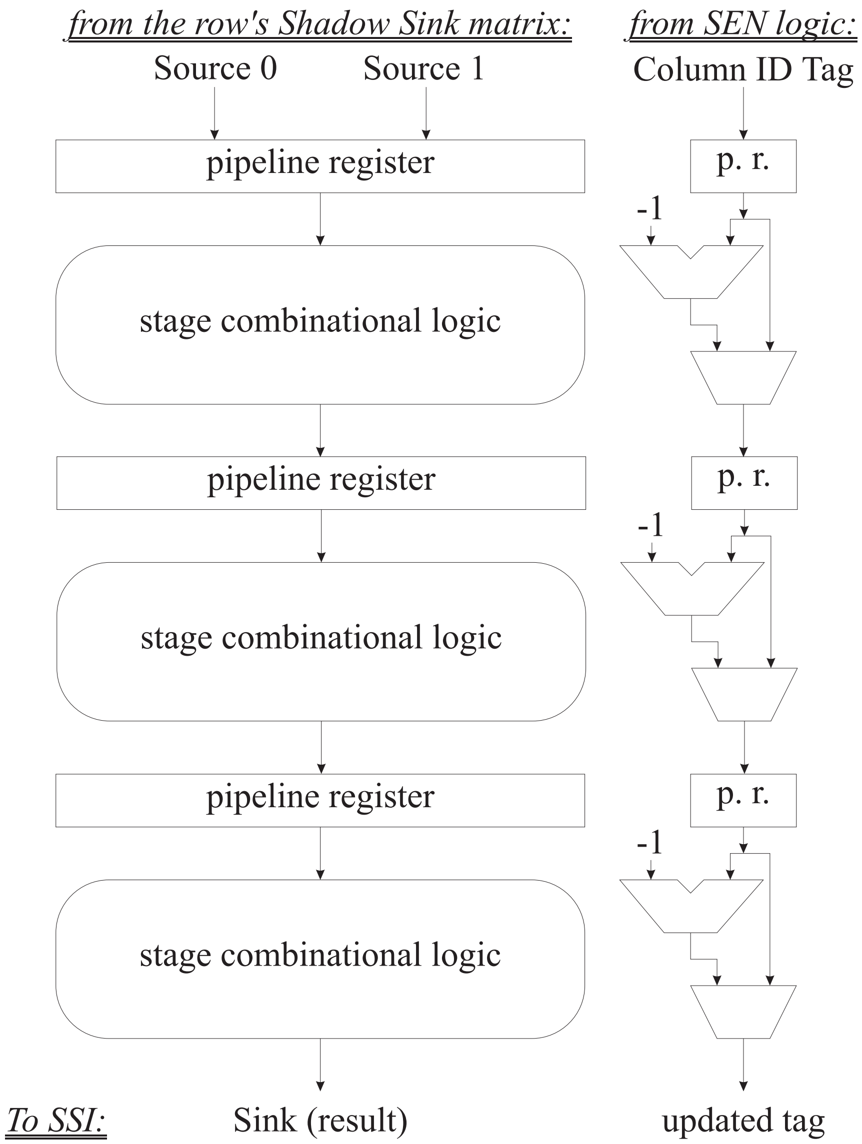

Figure 8: *Pipelined Processing Element (PE) data path.* A 3-stage PE is shown. The arithmetic or other logical function of the PE is divided into three stages, in the usual way. A column (destination) ID tag, accompanying the data as it moves through the PE, is initially set to indicate the column of issuance of the instruction being computed. If IQ columns are retired during the computation, shifting the IQ matrix's contents to the left, the tag is modified to indicate the revised instruction location. When the instruction's computation is finished, and appears at the end of the pipeline, the associated tag indicates the destination column of the instruction result. In this figure, integer tags are assumed, and are decremented at column retiring; they could instead be bit vectors, in which case they would be updated by shifting them left one bit when column 1 is retired.

