Register Renaming and Dynamic Speculation: an Alternative Approach*

Mayan Moudgill
Keshav Pingali
Stamatis Vassiliadis

TR 93-1379
August 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

Register Renaming and Dynamic Speculation: an Alternative Approach

Mayan Moudgill
Keshav Pingali
Department of Computer Science,
Cornell University,
Ithaca NY 14853

Stamatis Vassiliadis
IBM Corporation,
Enterprise Systems,
Poughkeepsie, NY 12602

School of Electrical Engineering
Cornell University,
Ithaca NY 14853

Abstract

In this paper, we present a novel mechanism that implements register renaming, dynamic speculation and precise interrupts. Renaming of registers is performed during the instruction fetch stage instead of the decode stage, and the mechanism is designed to operate in parallel with the tag match logic used by most cache designs. It is estimated that the critical path of the mechanism requires approximately the same number of logic levels as the tag match logic, and therefore should not impact cycle time.

[0]This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

1 Introduction

Superscalar processors attempt to exploit instruction level parallelism by issuing multiple instructions in parallel, thereby improving performance. Obviously, this improvement is limited by the amount of parallelism present in the code. Several mechanisms have been developed that increase the number of instructions that a processor can issue in parallel, and thereby increase performance. This paper focuses on two such mechanisms: register renaming and dynamic speculation. A number of recent studies [5, 13, 10] have shown that they can make a significant impact on performance. For instance, Johnson states in [5] that removing renaming and speculation from the superscalar implementation under consideration would reduce the speedup obtained by 36% and 30% respectively.

Register renaming improves performance in code not specifically compiled for a superscalar processor because such code tends to use registers in ways that reduce the amount of parallelism available. This reduction can limit the superscalar issue rate and lower potential performance gains. For example, consider the following code fragment:

```
(1) r1  := r2 / r3
(2) r2  := r1 + r3
(3) r1  := r5 + r8
(4) r4  := r1 - r7
```

Fig. 1 (a)

```
(1) r1  := r2 / r3
(2) r2  := r1 + r3
(3) r11 := r5 + r8
(4) r4  := r11 - r7
```

Fig. 1 (b)

All four instructions in Fig. 1(a) must execute serially. (2) must follow (1) since it uses the result of (1); similarly (4) must follow (3). Such dependences between instructions are known as *flow dependences*. Since (1) and (3) use the same result register, (3) must follow (1). This is an example of an *output dependence*. Also, (3) must follow (2), since (2) uses the value of `r1`. If (3) completes before (2), it would overwrite the value in `r1` needed by (2), which is incorrect. This is known as a *anti dependence*. The last two dependences are known as *false* dependences because serialization is caused by use of the same result register in two different instructions. This serialization can be eliminated by renaming the result register of the (3), as shown in Fig. 1(b), so (3) can be executed in parallel with (1) and (2). Register renaming hardware attempts to apply this transformation *dynamically*, thereby increasing the amount of parallelism available.

Another kind of dependence that inhibits parallelism is known as *control* dependence. It is caused by conditional branches and other transfers of control. Consider the following code fragment:

```
(1) r1   := r2 * r3
(2) breql r1, 0, L1
(3) r4   := 100 / r2
```

Clearly, (3) can be issued as soon as (1) can. However, at that point it is not known whether (3) will be executed; for instance, if either `r2` or `r3` were `0`, then (3) would not be executed. Dynamic speculation hardware issues an instruction before it is known whether or not the instruction should be executed. The hardware must be able to negate the effects of such a speculatively issued instruction if it is determined later that the instruction should not have been executed. Thus, if (3) is issued speculatively, the hardware must be able to restore `r4` to its original value if it is later determined that control branched to `L1`.

Implementations of register renaming described in the literature [5, 8, 11, 7] require a large amount of hardware, tend to increase the cycle time, or both. Extending these schemes to handle dynamic speculation and precise interrupts increases the complexity of the hardware required and may further impact machine cycle time.

In this paper, we describe a new mechanism for implementing register renaming, dynamic speculation and precise interrupts. Innovative features of this mechanism include the following:

- By folding renaming into the instruction fetch stage, instead of implementing it in the instruction decode stage as competing approaches do, we remove complexity from the already-complex decode stage.
- The critical path of the mechanism can be implemented in parallel with the tag compare logic of the instruction fetch stage, and therefore should not affect the machine cycle time.
- By eliminating the need for associative lookup, the mechanism removes a major source of hardware complexity.

The rest of the paper is organized as follow. Section 2 summarizes the state of the art of dynamic register renaming and speculation. Sections 3 and 4 describe our approach to register renaming. Sections 5 and 6 specify extensions for dynamic speculation and precise interrupts. Section 7 sketches out a design. Section 8 discusses an optimized design that merges register renaming, dynamic speculation and precise interrupt mechanisms. Section 9 elaborates the performance-critical portion of the designs, and shows that it should not impact cycle time. Section 10 compares our approach against previous approaches. Finally, Section 11 summarizes the results of the paper.

2 Prior Work

Tomasulo's Algorithm

The earliest implementation of register renaming was for the floating-point unit of the IBM 360/91 [11]. The IBM 360 is a CISC architecture, but in the following description, we will restrict our attention to register-to-register operations.

Every register is augmented by a busy bit. If the bit is off, the register holds the value needed by any instruction that references it. If the bit is set, the register holds the *tag* of the instruction that will produce the need value.

When an instruction enters the decode stage, it is assigned a tag. It reads each source register, obtaining an input value or a tag, depending on the busy bit. The instruction writes its tag into its output register and sets that register's busy bit. The instruction and its input values/tags are buffered in one of several *reservation stations*, depending on which functional unit it will execute in.

Every cycle, each functional unit searches the instructions stored in its reservation station. They may be *ready*, i.e., have both input values available, or *waiting*, i.e., one or more of its inputs is a tag, and therefore cannot execute till the as-yet-uncompleted instruction with that tag produces a result. The functional unit will select one of the ready instructions, and start executing it.

When an instruction completes, its result and tag are broadcast to all waiting instructions. Each waiting instruction compares this tag against the tags it is waiting on. If the tags match, the instruction overwrites the tag with the result.

This may change the instructions status from waiting to ready. The result is also sent to the register file. If the tag in the output register is the same as that of the instruction, the value is written to the register file.

Tomasulo's algorithm must be extended to implement precise interrupts [9]. Johnson [5] describes one such extension, based on the *future file* mechanism [9]. The registers described above correspond to the future file. There is also a duplicate register file called the *in-order file,* and a *reorder buffer*. Instructions issue as described, using the future file[1]. Additionally, results are written to the reorder buffer. The reorder buffer is a FIFO queue. When an instruction is issued, it receives a slot at the bottom of the queue. When the instruction completes, its result is written to the allocated slot. If the instruction at the top of the queue has completed, its result is used to update the in-order file, and the queue is advanced.

It is easy to see that the reorder buffer updates the in-order file with the results of the instructions in the order in which the instructions were issued. At the time an instruction's results are written to the in-order file, any exception caused by it is signalled. Therefore, when an instruction causes an exception, the in-order file contains the register state obtained by executing all instructions prior to the excepting instruction—exactly that required by a precise interrupt.

