3.2 SUPERSCALAR PROCESSOR DESIGN

In Section 3.1 we focus on the structural, or organizational, design of the superscalar pipeline and deal with issues that are somewhat independent of the specific types of instructions being processed. In this section we focus more on the dynamic behavior of a superscalar processor and consider techniques that deal with specific types of instructions. The ultimate performance goal of a superscalar pipeline is to achieve maximum throughput of instruction processing. It is convenient to view instruction processing as involving three component flows of instructions and data, namely, *instruction flow*, *register data flow* and *memory data flow* [Johnson 1991]. Consequently the overall performance objective is to maximize the volumes in all three of these flow paths. Of course what makes this task interesting is that the three flow paths are not independent and their interactions are quite complex. This section presents superscalar microarchitecture techniques, which are classified based on their association with the three flow paths. The three flow paths of instruction, register data, and memory data, correspond roughly to the processing of the three major types of instructions, namely branch, ALU, and load/store instructions, respectively.

1. **Instruction Flow:** Branch instruction processing.
2. **Register Data Flow:** ALU instruction processing.
3. **Memory Data Flow:** Load/store instruction processing.

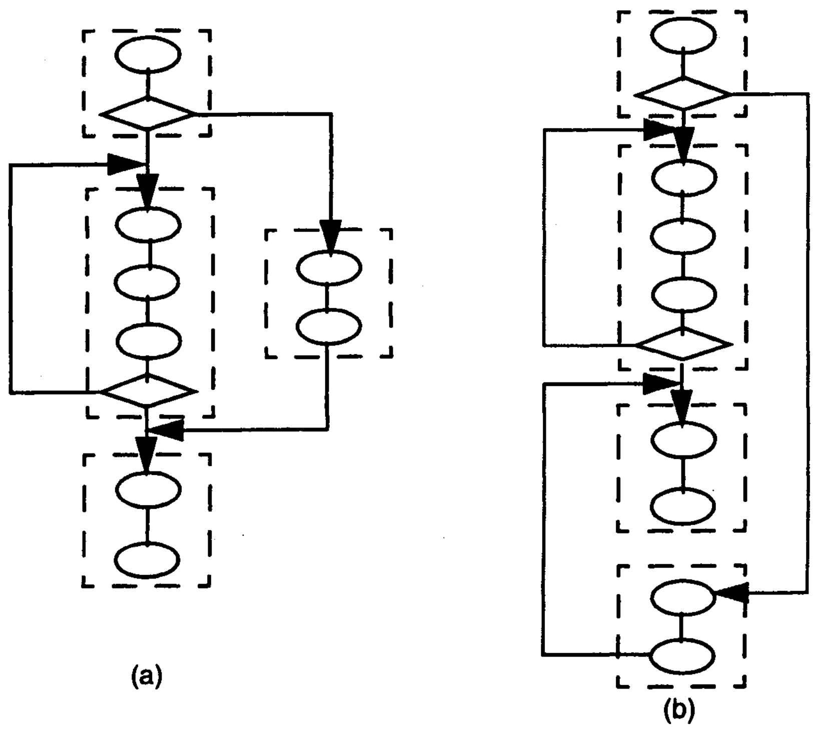
3.2.1 Instruction Flow Techniques

We present instruction flow techniques first because these deal with the early stages, e.g. Fetch and Decode stages, of a superscalar pipeline. The throughput of the early pipeline stages will impose an upper bound on the throughput of all subsequent stages. For contemporary pipelined processors, the traditional partitioning of a processor into control path and data path is no longer clear and effective. Nevertheless, the early pipeline stages along with the branch execution unit can be viewed as corresponding to the traditional control path whose primary function is to enforce the control flow semantics of a program. The primary goal for all instruction flow techniques is to maximize the supply of instructions to the superscalar pipeline subject to the requirements of the control flow semantics of a program.

3.2.1.1 Program Control Flow and Control Dependences

The control flow semantics of a program is specified in the form of the control flow graph (CFG), in which the nodes represent basic blocks and the edges represent transfer of control flow between basic blocks. Figure 21(a) illustrates a CFG with four basic blocks (dashed-line rectangles) each containing a number of instructions (ovals). The directed edges represent control flows between basic blocks. These edges are induced by conditional branch instructions. The run-time execution of a program entails the dynamic traversal of the nodes and edges of its CFG. The actual path of traversal is dictated by the branch instructions and their branch conditions which can be dependent on run-time data.

FIGURE 21 Program control flow: (a) the control flow graph (CFG); (b) mapping the CFG to sequential memory locations.

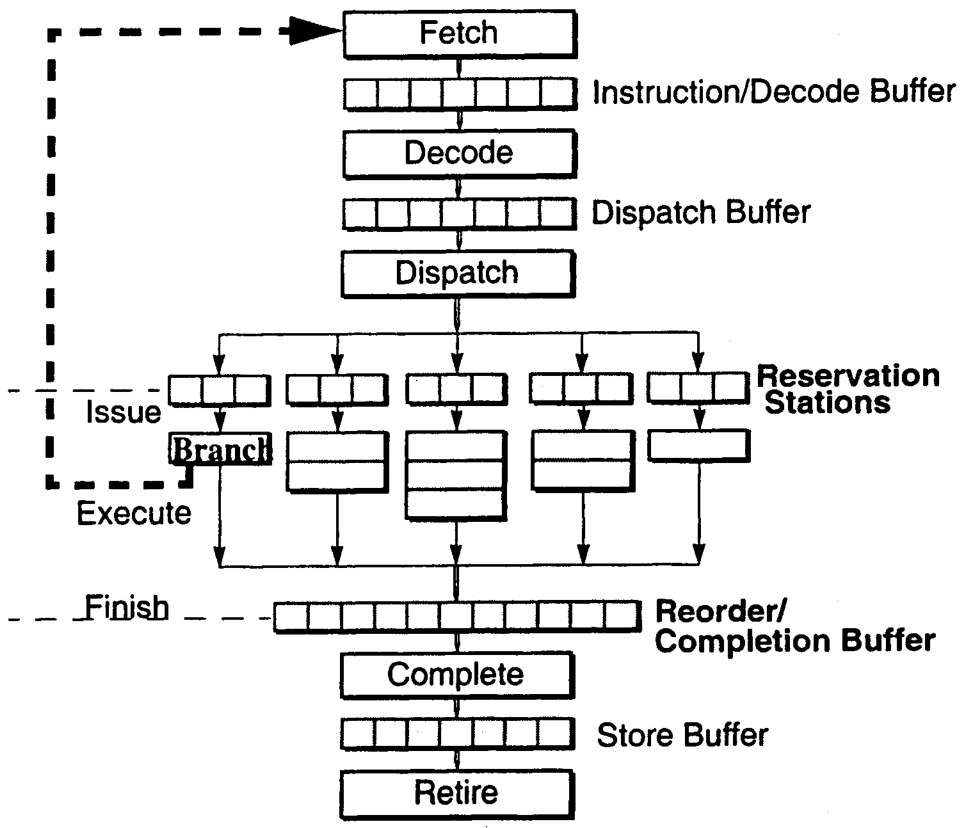


The basic blocks, and their constituent instructions, of a CFG must be stored in sequential locations in the program memory. Hence the partial ordered basic blocks in a CFG must be arranged in a total order in the program memory. In mapping a CFG to linear consecutive memory locations, additional unconditional branch instructions must be added as illustrated in Figure 21(b). The mapping of the CFG to a linear program memory facilitates an implied sequential flow of control along the sequential memory locations during program execution. However the encounter of both conditional and unconditional branches at run time induces deviations from this implied sequential control flow and the consequent disruptions to the sequential fetching of instructions. Such disruptions cause stalls in the instruction fetch stage of the pipeline and reduce the overall instruction fetching bandwidth. Subroutine jump and return instructions also induce similar disruptions to the sequential fetching of instructions.