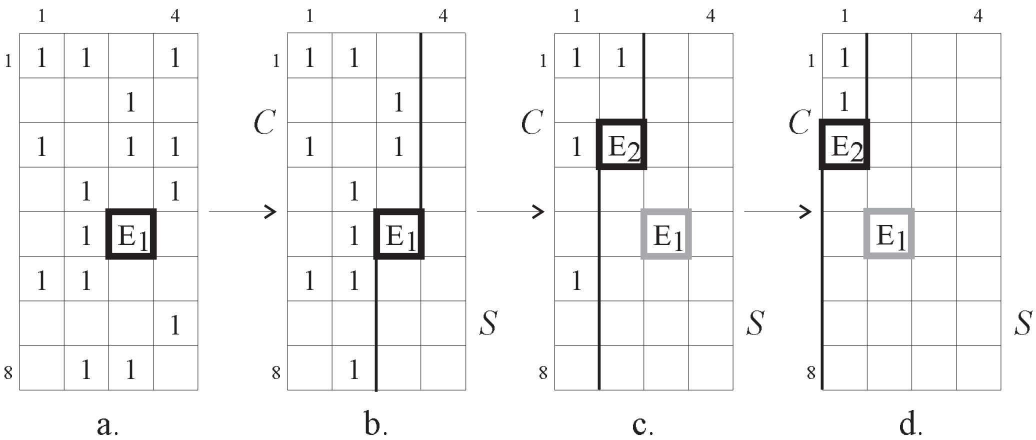

Figure 9: *Exception handling.* Each grid of squares represents the RE matrix, at different times, time increasing to the right. At time a. the first exception occurs. At time b. the instructions dynamically after this exception are squashed. These instructions are indicated by the Squashed area to the right of the bold lines. The other instructions, indicated by the Continued area to the left of the bold lines, are to finish execution. This is modified, however, by the occurrence of a second exception, E_2, dynamically earlier than the first. The RE matrix at time c. shows the result: the first exception E_1 is no longer serviced, but is squashed (indicated by its gray square), like the other instructions in the new, larger, S region. The first column then finishes execution and is retired. The remainder of the C instructions also finish execution (time d.), and exception E_2 is then serviced. This figure is more fully described in the text.

At time a., many instructions have executed, as indicated by the (scattered) 1's in RE (empty squares in the figure correspond to RE values of 0). Also at time a. instruction $I_{5,3}$ has generated an exception (E_1) during its execution. Thus, it does not finish execution immediately (not until its exception has been serviced), and its RE element remains at 0.

In order for E_1 to be handled correctly, all executed instructions dynamically after it must be squashed and kept from executing, while allowing all those instructions dynamically before it to complete execution. The former is illustrated by the RE matrix at time b., in which all of the 1's after E_1 have been cleared (set to 0). This region, the *Squashed* region, indicated by the S, is delimited from the *Continued*-to-be-executed region C by the vertical bold lines.

At time c. instruction $I_{3,2}$ generates another exception (E_2); again, the bold lines separate the two regions. Since E_1 falls into the Squashed region, it is <u>not</u> serviced. (Note that E_1 may recur, once E_2 has been serviced, and will be handled then.)

After time c. the instructions in the Continued region finish executing. Since all of the instructions in column 1 are executed, the column is shifted left and retired, with its sinks committed to memory. The picture of RE at time d. is taken after the shift, and after all of the Continued instructions have executed; thus, $I_{1,1}$ and $I_{2,1}$ have their RE bits set to 1. Once these instructions' sinks are committed to memory, exception E_2 is serviced. After servicing, the IQ is reloaded, starting at the PC value corresponding to the instruction causing E_2 (the address of instruction $\mathrm{IQ}_{3,1}$ as of time d.).

The exception has thus been handled, precisely.

4.2.2 Interrupt Handling

Since an interrupt does not correspond to a single instruction, the major issues have to do with the timing of the service, and the overall efficiency of operation. In all cases our basic stratagem is to allow one or more columns to completely finish execution, and to squash the rest, before jumping to the interrupt service routine. Thus, a precise interrupt is achieved, with a simple control structure.

The main design decision to be made is: how much of the current window should be fully executed before transferring control to the interrupt service routine? If the interrupt is *urgent*, such as due to a power interruption, then it would be wise to squash all but the overflow or first columns, allow their commits to occur, and then service the interrupt. In this fashion, the interrupt service routine might be invoked as early as one cycle after receipt of the interrupt signal.

On the other hand, if the interrupt is not urgent but rather *casual*, such as due to the receipt of a character from a dumb terminal, it might be better to let more of the active columns of the window finish executing, while squashing fewer, later columns. In this way, less work would be wasted than in the case of the urgent interrupt. Exactly how many columns should be allowed to finish execution is not clear at this time; study is needed.