Johnson's Algorithm

Johnson, in [5], describes another approach to register renaming. This approach replaces the future-file and reorder buffer of the previous section with an *associative* reorder buffer. As in the previous method, results are written first to the reorder buffer, which releases them to the in-order file in program order. The reorder buffer is also extended in the manner described below.

When an instruction is allocated a slot in the reorder buffer, its output register name and its tag are stored in that slot. When the instruction completes, the tag is overwritten with the result value.

To obtain an input value, the in-order file and the reorder buffer are accessed in parallel. If the reorder buffer contains no entry whose register matches the input register, the value returned by the in-order file is used. If there is some entry in

[1] For simplicity, we ignore the changes made in [5] to support speculation.

the reorder buffer with a matching register, the tag or value of the youngest such entry is returned. This, of course, requires an associative lookup prioritized by age, using the register name as the key.

Metaflow's DRIS

The Metaflow architecture [8] implements out-of-order execution and dynamic speculation as well as register renaming. Most of the mechanisms required are integrated into the *DRIS* (deferred-scheduling, register-renaming instruction shelf). Unlike the approaches described above, but like [1], instructions are not deferred in reservation stations at each execute unit, but in one central structure. Every cycle, this centralized structure is searched for instructions that can be executed, which are then dispatched to the appropriate execute unit. Apart from that, register renaming using the DRIS is very similar in spirit to the previous approach.

The DRIS is a queue, much like the reorder buffer. When an instruction is issued, it is allocated a slot at the bottom of the DRIS. When an instruction completes, its result is written to the allocated slot. If the top instruction in the DRIS has completed execution, it is popped from the DRIS, and the register file is updated with the instruction's result.

Every time an instruction is added to the DRIS, the DRIS is searched for instructions whose result registers match the incoming instructions source registers. If none is found, the required value can be read from the register file. If some match is found, then the source register is associated with the youngest such instruction. When the associated instruction completes, the required value can be read from that instruction's DRIS entry.

This mechanism differs from the tag-and-result broadcast of Tomasulo's algorithm, in that only the tag (i.e., the DRIS entry) is broadcast. It indicates that the result is available. All instructions in the DRIS waiting for that result mark it as available, and may become ready. The actual value is read from the DRIS entry[2] when the instruction is dispatched to an execute unit.

[2] Or from the register file, if the associated instruction has completed and been removed from the DRIS before the incoming instruction is issued.

RS/6000 Floating Point Renaming

The register renaming mechanism in the floating point unit of the RS/6000 [2] is quite different from those discussed above, both in motivation and in implementation. One of the motives for removing dependences was to enable the fixed-point unit to perform floating-point loads. Instead of having the fixed-point unit keep track of floating-point false dependences, the architects chose to rename the destination register of the load to eliminate the dependences.

As implemented, the floating-point unit has more physical registers than register names. There is a map-table containing a mapping of register names to physical registers. There is a queue containing free physical registers (roughly, those that are not mapped). When a floating-point instruction, other than a load, is decoded, its registers are renamed according to the map. When a floating-point load is encountered, a free physical register is removed from the queue, and used as the target of the load. The map is updated to reflect the fact that all subsequent references to the target register name will use the newly allocated physical register. We shall not discuss this mechanism in more detail since it is subsumed by the mechanism discussed below, and by the one we are proposing.

ES/9000

The IBM ES/9000 [7] extends the renaming mechanism described above to *all* registers for most instructions. It also implements dynamic speculation past at most 2 conditional branches.

As in the RS/6000, there are more physical registers than register names, and a map-table (called the *DRAL*) is used to keep track of the mapping of logical register names to physical registers. In the decode stage, the source register names in every instruction are renamed using the map. The instruction's output register is mapped to a new free physical register, and the map is updated appropriately, displacing the old physical register mapped by the output register.

Registers are controlled using information stored in an content-addressable structure called the *ACL*. Whenever all instructions prior to an instruction have completed, the ACL is associatively searched using the instruction's identifier, and the physical register displaced by that instruction is freed.

The value of the map at a branch that is speculated past is saved in structures called the *BRAL*. On a mispredicted branch, the DRAL is restored to the appropriate state by copying from the appropriate BRAL. When a branch is mispredicted and all instructions prior to the branch have completed, the ACL is associatively searched to identify physical registers allocated to instructions issued after the branch. These registers are then freed.

Finally, the ACL contains information for precise register state recovery. For every physical register, it keeps track of whether the physical register contains a value that is part of the precise state. On an interrupt, the precise state is recovered by searching the ACL for entries in the precise state, and using them to update the map appropriately.

3 Our Approach

The approach we are proposing implements register renaming in a direct fashion, using a single storage area. There is no separation of storage locations into an architected register file and some auxiliary storage. Instead, there is one set of *physical* registers, from which all storage locations are allocated and into which all values are written. The architected registers are mapped onto some subset of the available physical registers. This mapping changes as instructions are issued. The association of architected register name with physical register is maintained in a *mapping table* [6]. The mapping table has an entry for each register name, which contains the physical register currently associated with, or *mapped by*, that register name. There is a *free pool* of physical registers, used to provide new physical registers when necessary.

After an instruction is fetched from the I-cache, but before it accesses the physical registers, the instruction's source register names are *mapped* to physical registers. This is accomplished by indexing into the mapping table using the source register name.

The output register name is mapped to some new physical register, allocated from the free pool. The mapping table is modified to reflect this new association. The old physical register in the output register name's entry is replaced by the new physical register. The register name is said to have been *remapped* to the new physical register. The old physical register is said to have been *unmapped*.