3.2.1.2 Performance Degradation Due to Branches

A pipelined machine achieves its maximum throughput when it is in the streaming mode. For the fetch stage, streaming mode implies the continuous fetching of instructions from sequential locations in the program memory. Whenever the control flow of the program deviates from the sequential path, potential disruption to the streaming mode can occur. For unconditional branches, subsequent instructions cannot be fetched until the target address of the branch is determined. For conditional branches, the machine must wait for the resolution of the branch condition and if the branch is to be taken it must further wait until the target address is available. Figure 22 illustrates the disruption of the streaming mode by branch instructions. Branch instructions are executed by the branch functional unit. For a conditional branch, it is not until it exits the branch unit and when both the branch condition and the branch target address are known that the fetch stage can correctly fetch the next instruction.

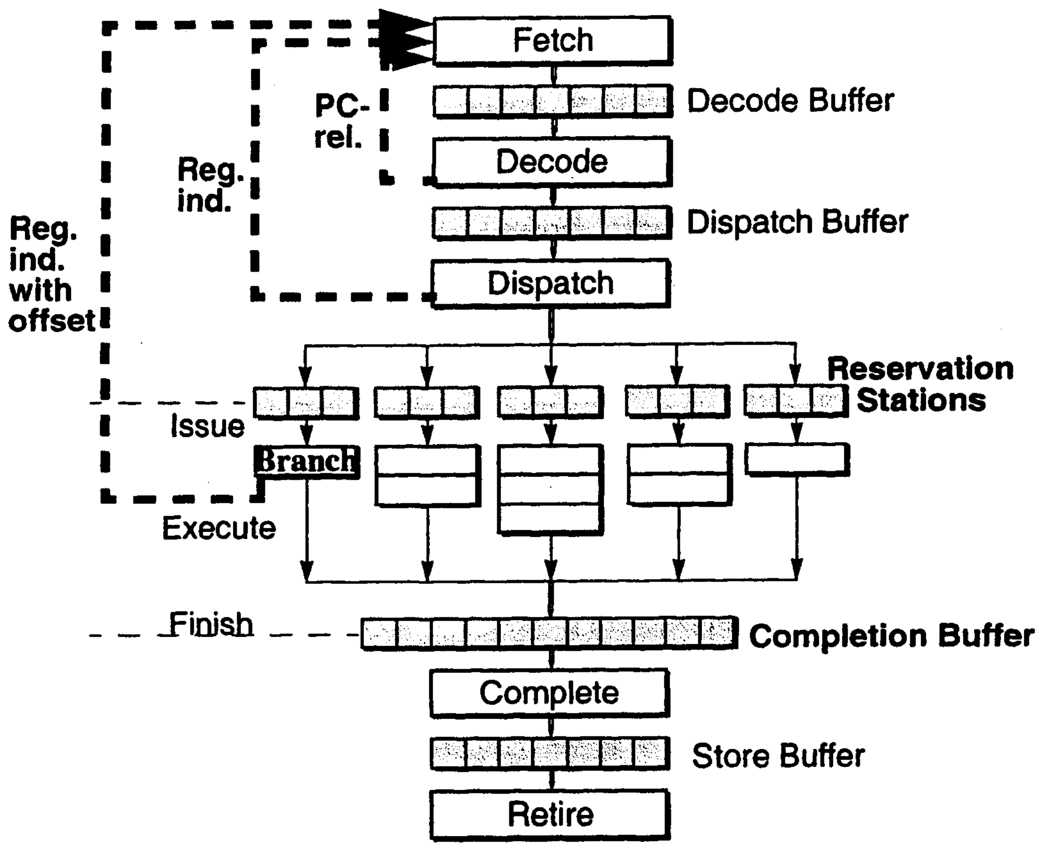
FIGURE 22 Disruption of sequential control flow by branch instructions.



As Figure 22 illustrates, this delay in processing conditional branches incur a penalty of three cycles in fetching the next instruction, corresponding to the traversal of the Decode, Dispatch and the Execute stages by the conditional branch. The actual lost-opportunity cost of three stalled cycles is not just three empty instruction slots as in the scalar pipeline but must be multiplied by the width of the machine. For example, for a 4-wide machine the total penalty is 12 instruction "bubbles" in the superscalar pipeline. Also recall from the last chapter, such pipeline stall cycles effectively correspond to the "sequential bottleneck" of the Amdahl's law and rapidly and significantly reduces the actual performance from the potential peak performance.

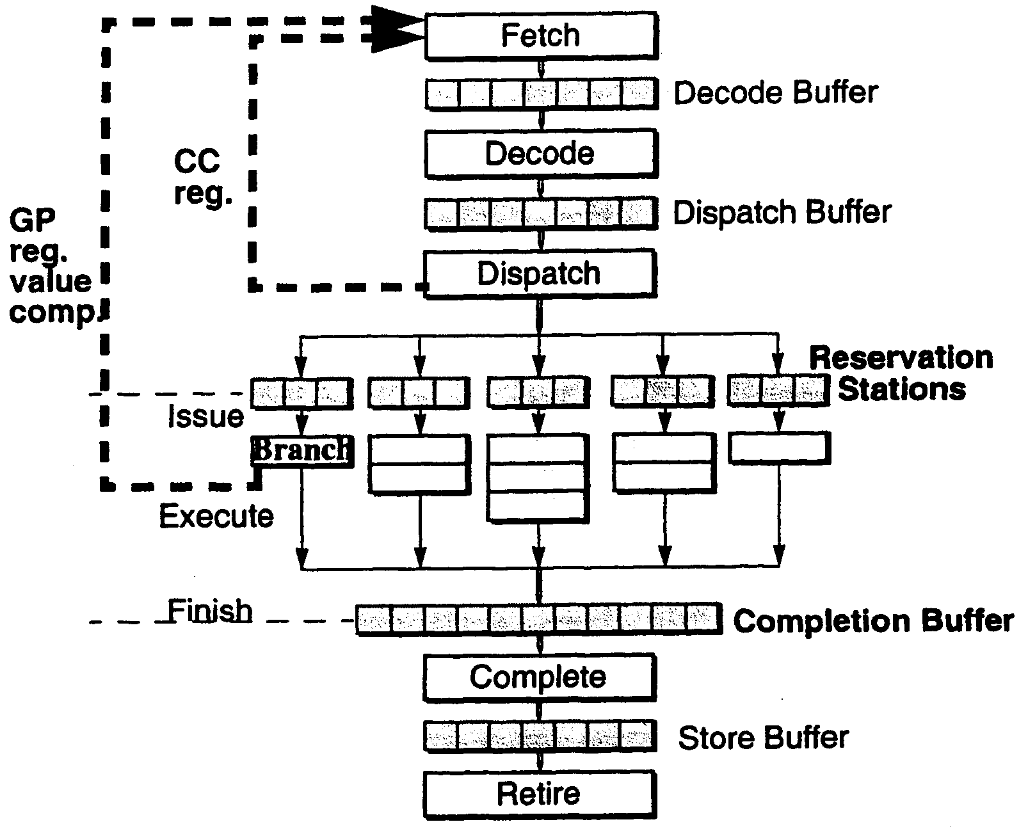
For conditional branches, the actual number of stalled or penalty cycles can be dictated by either target address generation or condition resolution. Figure 23 illustrate the potential cycles that can be incurred by target address generation. The actual number of penalty cycles is determined by the addressing modes of the branch instructions. For PC-relative addressing mode, the branch target address can be generated during the Fetch stage resulting in a penalty of one cycle. If register indirect addressing mode is used, the branch instruction must traverse the Decode stage to access the register. In this case a two-cycle penalty is incurred. With register indirect with an offset addressing mode the offset must be added after register access and a total three-cycle penalty can result. For unconditional branches, only penalties due to target address generation are of concern. For conditional branches, branch condition resolution latency must also be considered.