5 The Levo-32 Physical Implementation

We now describe our basic strategy for realizing our own multi-chip Levo prototype, Levo-32.

We plan to use standard field-programmable gate arrays (FPGAs) for the logic of Levo, and RISC processors such as MIPS R4400's for the Processing Elements (PEs), to reduce design and fabrication costs and time, as well as to be able to use existing software (compilers, operating systems, etc.). The theory and microarchitectural approaches of Levo are general enough to handle any instruction set architecture, however. Physically, Levo-32 will use thirty-two 32-bit MIPS processors to mimic the function of the 32 PE's. Each PE will likely have one SBlock associated with it.

6 Comparison to Other Work

Virtually all commercial and most research superscalar machines use only simple SP execution with minimal register data dependencies[4]. VLIW and other software-based machines are an alternate approach; see [12] for a discussion and comparison of these and other methods.

The multiscalar architecture[5] realizes concurrently executing tasks in multiple localities of SP executions. The realization of concurrent "localities" is a type of reduced control dependencies, originally devised in [6], and improved to minimal control dependencies in [7, 8]. However, minimal control dependencies are not realized throughout the machine, and SP execution is applied to the overall structure; that is, when there is a misprediction in a locality ("engine"), all state dynamically after it, including that in other localities, is squashed. This greatly restricts the ILP exploitable by the machine; this is confirmed by multiscalar measurements: the best IPC demonstrated for it to date is about 2.5 (this includes data speculation). It is not known what the potential (upper limit) ILP speedup of the multiscalar model is, as it has never been determined (to our knowledge).

In contrast, Levo realizes both of the advanced DEE and MCD models, giving potentially high ILP speedups; they will likely be much higher than the multiscalar ones when precisely comparable simulations are made.

The attributes of the multiscalar and Levo memory data dependency and memory speculation models are somewhat similar, but we believe the Levo model will result in higher effective memory bandwidths; see [9] for a detailed comparison. The multiscalar data speculation model is more advanced than Levo's currently is, but it is not clear that the difference is significant in terms of performance; Levo likely has ample exploitable ILP available to it. An interesting area of future research would be to determine the overlap of realized ILP between data speculation and advanced control speculation techniques.

It is likely that DEE, etc., could be applied to the multiscalar model. It is another area for future research.

7 Summary

We have described the basic microarchitecture and operation of Levo-32-chip, a hypothetical single-chip prototype of the Levo computer, embodying disjoint eager execution and minimal control dependencies. Future work will simulate Levo-32-chip and provide accurate measurements of its performance and dimensional tradeoffs.

References

[1] M. Franklin and G. S. Sohi. A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[2] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the Seventh Annual International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 138–147. ACM, October 1996.

[3] A. I. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24nd Annual International Symposium on Computer Architecture (ISCA-24)*. IEEE and ACM, June 1997.

[4] D. B. Papworth. Tuning the Pentium Pro Microarchitecture. *MICRO*, 16(2):8–15, April 1996.

[5] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. IEEE and ACM, June 1995.

[6] G. S. Tjaden. *Representation and Detection of Concurrency Using Ordering Matrices*. PhD thesis, The Johns Hopkins University, 1972.

[7] A. K. Uht. An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*. University of Hawaii, in cooperation with the ACM and the IEEE Computer Society, January 1986.

[8] A. K. Uht. A Theory of Reduced and Minimal Procedural Dependencies. *IEEE Transactions on Computers*, 40(6):681–692, June 1991.

[9] A. K. Uht. High Performance Memory System for High ILP Microarchitectures. Technical Report 0797-0002, Department of Electrical and Computer Engineering, University of Rhode Island, August 1997. Available via `http://www.ele.uri.edu/~uht`.

[10] A. K. Uht. Verification of ILP Speedups in the 10's for Disjoint Eager Execution. Technical Report 0697-0001, Rev. A, Department of Electrical and Computer Engineering, University of Rhode Island, July 1997. Available via `http://www.ele.uri.edu/~uht`.

[11] A. K. Uht and V. Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 313–325. IEEE and ACM, November/December 1995.

[12] A. K. Uht, V. Sindagi, and S. Somanathan. Branch Effect Reduction Techniques. *COMPUTER*, 30(5):71–81, May 1997.

[13] T.-Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual Symposium and Workshop on Microarchitecture (MICRO-24)*. ACM and IEEE Computer Society, November 18-20 1991.