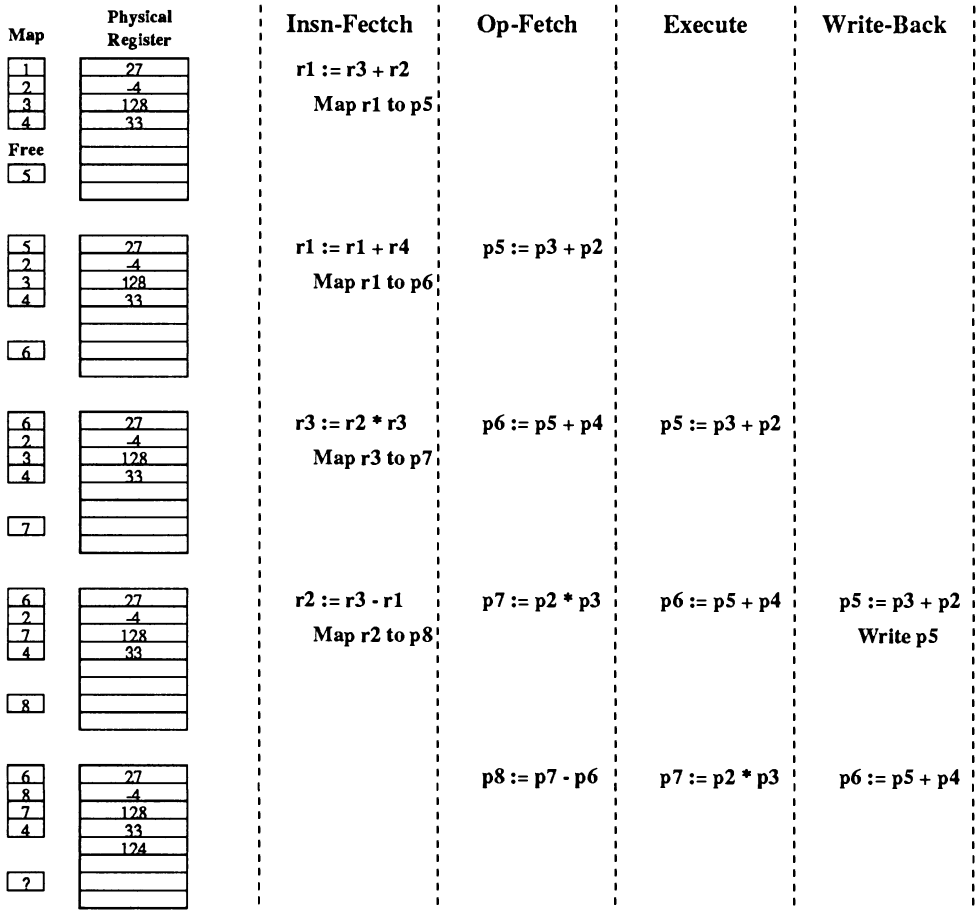

Figure 3: Mapping

The mapping process is illustrated in Fig. 3. The processor used in this example is a 4 stage pipelined processor with 4 register names `r1-r4`. The implementation uses 8 physical registers `p1-p8`. `Free` contains a physical register from the free pool.

After the first instruction, `r1:=r3+r2`, is fetched, its source register names are translated using the map to `p3` and `p2` respectively. The result of this instruction will be placed in the physical register `p5`. The map is updated to reflect the fact that `r1` is now mapped to `p5`. The remaining stages of the pipeline execute the instruction `p5:=p3+p2`.

When the next instruction is fetched, it is translated using the modified map. Thus, the source register `r1` is mapped to `p5`. The result is to be written to `p6`, the new physical register, and the map must be modified to show that `r1` is now associated with this physical register.

Notice that the mapping table has only to keep around the most recent physical register for each register name. This information is adequate since instructions are issued to the op-fetch stage in program order. A source register name, therefore, always refers to the value produced by the closest, and therefore most recently processed, instruction with the same (output) register name. Hence, the instruction uses the value stored in the physical register mapped most recently to that register name—which is exactly what the mapping table stores.

After the mapping, the rest of the stages in the processor use the physical registers. They never see the original register names. One way of viewing this transformation is to look on the mapping as a translation of instructions from an architecture with a small number (number of register names) of registers to an architecture with a larger number (number of physical registers) of registers, performed in a manner that increases instruction-level parallelism.

4 Reclaiming Physical Registers

Since there are only a finite number of physical registers, the processor must start reusing them at some point. A physical register can be reused when it is guaranteed that the value in it can never be used by any later instruction. This is true if all of the following hold:

1. The value has been written to the physical register[3].
2. All issued[4] instructions that need the value have read it.
3. The physical register has been unmapped.

The motivation for the first two conditions should be obvious. The third condition is motivated by the following: as long as the physical register is mapped by some register name, some later instruction may have that (source) register name, and therefore require that value. However, once that register name has been remapped (and, consequently, the physical register unmapped), no instruction can access that value.

The process of reclamation is illustrated by the following code fragment:

```
(1) r1 := ...
(2)    := r1 + ...
(3) r1 := ...
(4)    := r1 - ...
```

When instruction (1) is executed, it is allocated a new physical register, say `p11`. This is the register which the result of (1) will be written to. `p11` cannot be freed until (1) completes. The next instruction refers to `r1`, so it will read out of `p11`. Obviously, `p11` cannot be reused until (2) has been issued. Now, (3) again uses `r1` as an output. So, it will be allocated a new output register, say `p23`. `r1` will be mapped to `p23`, and `p11` is unmapped. Now, any subsequent reference to `r1`, such as in (4), will use `p23`. Clearly, no further uses of `p11` can be produced. So `p11` can be freed once (1) writes to it and (2) reads from it.

These conditions are easily detected using an approach which waits until the physical register has been written to and has been unmapped, and then waits for all instructions that use the physical register to complete. A counter per physical register is used to keep track of issued but unexecuted instructions in the processor pipeline that read that physical register. Since the total number of issued but unexecuted instructions in the processor is bounded and small, the counter does not have to be very big. An instruction that uses the value increments the counter as soon as it is fetched, and decrements the counter

[3]This is not strictly necessary; however, it greatly simplifies the implementation.

[4]In this paper, an instruction is said to be issued when it enters the instruction-decode/operand-fetch stage.

when it has read the value. A physical register can be reused when it has been written to, unmapped and its counter is zero.

To summarize—the processor maintains the following state for each physical register:

Complete Flag: The complete flag is false if the physical register has not been written into. It is set to false when the physical register is removed from the free pool, and is set to true when it is written to.

Unmap Flag: The unmap flag is false as long as the physical register is mapped by some register name. It is set to false when the physical register is first added to the mapping table, and is set to true when it is unmapped.

Counter: The counter keeps track of the number of instructions issued that need the value stored (or to be stored) in the physical register, but which have not yet read it. Every time an instruction is issued that needs this value, the counter is incremented. Every time an instruction reads the value, it is decremented.

A physical register can be reclaimed if the complete and unmap flags are true, and the counter value is zero.

Initially, each architected register name is mapped to the corresponding physical register. These registers are in state `F 0 T`. The remaining registers are in the free register pool, with states `T 0 T`.

Fig. 4 illustrates the process of reclaiming physical registers. The processor implementation has been extended by the addition of the update flag, the counter, and the complete flag for each physical register. If these fields have the value `T 0 T`, the register is reclaimable. One of the reclaimed register is inserted in the free slot.

The first instruction unmaps `p1`. `p1` is now reclaimable, since it has been written to, and there are no outstanding reads to it. Register `p5` is mapped by `r1`, so its unmap flag is set to false in cycle 1. The next instruction has `r1` as a source register name, so the counter for `p5` is incremented. In cycle 3, the fetched instruction has `r1` as an output register. This means that `p5` is now unmapped, and its unmap flag is set to true. Also, the result of the first instruction is forwarded to the second instruction. Since the second instruction has read the value for `p5`, it decrements the counter for `p5`. Finally, in cycle 4, the first