FIGURE 23 Branch target address generation penalties.



Different methods for performing condition resolution can also lead to different penalties. Figure 24 illustrates two possible penalties. If condition code registers are used, and assuming that the relevant condition code register is accessed during the Dispatch stage then a penalty of two cycles will result. If the ISA permits the comparison of two general purpose registers to generate the branch condition then one more cycle is needed to perform an ALU operation on the contents of the two registers. This will result in a penalty of three cycles. For a conditional branch, depending on the addressing mode and condition resolution method used, either one of the penalties may be the critical one. For example, even if the PC-relative addressing mode is used, a conditional branch that must access a condition code register will still incur a two-cycle penalty instead of the one-cycle penalty for target address generation.

FIGURE 24 Branch condition resolution penalties.



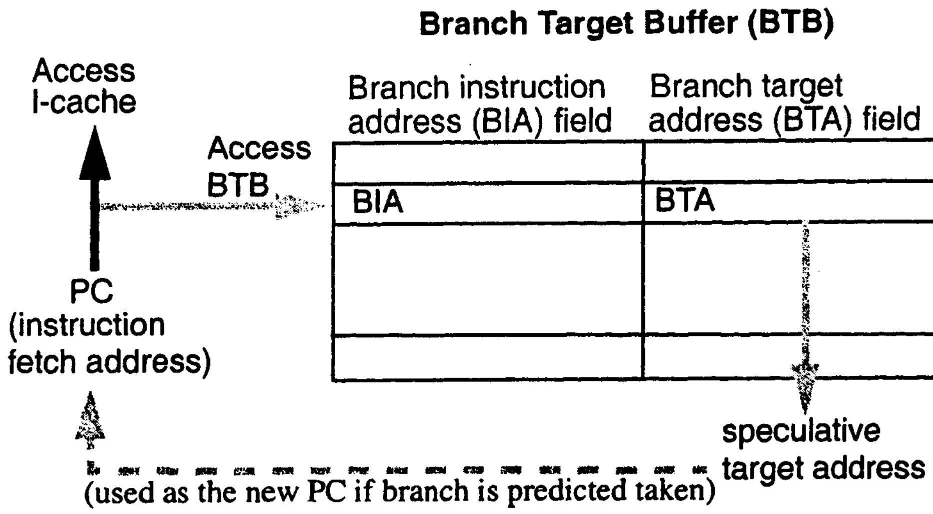
Maximizing the volume of the instruction flow path is equivalent to maximizing the sustained instruction fetch bandwidth. In order to do this the number of stall cycles in the fetch stage must be minimized. Recall that the total lost-opportunity cost is equal to the product of the number of penalty cycles and the width of a machine. For an n-wide machine each stalled cycle is equal to fetching n no-op instructions. The primary aim of instruction flow techniques is to minimize the number of such fetch stall cycles and/or to make use of these cycles to do potentially useful work. The current dominant approach to accomplishing this is via branch prediction which is the subject of the next subsubsection.

3.2.1.3 Branch Prediction Techniques

Experimental studies have shown that the behavior of branch instructions are highly predictable. A key approach to minimize branch penalty and maximize instruction flow throughput is to speculate on both branch target addresses and branch conditions of branch instructions. As a static branch instruction is repeatedly executed at run time, its dynamic behavior can be tracked. Based on its past behavior, its future behavior can be effectively predicted. Two fundamental components of branch prediction are *branch target speculation* and *branch condition speculation*. With any speculative technique, there must be mechanisms to validate the prediction and to safely recover from any mispredictions. Branch misprediction recovery will be covered in the next subsubsection.

Branch target speculation involves the use of a branch target buffer (BTB) to store previous branch target addresses. BTB is a small cache memory accessed during the instruction fetch stage using the instruction fetch address (PC). Each entry of the BTB contains two fields: branch instruction address (BIA) and branch target address (BTA). When a static branch instruction is executed for the first time, an entry in the BTB is allocated for it. Its instruction address is stored in the BIA field and its target address is stored in the BTA field. Assuming the BTB is a fully associative cache, the BIA field is used for the associative access of the BTB. The BTB is accessed concurrently with the accessing of the I-cache. When the current PC matches the BIA of an entry in the BTB, a hit in the BTB results. This implies that the current instruction being fetched from the I-cache has been executed before and is a branch instruction. When a hit in the BTB occurs, the BTA field of the hit entry is accessed and can be used as the next instruction fetch address if that particular branch instruction is predicted to be taken; see Figure 25.

FIGURE 25 Branch target speculation using a branch target buffer (BTB).


Branch Target Buffer (BTB)
Access I-cache
Access BTB
Branch instruction address (BIA) field
Branch target address (BTA) field
BIA
BTA
PC (instruction fetch address)
speculative target address
(used as the new PC if branch is predicted taken)

By accessing the BTB using the branch instruction address and retrieving the branch target address from the BTB all during the Fetch stage, the speculative branch target address will be ready to be used in the next machine cycle as the new instruction fetch address if the branch instruction is predicted to be taken. If the branch instruction is predicted to be taken and this prediction turned out to be correct, then the branch instruction is effectively executed in the Fetch stage incurring no branch penalty. The non-speculative execution of the branch instruction is still performed for purpose of validating the speculative execution. The branch instruction is still fetched from the I-cache and executed. The resultant target address and branch condition are compared with the speculative version. If they agree, then correct prediction was made, otherwise misprediction has occurred and recovery must be initiated. The result from the non-speculative execution is also used to update the content, i.e. BTA field, of the BTB.

There are a number of ways to do branch condition speculation. The simplest form is to design the hardware to be biased for not taken, i.e. always predict not taken. When a branch instruction is encountered, prior to its resolution, the Fetch stage continues fetching down the fall through path without stalling. This form of minimal branch prediction is easy to implement but not very effective. For example, many branches are used as loop closing instructions, which are mostly taken during execution except when exiting loops. Another form of prediction employs software support and can require ISA changes. For example, an extra bit can be allocated in the branch instruction format that is set by the compiler. This bit is used as a hint to the hardware to perform either predict not taken or predict taken depending the value of this bit. The compiler can use branch instruction type and profiling information to determine the most appropriate value for this bit. This allows each static branch instruction to have its own specified prediction. However, this prediction is static in the sense that the same prediction is used for all dynamic executions of the branch. Such static software prediction technique is used in the Motorola M88110 [keith]. A more aggressive and dynamic form of prediction makes prediction based on the branch target address offset. This form of prediction first determines the relative offset between the address of the branch instruction and the address of the target instruction. A positive offset will trigger the hardware to predict not taken; whereas a negative offset, most likely indicating a loop closing branch, will trigger the hardware to predict taken. This branch offset based technique is used in the original IBM RS/6000 design [IBMrs6k] and has been adopted by other machines as well. The most common branch condition speculation technique employed in contemporary superscalar machines is based on the history of previous branch executions.

