

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

Technical Report No. 072001-100

HDLevo – VHDL Modeling of Levo Processor Components

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

Email: {iota,uht}@ele.uri.edu
Web: www.ele.uri.edu/~uht

July 20, 2001

Abstract¹

Levo is a prototype uniprocessor whose goal is to achieve ILP in the 10's. HDLevo is an effort to create VHDL models of key components of the Levo prototype as a verification of the feasibility of constructing Levo, and to provide very accurate hardware size estimates for those components. To date, models of the Active Station and the Register Filter Unit have been completed, along with a behavioral VHDL test harness. Gate and transistor counts for the completed hardware models are presented. The test environment and methodology are explained.

Copyright 2001 University of Rhode Island.

¹ This work was partially supported by the National Science Foundation through grants: MIP-9708183, DUE-9751215, EIA-9729839, by the URI Office of the Provost, by an equipment grant from the Champlin Foundations, by software donations from Mentor Graphics Corporation and Xilinx Corporation. Patents applied for. This work has been submitted for publication

Contents

1 Introduction	3
2 Active Station	5
2.1 Load Bus Interface	5
2.2 Register Forwarding Bus Interface	6
2.3 Predicate Forwarding Bus Interface	8
2.4 Register Backwarding Bus Interface	11
2.5 Memory Forwarding Bus Interface	11
2.6 Memory Backwarding Bus Interface	12
2.7 Processing Element Bus Interface	13
2.8 Processing Element Return Bus Interface	13
2.9 Active Station Register File	13
2.10 Active Station Control Unit	13
3 Register Filter Unit	15
4 HDLevo Test Harness	16
4.1 Processing Elements	17
4.2 Levo Testbed	17
4.3 MIPS Assembler	17
5 Synthesis Results	18
6 Conclusions	18
7 References	20
8 Appendices	21
8.1 Active Station Component Design Reports	21
8.1.1 Register Forwarding Bus Interface	21
8.1.2 Register Backwarding Bus Interface	24
8.1.3 Memory Forwarding Bus Interface	25
8.1.4 Memory Backwarding Bus Interface	26
8.1.5 Predicate Forwarding Bus Interface	27
8.1.6 Processing Element Bus Interface	29
8.1.7 Processing Element Return Bus Interface	30
8.1.8 Active Station Register File	31
8.2 Output from the Xilinx Alliance Synthesis Tools	34
8.2.1 Active Station	35
8.2.2 Register Filter Unit (Column Head)	35
8.2.2 Register Filter Unit (standard)	35

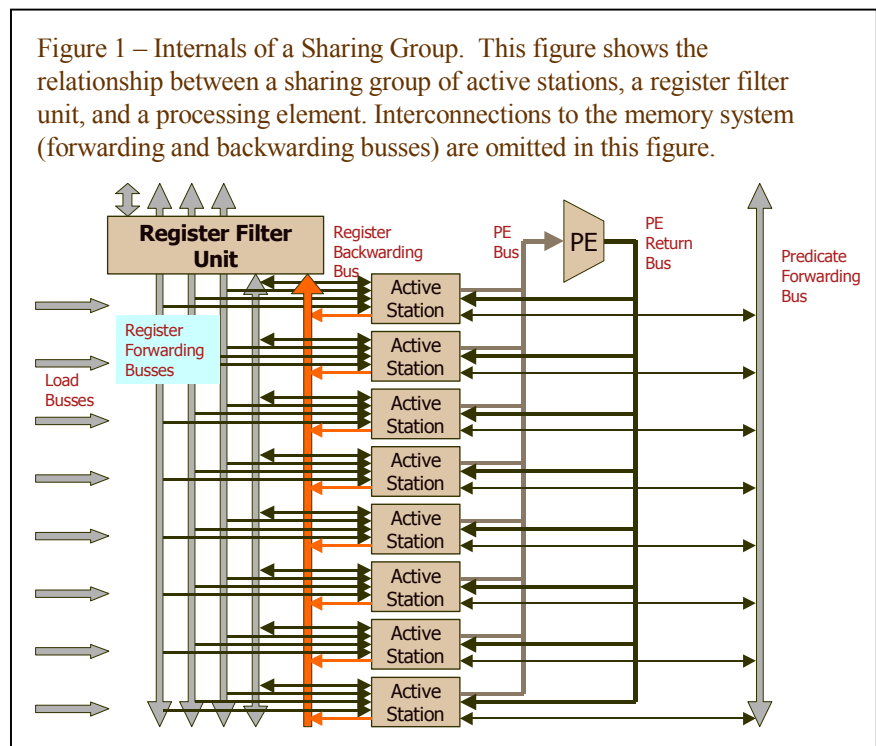
1 Introduction

The Levo project seeks to develop a new architecture for uniprocessors, which will allow very high levels of parallelism to be exploited, without requiring any changes to existing instruction set architectures. The goal of Levo is to develop a MIPS R3000 compatible processor, which can achieve an average of 20 instructions executed per cycle. Levo can achieve such high levels of ILP by using a novel execution model called *resource flow* computing.

Resource flow computing is an execution architecture whose guiding philosophy is to attempt to utilize all available execution resources in every cycle. Instructions are speculatively executed if an execution resource is available, even if the inputs for that instruction are not yet available. Instead the inputs to the instruction are speculated, branch direction and data values are predicted and the instruction is speculatively executed. Later, when the inputs of the instruction do become available, if it is discovered that the speculative execution result is incorrect, instructions are re-executed as necessary. The name “resource flow” comes from the mental image of processor resources “flowing” to active stations that have an instruction to execute (possibly speculatively), rather than being driven by the availability of final committed values for an instructions inputs. The availability of resources drives execution, not the availability of inputs.

In Levo, the traditional instruction reorder buffer is replaced by an array of *active stations* called the *execution window*. The execution window is organized into *columns*, and the columns are further sub-divided into *sharing groups*. The

HDLevo model realizes a machine with 8 columns of 4 sharing groups, with 8 active stations per sharing group, for a total of 256 active stations. These dimensions represent only one among many of the possible machine configurations being studied in the Levo research effort, however, results thus far have been promising for such a setup. Each active station tracks the state of a single instruction in the processor’s static execution path. Figure 1 shows a diagram of a sharing group.



The execution window is loaded with instructions starting with the first active station, and further instructions are loaded into the execution window following the most likely path through a process’s static control graph. When all the instructions in the first column have fully executed, the instructions in the column are *committed*, i.e., writes to the memory system are completed, processor exceptions are raised, and the column is *retired*. Once a column has been

retired, all the columns in the system are renamed, and the now empty column is loaded with the next set of instructions.

In addition to a set of active stations, each sharing group also contains a single *processing element*. A processing element is a set of execution resources, such as an ALU and FPU, and may be pipelined, or multiple-issue. In HDLevo, each processing element is a single-issue processor that can execute an instruction or perform an address calculation in one cycle. As there is one processing element per sharing group, there are a total of 32 processing elements in HDLevo.

The active station is the key component of Levo, responsible for the proper execution of a single dynamic instruction from loading to retirement. The active station is similar in concept to the reservation station described in [1]. The role of an active station is to monitor a set of “forwarding” busses for inputs to its instruction, and, as updates to each input are received, dispatch the instruction for execution. Once an execution result is returned from the processing element, the active station broadcasts the output of the instruction on the forwarding busses. In this way, active stations pass their results to one another over the forwarding busses. One of the key features of the forwarding bus architecture is that the length and complexity of each forwarding bus grows linearly with the number of active stations. This is in contrast to the complexity of many schemes for explicitly calculating the data and control dependencies among instructions, whose complexity generally grows quadratically with the number of instructions. By allowing instructions to execute speculatively and then re-execute when correct inputs become available, Levo does not have to precalculate the dependencies between instructions; incorrect results are simply discarded and recalculated.

As the number of active stations increases, so too would the number of loads on the forwarding busses. As processor clock speeds continue to increase, it is the distance signals must travel on bus lines and the number of capacitive loads that limits clock rates – more so than the complexity of the logic a signal must pass through. Thus, it is unrealistic to imagine a single forwarding bus that can connect 256 or even greater numbers of active stations, and still carry signals with reasonably delay. Additionally, since in any particular cycle, many active stations will generate outputs to the forwarding busses, bus contention would become a problem. Thus, for each of the kinds of forwarding busses in the system, the bus is divided into segments called *forwarding spans*. Each forwarding span connects a subset of the total number active stations in the processor. Some bus systems are local to a single sharing group. Others may span multiple sharing groups or an entire column. At the boundaries between consecutive forwarding spans sit *filter units*. Each filter unit acts as a buffer, receiving signals on its input bus, and competing with other bus masters on its output bus to forward the buffered signals. Thus, it costs at least one cycle for a bus signal to move from one forwarding span to the next. Longer forwarding spans mean the total latency for a signal to reach all active stations is lower. Shorter forwarding spans mean fewer loads on each bus and less capacitance in the bus, resulting in better clock speeds.

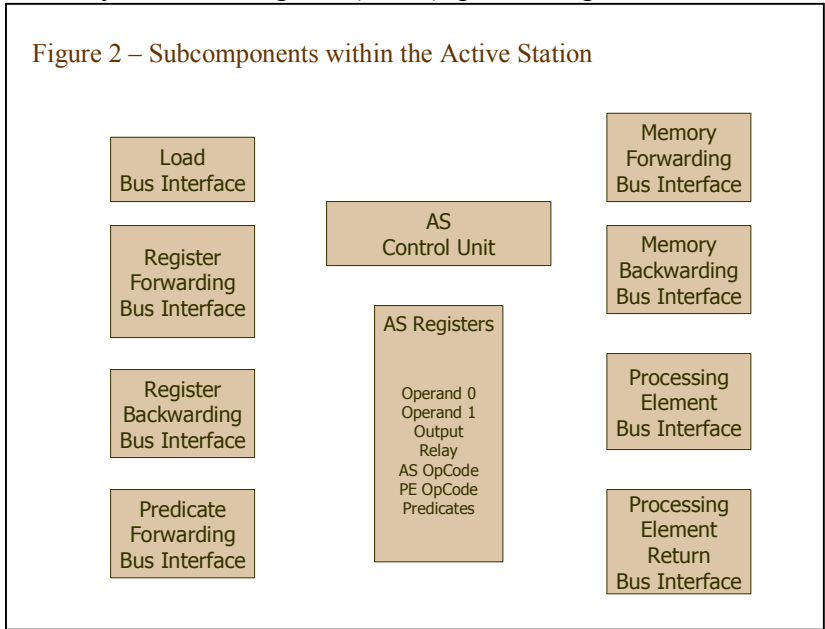
Filter units also have additional duties beyond simply relaying signals from one forwarding span to the next. For example, the register filter units maintain copies of the values of each of the MIPS architectural registers in Levo, and the memory filter units interact with the memory hierarchy to perform loads and stores.