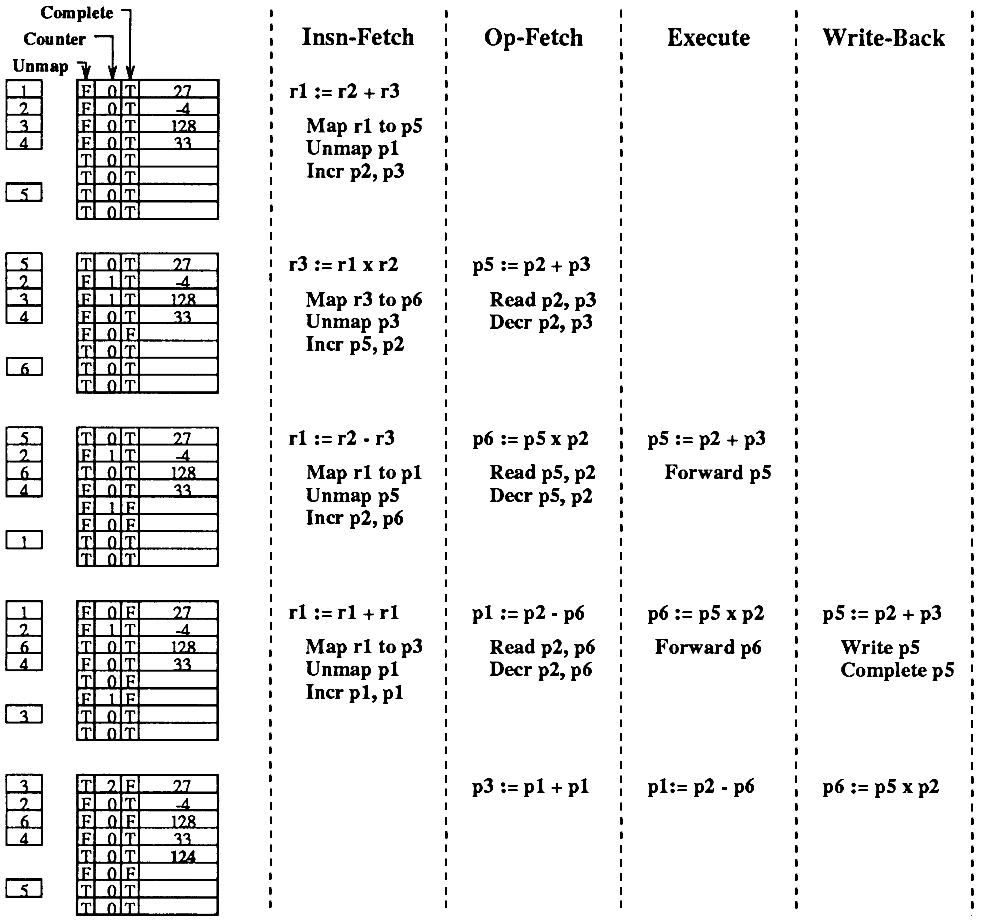

Figure 4: Reclaiming Registers

instruction completes, writes its result to the p5, and sets the complete flag for p5 to true. p5 can now be reclaimed, and is placed in the free slot.

5 Branches and Speculation

Branches are another cause of serialization. There may be an interval between the time a branch is issued and the time it is *resolved*, i.e., the branch's direction and target are determined. Halting instruction issue until a branch is resolved results in a serious performance loss. The alternative, predicting a target for the branch, and continuing to issue instructions from that target before the branch is resolved, can cause problems when the prediction proves to be incorrect. For best results, the processor should be able to deal with several simultaneously unresolved branches. To simplify the hardware, branches are assumed to be resolved in program order.

An instruction that is issued from below a branch before the branch is resolved is said to have been *speculatively* issued. If a speculative instruction causes an exception, then the exception must not be reported till it is determined whether the predicted target was correct. If it was not correct, i.e., the branch was *mispredicted*, the interrupt must be ignored. Otherwise the interrupt must be reported in a manner consistent with the program order.

Branches are mispredicted fairly often. The mechanism that recovers from a mispredicted branch must be efficient. Prior work on recovery treats a mispredicted branch as a kind of interrupt, and modifies the precise-interrupt recovery mechanism to implement mispredicted branches. But mispredicted branches do not require the full power of the precise-interrupt recovery mechanisms. Consider the following example:

```
r1 := r2 / r3
brgt0 r5, r8
```

Assume that at the time the branch is resolved, the divide has not completed. Precise interrupt recovery mechanisms would halt further execution till the divide completed. However, it is not necessary to do so just to recover from a mispredicted branch.

What actions must be taken to recover from a mispredicted branch? Consider the following example:

```
(1) r1 := r2 + r3
(2) r3 := r7 / r4
    brgt0 r5, L10
(3) r1 := r1 - r4
L10:
(4) r1 := r8 * r1
```

Assume that the branch was predicted (incorrectly) to be not taken. Then, to correctly execute instruction `(4)` after resolution, source registers `r8` and `r1` must contain the values they would have had, assuming none of the speculative instructions had executed. This means that the value read from `r1` must not be the value produced by `(3)`, but the value produced by `(1)`. Further, all values produced by the speculative instructions should be discarded. Specifically, the value produced by `(3)` could be used only by other speculative instructions, and since they, too, are mispredicted, their reading the value is irrelevant. Lastly, all interrupts produced by speculative instructions must also be discarded.

Modifying our design to satisfy these aims turns out to be fairly simple. Since multiple occurrences of the same output register name are given different physical registers, values stored in the register are never overwritten until the register is reclaimed and reassigned to another instruction. Therefore, the design must be modified to ensure that a physical register is not reclaimed before the branch is resolved. When the branch is discovered to have been mispredicted, the logical register names must be set to map to the physical registers containing the old values. In our design, this means that the mapping table must be restored to the state it was in at the time the branch was issued. These changes are described in more detail below.

The mapping table is small enough that it can be saved each time we speculate past a branch. If the branch is correctly predicted, the saved map is discarded. If the branch was mispredicted, the saved map is used to restore the mapping table, and all other saved maps are discarded. The depth of speculation in the scheme being presented is constrained by the number of maps that can be saved. There is an alternative scheme, described later, that does not require the map to be saved at each branch; instead it recovers the map from the information saved for precise interrupts. This scheme has no a priori limits on the depth of speculation.

In addition, the unmapping rule must be modified. A physical register is now considered unmapped if it is not mapped in the current mapping table, *or any of the saved mapping tables*[5]. Under the previous definition, a physical register would be considered unmapped (and therefore reclaimable) if its associated register name was the output register for some speculated instruction. This case would arise in the example below (assume branch is predicted not taken)

```
(1) r5 := r10/r2
(2) r1 := r2 + r3
    brgt0 r5, r8
(3) r1 := r3 * r4
```

Reclaiming the physical register storing the result of `(2)` would be correct as long as the branch was correctly predicted. However, if it was mispredicted, the value might be needed by some instruction on the other path. Therefore, it cannot be reclaimed as long as the branch is unresolved. The modified unmapping rule (in combination with the fact that saved maps are discarded as soon as they are resolved) ensures that physical registers are not reclaimed too soon, and that they can be reclaimed if necessary after the branch is resolved.

As long as branches are predicted correctly, the complete flag and the counters are left untouched. Resetting them at a mispredicted branch is closely tied to the way the pipeline is flushed of the speculative instructions. If, for instance, all mispredicted speculated instructions in the pipeline are run to completion, and then discarded, complete flags and counters can be handled unchanged.

An interesting case is when the processor waits for all instructions issued before a mispredicted branch to complete before starting recovery, and then flushes all instructions in the pipe. At that point, since all instructions prior to the branch have completed execution, all values must have been read, so all counters must be set to zero. Further, all instructions prior to the branch have completed, so that all complete flags can be set to true. The speculative instructions do not matter, since they are all being discarded.

[5]There is an equivalent, but more complex definition, that can be implemented more cheaply and will be described later.