History-based branch prediction makes prediction of the branch direction, whether taken (T) or not taken (N), based on previously observed branch directions. The assumption is that historical information on the direction that a static branch takes in previous executions can give helpful hints on the direction that it is likely to take in future executions. Design decisions for such type of branch prediction includes how much history should be tracked and for each observed history pattern what prediction should be made. The specific algorithm for history-based branch direction prediction can be characterized by a finite state machine (FSM); see Figure 26. The n state variables encode the directions taken by the last n executions of that branch. Hence each state represents a particular history pattern in terms of a sequence of takens and not takens. The output logic generates a prediction based on the current state of the FSM. Essentially, a prediction is made based on the outcome of the previous n executions of that branch. When a predicted branch is finally executed, the actual outcome is used as an input to the FSM to trigger a state transition. The next state logic is trivial; it simply involves chaining the state variables into a shift register, which records the branch directions of the previous n executions of that branch instruction.

FIGURE 26 FSM model for history-based branch direction predictors.

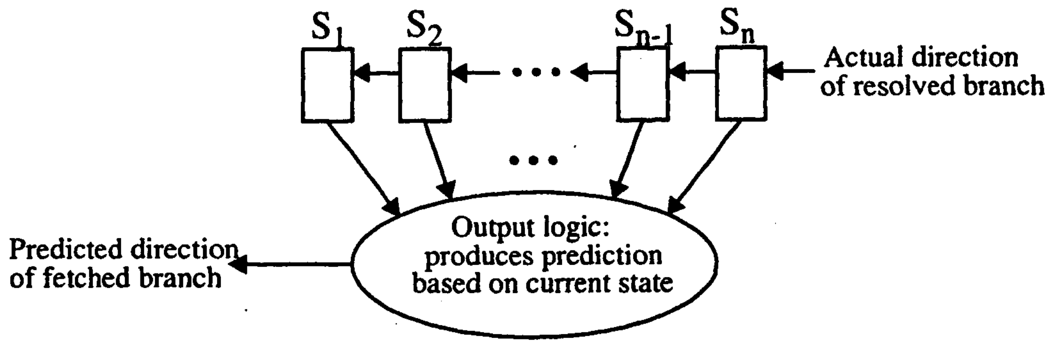
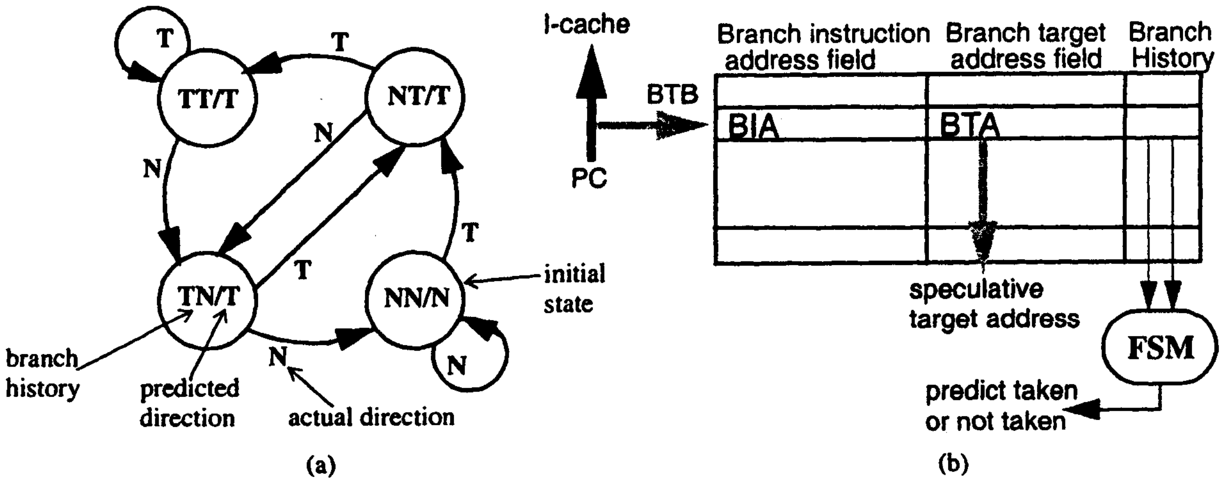

S_1
S_2
S_{n-1}
S_n
Actual direction of resolved branch
Output logic: produces prediction based on current state
Predicted direction of fetched branch

Figure 27 (a) illustrates the FSM diagram of a typical "2-bit" branch predictor that employs two history bits to track the outcome of two previous executions of the branch. The two history bits constitute the state variables of the FSM. The predictor can be in one of four states: NN, NT, TT, or TN representing the directions taken in the previous two executions of the branch. The NN state can be designated as the initial state. An output value of either T or N is associated with each of the four states representing the prediction that would be made when a predictor is in that state. When a branch is executed, the actual direction taken is used as an input to the FSM and state transition occurs to update the branch history which will be used to do the next prediction.

The particular algorithm implemented in the predictor of Figure 27 (a) is biased towards predicting branches to be taken; note that 3 of the 4 states predict the branch to be taken. It anticipates either long runs of N's (in the NN state) or long runs of T's (in the TT state). As long as at least one of the two previous executions was a taken branch, it will predict the next execution to be taken. The prediction will only be switched to not taken, when it has encountered two consecutive N's in a roll. This represents one particular branch-direction prediction algorithm; clearly there are many possible designs for such history-based predictors and many designs have been evaluated by researchers.

FIGURE 27 History based branch prediction: (a) a 2-bit branch predictor algorithm; (b) branch target buffer (BTB) with an additional field for storing branch history bits.



To support history-based branch-direction predictors, the BTB can be augmented to include a history field for each of its entries. The width, in number of bits, of this field is determined by the number of history bits being tracked. When a PC address hits in the BTB, in addition to the speculative target address, the history bits are also retrieved. These history bits are fed to the logic that implements the next-state and output functions of the branch predictor FSM. The retrieved history bits are used as the state variables of the FSM. Based on these history bits, the output logic produces the 1-bit output that indicates the predicted direction. If the prediction is a taken branch, then this output is used to steer the speculative target address to the PC to be used as the new instruction fetch address in the next machine cycle. If the prediction turns out to be correct, then effectively the branch instruction has been executed in the Fetch stage without incurring any penalty or stalled cycle.