The remainder of this technical report describes the details of the HDLevo model components that have been completed to date and the test environment for verifying proper execution. Section 2 describes the details of the active station component and its HDLevo

realization. Section 3 describes the register filter unit. Section 4 describes the HDLevo test harness. Section 5 presents hardware size results for the synthesis of the active station and register filter unit. Section 6 presents some conclusions.

2 Active Station

As described in the introduction above, the active station is the component of the Levo machine which tracks the inputs and outputs of an instruction and competes for the execution of that instruction. Each active station is connected to 8 bus systems, the load bus (LB), register forwarding bus (RFB), predicate forwarding bus (PFB), register backwarding bus (RBB), memory forwarding bus (MFB), memory backwarding bus (MBB), processing element bus (PEB) and processing element return bus (PERB). The active station can be thought of as a type of switch board and memory – it listens for its inputs on the various busses, updates its state based on them, and then generates bus signals to communicate with other parts of the Levo processor. The HDLevo active station model is organized as a set of sub components, each of which is responsible for interfacing to one of these bus systems. In addition to these interface components, the active station model contains one component which stores all the execution state (operand and result values and addresses), and a final component which generates control signals and coordinates the efforts of all the other components.



The active station model contains one component which stores all the execution state (operand and result values and addresses), and a final component which generates control signals and coordinates the efforts of all the other components.

The following subsections detail each sub-component within the active station model. The concepts and details of the operation of the Levo processor as a whole are introduced in the discussion of the relevant sub-components. For most components, a component design report can be found in the appendices detailing the input and output signals from the component and explaining specific VHDL implementation issues regarding the component.

2.1 Load Bus Interface

The load bus connects each active station to the *load buffer*. The instruction fetch unit continually loads new instructions from memory following the most likely path across branches and places those instructions in the load buffer. When a column is retired, the entire column is loaded with a new set of instructions in a single cycle via the load bus. The load bus carries the instruction op code for the active station along with branch prediction values from the fetch hardware. In the HDLevo model, instruction op codes are partially decoded by the fetch hardware, and a decoded op code is transmitted to the active station, along with additional *predication* information used to control instruction execution.

The HDLevo load bus is a total of 100 bits wide. There are 32 parallel load busses in HDLevo, such that an entire column of active stations can be loaded simultaneously. The load busses for a particular row of active stations are shared across the columns. The total width of the load bus breaks down as follows. 32 bits carry the original undecoded MIPS R3000 op code. The intention of HDLevo is to eventually use real MIPS processor cores as the processing elements, thus it is advantageous to include the full op code in addition to the partial decodings. Note that in an optimal implementation of Levo, some of the duplicate information carried in these 32 bits could be eliminated. 4 bits carry a simplified op code for the active station, used to distinguish ALU, branch, load, store, etc instruction classes. 10 bits carry the two register operand addresses for the instruction. One or both addresses may be unused. 5 bits carry the output register address of the instruction, which again may not apply to all instructions. 49 additional bits carry predication information for the active station. These 49 bits are composed of: a predicate address (8 bits), an initial predicate value (1 bit), an array of four canceling predicate addresses (8 bits each), an array of four initial canceling predicate values (1 bit each), and four 1 bit flags to indicate if the canceling predicates are in use. See the section on the Predicate Forwarding Bus below for a discussion of predication in Levo and details on how these predicate addresses and values are used.

2.2 Register Forwarding Bus Interface

The register forwarding bus system allows the output values of assignment instructions to be forwarded from one active station to the next. Each active station is connected to a number of RFBs; in the case of the HDLevo model, 4. One of these RFBs is both read and written to, the others are only monitored. Each RFB has a *forwarding span* (measured in number of sharing groups) equal to the number of RFBs. The start of each RFB is offset by one sharing group from the previous RFB. Thus each active station monitors 4 different RFBs, each having overlapping forwarding spans. One of these RFBs has its origin in the sharing group to which the active station belongs. This RFB (designated as index 0 relative to the active station) is the RFB to which the active station may write. A single transaction on an RFB transmits three pieces of information: the address of the register (5 bits), its value (32 bits), and a *time tag* (nominally need only be 8 bits, but is decoded to 13 for efficiency, as explained below). Time tags as they are used on the RFB are a key innovation of the Levo processor architecture.

A time tag is a designation for the active station that originated a particular register value. When an active station forwards a value on an RFB, it also forwards its address in the instruction window as a time tag. Other active stations that observe the forwarding transaction on the RFB compare the time tag of the forwarded address and value to the time tag of any value they have previously observed and used for computation. If the time tag of the forwarding transaction is equal to or newer than the time tag of the previously observed value, the receiving active station captures or *snarfs* the new value and discards any computations that have been performed with the previous value. A higher time tag indicates that a value was generated by an instruction further forward in the dynamic instruction stream. For correct operation, each active station must compute its final output value using the output from nearest preceding instruction that writes to its operands.

As columns of instructions are retired from the instruction window and columns are renamed, simultaneously all time tags in the system are decreased by the number of active stations retired (the number of active stations in a column). In this way, as an active station's index in the instruction window changes due to its column index changing, so, too, do the time

tags of every value in the instruction window that originated at that active station. Time tags decrease to a minimum time tag of 0, indicating a value that originates from an instruction prior to any instruction in the execution window. Note that a time tag of 0 indicates that a particular value is no longer speculative – the instruction that generated it has retired.

In the HDLevo model, time tags are realized as the combination of an encoded binary index for the sharing group in which the originating active station is a member (requiring 5 bits), plus a one-hot encoding of the active stations index within its sharing group (an additional 8 bits for a total of 13). The advantage of this partially decoded format lies in the nature of the RFBs within a particular sharing group. Active stations must be able to perform many fast and hardware efficient comparisons on time tags. It is possible for the same register address to appear on all four RFBs visible to a particular active station in the same cycle with different values and different time tags. In this situation, time tags across all four RFBs must be compared as well as the time tag of any previously stored value. Additionally, since an RFB is a single tri-stated bus within a sharing group, an active station may observe a value with a time tag higher than its own at its inputs, and must avoid snarfing these values. The partially decoded time tag allows for optimizations of this comparison hardware.

The register forwarding bus interface is perhaps the single most complex sub-component of the active station. This is due to the number of time tag comparisons necessary to ensure that the component always selects the correct RFB from which to snarf a value, and avoids snarfing values when unnecessary.

A fundamental assumption in the HDLevo model is that insufficient time is available in a single clock cycle for values to propagate over a bus, and for time tag comparisons (or other computation) to be performed. Thus, each register forwarding bus transaction must be registered upon reception in the RFB interface. Then, in the following clock cycle, the values from the bus transaction are passed through the comparison hardware to determine if the active station should snarf the bus transaction for one or both of its operands, or as its *relay value*. (Relay values and their use are discussed in the section on the Predicate Forwarding Bus below. For now, they can be considered as a third operand to an assignment instruction).

The register forwarding bus interface must contain one comparator to match operand addresses and bus transaction addresses for each combination of bus and operand, for a total of 12 5-bit comparators. An active station is only interested in snarfing values that are for its operand's register addresses, and ignores all other bus transactions.

Once it has been established that one or more RFBs carry register addresses that the active station is interested in, the time tags of the bus transactions must be considered. First, each transaction time tag must be compared against the output time tag of the active station itself. If the transaction's time tag is greater than the output time tag of the active station, then the transaction is the output of an instruction after this active station's, and should not be snarfed. Note that this comparison is only necessary for RFB 0 (other RFBs can only be written to by active stations which precede this active station, as they must be in preceding sharing groups). Further, because the active station index portion of a time tag is decoded, this comparison can be performed in a very hardware efficient manner.

With these two conditions met, the time tag must be compared against the time tags of previously snarfed operand values. Only bus transactions with equal or greater time tag will be snarfed. Finally, after all these time tag comparisons, it is still possible that more than one RFB carries a transaction appropriate for snarfing. Among these candidates, the transaction with the

greatest time tag must be selected. Thus, another set of time tag comparators is required, comparing each combination of RFBs.

In addition to address and time tag comparators, HDLevo also compares the value in a bus transaction to any previously snarfed operand values, and suppresses a snarf when the values match. This comparison is not necessary for correct execution, but is essential for performance reasons, as every snarf forces a re-execution of the active stations instruction.

The register forwarding bus interface is also responsible for writing output values to RFB 0 when directed to do so by the active station's control unit. The hardware required to implement this is a set of tri-state drivers and bus request logic, and is small relative to the hardware necessary for snarfing.

2.3 Predicate Forwarding Bus Interface

One type of instruction that has largely been omitted in the above discussions is the branch instruction. In general, all branches are predicted at fetch time, and the instruction window is loaded with instructions following the predicted path of execution. In the case of backwards branches (loops), this indicates that a particular instruction in a program may appear in several active stations in the instruction window; each occurrence corresponding to a different iteration of the loop. A misprediction of the end of such a loop, or a misprediction of a branch which targets instructions outside the instruction window, results in the remainder of the instruction window having to be flushed and reloaded with instructions starting at the correct address. Levo strives to avoid this expensive situation using two key novel approaches. The first of these approaches is *veiled-explicit predication*, and the second of these is *disjoint eager execution* (DEE) [2]. The latter technique supplements the instruction window with additional columns of active stations called DEE-columns, which are loaded with instructions following the non-predicted path of execution at a particular branch. Should execution of the branch determine that it was indeed mispredicted, the contents of the DEE column are copied into the instruction window and only a minimum of cycles are lost. The intuition behind DEE is that an instruction that appears on the non-predicted execution path for the first branch in the execution window is in many cases more likely to be executed than the instruction at the 256th position in the execution window. The 256th instruction may only be executed if perhaps 20 or more branch predictions are correct, while the DEE instruction's result may be required if only a single prediction is incorrect. The degree to which DEE may be able to improve performance depends on the frequency and accuracy of branch prediction. DEE is not included in the current HDLevo test harness, and no further discussion of its implementation is made in this report. DEE is planned for addition in the next version of HDLevo.

