## Differentiating Between Scalar and Vector Instructions

Vector instructions are detected when a register spec field indicates a vector register is involved and the register spec is valid. Otherwise, the instruction is assumed to be a scalar instruction.

## Register Renaming

Scalar registers are renamed to reduce the number of hazards encountered causing the core to stall. Vector registers are not renamed due to the large number of registers and hardware requirements. Since vector elements are independent of each other stalls occur much less frequently during the execution of vector instructions than they would for scalar ones.

The renaming mechanism retains the architectural register number as part of the rename.

## Register Tags

Registers are tagged with a 12-bit tag number to allow register dependencies to be determined. Scalar registers effectively use a nine-bit tag number as the upper three bits of the tag are always 111.

Scalar Register

|  |  |  |
| --- | --- | --- |
| 11 9 | 8 6 | 5 0 |
| 111  row ‘7’ from  vector register file | Rename bucket number | Architectural Scalar Register Number |

The lower six bits of the scalar register tag contain the architectural or logical register number. When a scalar instruction commits only the lower six bits of the register tag control the physical register written to. The rename bucket number is forced to seven.

Vector Register Tag

|  |  |
| --- | --- |
| 11 6 | 5 0 |
| row 00 to 67 (octal)  row ‘7x’ equals scalar  register file | Architectural Vector Register Number |

## Rename Buckets

Each of the 64 architecturally visible registers has eight rename buckets. This allows registers to be renamed up to eight times. On the ninth rename requirement the machine will stall until a register bucket becomes available.

For each physical register there is a bitmap of the buckets in use. A bitmap rather than a simple counter is used to allow instructions to execute out-of-order. Buckets might be assigned 1,2,3 but instructions might be done in order 3,2,1.

|  |  |  |
| --- | --- | --- |
| 11 9 | 8 6 | 5 0 |
| 111  row ‘7’ from  vector register file | Rename bucket number | Architectural Scalar Register Number |

When the instruction is committed to the machine state from the reorder buffer, the rename bucket number associated with the target register is marked as available. A reset operation also marks all buckets as available.

The physical register available bitmap is a bitmap of registers available for rename use.

## Register Rename Map

The register rename map contains the mapping of renamed registers. There are 64, three-bit entries in a map. Each entry identifies the bucket used for the current value of the register. When register fetch occurs the value of the register identified in the rrmap is effectively used.

## Register Rename Map History

There is an eight-entry register rename map history that allows previous mappings of registers to be restored, should a branch miss occur. Whenever a branch is executed, a new map is allocated and the current map values copied to the new map. To rollback on a branch miss the map number associated with the branch is set as the active map. Because rename history is only eight entries deep speculation beyond eight branches is not allowed. The machine will stall when evaluating the eighth branch speculation until prior speculations are resolved.

## Register Value Source Tracking

The source of data values for a register is tracked by the core. It is determined at instruction queue time.

The source for a given register targeted by an instruction is the re-order buffer tail id. If there is a branch miss the source is set back to the latest id targeting the register before the branch. On reset the source is set to a special value indicating no source (all ones’).

Read Ports

Up to nine read ports are required on the register source tracking file. Five required during argument matching (there could be up to five register as arguments). And four more during the commit stage to valid the registers.

Write Ports

Because up to four instructions may queue in a given cycle, there are up to six different register targets that have to be set in a single clock cycle. One for each queued instruction, one for the branch miss and one for reset. This is six write ports on the source tracking file.

Since there is a source for every architectural register in the machine with 4096 registers this turns out to be quite a large file. Six write and nine read ports are present. Fortunately, the id’s tracked are small for example four bits.. It is desirable then to reduce the size of this register file. One way to do so is to reduce the number of write ports. The number of write ports active during any clock cycle may be reduced by time domain multiplexing the writes. There are three independent selections for write ports on a given clock cycle. These are: reset selection, branch miss, and normal operations. Since these operations are exclusive there are only a max of four write ports required at any one time.

Using a five times clock, the four write ports could be time-domain multiplexed so that only a single write occurs during the clock cycle. Additionally, a hand-shaking signal could be present to indicate when all relevant writes have taken place. Note that many instructions do not update the register file. These instructions would not need to update source tracking.

## Branch Miss Logic

The original code uses a bitmask of registers, one bit for each architectural register. In the vector machine this is not practical as there more than 4,000 registers which makes manipulating multiple wide bit masks hardware inefficient.

There is potentially only one live target register per queue entry. A target register spec requires only 13 bits. The bitmasks were essentially a convenient way to build a list of registers. A list of registers may also be built using an array of register specs. The array of register specs requires 13 bits times the number of queue entries supported. For 16 queue entries only 208 bits are required. This is much less than the 4,000+ bits of a bitmask. It does mean that the logic for accumulation of registers changes.

For the original code, iq\_livetarget[n] would be a 4,000 bit wide bitmask which has only a single bit set in it corresponding to the live target register of queue entry ‘n’. Since there is only a single bit set, this can be replaced by a 13-bit register spec which is either the live target register or zero to indicate no live target.

iq\_livetarget[n] is the target register id for queue entry ‘n’ when the queue entry is valid and not stomped on.

VEX Instruction

The VEX instruction needs to execute in two stages because the vector register element to read is not known until after the value of register Ra is known. So, VEX must first wait until Ra becomes valid. Once Ra is valid it is used to read the vector register file. The Vb cannot start out as valid for the VEX instruction, it must always be flagged as invalid during decode / queue. This means a value for the register file source is needed that will never match as results are placed in the reorder buffer. Once Ra is known the register file source for Vb can be reset to a valid source, allowing Vb to be updated when results for it are available.