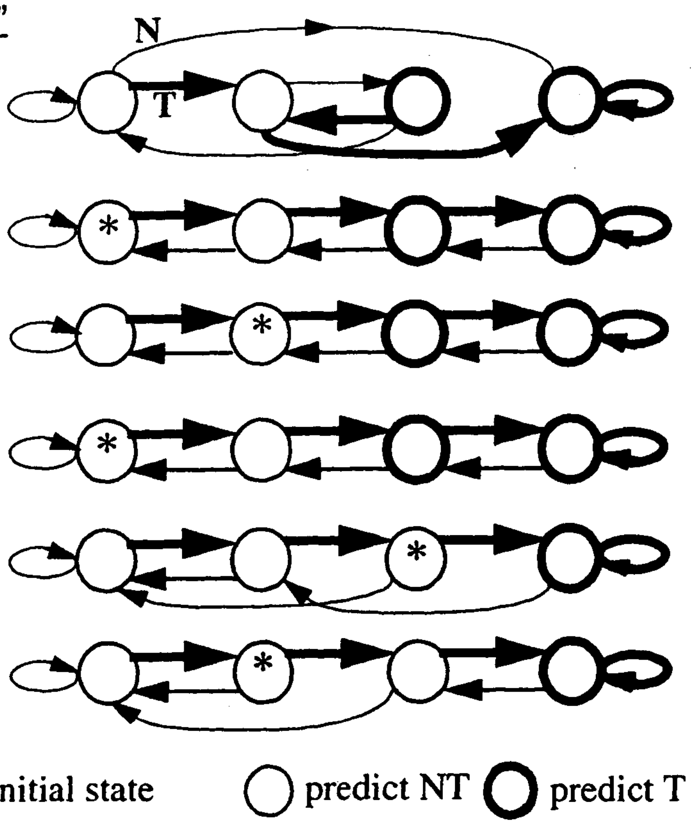
A classic experimental study on branch prediction was done by Lee and Smith [lee&smith84]. In this study, 26 programs from six different types of workloads for three different machines (IBM 370, DEC PDP-11, and CDC 6400) were used. Averaged across all the benchmarks, 67.6% of the branches were taken while 32.4% were not taken. Branches tend to be taken more than not taken by a ratio of two to one. With static branch prediction based on the op-code type, the prediction accuracy ranged from 55% to 80% for the six workloads. Using only one bit of history, history-based dynamic branch prediction achieved prediction accuracy ranging from 79.7% to 96.5%. With two history bits, the accuracy for the six workloads ranged from 83.4% to 97.5%. Continued increase of the number of history bits brought additional incremental accuracy. However beyond four history bits there is very minimal increase in the prediction accuracy. They implemented a 4-way set associative BTB that had 128 sets. The averaged BTB hit rate was 86.5%. Combining prediction accuracy with the BTB hit rate, the resultant average prediction effectiveness was approximately 80%.

A more recent experimental study was done at IBM by Nair using the RS/6000 architecture and SPEC benchmarks [Nair92]. This is a very comprehensive study of possible branch prediction algorithms. The goal for branch prediction is to overlap the execution of branch instructions with that of other instructions so as to achieve "zero cycle" branches or accomplish "branch folding," i.e. branches are folded out of the critical latency path of instruction execution. This study performed an exhaustive search for optimal 2-bit predictors. There are 2^{20} possible FSM's of 2-bit predictors. Nair determined that many of these machines are uninteresting and pruned the entire design space down to 5,248 machines. Extensive simulations are performed to determine the optimal (achieves the best prediction accuracy) 2-bit predictor for each of the benchmarks. The list of SPEC benchmarks, their best prediction accuracies, and the associated optimal predictors are shown in Figure 28.

In Figure 28, the states denoted with bold circles represent states in which the branch is predicted taken; the non-bold circles represent states that predict not taken. Similarly the bold edges represent state transitions when the branch is actually taken; the non-bold edges represent transitions corresponding to the branch actually not taken. The state denoted with "*" indicates the initial state. The prediction accuracy for the optimal predictors of these six benchmarks range from 87.1% to 97.2%. Notice that the optimal predictors for *doduc*, *gcc* and *espresso*, are identical (disregarding the different initial state of the *gcc* predictor) and exhibit the behavior of a 2-bit up/down saturating counter. We can label the four states from left to right as "0", "1", "2", and "3" representing the four count values of a 2-bit counter. Whenever a branch is resolved taken the count is incremented, and decremented otherwise. The two lower-count states predict a branch to be not taken while the two higher-count states predict a branch to be taken. Figure 28 also provides the prediction accuracies for the six benchmarks if the 2-bit saturating counter predictor is used for all six benchmarks. The prediction accuracies for *spice2g6*, *li*, and *eqntott* only decrease minimally from their optimal values, indicating that the 2-bit saturating counter is a good candidate for general use on all benchmarks. In fact, the 2-bit saturating counter has become a popular prediction algorithm in real and experimental designs.

FIGURE 28 Optimal 2-bit branch predictors for six SPEC benchmarks [Nair92].

Benchmark	Optimal	"Counter"
spice2g6	97.2	97.0
doduc	94.3	94.3
gcc	89.1	89.1
espresso	89.1	89.1
li	87.1	86.8
eqntott	87.9	87.2

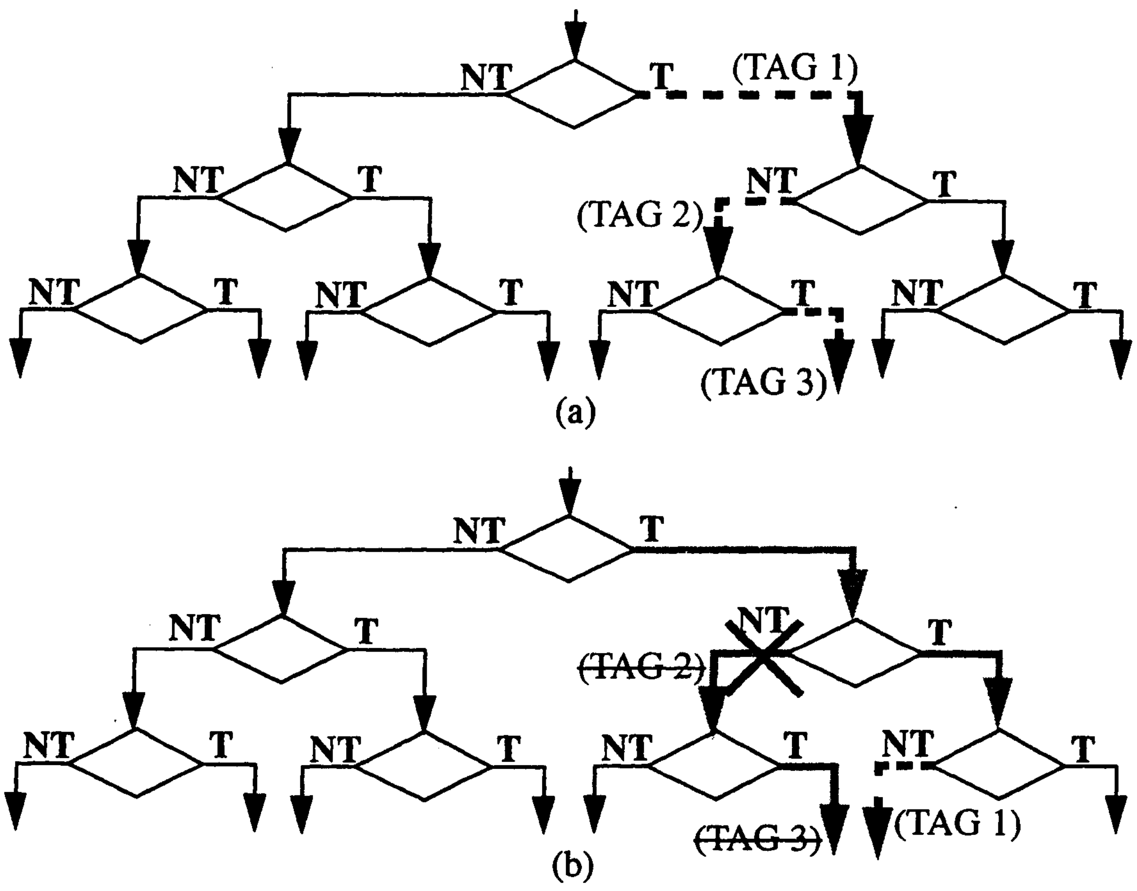


The same study also investigated the effectiveness of counter based predictors. With an 1-bit counter as the predictor, i.e. remembers the direction taken last time and predicts the same direction for the next time, the prediction accuracies ranged from 82.5% to 96.2%. As we have seen in Figure 28, a 2-bit counter yields accuracy range of 86.8% to 97.0%. If a 3-bit counter is used the increase in accuracy is minimal, it ranges from 88.3% to 97.0%. Based on this study, the 2-bit saturating counter appears to be a very good choice for a history-based predictor. Direct-mapped branch history tables are assumed in this study. While some programs, such as *gcc*, have over 7,000 conditional branches, for most programs, the branch penalty due to aliasing in finite-sized branch history tables levels out at about 1,024 entries for the table size.