Veiled explicit predication, on the other hand, is essential to Levo and is modeled in HDLevo. Predication refers to the well-known technique of replacing forward conditional branches with an assignment to a boolean predicate register, and then conditioning the execution of instructions between the branch instruction and its target (the branches *domain*) on this predicate. For example, the sequence of instructions:

```
10  if (a == b) goto 30
20  c := d
30  ...
```

can be replaced with the following:

```
10  pred := (a == b)
20  if (not pred) then c := d
30  ...
```


In many modern instruction set architectures (for example the Intel IA-64), predicate registers appear as part of the programmer-visible set of architectural registers, and instructions must be explicitly assigned a predicate register address. Hence, predication is called visible explicit. In the Levo architecture, however, forward branch instructions with a domain smaller than the instruction window are automatically converted into predicated assignments by the instruction loading hardware, and the predicate register addresses may not be directly selected by the programmer. Therefore, this novel form of predication is called veiled explicit predication.

The instructions in the instruction window can be divided into three classes, forward branch instructions eligible for predication (subject to the domain size restriction; namely that the branch and its target will both appear in the instruction window, that is, they are less than 256 instructions apart), targets of such a branch, and any other instructions. Note that it is possible for an instruction to be both a forward branch, and itself a target, in which case it belongs to both classes. As the instruction fetch hardware loads each instruction into the execution window, it assigns to each instruction an input predicate address, called its p_{in} . Initially, this p_{in} is a special code indicating an always-true predicate. When a branch instruction is encountered, the branch is assigned two output predicate addresses. The first is called the branch's output predicate or p_{out} . The value of this predicate will be the inverse of the result of the branch's condition, once it is resolved, logically ANDed with the branch's p_{in} . The second output from a branch is called its canceling predicate or cp_{out} , and its value is the branches condition ANDed with the branch's p_{in} . A particular p_{out} address and cp_{out} address always correspond to one another (ie the same address is used for both). Any instruction following a branch, including another branch, uses the p_{out} of the branch as its p_{in} . Expressed in words, this means that the input predicate controlling the execution of instructions following a branch is the output predicate of that branch. Following instructions in the branch's domain will only be executed if the branch is not taken.

When a second branch follows the first, its p_{out} will only be true if both its own condition is false (it is not taken) and the previous branch's condition was false (its p_{in} is true). In this way, branch condition chain their predicates together, and instructions following the second branch are conditioned upon the result of both comparisons. Branch targets, on the other hand, should be executed

address	Instruction	p_{in}	cp_{in}	p_{out}	cp_{out}	p_i
1	d = e + f	p_0	0	-	-	p_0
10	if bc goto 40	p_0	0	$p_{10} = \overline{bc} p_0$	$cp_{10} = bc p_0$	1
20	x = y + z	p_{10}	0	-	-	$\overline{bc} p_0$
30	u = v + w	p_{10}	0	-	-	$\overline{bc} p_0$
40	r = s + t	p_{10}	cp_{10}	$p_{40} = p_{10} + cp_{10}$ $= \overline{bc} p_0 + bc p_0 = p_0$	-	$= p_{out} = p_0$

p_{in} is effective input predicate for Instruction; its address is kept with Instruction.

cp_{in} is effective input cancelling predicate for Instruction; its address is kept with Instruction.

p_{out} is effective output predicate for Instruction. Is only set by branches and their targets. In general, for branches with branch condition bc , $p_{out} = \overline{bc} p_{in}$; for branch targets, $p_{out} = p_{in} + cp_{in}$. Address = Instruction's Time Tag.

cp_{out} is effective output cancelling predicate; only set by branches. In general, $cp_{out} = bc p_{in}$. Address = Instruction's Time Tag.

p_i is effective predicate for corresponding (same line) Instruction. For non-branch and non-branch targets, $p_i = p_{in}$. For branches, $p_i = 1$. (All branches are completely independent and may execute at any time.)
For branch targets, $p_i = p_{out}$.

in either of two cases. First, if their p_{in} is true, indicating that they were reached from the previous instruction. However, a branch target should also be executed if a branch that targets it is taken. When a branch is resolved as taken, its cp_{out} will be true. Thus, in addition to having a p_{in} input, branch targets also have one cp_{in} input for every branch that targets them. A branch target is executed if its p_{in} or any of its cp_{in} inputs is true. Furthermore, instructions following the branch target also must execute. Therefore, the branch target instruction also generates a p_{out} , and instructions following it use this value as their p_{in} . Using these rules, any sequence of forward branches can be converted into veiled explicit predication. Figure 3 shows a diagram of veiled explicit predication.

What is needed in HDLevo, then, is a hardware realization for the assignment of predicate and canceling predicate addresses, a method for communicating predicate values from active station to active station, and a protocol to follow in the case that an active station's input predicate is false. Assignment of predicate and canceling predicate addresses occurs in the load hardware, and is loaded into the active station on the load bus as mentioned above. Every instruction has an input predicate address, either address 0 for instructions with an always-true input predicate, or the active station time tag of the nearest preceding branch or branch target. Branch instructions have an output predicate and output canceling predicate address equal to their active station index. Branch targets have a set of input canceling predicate addresses, each corresponding to the address of the branch's source. In HDLevo, four canceling predicates are supported per instruction. If a particular instruction is targeted by more than four branches, the load hardware must add NOP instructions to the instruction window ahead of the targeted instruction, to be able to support the additional canceling predicate requirement. Note that as columns are retired from the instruction window, the column portion of active station time tags change, and predicate addresses throughout the execution window must also be adjusted to remain synchronized.

Predicate values are communicated from one active station to the next on the predicate forwarding bus. The predicate forwarding bus is in many respects similar to the register forwarding bus. However, in the current HDLevo model, there is only one predicate forwarding bus, not four in parallel as for the RFB. The forwarding span of each predicate forwarding bus segment is a full column, and predicate filter units (which forward predicate values from one span to the next, and arbitrate the bus for their column) sit at the top of each column. The predicate forwarding bus is 10 bits wide. Eight bits identify the predicate address for the bus transaction (the index of the initiating active station), one bit indicates the predicate value, and the other the canceling predicate value. Note that predicate and canceling predicate values are always generated and forwarded together. A taken branch generates a predicate value of '0' and a canceling predicate value of '1'. A not-taken branch generates a predicate of '1' and a canceling predicate of '0'. A branch instruction whose input predicate is '0' (that is to say, that is itself skipped by other branches), generates predicate and canceling predicate of '0'. A branch target generates a predicate of '1' if either its input predicate, or any of its canceling predicates are '1', and always generates a canceling predicate of '0'.

Snarfing predicates takes very little hardware, as only the address must be compared. The complexity of predication is not in the hardware of the predicate forwarding bus interface. The difficulty of predication is illustrated by the following example.

Consider an assignment instruction, say $R3 = R1 + R2$. Like any other instruction, this assignment is predicated, and an initial predicate prediction is loaded into the active station along with the instruction. Suppose that the initial prediction of the predicate is '1', and therefore, the

assignment instruction is executed, and forwards an output value for R3 equal to $R1 + R2$. Now, suppose that a branch prior to this instruction resolves as mispredicted, and the active station receives a new predicate value of '0'. The active station must suppress its execution. However, the instruction has already executed, and its output value has already been sent on to other active stations, which are now potentially performing calculations with an incorrect input. These other active stations need the value of R3 prior to the addition of $R1+R2$.

In order to correct this situation, each active station maintains a *relay* value, along with its operands and output value. The relay value of an instruction is the value its output register had prior to the instruction's execution. This relay value is snarfed from the register forwarding bus just as operand values are, and may be updated over time as previous active stations execute or re-execute. When an assignment instruction's predicate changes from a '1' to a '0', the assignment instruction outputs its relay value as if it was the instructions output value. This will force any later active stations, which have performed computations using the incorrect register value to recalculate their outputs using the correct value. If an update for an active station's relay value arrives while its input predicate is '0', it must relay the new value. In this way, correct execution is ensured.

2.4 Register Backwarding Bus Interface

In order for an active station to compute its output values, it must observe its operands on a register forwarding bus. In general, the active station will see its operands and outputs because of forwarding operations that are initiated by another active station. However, it is possible that an active station's input is not written to by any other instruction earlier in the window. For example, a register may be storing a variable that has not been written to for several hundred machine cycles (many more instructions than fit in the execution window).

To avoid the situation where an active station "starves" waiting for a register to be forwarded to it, each active station makes an explicit request for all of its operands. This request is made on the register backwarding bus. Each active station is connected to exactly one RBB. This RBB connects all the active station in a sharing group to the register filter unit situated at the top of the sharing group. Active stations compete for the RBB to request their operands. The register filter unit monitors these requests and compares them against values that it has observed on the bus, filling any requests that have not been filled by prior forwarding operations, using the most up-to-date value the register filter unit has observed.

In the current iteration of HDLevo, the register backwarding bus interface of each active station is very simple: it requests each of the instruction's operands one by one, and pays no attention to requests made by other active stations. The register filter unit ensures that requests for a particular address are filled only once, even if requested by multiple active stations. Contention for the register backwarding bus has not been a bottleneck in current HDLevo simulations.

The register backwarding bus is 5 bits wide, carrying a single register address per cycle. The register filter unit acts as the bus arbiter.

2.5 Memory Forwarding Bus Interface

As with registers, memory addresses that are written by a store instruction are often used by load instructions shortly after the store. Levo exploits this temporal locality by forwarding the output of store operations on memory forwarding busses, so that they can be snarfed by later

loads. Additionally, when a load requests an address (see the next section on the memory backwarding bus), the requested value is sent to the load on the memory forwarding bus system.

In HDLevo, there are four memory forwarding busses. The four busses are interleaved, so a particular double-32 bit word in memory may only appear on one of the four busses. Memory filter units (much like the register filter units) sit at the top of each column, forwarding memory operations from previous columns.

When a store operation is executed, the value to be stored is written by the active station to the appropriate memory forwarding bus based on the low order bits of the address. Along with the value, the time tag and the necessary bits of the address are written, and a flag indicating that the forward is a store operation. If the store operation is re-executed, it writes to the memory forwarding bus again with the same time tag. If any load operation after the store is waiting for the same address, it will snarf the stored value. At the boundary between columns, a memory filter unit receives the store operation.

The memory filter unit has two duties. First, it forwards the written value on to the subsequent column's memory forwarding bus. Second, it caches the value locally, so that it can satisfy any requests for that address that arrive on the memory backwarding bus. The cached writes will be written to the memory hierarchy when the column containing the memory filter unit is retired. If a second forward occurs for the same address, the value cached in the memory filter unit is overwritten.