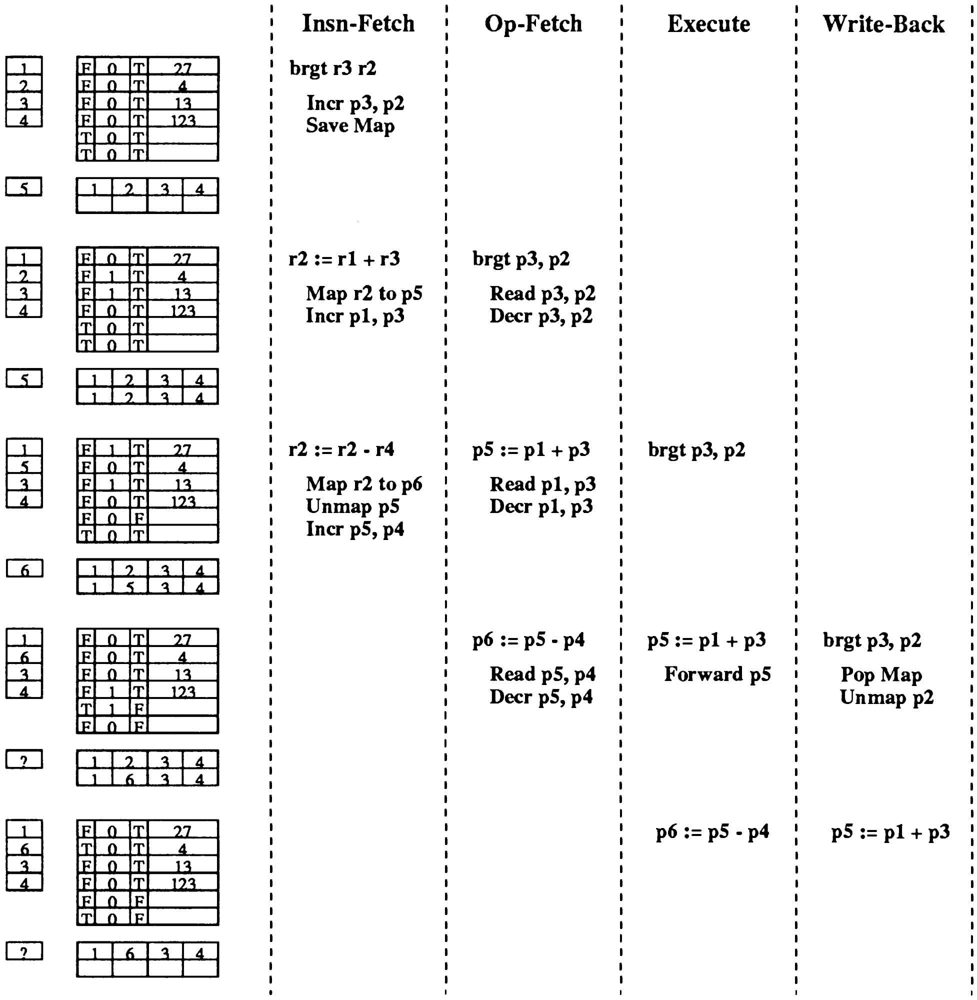

Figure 5: Correctly Predicted Branch

The examples in Fig. 5 and Fig. 6 illustrate the way our design handles speculation. The architecture is further extended by the addition of a 2-entry map save stack. However, the current map is always stored in this stack. Therefore, the processor, as shown, can speculate past at most 1 branch. Mispredicted branches are assumed to be issued only after all prior instructions have completed. Counters and complete flags are handled according to the above scheme.

When the branch instruction is issued, the processor saves the current map on the map save stack. The next instruction remaps `r2` to `p5`. However, the previously mapped physical register `p2` is still not consider unmapped, since it is mapped in the map save stack. The next instruction again remaps `r2`. `p5` *is* considered unmapped, since it is no longer mapped by the current or any of the saved maps. Finally, the branch is resolved in the 4th cycle.

Fig. 5 shows the processor behavior when the branch was correctly predicted. The saved map is popped from the stack. `p2` is now no longer mapped by any map, and its unmap flag is set to true. In fact, it can now be reclaimed.

Fig. 6 shows the processor behavior if the branch was mispredicted. All but the top saved map are flushed from the map save stack. This map is used to restore the mapping table. The physical registers not mentioned in this map are readied for reclamation by setting their unmap and complete flags, and zeroing their counters. Of course, the instructions currently in the pipeline are flushed.

6 Precise Interrupts

Any one of several precise interrupt mechanisms [9] can be adapted to the proposed design. The major modification necessary is that when an interrupt occurs, not only must the register state reflect the precise state, but the mapping table must be restored to its value at the time the interrupting instruction was issued. Actually, it is not necessary for all the physical registers to be restored to their value at the time the interrupting instruction was issued; it is enough to restore only those registers that were actually mapped at that point.

For instance, if we wish to adopt a checkpoint-retry mechanism [4] then, at the checkpoint, we would need to save the current map, and the values of the physical registers mapped by the current map. On an interrupt, the checkpointed values would be used to reset the map and the relevant physical registers.