3.2.1.4 Branch Misprediction Recovery

Branch prediction is a speculative technique. Any speculative technique requires mechanisms for validating the speculation. Dynamic branch prediction can be viewed as consisting of two interacting engines. The leading engine performs speculation in the front-end stages of the pipeline while a trailing engine performs validation in latter stages of the pipeline. In the case of misprediction the trailing engine also performs recovery. These two aspects of branch prediction are illustrated in Figure 29.

FIGURE 29 Two aspects of branch prediction: (a) branch speculation; (b) branch validation/recovery.

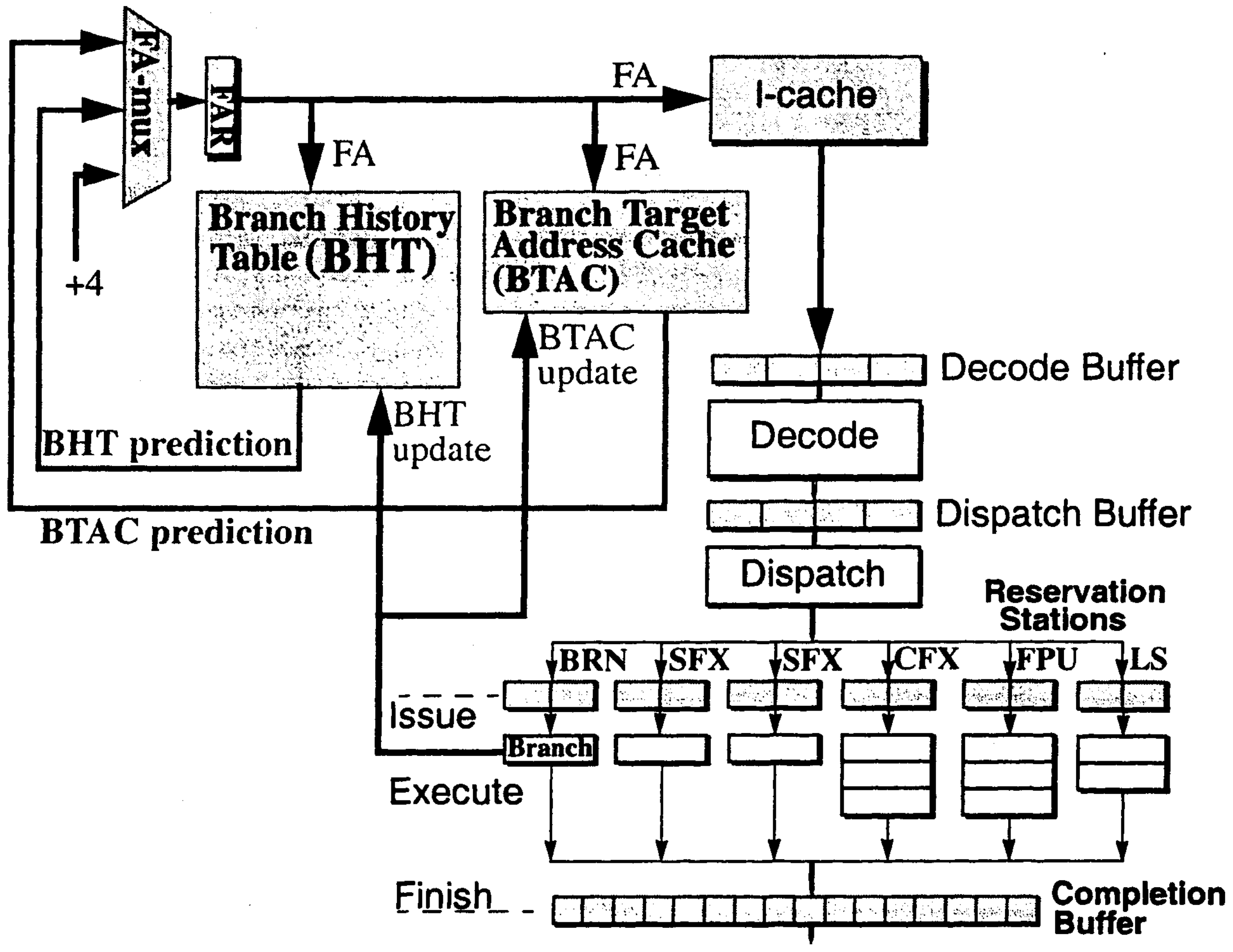


Branch speculation involves predicting the direction of a branch and then proceeding to fetch along the predicted path of control flow. While fetching from the predicted path, additional branch instructions may be encountered. Prediction of these additional branches can be similarly performed, potentially resulting in speculating passed multiple conditional branches before the first speculated branch is resolved. Figure 29 (a) illustrates speculating passed three branches with the first and the third branches being predicted taken and the second one predicted not taken. When this occurs, instructions from three speculative basic blocks are now resident in the machine and must be appropriately identified. Instructions from each speculative basic block are given the same identifying tag. In the example of Figure 29 (a), three distinct tags are used to identify the instructions from the three speculative basic blocks. A tagged instruction indicates that it is a speculative instruction, and the value of the tag identifies which basic block it belongs to. As a speculative instruction advances down the pipeline stages, the tag is also carried along. When speculating, the instruction addresses of all the speculated branch instructions (or the next sequential instructions) are buffered in the event that recovery is required.

Branch validation occurs when the branch is executed and the actual direction of a branch is resolved. The correctness of the earlier prediction can then be determined. If the prediction turns out to be correct, the speculation tag is deallocated and all the instructions associated with that tag become non-speculative and are allowed to complete. If a misprediction is detected, two actions are required, namely the incorrect path must be terminated and fetching from a new correct path must be initiated. To initiate a new path, the PC must be updated with a new instruction fetch address. If the incorrect prediction was a not-taken prediction, then the PC is updated with the computed branch target address. If the incorrect prediction was a taken prediction, then the PC is updated with the sequential (fall through) instruction address, which is obtained from the previously buffered instruction address when the branch was predicted taken. Once the PC has been updated, fetching of instructions resumes along the new path and branch prediction begins anew. To terminate the incorrect path, speculation tags are used. All the tags that are associated with the mispredicted branch are used to identify the instructions that must be eliminated. All such instructions that are still in the decode and dispatch buffers as well as those in reservation station entries are invalidated. Reorder buffer entries occupied by these instructions are deallocated. Figure 29 (b) illustrates this validation/recovery task when the second of the three predictions is incorrect. The first branch is correctly predicted and therefore instructions with Tag 1 become non-speculative and are allowed to complete. The second prediction is incorrect and all the instructions with Tag 2 and Tag 3 must be invalidated and their reorder buffer entries must be deallocated. After fetching down the correct path, branch prediction can begin once again and Tag 1 is used again to denote the instructions in the first speculative basic block. During branch validation, the associated BTB entry is also updated. <speculative updates?>

We now use the PowerPC 604 superscalar microprocessor to illustrate the implementation of dynamic branch prediction in a real superscalar processor. The 604 is a 4-wide superscalar capable of fetching, decoding and dispatching up to four instructions in every machine cycle. Instead of a single unified BTB, the 604 employs two separate buffers to support branch prediction, namely the Branch Target Address Cache (BTAC) and the Branch History Table (BHT); see Figure 30. The BTAC is a 64-entry fully-associative cache that stores the branch target addresses while the BHT, a 512-entry direct-mapped table, stores the history bits of branches. The reason for this separation will become clear shortly.