The memory system must also account for two other possible changes to a cached write. First, a store instruction may be predicated false, that is to say, it may receive an updated predicate value indicating it should be skipped by branches. Secondly, since many memory operations calculate their target address based on a register operand, and this register operand may change, it is possible that a write on the memory bus may be issued with an incorrect address. In both situations, a write operation must be "rolled back", and its effects cancelled. In order to facilitate this, the memory forwarding bus system supports a *nullify* operation. When issuing a nullify, the active station writes the address and time tag to be nullified, and a flag indicating that a nullify transaction is occurring to the memory forwarding bus. The memory filter unit receives this nullify, passes it on to the next column, and then deletes any cached value for that address. If an active station snarfs a nullify transaction, it re-requests its value on the memory backwarding bus.

The memory forwarding bus interface in HDLevo has a number of duties. If the active station is a load instruction, it must monitor the appropriate memory forwarding bus, based on the load address, waiting for the value to arrive either from the memory hierarchy or from another active station. For a store instruction, the memory forwarding bus must issue forwarding transactions when the stores output and address are available, and issue nullify transactions when the write address changes or the instructions input predicate changes from true to false. However, since only one of the memory forwarding busses is used at a time per active station (based on its address) very few comparators are required relative to the register forwarding bus interface, which must always monitor all busses.

2.6 Memory Backwarding Bus Interface

Much like the register backwarding bus interface, the memory backwarding bus interface is used to request instructions' input values. Like the memory forwarding busses, there are four interleaved memory backwarding busses. Load instructions issue a request on the memory backwarding bus once they calculate their load address. These requests are received by the

preceding memory filter unit. The memory filter unit firsts attempts to satisfy the load request with values from its local cache. If there is no match, the memory filter unit issues the load to the remainder of the memory hierarchy. Once the loaded value is available from memory, the memory filter unit forwards the requested value on the appropriate memory forwarding bus, with a time tag of 0 (indicating a value that is older than any instruction in the instruction window).

Currently, there is no mechanism by which loads of incorrect addresses are “canceled”, however, such an enhancement may be possible in a future version of HDLevo.

2.7 Processing Element Bus Interface

The processing element bus connects each active station in a sharing group to the processing element belonging to that sharing group. Active stations dispatch instructions for execution by a PE on the processing element bus. The active station forwards the op code, operand values, and its own address within the sharing group. The bus arbitration of the processing element bus favors granting the bus to active stations with low indices, as these instructions results are more likely to be needed by later instructions and less likely to require re-execution.

In HDLevo, the processing element bus carries two 32 bit operands, one 32-bit op code, and a 5 bit active station index.

2.8 Processing Element Return Bus Interface

The processing element return bus returns results that have been calculated by the processing element to the appropriate active station. In HDLevo, all calculations are assumed to take only a single cycle. However, the HDLevo model also can support other execution models, where the processing element may be pipelined or multi-issue. Results on the processing element return bus are accompanied by the index of its destination active station.

2.9 Active Station Register File

The active station register file is not a register file in the traditional sense; it is the name of the component of the active station which stores all of the state of the active station. The register file stores all the operands, outputs, addresses, and time tags of the register station.

The register file stores a total of 377 bits of information. 50 bits each are for two operands (32 bit value, 5 bit register address, and 13 bit time tag). An additional 5 store the output address, 32 for the calculated output value, 32 for the relay value, and 13 to store the time tag of the last observed relay value. The time tag of the output of the active station is also calculated in the register file, based on the active station’s current index in the window. 32 bits store the instruction op code for the processing element, and 4 store the instruction class for the active station’s control unit. Another 32 bits store the memory address for load and store instructions. There are 8 bits for each of the predicate addresses (1 input predicate and 4 canceling predicates), 1 bit to store each of these 5 predicates, and 4 1-bit flags to indicate if each canceling predicate is in use. In order to support 64-bit floating-point operations, an additional 32 bits are required for each of the operands, relay, and output data registers. In the HDLevo model, all of these values are stored in flip-flops.

2.10 Active Station Control Unit

The active station control unit is the final component of the active station. This component generates control signals for the remainder of the active station based on the active

station's 4-bit op code class, the state of the input predicates (both stored in the register file), and the present state of a number of small internal state registers. The internal state registers indicate whether the data registers in the active station register file contain valid data, and direct the actions of the active station. These state registers are all 1 bit, and track whether a certain condition is true or not. The actions of the active station are determined by combinational logic based on the op code class and these state registers. Each of these state registers is now examined.

There are three state bits which track whether or not operand and relay values have been requested on the register backwarding bus. When an instruction is loaded into the active station, these state bits are cleared. Each bit is set when the corresponding operand or relay is requested on the register backwarding bus. Note that it may require several cycles for a request to be issued on the RBB due to bus contention. An instruction which is currently predicated true (its input predicate or any input canceling predicate is true) that requires an operand may not retire until the corresponding state bit is set. An instruction that is predicated false may not retire until the state bit indicating that the relay value has been requested is set. This ensures that all required values will be requested before the column can retire. If a value is requested and has not been supplied to the active station, the register filter unit owning the register backwarding bus will be aware of this and will prevent the column from retiring. In the HDLevo VHDL model, these state bits are referred to as the `sReg_OperandRequested` and `sReg_RelayRequested` state registers.

The `sReg_OutputValid` state register indicates whether the data value stored in the register file's output register is valid. This state bit is reset upon load, and upon any operand snarf. It is set whenever an execution result returns from the processing element. `sReg_OutputValid` is always set for NOP instructions.

The `sReg_OutputDirty` state register indicates whether an updated value in the output data register has been forwarded on the register forwarding bus or not. `sReg_OutputDirty` is cleared when the active station is loaded and anytime the output data register is successfully forwarded on the register forwarding bus system. `sReg_OutputDirty` is set whenever a result is returned from the processing element and the active station is predicated true. `sReg_OutputDirty` is also set anytime the active station's input predicate changes from false to true. The active station may not retire while `sReg_OutputDirty` is set.

The `sReg_RelayDirty` state register indicates whether the data value stored in the register files relay register requires forwarding. Note that this is only the case when the active station is predicated false. `sReg_RelayDirty` is set when a new relay value is snarfed, or when the active station's input predicate changes from true to false. `sReg_RelayDirty` is cleared on load and whenever the relay value is forwarded on the register forwarding bus. The active station may not retire if `sReg_RelayDirty` is set and the active station is predicated false.

`sReg_ExecutionPending` is set whenever the active station issues its instruction for execution to the processing element. It is cleared when results are returned from the processing element.

`sReg_PendingExecutionStale` is set if a snarf of any operand value occurs while `sReg_ExecutionPending` is set. This indicates that an operation has been sent to the processing element for execution, but that the operand values that were sent have since been updated. The execution result must be discarded. The operation can be issued for re-execution immediately. If a processing element takes more than one machine cycle to return an execution result, `sReg_PendingExecutionStale` must be replaced with an up-down counter sufficient to count the maximum number of cycles an execution could take. Thus, when repeated snarfs occur one after

the next, `sReg_PendingExecutionStale` counts the number of invalid results that will return from the processing element, before the final, valid result is returned. `sReg_OutputValid` may only be set when `sReg_PendingExecutionStale` is clear (zero if it is realized as a counter).

`sReg_PredsDirty` indicates whether a branch instruction or branch target must write an output predicate to the predicate forwarding bus before the active station may be retired. `sReg_PredsDirty` is always clear if an instruction is neither a branch, nor a branch target (branch targets have at least one valid input canceling predicate). For branches, `sReg_PredsDirty` is set whenever the branch condition is reevaluated and the result is different than the last result written to the predicate forwarding bus, or the branch's input predicate changes. For branch targets, `sReg_PredsDirty` is set when the input predicate or canceling predicates of the instruction change and cause a change in its output predicate. In either case, `sReg_PredsDirty` is cleared when the new predicate is forwarded on the predicate forwarding bus.

3 Register Filter Unit

The register filter unit is the component of the Levo machine which maintains copies of all the MIPS architectural registers and manages the flow of register values on the register forwarding busses.

The register filter unit has four input register forwarding busses, four output register forwarding busses, and one input register backwarding bus. The register filter unit must maintain a copy of each of the 32 MIPS architectural registers, along with a time tag corresponding to the value stored in each register. Whenever new values for any register appear on one of the input register forwarding busses, the register filter unit must compare the time tag of the new value with that of the old, and store the new value if the time tag is newer. Since it is possible for all four input busses to be active in a single cycle, the architectural register file must have at least 4 input ports. Furthermore, it is also possible for any combination of busses to carry values for the same address. In this case, time tags must be compared to verify which value, if any, should be snarfed. Because of the number of comparators required for this, the register filter unit requires the most hardware of any HDLevo component.

Three of the input register forwarding busses “pass through” the register filter unit. That is to say, the register filter unit monitors these busses, but does not write to them. These busses are designated as RFB 3, 2, and 1. The fourth input register forwarding bus (RFB 0) terminates at the register filter unit. This RFU is at the end of the RFB 0's forwarding span. The three “pass through” busses are re-indexed as 2, 1, and 0 when they leave the register filter unit, and a new register forwarding bus originates at the RFU, and is designated as the new RFB 3. In this way, each register forwarding bus has a forwarding span of four sharing groups.

The register filter unit writes values to its output RFB 3. Any value that is snarfed from input RFB 0 must be re-forwarded on output RFB 3 as soon as the bus becomes available. Note that if a value arrives on RFB 3 but is not snarfed, either because the value is unchanged or its time tag is older than that of the previously stored value, it is NOT rebroadcast, reducing unnecessary bus traffic. Additionally, any values requested on the register filter unit's backwarding bus must be forwarded on RFB 0. By performing these two functions, the register filter unit guarantees that every active station sees all the operand values that it requires to perform its computations. A register filter unit prevents the column it is in from being retired until it has performed all pending forwarding operations.

There are actually two varieties of register filter unit in the Levo machine. One variety is located at the top of each Levo column, at the head of the first sharing group in each column. These register filter units are special because they could be located at the boundary between the lowest and highest indexed columns in the Levo instruction window. (Recall that columns are renamed as instructions are retired in the window in a circular fashion, thus, each column spends some time as “column 0”). These register filter units have the double duty of maintaining the oldest register values in the machine (those with time tag 0), while at the same time, collecting the “most speculative” register values (those with the highest time tags) at their inputs. Thus, these register filter units must maintain two copies of all the MIPS architectural registers, one of which is only used when the register filter unit sits at the head of column 0.