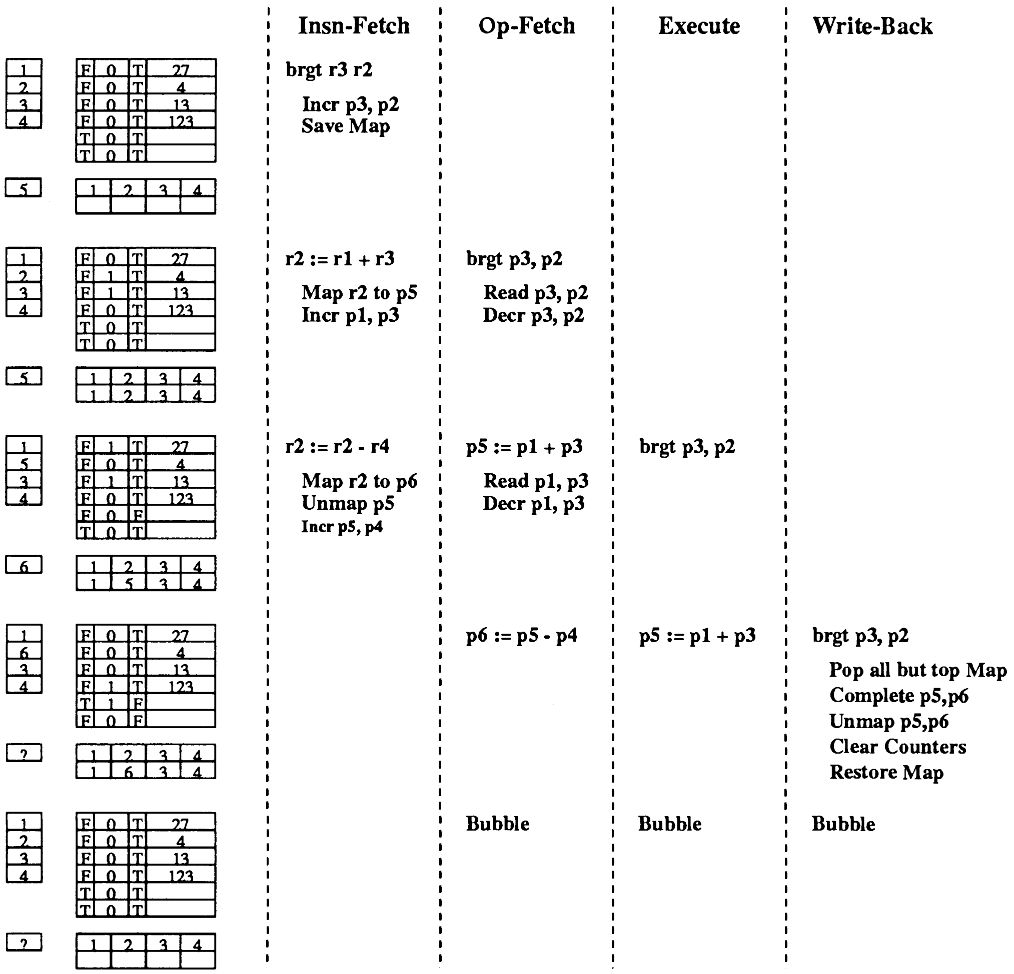

Figure 6: Mispredicted Branch

The history buffer mechanism is a particularly attractive candidate. A history buffer is a FIFO, to which every instruction is added when it is fetched. When the instruction writes to the register file, the value it overwrites is saved in with it. Instructions are removed from the top of the history buffer when they complete. Any exception caused by an instruction is reported only when it comes to the top of the stack. At that point the register file values are restored by "roll back"—basically, starting from the bottom of the buffer, all the saved register values are written back to the register file. The straight-forward adaption of a history buffer to our scheme will save with each instruction the *old* physical register mapped by its output register. On an interrupt, both the map and the register file are restored to their precise state by rolling back the history buffer.

We have discussed the precise interrupt mechanism as a separate component of the processor. This enables us to develop a design that maximizes speculation, described in the Section 7. However, by accepting some restrictions on register reuse and speculation, it is possible to integrate precise interrupt support with renaming and speculation, as shall be shown in Section 8.

7 Design

In this section, we shall partially detail an implementation of the general scheme described above. For concreteness, we shall assume that the implementation is of a RISC architecture with 32 architected registers. We shall, for simplicity, assume that there is only one register file, i.e., there is no separate floating point register file. The implementation has 64 physical registers. The machine fetches and issues upto 4 instructions per cycle from a 2-way set associative cache with a cache line containing 4 instructions. These instructions are vanilla RISC instructions, with upto 2 input and 1 output register each. The implementation can speculate past 7 branches. The pipeline is purged of mispredicted speculated instructions by allowing them to run to completion, but inhibiting any exceptions they might cause. The success or failure of a branch prediction is reported only after all prior branches have been resolved.

7.1 Mapping Logic

The mapping hardware is responsible for translating each 5-bit source register specifier to the corresponding 6-bit physical register. As shown in Fig. 7, this logic is implemented in parallel with the tag match logic. Therefore, it must translate the source registers for instructions from the cache lines from *both* sets. Thus there are upto 16 source registers that must be translated.

The implementation uses the the source register specifiers to select the relevant physical register from the current map using 32:1 multiplexors. Each of the 32 current map entries contains 6-bits. Thus, the source register must control six 32:1 multiplexors. There are a total of 96 such 32:1 multiplexors.

Further translation may have to take place in the instruction decode stage. This becomes necessary when some instruction has a source register that is the same as the output register of some other instruction issued simultaneously, but prior to it in the program order. In this case, the source register must be mapped not to the physical register selected by the multiplexor, but to the free physical register allocated to the prior instruction. Fortunately, both of these instructions cannot be issued in the same cycle; the instruction must wait for the prior instruction to complete, since it uses the results of the prior instruction. This gives a full cycle to complete the remapping process.

The same problem arises when the decode stage does not process all the fetched instructions. For instance, a full reservation station may force the decode stage to decode only 2 out of 4 instructions. The unissued instructions must be examined for a similar source-output equality, and the mappings modified appropriately.

7.2 Map Modify Logic

The map modify logic is responsible for updating the current map after each cycle. Each of the 32 entries are modified in parallel. Briefly, the mapping for each architected register is updated as follows:

- On a reset or a context switch, the entry is set to some default value, such as the entry number. Thus, initially, architected register N maps to physical register N.

- On a mispredicted branch, the map is updated from the map at the top of the map save stack.
- If the architected register is the same as the output register from one of the fetched instructions, then it is replaced by the free physical register allocated to that instruction. This involves decoding the instruction to see if it had an output register. Also, if the architected register is the output register for multiple instructions, then it must be replaced by the free register allocated to the last such instruction.
- Otherwise it is set to its old value.

This logic executes in parallel with the cache-access—while the next group of instructions is being fetched, the map is being modified to reflect the affects of the instructions from the previous cycle.

7.3 Map Save Table

The map save table, as indicated before, saves the map values at each branch, and the current map. It is easily implemented using a array with 8 entry of 192 bit each, having 1 read and 1 write port, and two pointers, one to the top of the table and one to the bottom. Every cycle, the top of the table is read and sent to the map modify logic. Further, the new map is saved at the bottom of the table every cycle. On a branch, the bottom of the table is incremented (if possible; otherwise fetching is stalled). On a mispredicted branch the bottom of table is set to the top of the table. On a correctly predicted branch the top of the table is incremented.

7.4 Free Physical Register Logic

Freeing physical registers is possibly the most complex part of the design. It involves keeping track of when an instruction has completed, the number of outstanding uses, and whether it has been unmapped. Completion is easy; whenever a physical register is written, the register's completion flag is set. Keeping track of outstanding reads involves a multi-input up-down counter, which is incremented by the upto 4 instructions that can be fetched each cycle, and decremented by the instructions executed each cycle.

Unmapping is implemented using a table paralleling the map save table, called the register save table. There is an entry in the register save table for every entry in the map save table. This entry is a bit-vector representing the physical registers that are stored in the corresponding map save table entry. Thus, in the implementation being discussed, the register save table contains 8 entries of 64 bits each. Each entry will have exactly 32 bits set.

A register is unmapped if it is not in any of the tables. Implementing this directly would be expensive. Instead, we make use of the following observation: If a physical register is present in some map, then all the maps it is present are contiguous. Once a physical register is added to the current map, it stays in the map till it is remapped by some other instruction. Every time the current map is saved at a branch, the physical register will be saved in that map. These saved maps (and the current map, if the physical register is mapped) are contiguous. Once the register has been removed from the current map, it can only be added to the map after it has been freed. However, it can only be freed after it appears in none of the saved maps, or the current map. Thus, as long as the register is mapped in some contiguous set of saved maps, it cannot be present in some other saved map. From this it follows that, when a physical register is being removed from a map, it is sufficient to check the adjacent map to see if it should be unmapped.

From this observation, it follows that a register that is removed from the current map must be unmapped only if it is not mapped in the previously saved map, i.e., the next-to-bottom entry in the map save table. On a correctly predicted branch, all registers mapped by the top of stack are compared with those mapped by the next entry. Those not present in that next entry are unmapped. Finally, on a mispredicted branch, all registers not in the top of stack are unmapped. These operations are implemented using the register save stack and combinatorial logic.