FIGURE 30 Branch prediction in the PowerPC 604 superscalar microprocessor.



Both the BTAC and the BHT are accessed during the Fetch stage using the current instruction fetch address in the PC. The BTAC responds in one cycle, however, the BHT requires two cycles to complete its access. If a hit occurs in the BTAC, indicating the presence of a branch instruction in the current fetch group, a predict taken occurs and the branch target address retrieved from the BTAC is used in the next fetch cycle. Since the 604 fetches four instructions in a fetch cycle, there can be multiple branches in the fetch group, hence, the BTAC entry indexed by the fetch address contains the branch target address of the first branch instruction in the fetch group that is predicted to be taken. In the second cycle, or during the Decode stage, the history bits retrieved from the BHT are used to generate a history-based prediction on the same branch. If this prediction agrees with the taken prediction made by the BTAC, the earlier prediction is allowed to stand. On the other hand, if the BHT prediction disagrees with the BTAC prediction, the BTAC prediction is annulled and fetching from the fall-through path, corresponding to predict not taken, is initiated. In essence, the BHT prediction can overrule the BTAC prediction. As expected, in most cases the two predictions agree. In some cases, the BHT corrects the wrong prediction made by the BTAC. It is possible, however, for the BHT to erroneously change the correct prediction of the BTAC; this occurs very infrequently. When a branch is resolved, the BHT is updated and based on its updated content the BHT in turns updates the BTAC by either leaving an entry in the BTAC if it is to be predicted taken the next time, or deleting an entry from the BTAC if that branch is to be predicted not taken.

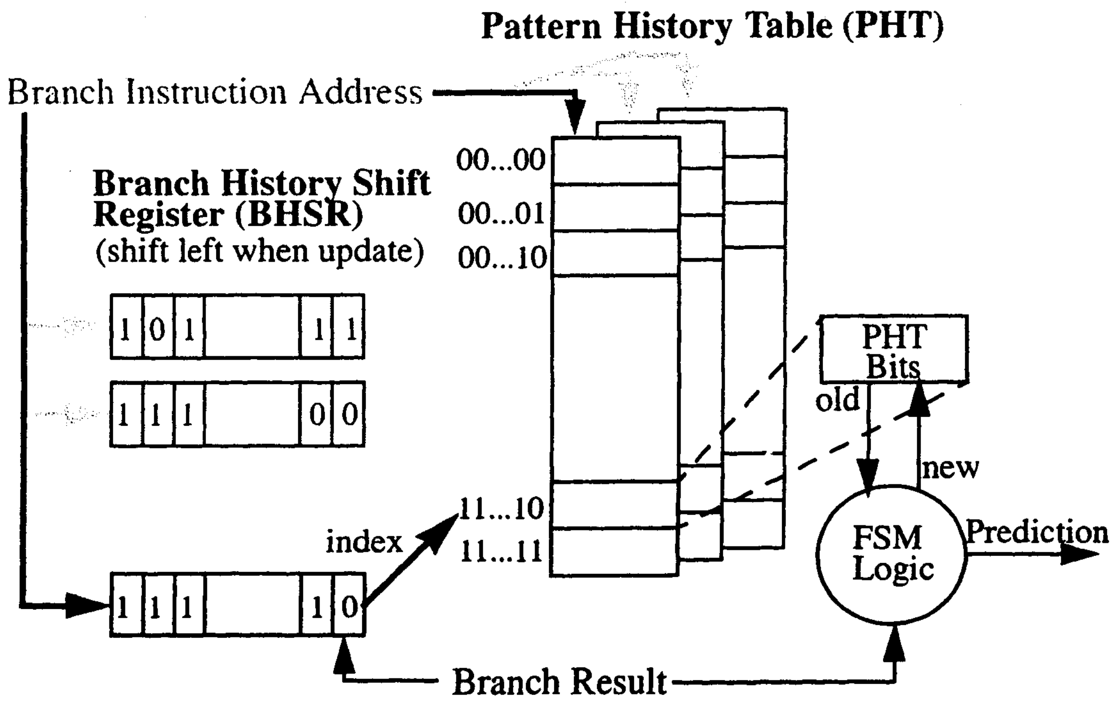
The 604 has four entries in the reservation station that feed the branch execution unit. Hence it can speculate passed up to four branches, i.e. there can be a maximum of four speculative branches present in the machine. To denote the four speculative basic blocks involved, a 2-bit tag is used to identify all speculative instructions. After a branch resolves, branch validation takes place and all speculative instructions are either made non-speculative or are invalidated via the use of the 2-bit tag. Reorder buffer entries occupied by mis-speculated instruction are deallocated. Again, this is performed using the 2-bit tag. <overall prediction accuracy of the BTAC and BHT and misprediction penalties???>

3.2.1.5 Advanced Branch Prediction Techniques

The dynamic branch prediction schemes discussed thus far have a number of limitations. Prediction for a branch is made based on the limited history of only that particular static branch instruction. The actual prediction algorithm does not take into account the dynamic context within which the branch is being executed. For example, it does not make use of any information on the particular control flow path taken in arriving at that branch. Furthermore the same fixed algorithm is used to make the prediction regardless of the dynamic context. It has been observed experimentally that the behavior of certain branches are strongly correlated with the behavior of other branches that precede them during execution. Consequently more accurate branch prediction can be achieved with algorithms that take into account the branch history of other correlated branches and that can adapt the prediction algorithm according to the dynamic branching context.

In 1992, Yeh and Patt proposed a two-level adaptive branch prediction technique [Yeh&Patt92] that can potentially achieve better than 95% prediction accuracy by having a highly flexible prediction algorithm that can adapt to changing dynamic contexts. In previous schemes, a single branch history table is used and indexed by the branch address. For each branch address there is only one relevant entry in the branch history table. In the two-level adaptive scheme, a set of history tables is used. These are identified as the Pattern History Table (PHT); see Figure 31. Each branch address indexes to a set of relevant entries; one of these entries is then selected based on the dynamic branching context. The context is determined by a specific pattern of recently executed branches stored in a Branch History Shift Register (BHSR); see Figure 31. The content of BHSR is used to indexed into the PHT to select one of the relevant entries. The content of this entry is then used as the state for the prediction algorithm FSM to produce a prediction. When a branch is resolved, the branch result is used to update both the BHSR and the selected entry in the PHT.

FIGURE 31 Two-level adaptive branch prediction [Yeh&Patt92].



The two-level adaptive branch prediction technique actually specifies a framework within which many possible designs can be implemented. There are two options to implementing the BHSR: *global* (G) or *individual* (P). The global implementation employs a single BHSR of k bits that tracks the branch directions of the last k dynamic branch instructions in program execution. These can involve any number (one to k) of static branch instructions. The individual (called "per-branch" in [Yeh&Patt92]) implementation employs a set of k-bit BHSR's as illustrated in Figure 31, one of which is selected based on the branch address. Essentially the global BHSR is shared by all static branches; whereas with individual BHSR's each BHSR is dedicated to each static branch or a subset of static branches if there is address aliasing when indexing into the set of BHSR's using the branch address. There are three options to implementing the PHT: *global* (g), *individual* (p), or *shared* (s). The global PHT uses a single table to support the prediction of all static branches. Alternatively, individual PHT's can be used in which each PHT is dedicated to each static branch (p) or a small subset of static branches (s) if there is address aliasing when indexing into the set of PHT's using the branch address. A third dimension to this design space involves the implementation of the actual prediction algorithm. When a history-based FSM is used to implement the prediction algorithm Yeh and Patt identified such schemes as *adaptive* (A).