When one of these “split” RFUs is at index 0, it receives and stores values from its input RFBs and stores these values in one set of registers. Backwarding requests are satisfied from a second set of registers. Once column 0 is retired, the values from the first set of registers are copied into the second, and only the second set of registers is used (as per a normal RFU), until the column again becomes column 0. This prevents values with high time tags from “wrapping around” the instruction window and reappearing at the lowest-indexed active station.

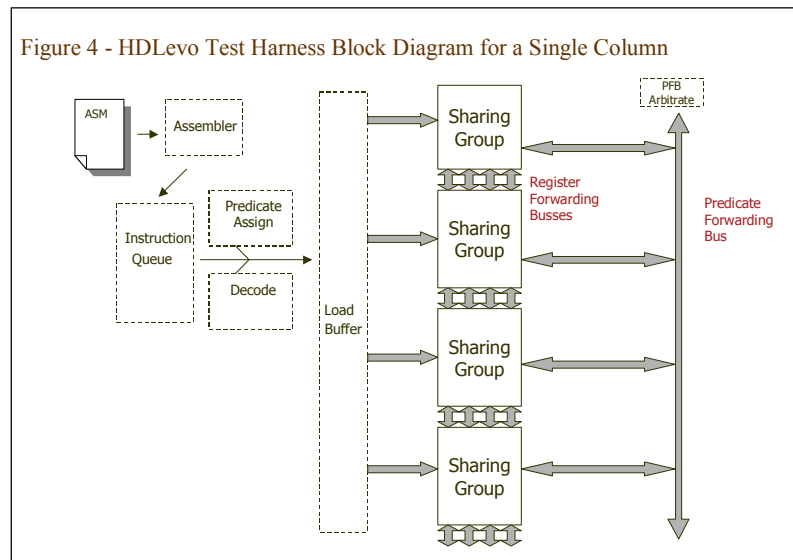
4 HDLevo Test Harness

Since not all of Levo has been implemented in the synthesizable VHDL model, a test harness is needed to simulate the function of parts of Levo that are still missing in the HDLevo model. The test harness must play the part of the instruction fetch and load system, the memory system, the instruction window control logic, the processing elements, and predicate and memory filter units, in order to provide a simulation environment in which the register filter unit and active station models can be tested.

The test harness creates a Levo testbed which wires the active stations and register filter

units together into an execution window, and provides behavioral (non-synthesizable) VHDL implementations of the surrounding hardware. In addition, to facilitate testing, the test harness has a built-in MIPS assembler which assembles input programs and loads them through simulated instruction fetch hardware into the modeled active stations. This built-in assembler greatly eases testing with HDLevo as MIPS programs can be fed directly into the simulation environment.

The following sub-sections describe how various parts of the test harness are implemented.



4.1 Processing Elements

Processing elements are simulated using behavioral VHDL. The processing element is implemented as a VHDL process, which uses a large tree of if statements to match MIPS op codes to instruction types, decode the MIPS operation, and perform the operation using standard VHDL library components. Currently, only a representative subset of all MIPS instructions is supported. As HDLevo becomes capable of more sophisticated simulations, the full MIPS instruction set will be supported. Only BEQ, ADD, and ADDI, and NOP instructions are currently supported. The BEQ instruction is sufficient for HDLevo to verify predication functionality (all other direct branches are functionally equivalent to BEQ except for the branch condition calculation, which does not affect the logic in the active station). ADD instructions are used to represent all 2 operand ALU operations, which again are functionally equivalent from the point of view of the active station. The ADDI instruction represents 1 operand ALU instructions and is also used to calculate the target address of memory instructions. Thus, these three instructions demonstrate the capability of HDLevo to carry out a very large subset of all MIPS R3000 instructions. Floating point operations are currently omitted from the HDLevo processing element, however, the HDLevo active station model is capable of storing and forwarding 64 bit floating point operands and results.

4.2 Levo Testbed

The Levo testbed is the top-level component of the HDLevo test harness. The testbed component instantiates two columns of active stations and register filter units, and wires them together. The testbed is also responsible for invoking the assembler and loading the columns with instructions. As each column reports that it has completed execution of all its instructions, the testbed renames the two columns and loads the now-empty column with a new set of instructions. The testbed also generates global clock, reset, and other configuration signals.

The testbed component also must perform the conversion from branches to predication, assigning predicates and canceling predicates as required. Assigning predicates to branches is simple, since the predicate address is equal to the index of the active station into which the instruction is loaded. However, the targets of these branches must be loaded with the correct canceling predicate addresses. Several VHDL variables maintain arrays of branch targets and their corresponding predicates and these arrays are searched as instructions are loaded into active stations for branches targeting the instruction.

The testbed must also process branch instructions which branch backwards or out of the instruction window, clearing the window and reloading it from the new starting address. This functionality is omitted in the current version of the testbed and is planned for addition in the next version. Thus, at this time, only forward branches with targets in the instruction window can be tested. However, this is sufficient to verify that predication is working properly.

It is planned to expand the testbed to simulate eight columns of active stations. However, to increase the speed of simulation and the readability of results, only two columns are currently simulated.

4.3 MIPS Assembler

A lexical analyzer and finite state automata parser have been implemented using the VHDL textio facilities to create an assembler which can assemble directly from MIPS .asm files into the simulated memory of the HDLevo machine. The assembler analyzes its input line by line, and breaks each line into a stream of tokens using the textio library. These tokens are then

fed to the state machine, which parses MIPS assembly instructions and writes MIPS machine language into the simulated memory. The assembler issues diagnostic messages any time it cannot determine how to traverse the parser’s state transition tree due to faulty input files. The state machine itself is implemented as a large tree of “case” statements based on the present state and the next input token. The assembler currently supports the same subset of instructions that the simulated processing element supports, namely BEQ, ADD, ADDI, and NOP. Even this simple set of instructions allows for the construction of fairly complex test cases, which ensure that the active station and register filter unit models are functioning properly.

5 Synthesis Results

Hardware sizes for the active station and register filter unit models were derived by running the VHDL input for these models through VHDL synthesis tools. The models were optimized and translated into net lists using LeonardoSpectrum 1999j. The net lists were then mapped for specific FPGAs using the Xilinx Alliance 3.1i tool chain. Synthesis targeted a Xilinx VirtexE V1000efg1156 FPGA. The Levo project is planning to implement a prototype of the Levo processor using the VirtexE 1000 FPGAs, hence, this FPGA was chosen as the synthesis target to generate size estimates.

The Xilinx Alliance tool chain provides a metric at the end of the mapping process which measures the equivalent gate count for a design. One “equivalent gate” is a 2 input NAND gate, or roughly 4 transistors [3]. The following table presents the equivalent gate and estimated transistor counts for the active station and both types of register filter units. Also included are estimates for a memory filter unit, and predicate filter unit. These estimates are based on scaling the numbers for the register filter unit by the relative complexity we estimate for the other filter units, and are not based on VHDL models.

Component Model	Equivalent Gates	Estimated Transistors
Active Station	21621	86484
Register Filter Unit (standard)	60040	240160
Register Filter Unit (column-head)	75207	300828
Predicate Filter Unit (<i>estimated</i>)	5000	20000
Memory Filter Unit (<i>estimated</i>)	37604	150414

6 Conclusions

The HDLevo project demonstrates the feasibility of realizing a full Levo prototype using FPGA’s, and the feasibility of implementing the Levo design in a custom ASIC using today’s state-of-the-art fabrication processes. The key component of Levo, the active station, uses roughly 86000 transistors. Eight active stations (one sharing group) fit easily on a single VertexE 1000 FPGA, each using roughly 8.5% of available logic cells, leaving sufficient space for routing and interconnection logic. The register filter unit is currently anticipated to be the largest of the Levo components, and is 240000 (300000 for column-head) transistors. A register filter unit uses roughly 20% of the available logic on a VertexE 1000. It is the hope of the Levo

project to implement a prototype Levo processor using interconnected VertexE FPGA's, a goal which appears realistic given the gate counts established here.

For a full Levo processor, 512 active stations are required (256 each for the execution window, and DEE paths), for a total of 44 million transistors. 32 register filter units are required (1 per sharing group), 8 of which are the larger column head variety. These require a total of just over 8 million transistors. Thus, including the estimates for predicate and memory filter units, the bulk of the Levo processor fits in less than 59 million transistors. The Intel Itanium Processor uses a total of 25 million transistors in the processor and 300 million in the cache [4]. Based on this comparison, it seems clear the Levo could be realized on a single chip using current or very near future fabrication technology.

7 References

- [1] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11 pp. 25-33, January 1967.
- [2] 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)*, pp. 313-325, IEEE and ACM, November/December 1995.
- [3] "Definition of a 'gate', when defining number of logic gates in a FPGA" Xilinx Answer Database Record # 2155. Xilinx Corporation.
http://support.xilinx.com/xlnx/xil_ans_display.jsp?getPagePath=2155
- [4] Intel Itanium Product Brief, Intel Corporation.
<http://www.intel.com/ebusiness/pdf/prod/ia64/ds010401.pdf>

8 Appendices

8.1 Active Station Component Design Reports

The following sections of this appendix are component design reports for various sub-components of the active station. They are a detailed reference useful as a guide for examining and understanding the VHDL source of the active station.

8.1.1 Register Forwarding Bus Interface

Purpose:

The *Register Forwarding Bus Interface* is the component of Levo which reads from and writes to the register forwarding bus system. The *RFBI* monitors all four of the RFBs to which an active station is connected for bus transactions whose address matches any of the active station's operand or relay addresses. The *RFBI* compares the time tag of these transactions to those of the previously stored values, and sends a signal to the *Active Station Control Unit* when a “snarf” occurs.

When writing, the *RFBI* signals the bus arbitration unit that it would like to write to its output register forwarding bus. When the *RFBI* is granted the bus, it writes the output or relay value to be forwarded to the bus and signals the *Active Station Control Unit* to indicate that the forwarding operation is complete.

Entity: eRegisterForwardingBusInterface
entities/register_forwarding_bus_interface_entity.vhd

Architectures: implementation
architectures/register_forwarding_bus_interface_architecture.vhd
debug_implementation
debug_architectures/register_forwarding_bus_interface_architecture.vhd

Component Interactions:

Strobe and Reset signals for all registers come from the *AS Control Unit*.

Current values, addresses, and time tags for operands, relay, and output values come from the *Active Station Register File*.

New operand and relay values and time tags go to the *Active Station Register File*.

Control signals indicating that a snarf of an operand/relay has occurred go to the *AS Control Unit*.

Control signals indicating that a the output or relay value should be forwarded come from the *AS Control Unit*.