8 Optimization

The implementation described above is fairly general, and can be made even more general. For instance, it could be adapted to handle branch prediction resolution in any order. In this section, we shall do the opposite: we shall introduce certain constraints on the implementation. These constraints enable us to greatly simplify the implementation.

One major source of complexity is determining when physical registers can be reused. This happens when the register has been written to, there are no outstanding reads and the associated architected register has been remapped. These must all be true if all instructions upto and including the instruction that displaced the physical register have completed. Clearly, the instruction that was allocated the physical register must have completed, and therefore the register must have been written to. All the instructions that could access this register must precede the displacing instruction, and since they must have completed, all reads of the register must have been accomplished. And, of course, the register must have been remapped. This criteria—all instructions upto and including the instruction that remaps the physical register must have completed—is fairly simple to implement.

The implementation can be built around a history buffer. An instruction is added to the bottom of the history buffer when it is fetched. An instruction is removed from the top of the history buffer after it completes. Thus, the history buffer contains instructions in program order. To support register freeing, when an instruction is added to a history buffer, the physical register it remapped is saved with it. When the instruction is removed from the top of the history buffer, this physical register is freed.

A mechanism that frees registers as described above can be used to significantly simply precise state recovery. we only need to recover the values in those registers that were mapped at the time the excepting instruction was issued. In our scheme, a register can only be overwritten after it is freed, and it is freed only after all instructions that were issued while it was mapped complete. Thus, all values which are part of the precise state are present in physical registers, which have not been freed. Therefore, when an instruction traps, only the map has to be restored to what it was when the instruction was issued; none of the registers that were mapped at that time would have been freed and overwritten.

Interrupt recovery and misprediction recovery can be facilitated by maintaining an *in-order map*, i.e., the map used by the instruction at the top of the history. This can be implemented by saving, for each instruction in the history buffer, the physical register which the instruction's output register was mapped to before being remapped by the instruction. When an instruction is retired from the top of the history buffer, the in-order map is updated appropriately. If exceptions are reported only when the excepting instruction reaches the top of the history buffer, the map can be restored in a single cycle by copying the in-order map instead of rolling back the history buffer. Similarly, if mispredicted branches are handled only when the branch reaches the top of the history buffer, the map can be reloaded using the in-order map.

The in-order map also suggests another optimization: there is no need to save the physical register that was displaced physical register in the history buffer to support freeing. Instead, the register to be freed can be selected from the inorder map by using the architected output register of the instruction at the top of the history buffer.

To summarize: the history buffer has an entry for every instruction. The information contained in this entry includes the instruction, the physical register displaced from the current map by the instruction when it was issued, and a bit indicating whether the instruction has completed. There is an in-order map, which satisfies the invariant that it is the same as the current map used by the instruction at the top of the history buffer. The various actions performed are:

- When an instruction is fetched, it is added to bottom of the history buffer. The physical register that the instructions output register is remapped to is also added (assuming, of course, that the instruction has an output register).
- When the instruction at the top of the history buffer completes, it is removed from the history buffer. The physical register in the in-order map that corresponds to the architected output register of the instruction is freed. It is replaced by the physical register saved with the instruction being removed.
- If the instruction at the top of the history buffer is a mispredicted branch, then all registers not in the in-order map are freed. The in-order map is copied to the current map.
- The behavior on an interrupt is identical to the behavior at a mispredicted branch.

Thus, it is possible to dispense with the counters, map save stack, register save stack etc., and implement physical register freeing and mispredict recovery economically using the history buffer mechanism described above. This mechanism also allows us to implement single-cycle precise state recovery. All this is accomplished at the cost of some generality.

9 Implementation

We have discussed the need for additional hardware, including logic to perform the mapping, modify the map, free registers, etc. We also assumed the existence of logic for branch prediction and an instruction dispatch mechanism[6]. However, lack of space prevents us from discussing the detailed implementation of all of this logic. Instead, we shall concentrate on the mechanism for actually performing the mapping, since this is the only logic which could potentially increase the machine cycle time.

In the past, renaming has been implemented in the instruction dispatch stage. This stage is considered to be one of the most critical stages in superscalar processors in determining cycle time [3, 12]. We propose to implement the renaming scheme we introduce in the instruction fetch stage, as shown in Fig. 7. The critical paths are shown by the solid lines; paths shown by the dashed lines represent feed-back paths, i.e., values used in the *next* cycle.

The proposed scheme requires that each source register name be mapped using the mapping table to its associated physical register. If there are are 32 architected registers, the mapping is implemented in the *mapping logic* as a 32:1 mux. Using a pass-transistor based multiplexor implementation in CMOS, this will probably involve 4 logic stages. The source register fields must control several multiplexors; this requires they be initially powered up to obtain a fanout estimated to be about 40.

As shown in the figure, the critical path is determined either by the tag match logic required for cache operations or by the mapping logic discussed previously. The path through the tag match logic is influenced by four factors, namely, the associativity of the cache, its size, the address size and the fan-out requirements of the tag compare. For the purposes of this discussion, we assume that the cache is not direct mapped and that the address size is 32 bits.

[6]We have deliberately not specified whether instruction dispatch uses reservation stations or a centralized mechanism; our approach works with both.

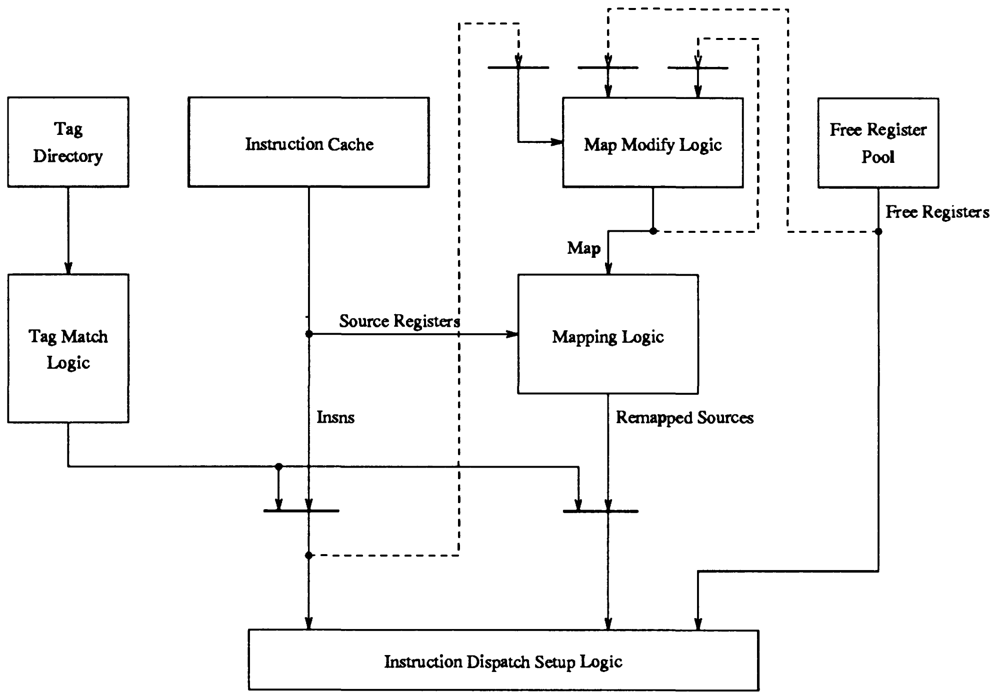

Figure 7: Instruction Fetch Stage

The tag match logic starts off by checking, depending on the cache size, possibly the high 18–24 bits of the address and the program counter for equality, which can be implemented in about 4 stages of logic stages. The produced bits then serve as control for a number of multiplexors. If the implementation attempts to issue 4 instructions per cycle, this implies a fanout of about 128[7]. This requires a powering tree that will add 2–3 logic stages (assuming some of the powering is buried in the compare).