Ports:

Asc2Rfb_En_OperandSnarf (in) Enables or disables snarfing of each operand.
Asc2Rfb_En_RelaySnarf (in) Enables or disables snarfing of the relay.

Asc2Rfb_En_FloatSnarf	(in) Set when floating point values should be snarfed. When floating point snarf is enabled, the LSB of operand and relay addresses is masked for purposes of matching addresses, and is instead used to distinguish high doubleword from low doubleword of a floating point value.
Asc2Rfb_Req_Forwarding	(in) Control signal which indicates that the <i>RFBI</i> should forward a value on the register forwarding bus.
Asc2Rfb_Sel_ForwardingSignal	(in) Indicates whether output or relay should be forwarded.
Asc2Rfb_Sel_ForwardingFloat	(in) When set, indicates that the low byte of a floating point output/relay value should be written to the bus.
Out2Rfb_BusGrant_RFB	(in) Indicates that write access has been granted for the register forwarding bus.
Out2All_ColumnShift	(in) Indicates that a column is being retired this cycle and that all time tags must be shifted.
Out2All_SystemClock	(in) System clock
Out2All_SystemReset	(in) System reset
Rf2All_OperandData	(in) Current operand values
Rf2All_FloatOperandData	(in) Current floating point low doubleword operand values
Rf2All_OperandAddress	(in) Current operand addresses
Rf2All_OperandTimeTag	(in) Current operand time tags
Rf2All_OutputData	(in) Current output value
Rf2All_FloatOutputData	(in) Current floating point low doubleword output value
Rf2All_OutputAddress	(in) Current output/relay address
Rf2All_OutputTimeTag	(in) Current output time tag
Rf2All_RelayData	(in) Current relay value
Rf2All_FloatRelayData	(in) Current floating point low doubleword relay value
Rf2All_RelayTimeTag	(in) Current relay time tag
Rfb2Asc_Ack_Forwarding	(out) Acknowledges to the <i>ASC</i> that a forwarding operation has been completed.
Rfb2Out_BusReq_RFB	(out) Requests write access to the register forwarding bus.
Rfb2Asc_OperandSnarf	(out) Signal to <i>ASC</i> indicating that an operand value has been snarfed.
Rfb2Asc_RelaySnarf	(out) Signal to <i>ASC</i> indicating that the relay value has been snarfed.
Rfb2Asc_FloatOperandSnarf	(out) Signal to <i>ASC</i> indicating that low byte of a floating point operand value has been snarfed.
Rfb2Asc_FloatRelaySnarf	(out) Signal to <i>ASC</i> indicating that low byte of the floating point relay value has been snarfed.
Rfb2Asc_OperandFreshen	(out) Indicates that an operand value has been <i>freshened</i> . Freshening means that a newer time tag, but an unchanged value has been snarfed.
Rfb2Asc_RelayFreshen	(out) Indicates that an operand value has been <i>freshened</i> .
Rfb2Rf_NewOperandData	(out) New operand values.
Rfb2Rf_NewOperandTimeTag	(out) New operand time tags.
Rfb2Rf_NewRelayData	(out) New relay value.
Rfb2Rf_NewRelayTimeTag	(out) New relay time tag.

RegisterForwardingBus (inout) Read/Write RFB.
PassthruRegisterForwardingBusses (inout) Read-only RFBs

Internal Logic:

Diagram not available

VHDL Implementation Issues:

Note that it is assumed in HDLevo that there is not enough time in a single cycle for forwarded signals to propagate across a bus, and then be processed at the end of the bus. Thus, values off of all four register forwarding busses are registered in the *RFBI* each cycle, and then compared to values stored in the *ASRF* in the following cycle. The only processing done in the cycle in which bus transactions are received is the shifting of time tags if a column shift is signaled during the transaction cycle.

Once registered, the addresses and time tags from each of the busses are compared against the operand and relay addresses and time tags. These comparators are instantiated in the `GenAddressMatch` generate block in the *RFBI* architecture. The first nine lines within the generate block instantiate comparators for address comparison. The address comparators are broken up over several signals in order to efficiently match addresses for floating point operations. In floating point operations, 64 bit floating point values are stored in two consecutive architectural registers, where the register with address ending in a '0' is the most significant doubleword, and the address ending with '1' the least significant.

The next several lines evaluate the time tags of each bus signal. First, `sTimeTagInPast` ensures that the bus transaction comes from an active station above this active station in the execution window. This prevents an active station from snarfing updates from other active stations in their sharing group which have higher time tags. The next several comparisons evaluate if the stored value for each operand/relay is more recent than that on the bus. The final set of comparators compare the data value from the bus transaction to that of the stored value. If the data values match, a freshen, instead of a snarf operation is performed. In a freshen operation, time tag values are updated, but instructions are not re-executed nor are new results forwarded on any bus.

Another problem that must be solved in the *RFBI* is that the same address may appear on multiple register forwarding busses in the same cycle. A set of comparators compare the time tags across each combination of busses. When the same address appears on multiple busses, the output of these comparators is used to determine which value to snarf.

Finally, freshen and snarf output signals are generated based on the results of all the comparators. These signals also determine which bus value should be copied to the `Rfb2Rf_NewOperandData` and other output signals to the *ASRF*.

8.1.2 Register Backwarding Bus Interface

Purpose:

The *Register Backwarding Bus Interface* is used by Levo to request operand and relay values from the *Register Filter Unit* nearest the active station. These requests are necessary to ensure that the *RFU* forwards the required value on the register forwarding bus. The *RFU* will ignore requests for values which have already been forwarded, or which have been requested by other active stations.

Entity: eRegisterBackwardingBusInterface
entities/register_backwarding_bus_interface_entity.vhd

Architectures: implementation
architectures/register_backwarding_bus_interface_architecture.vhd
debug_imlementation
debug_architectures/register_backwarding_bus_interface_architecture.vhd

Component Interactions:

Current addresses for operands, relay, and output values come from the *Active Station Register File*.

Control signals indicating that an operand or relay should be requested come from the *AS Control Unit*.

Control signals indicating that a request has been made go to the *AS Control Unit*.

Ports:

Asc2Rbb_Req_Operand	(in) Signal from <i>ASC</i> indicating that a particular operand should be requested.
Asc2Rbb_Req_Relay	(in) Signal from <i>ASC</i> indicating that the relay should be requested.
Asc2Rbb_Req_FloatOperand	(in) Signal from <i>ASC</i> indicating that a the least significant doubleword of a floating point operand should be requested.
Asc2Rbb_Req_FloatRelay	(in) Signal from <i>ASC</i> indicating that a the least significant doubleword of the floating point relay should be requested.
Rbb2Asc_Ack_OperandReq	(out) Signal to the <i>ASC</i> acknowledging a request for an operand.
Rbb2Asc_Ack_RelayReq	(out) Signal to the <i>ASC</i> acknowledging a request for the relay.
Rbb2Asc_Ack_FloatOperandReq	(out) Signal to the <i>ASC</i> acknowledging a request for the least significant doubleword of a floating point operand.
Rbb2Asc_Ack_FloatRelayReq	(out) Signal to the <i>ASC</i> acknowledging a request for the least significant doubleword of the floating point relay.
Rbb2Out_BusReq_RBB	(out) Requests write access to the register backwarding bus.
Out2Rbb_BusGrant_RBB	(in) Indicates that write access has been granted for the register backwarding bus.
Out2All_SystemClock	(in) System clock

Out2All_SystemReset	(in) System reset
Rf2All_OperandAddress	(in) Current operand addresses
Rf2All_OutputAddress	(in) Current output/relay address
RegisterBackwardingBus	(out) Register backwarding bus.

Internal Logic:

Diagram not available

VHDL Implementation Issues:

No particular implementation issues.

8.1.3 Memory Forwarding Bus Interface

Purpose:

The *Memory Forwarding Bus Interface* is used by Levo to forward the outputs of store instructions and monitor for the inputs of load instructions. There are four memory forwarding busses in Levo, each of which are interleaved to carry different memory addresses. One memory bus carries transactions for addresses whose least significant bits are “00”, the next for “01” and so on. The *MFBI* performs update and nullify transactions for store instructions. An update transaction forwards a memory address, new value, and time tag on the bus, and indicates a write to that memory address. A nullify transaction forwards an address and time tag, and indicates that a previous write to that address should be rolled back. For load instructions, the *MFBI* functions in a similar fashion to the *RFBI*, snarfing updates to its memory address. However, since loads always load from a single address, only one memory forwarding bus need be monitored at a time, saving a considerable amount of hardware relative to the *RFBI*.

Entity: eMemoryForwardingBusInterface
entities/memory_forwarding_bus_interface_entity.vhd

Architectures: implementation
architectures/memory_forwarding_bus_interface_architecture.vhd
debug_implementation
debug_architectures/memory_forwarding_bus_interface_architecture.vhd

Component Interactions:

Current memory addresses value, and time tags come from the *Active Station Register File*. Control signals indicating that the *MFBI* should snarf or perform an update or nullify transaction come from the *AS Control Unit*. Control signals indicating that a snarf or transaction has occurred go to the *AS Control Unit*.

Ports:

Asc2Mfb_En_MemorySnarf	(in) Signal from <i>ASC</i> indicating the <i>MFBI</i> should monitor the appropriate forwarding bus to snarf memory updates.
Asc2Mfb_Req_Store	(in) Signal from <i>ASC</i> indicating a transaction should be performed.
Asc2Mfb_StoreNotNullify	(in) Signal from <i>ASC</i> indicating whether an update or nullify transaction should be performed.
Mfb2Asc_Ack_Request	(out) Signal to <i>ASC</i> indicating the requested transaction has been completed.
Mfb2Out_BusReq_MFB	(out) Requests write access to the memory forwarding bus.
Out2Mfb_BusGrant_MFB	(in) Indicates that write access has been granted for the memory forwarding bus.
Mfb2Asc_LoadSnarf	(out) Signal to the <i>ASC</i> indicating that an update transaction has been snarfed.
Mfb2Asc_NullifySnarf	(out) Signal to the <i>ASC</i> indicating that a nullify transaction has been snarfed.
Out2All_ColumnShift	(in) Indicates that all time tags must be shifted due to a column being retired.
Out2All_SystemClock	(in) System clock
Out2All_SystemReset	(in) System reset
Rf2All_OutputTimeTag	(in) Time tag used when writing nullify and update transactions
Rf2All_MemoryData	(in) Value currently stored for memory data
Rf2All_MemoryAddress	(in) Value currently stored for memory address
Rf2All_MemoryTimeTag	(in) Value stored for the last snarfed memory time tag.

Internal Logic:

Diagram not available

VHDL Implementation Issues:

As with the *RFBI*, the assumption is made in the *MFBI* that bus transactions cannot be received and compared in the same cycle. Thus, all bus transactions are registered in the cycle they occur, and then evaluated in the following cycle.

Note that the *MFBI* always deals with only one memory address at a time, thus, only one set of comparators is needed. The least significant bits of the memory address select which of the memory forwarding busses should be read from / written to.

8.1.4 Memory Backwarding Bus Interface

Purpose:

The *Memory Backwarding Bus Interface* is used by load instructions to request that values be retrieved from the memory hierarchy. Memory backwarding requests are sent to the nearest

memory filter unit. The memory filter unit may then satisfy the request from a local buffer, from the write queue, or from memory.

Entity: eMemoryBackwardingBusInterface
entities/memory_backwarding_bus_interface_entity.vhd

Architectures: implementation
architectures/memory_backwarding_bus_interface_architecture.vhd
debug_implementation
debug_architectures/memory_backwarding_bus_interface_architecture.vhd

Component Interactions:

The memory address to be requested comes from the *Active Station Register File*.
Control signals indicating that a request should be made comes from the *AS Control Unit*.
Control signals indicating that a request has been made go to the *AS Control Unit*.

Ports:

Asc2Mbb_Request_Load	(in) Signal from <i>ASC</i> indicating that the <i>MBBI</i> should issue a memory request.
Out2Mbb_BusGrant_MBB	(in) Indicates that write access has been granted for the memory backwarding bus.
Out2All_SystemClock	(in) System clock
Out2All_SystemReset	(in) System reset
Rf2All_MemoryAddress	(in) Memory address to be requested
Mbb2Asc_Ack_LoadReq	(out) Signal to <i>ASC</i> acknowledging that a memory request has been made.
Mbb2Out_BusReq_MBB	(out) Requests write access to the memory backwarding bus.
MemoryBackwardingBusses	(inout) Bus for requesting memory addresses.

Internal Logic:

Diagram not available

VHDL Implementation Issues:

As with the memory forwarding bus, the memory backwarding busses are interleaved, each carrying addresses with different values in the least two significant bits.

8.1.5 Predicate Forwarding Bus Interface

Purpose:

The *Predicate Forwarding Bus Interface* is used to send output predicates and canceling predicates from branch and branch target instructions. The *PFBI* also snarfs input predicate and input canceling predicate forwarding transactions for all instructions.

Entity: ePredicateForwardingBusInterface
 entities/predicate_forwarding_bus_interface_entity.vhd

Architectures: implementation
 architectures/predicate_forwarding_bus_interface_architecture.vhd
 debug_inlmentation
 debug_architectures/predicate_forwarding_bus_interface_architecture.vhd

Component Interactions:

The addresses and current values for input predicates and canceling predicate come from the *Active Station Register File*.

Newly snafed input predicate values go to the *Active Station Register File*.

Output predicate and canceling predicate values come from the *AS Control Unit*.

Control signals indicating that predicate should be forwarded and that snarfing should be enabled come from the *AS Control Unit*.

Control signals indicating that predicates have been snarfed go to the *AS Control Unit*.

Ports:

Asc2Pfb_En_pSnarf	(in) Signal from <i>ASC</i> enabling snarfing of the input predicate.
Out2All_SystemClock	(in) System clock
Out2All_SystemReset	(in) System reset
Asc2Pfb_En_cpSnarf	(in) Signal from the <i>ASC</i> enabling snarfing for each of the canceling predicates
Asc2Pfb_Req_Forwarding	(in) Signal from the <i>ASC</i> requesting that the output predicate and canceling predicate be forwarded.
Pfb2Asc_Ack_Forwarding	(out) Signal to the <i>ASC</i> acknowledging a forwarding operation.
Pfb2Out_BusReq_PFB	(out) Requests write access to the predicate forwarding bus.
Out2Pfb_BusGrant_PFB	(in) Grant of write access to the predicate forwarding bus.
Out2All_ColumnShift	(in) Signal indicating all predicate addresses should be shifted due to a column retiring.
Pfb2Asc_pSnarf	(out) Signal to the <i>ASC</i> indicating the input predicate has been snarfed.
Pfb2Asc_cpSnarf	(out) Signal to the <i>ASC</i> indicating the input canceling predicate has been snarfed.
Rf2All_PredValue	(in) Current input predicate value.
Rf2All_PredAddress	(in) Current input predicate address.
Rf2All_cPredValue	(in) Current input canceling predicate value.
Rf2All_cPredAddress	(in) Current input canceling predicate address.
Pfb2Rf_NewpValue	(out) Snarfed input predicate value.
Pfb2Rf_NewcpValue	(out) Snarfed input canceling predicate value.
Rf2All_ASID	(in) Active station index (used to calculate output predicate and canceling predicate address).
Asc2Pfb_poutValue	(out) Output predicate value.
Asc2Pfb_cpoutValue	(out) Output canceling predicate value.

PredicateForwardingBus

(inout) The predicate forwarding bus.

Internal Logic:

Diagram not available

VHDL Implementation Issues:

No special VHDL considerations.

8.1.6 Processing Element Return Bus Interface

Purpose:

The *Processing Element Return Bus Interface* monitors the Processing Element Return Bus for execution results sent back from the *Processing Element* to this *Active Station*. It compares the address on the PERB to the address of this Active Station, and signals the Active Station Control Unit if there is a match.

Entity: ePEReturnBusInterface

entities/pe_return_bus_interface_entity.vhd

Architectures: implementation

architectures/pe_return_bus_interface_architecture.vhd

debug_implementation

debug_architectures/pe_return_bus_interface_architecture.vhd

Component Interactions:

Execution results are sent from the *Processing Element*.

Execution results are sent to the *AS Register File*. The AS Address comes from the *ASRF*

A signal is sent to the *AS Control Unit* when an execution result is snarfed.

Ports:

Rf2All_ASID

(in) Address of the *Active Station* containing this *PEBI*

PERBus

(tri-out) The *Processing Element Return Bus*

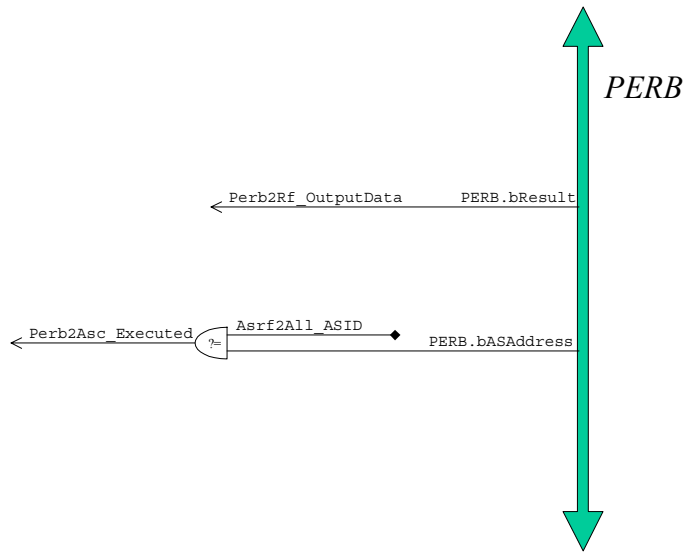
Perb2Rf_OutputData

(out) Execution Result sent to *ASRF*

Perb2Asc_Executed

(out) Signal to *ASC* to indicate an execution result has been received

Internal Logic:



VHDL Implementation Issues:

No special VHDL considerations.

8.1.7 Processing Element Bus Interface

Purpose:

The *Processing Element Bus Interface* is signaled by the *AS Control Unit* when an instruction should be dispatched to the *Processing Element* for execution. It competes for the Processing Element Bus and then dispatches the instruction and operands to the *PE*.

Entity: ePEBusInterface entities/pe_bus_interface_entity.vhd
Architectures: implementation architectures/pe_bus_interface_architecture.vhd
 debug_imlementation debug_architectures/pe
 _bus_interface_architecture.vhd

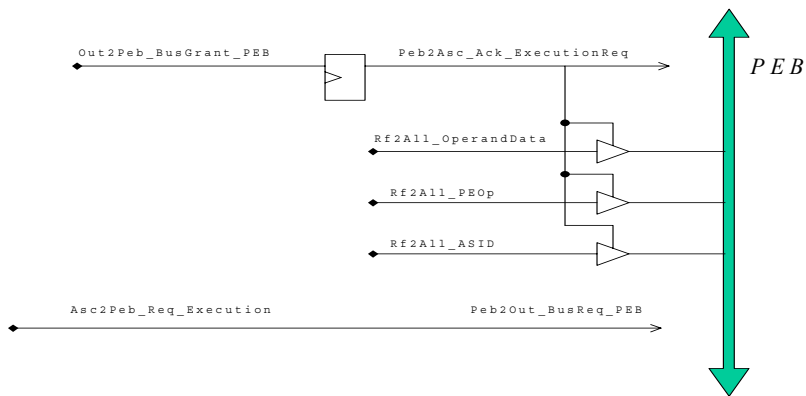
Component Interactions:

Instructions and operands are dispatched to the *Processing Element*.
 Operands and opcodes for instructions come from the *AS Register File*.
 Control signals come from the *AS Control Unit*
 Clock and Reset signals are global.

Ports:

Out2All_SystemClock	(in) Clock
Out2All_SystemReset	(in) Reset (clears the <code>PEBGranted</code> flip flop)
Rf2All_OperandData	(in) Operand values from the <i>AS Register File</i>
Rf2All_PEOp	(in) <i>Processing Element</i> op code
Rf2All_ASID	(in) Address of the <i>Active Station</i> containing this <i>PEBI</i>
PEBus	(tri-out) The <i>Processing Element</i> Bus
Asc2Peb_Req_Execution	(in) Signal from <i>ASC</i> to dispatch the instruction
Peb2Asc_Ack_ExecutionReq	(out) Acknowledgement to <i>ASC</i> to indicate excution has been requested
Peb2Out_BusReq_PEB	(out) Bus request signal to <i>PEB</i> arbitration
Out2Peb_BusGrant_PEB	(in) Bus grant signal from <i>PEB</i> arbitration

Internal Logic:



VHDL Implementation Issues:

`sReg_PEBGranted` and `sPEBGranted` together synthesize a flip flop used to store Bus Grant signals from the cycle in which the bus is granted, to the cycle in which the bus signals are enabled. All connections to the `PEB` are tri-stated, using `sReg_PEBGranted` as the enable.