Putting all the requirements together, we observe that both paths require about the same number of logic stages: about 6–7. This suggests that, assuming CMOS technology and a set associative cache, the proposed scheme may not increase the machine cycle time.

Notice that only the source registers go through the mapping logic; specifically, the time at which the opcodes become available to the instruction decode stage does not change. This allows us, if necessary, to possibly overlap some of the mapping with the initial phases of the instruction decode in the decode stage.

None of the other mechanisms required add to the cycle time. Updating the map to reflect any remapped registers is not time critical. It can be done in the *next* cycle, in parallel with the I-cache access. The mechanism to free physical registers is also not time-critical. All that matters is that there be enough free physical registers, not which ones, i.e. throughput matters, not latency. Adding an extra cycle to the time to free a physical register can be compensated by expanding the number of physical registers by the number required per cycle.

So far in the discussion, we have assumed that all instructions are 3-address instructions. There will be several instructions with no output register (e.g., branches), and some with only one input register. Superscalar issue adds a related problem—if some source register name is the same as the result register name of some instruction fetched at the same time, but prior in the program order, it must use the new physical register allocated to that register name, rather than the entry in the mapping table. It would seem that we would need to do some decoding in the instruction-fetch stage.

However, we can delay these decisions till the next cycle. All instructions are initially mapped as though they were independent 3-address instructions.

[7]The fanout can be of this order, or worse, even if less instructions are issued per cycle, depending on the physical organization and read out of the cache arrays.

In the next cycle, as part of the remap logic, the instructions are decoded to determine whether they actually had an output, and source and result registers are compared for identity. If the instruction did not have an output, or multiple instructions had the same output register, the mapping is updated appropriately. Thus, decodes are *not* added to the critical path.

A similar fixup has to be performed in the instruction decode stage. Determining the appropriate register to use can be done in parallel with register access. If the physical register provided by mapping table is incorrect the value read can be discarded. This results in no loss in performance. Either the instruction did not need that value, or the wrong register was accessed because some other instruction fetched in parallel had the same output register name. In the second case, the value needed is the one that will be provided by the other instruction. That instruction, obviously, cannot have completed, so the instruction with the incorrectly mapped register cannot be issued immediately, anyway. Thus, there is at least a full cycle in which to determine the correct register mapping.

10 Comparison

The performance gained by adding a new mechanism has to be balanced against the amount of hardware required to implement the mechanism, and that implementation's potential impact on the cycle-time of the processor. Both of these are extremely difficult to quantify without actually an actual implementation, and even with such an implementation it is sometimes difficult to isolate the impact caused solely by the new mechanism, or by its interactions with other processor elements. So, the comparison of our scheme with other schemes that implement register renaming, dynamic speculation and precise interrupts will be more qualitative than quantitative.

One critical path in register renaming is the path from the instruction fetch to execution. On a vanilla RISC processor, this path is implemented in two stages, Instruction-Fetch and Instruction-Decode/Operand-Fetch. Logically, it follows the following steps:

$$Fetch \rightarrow Decode \rightarrow ReadValues \rightarrow Execute$$

Adding renaming adds an extra step:

$$Fetch \rightarrow Decode \rightarrow Rename \rightarrow ReadValues \rightarrow Execute$$

If these steps are implemented serially, obviously either an extra stage will be required, as in [2, 8], or the cycle-time of the existing two stages will be increased. The only way to avoid this is to implement some of the steps in parallel. [5] discusses several ways of folding together register renaming with register read, all of which require content-addressable storage. The time to perform an associative lookup is obviously going to be greater than that of a register access. It may turn out that the associative access will end up increasing the cycle time.

Our approach chooses to fold renaming with instruction fetch. A preliminary evaluation suggests that the mapping mechanism used can execute in parallel with elements of the instruction fetch stage without impacting the critical path through that stage. The rest of the path should not change from that in a vanilla RISC processor. In particular, reading values involves a normal register access.

Associative lookup requires content-addressable storage to implement efficiently. This storage requires complex and expensive hardware. The designs proposed by [2] and [5] use associative lookup to access register values. Similarly, the implementation described in [7] uses content-addressable storage for register control. Our scheme, on the other hand, requires no such complexity.

Finally, our scheme like [2] and [5], but unlike [7], enables us to recover precise register state in a single cycle. This in turn lets us implement single cycle interrupt recovery and speculation past an arbitrary number[8] of branches.

11 Conclusion and Future Work

In this paper, we have presented a proposal that is in line with the current thrust of processor design and implementation. It exploits both the regular nature of RISC instructions and the fast, pass-gate based, multiplexors available in CMOS technology to efficiently implement renaming in the instruction fetch stage. This early implementation of renaming has simplifying repercussions for the rest of the design.

[8]Not entirely arbitrary; it is limited to the number of slots in our "history buffer"

The optimized implementation proposed integrates speculation, renaming and precise interrupts. The synergy resulting from this integration results in a simple and potentially cheap implementation of the three mechanisms.

It is seems fairly clear that the hardware costs of our proposal are fairly modest, especially in the context of the 3 million or more transistors being used to implement current microprocessors. The open question, therefore is, can the proposal be implemented without increasing the cycle-time? We hope in the future to add our mechanisms to an existing microprocessor, and demonstrate, at least via simulation, that the addition will not impact the cycle-time of the proposal.

12 Acknowledgment

We would like to thank our colleagues at IBM Glendale Laboratory, Endicott, in particular Steve Gregor and Tom Jeremiah, for verifying our estimates of the critical paths.

References

[1] H. Dwyer and H.C. Torng. Out-of-order superscalar processor with speculative execution and fast, precise interrupts. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 272–281, 1992.

[2] G.F. Gohoski. Machine organization of the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):37–58, January 1990.

[3] J.L. Hennessy and N.P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, 24:18–29, September 1991.

[4] W.M. Hwu and Y.N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, June 1987.

[5] W.M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[6] R.M. Keller. Look-ahead processors. *Computing Surveys*, 7(4):177–195, December 1975.

[7] J.S. Liptay. Design of the IBM Enterprise System/9000 high-end processor. *IBM Journal of Research and Development*, 36(4):713–731, July 1992.

[8] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The Metaflow architecture. *IEEE Micro*, pages 10–13,63–73, June 1991.

[9] J.E. Smith and A.R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[10] M.D. Smith, W.M. Johnson, and M.A. Horowitz. Limits on multiple instruction issue. In *Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, April 1989.

[11] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[12] S. Vassiliadis, B. Blaner, and R. Eickemeyer. SCISM: A scalabale compound instruction set machine. *IBM Journal of Research and Development*, to appear 1993.

[13] D.W. Wall. Limits of instruction-level parallelism. In *Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.