8.1.8 Active Station Register File

Purpose:

The *Active Station Register File* contains all state information stored in the active station, except control state stored in the *Active Station Control Unit*. The *ASRF* contains registers for operand, relay, and output data addresses, and time tags, op codes, active station address, predicates and canceling predicates and their addresses.

Entity: eASRegisterFile entities/as_register_file_entity.vhd
Architectures: implementation architectures/as_register_file_architecture.vhd

Component Interactions:

Strobe and Reset signals for all registers come from the *AS Control Unit*.

Opcodes, operand addresses, output address, predicates and their addresses come from the *Load Bus Interface*.

New operand and relay values and time tags come from the *Register Forwarding Bus Interface*.

New output values come from the *Processing Element Return Bus Interface*.

New input predicate values come from the *Predicate Forwarding Bus Interface*.

New output predicate values come from the *AS Control Unit*.

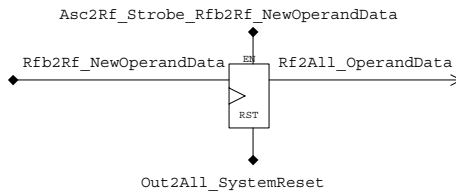
Values contained in the *ASRF* are made available to all other components.

Ports:

Asc2Rf_Strobe_Rfb2Rf_OperandData	(in) Strobe signal for operand data coming from <i>RFBI</i> .
Asc2Rf_Strobe_Lb2Rf_OperandAddress	(in) Strobe signal for operand address coming from <i>RFBI</i> .
Asc2Rf_Strobe_Rfb2Rf_OperandTimeTag	(in) Strobe signal for operand time tag coming from <i>RFBI</i> .
Asc2Rf_Strobe_Perb2Rf_OutputData	(in) Strobe signal for output data coming from <i>PERB</i> .
Asc2Rf_Strobe_Lb2Rf_OutputAddress	(in) Strobe signal for output address coming from <i>LBI</i> .
Asc2Rf_Strobe_OutputTimeTag	(in) Strobe signal to generate output time tag.
Asc2Rf_Strobe_Rfb2Rf_RelayData	(in) Strobe signal for relay data coming from <i>RFBI</i> .
Asc2Rf_Strobe_RelayTimeTag	(in) Strobe signal for relay time tag coming from <i>RFBI</i> .
Asc2Rf_Strobe_Lb2Rf_PEOpCode	(in) Strobe signal for PE op code coming from <i>LBI</i> .
Asc2Rf_Strobe_Lb2Rf_ASOp	(in) Strobe signal for AS op code coming from <i>LBI</i> .
Asc2Rf_Strobe_Lb2Rf_PredAddress	(in) Strobe signal for predicate addresses coming from <i>LBI</i> .
Asc2Rf_Strobe_Lb2Rf_PredValue	(in) Strobe signal for predicate values coming from <i>LBI</i> .
Asc2Rf_Strobe_Lb2Rf_cPredAddress	(in) Strobe signal for canceling predicate addresses coming from <i>LBI</i> .
Asc2Rf_Strobe_Lb2Rf_cPredValue	(in) Strobe signal for canceling predicate values coming from <i>LBI</i> .
Asc2Rf_Strobe_Lb2Rf_cPredValid	(in) Strobe signal for canceling predicate valid bits coming from <i>LBI</i> .
Asc2Rf_Strobe_Pfb2Rf_NewpValue	(in) Strobe signal for predicate values coming from <i>PFBI</i> .
Asc2Rf_Strobe_Pfb2Rf_NewcpValue	(in) Strobe signal for canceling predicate values coming from <i>PFBI</i> .
Asc2Rf_Rst_OperandTimeTag	(in) Reset signal to clear operand time tags when a new instruction is loaded.
Asc2Rf_Rst_RelayTimeTag	(in) Reset signal to clear relay time tags when a new instruction is loaded.
Rfb2Rf_NewOperandData	(in) New operand value from <i>RFBI</i> .
Rfb2Rf_NewOperandTimeTag	(in) New operand time tag from <i>RFBI</i> .
Rfb2Rf_NewRelayData	(in) New relay value from <i>RFBI</i> .
Rfb2Rf_NewRelayTimeTag	(in) New relay time tag from <i>RFBI</i> .
Lb2Rf_OperandAddress	(in) New operand address from <i>LBI</i> .

Perb2Rf_OutputData	(in) New output value from <i>PERBI</i>
Lb2Rf_OutputAddress	(in) New output address from <i>LBI</i>
Lb2Rf_PEOpCode	(in) New PE op code from <i>LBI</i>
Lb2Rf_ASOp	(in) New AS op code (pre-decoded from PE op) from <i>LBI</i>
Out2Rf_ASID	(in) Address of this <i>AS</i> within Levo.
Lb2Rf_PredAddress	(in) New predicate address from <i>LBI</i>
Lb2Rf_PredValue	(in) Initial input predicate value from <i>LBI</i>
Lb2Rf_cPredAddress	(in) New canceling predicate address from <i>LBI</i>
Lb2Rf_cPredValue	(in) Initial canceling predicate value from <i>LBI</i>
Lb2Rf_cPredValid	(in) Valid bit for each canceling predicate from <i>LBI</i>
Pfb2Rf_NewpValue	(in) New input predicate value from <i>PFBI</i>
Pfb2Rf_NewcpValue	(in) New canceling predicate value from <i>PFBI</i>
Out2All_SystemClock	(in) System clock
Out2All_SystemReset	(in) System reset
Rf2All_OperandData	(out) Current operand values
Rf2All_OperandAddress	(out) Current operand addresses
Rf2All_OperandTimeTag	(out) Current operand time tags
Rf2All_OutputData	(out) Current output value
Rf2All_OutputAddress	(out) Current output/relay address
Rf2All_OutputTimeTag	(out) Current output time tag
Rf2All_RelayData	(out) Current relay value
Rf2All_RelayTimeTag	(out) Current relay time tag
Rf2All_PEOp	(out) Current PE op code
Rf2Asc_ASOp	(out) Current AS op code
Rf2Asc_BranchTaken	(out) indicates if the last branched executed was taken (1) or not taken (0). This is actually another name for bit 0 of <i>Rf2All_OutputData</i>
Rf2Asc_p	(out) Latched version of <i>Rf2All_PredValue</i> . This output follows input values that will change <i>Rf2All_PredValue</i> after the next clock edge
Rf2Asc_cp	(out) Latched version of <i>Rf2All_cPredValue</i> . This output follows input values that will change <i>Rf2All_PredValue</i> after the next clock edge
Rf2All_ASID	(out) Current address of this <i>AS</i> in Levo
Rf2All_PredAddress	(out) Current input predicate address
Rf2All_PredValue	(out) Current input predicate value
Rf2All_cPredAddress	(out) Current input canceling predicate addresses
Rf2All_cPredValue	(out) Current input canceling predicate values
Rf2All_cPredValid	(out) Current input canceling predicate valid flags

Internal Logic:



Note: diagram illustrates a sample register. All other registers are realized in a similar fashion. See implementation notes.

VHDL Implementation Issues:

All of the output signals from the *ASRF* should be generated as the Q signals from registers, except the *Rf2Asc_** signals, which must be implemented as latches. The output signals of *Rf2Asc_** signals must follow the inputs when any of the strobe signals are asserted, to assure proper behavior of the *ASC*.

When a register is attached to multiple input signals, the inputs are connected by a mux, whose output is selected based on the strobes, and the strobe signals are OR'd together. The muxes should give priority to signals coming from the *LBI*. After the *LBI* signals, priority is arbitrary, and it is the responsibility of the *ASC* to assure that only one strobe is active at a time.

Reset signals are synchronous, and take priority of strobe signals. Reset is to all '0's except for predicate value, which resets to '1'.

The *ASRF* generates the output time tag of the AS by taking the top 5 bits of the ASID, and then appending to it a special one-hot encoding of the bottom 3 bits. In this encoding "000" maps to "00000000"; "001" maps to "00000001"; "010" maps to "00000011"; and so on, up to "111" mapping to "11111111".

8.2 Output from the Xilinx Alliance Synthesis Tools

8.2.1 Active Station

Excepted from the Xilinx Alliance map report for the Active Station

Number of Slices:	1,110 out of 12,288	9%
Number of Slices containing unrelated logic:	0 out of 1,110	0%
Total Number Slice Registers:	735 out of 24,576	2%
Number used as Flip Flops:	725	
Number used as Latches:	10	
Total Number 4 input LUTs:	1,950 out of 24,576	7%
Number used as LUTs:	1,941	
Number used as a route-thru:	9	
Number of bonded IOBs:	1,031 out of 660	156%
Number of GCLKs:	1 out of 4	25%
Number of GCLKIOBs:	1 out of 4	25%
Total equivalent gate count for design:	21,621	

8.2.2 Register Filter Unit (Column Head)

Excepted from the Xilinx Alliance map report for the Column-Head Register Filter Unit

Design Summary:

Number of errors:	0		
Number of warnings:	36		
Number of Slices:	4,615 out of 12,288	37%	
Number of Slices containing unrelated logic:	0 out of 4,615	0%	
Total Number Slice Registers:	3,189 out of 24,576	12%	
Number used as Flip Flops:	3,139		
Number used as Latches:	50		
Total Number 4 input LUTs:	7,880 out of 24,576	32%	
Number used as LUTs:	7,843		
Number used as a route-thru:	37		
Number of bonded IOBs:	301 out of 660	45%	
Number of GCLKs:	1 out of 4	25%	
Number of GCLKIOBs:	1 out of 4	25%	
Total equivalent gate count for design:	75,207		

8.2.2 Register Filter Unit (standard)

Excepted from the Xilinx Alliance map report for the standard Register Filter Unit

Design Summary:

Number of errors:	0		
Number of warnings:	4		
Number of Slices:	3,818 out of 12,288	31%	
Number of Slices containing unrelated logic:	0 out of 3,818	0%	
Total Number Slice Registers:	1,733 out of 24,576	7%	
Number used as Flip Flops:	1,682		

Number used as Latches:			51	
Total Number 4 input LUTs:	7,314	out of	24,576	29%
Number used as LUTs:			7,308	
Number used as a route-thru:			6	
Number of bonded IOBs:	296	out of	660	44%
Number of GCLKs:	1	out of	4	25%
Number of GCLKIOBs:	1	out of	4	25%
Total equivalent gate count for design: 60,040				