All possible implementations of the two-level adaptive branch prediction can be classified based on these three dimensions of design parameters. A given implementation can then be denoted using a three-letter notation, e.g. GAs represents a design that employs a single global BHSR, an adaptive prediction algorithm, and a set of PHT's with each being shared by a number of static branches. Yeh and Patt presented three specific implementations that are able to achieve prediction accuracy of 97% for their given set of benchmarks. These are listed below:

- GAg: (1) BHSR of size: 18 bits; (1) PHT of size: 2^{18} x 2 bits
- PAg: (512 x 4) BHSR's of size: 12 bits; (1) PHT of size: 2^{12} x 2 bits
- PAs: (512 x 4) BHSR's of size: 6 bits; (512) PHT's of size: 2^6 x 2 bits

All three implementations use an adaptive (A) predictor that is a 2-bit FSM. The first implementation employs a global BHSR (G) of 18 bits and a global PHT (g) with 2^{18} entries indexed by the BHSR bits. The second implementation employs 512 sets (4-way set associative) of 12-bit BHSR's (P) and a global PHT (g) with 2^{12} entries. The third implementation also employs 512 sets of 4-way set associative BHSR's (P), but each being only 6-bit wide. It also uses 512 PHT's (s), each having 2^6 entries indexed by the BHSR bits. Both the 512 sets of BHSR's and the 512 PHT's are indexed using 9 bits of the branch address. Additional branch address bits are used for the set-associative access of the BHSR's. The 512 PHT's are direct mapped and there can be aliasing, i.e. multiple branch addresses sharing the same PHT. From experimental data, such aliasing had minimal impact on degrading the prediction accuracy. Achieving great than 95% prediction accuracy by the two-level adaptive branch prediction schemes is quite impressive; the best traditional prediction techniques can only achieve about 90% prediction accuracy. The two-level adaptive branch prediction approach has been adopted by a number of real designs, including the Intel Pentium Pro and the AMD/NexGen Nx686 [???, ???].

Following the original Yeh and Patt proposal, other recent studies have gained further insights into two-level adaptive, or more recently called *correlated*, branch predictors [McFarling93, Young et al.95, Gloy et al.96]. Figure 32 illustrates a correlated branch predictor with a global BHSR (G) and a shared PHT (s). The 2-bit saturating counter is used as the predictor FSM. The global BHSR tracks the directions of the last k dynamic branches and captures the dynamic control-flow context. The PHT can be viewed as a single table containing a two-dimensional array, with 2^j columns and 2^k rows, of 2-bit predictors. If the branch address has n bits, a subset of j bits are used to index into the PHT to select one of the 2^j columns. Since j is less than n, some aliasing can occur where two different branch addresses can index into the same column of the PHT. Hence the designation of "shared" PHT. The k bits from the BHSR are used to select one of the 2^k entries in the selected column. The 2 history bits in the selected entry is used to make a history-based prediction. The traditional branch history table (BHT) is equivalent to having only one row of the PHT that is indexed only by the j bits of the branch address, as illustrated in Figure 32 by the dashed rectangular block of 2-bit predictors in the first row of the PHT.

FIGURE 32 Correlated branch predictor with global BHSR and shared PHT's (GAs) [Gloy et al. 96].

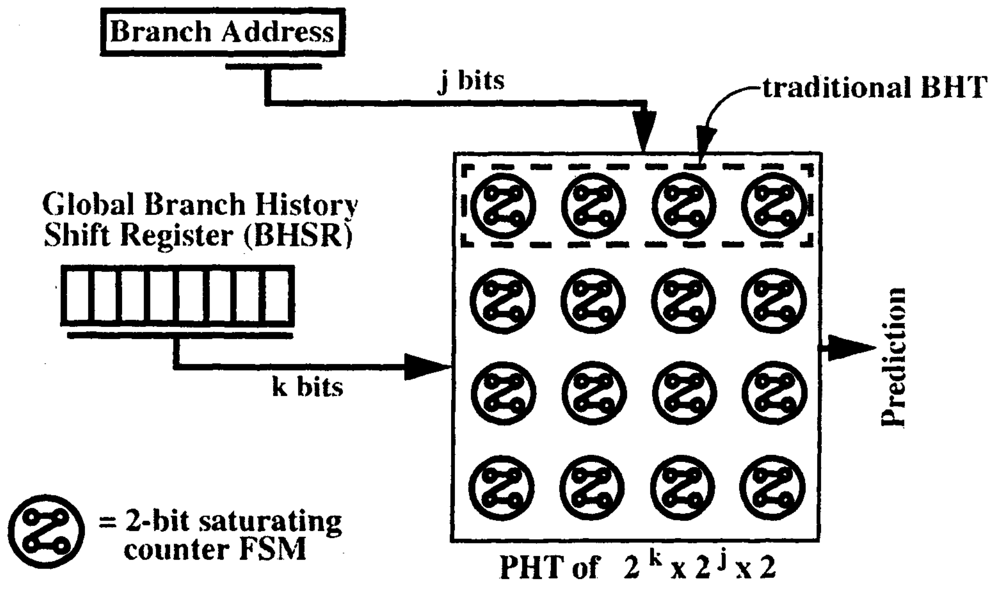

Branch Address
j bits
traditional BHT
Global Branch History Shift Register (BHSR)
k bits
Prediction
= 2-bit saturating counter FSM
PHT of $2^k \times 2^j \times 2$

FIGURE 33 Correlated branch predictor with individual BHSR's and shared PHT's (PAs) [Gloy et al. 96].

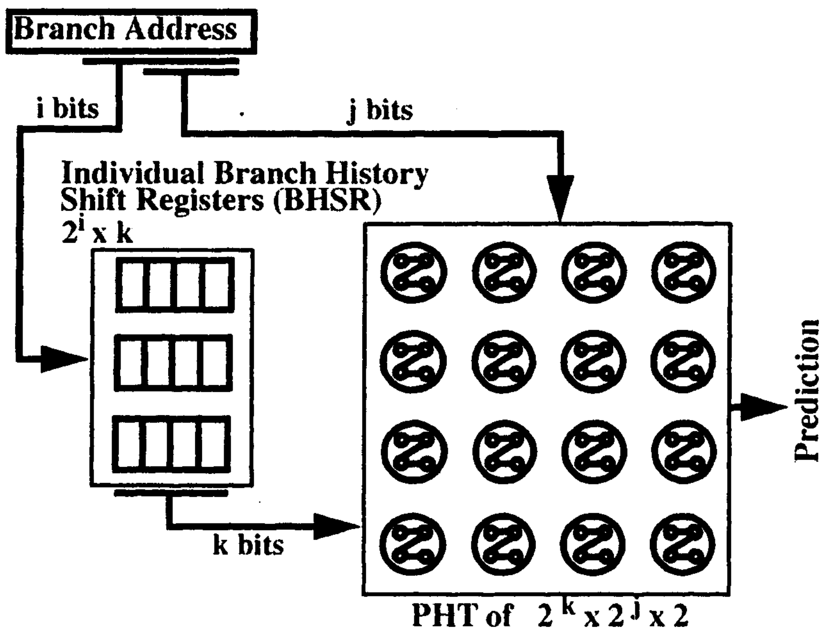
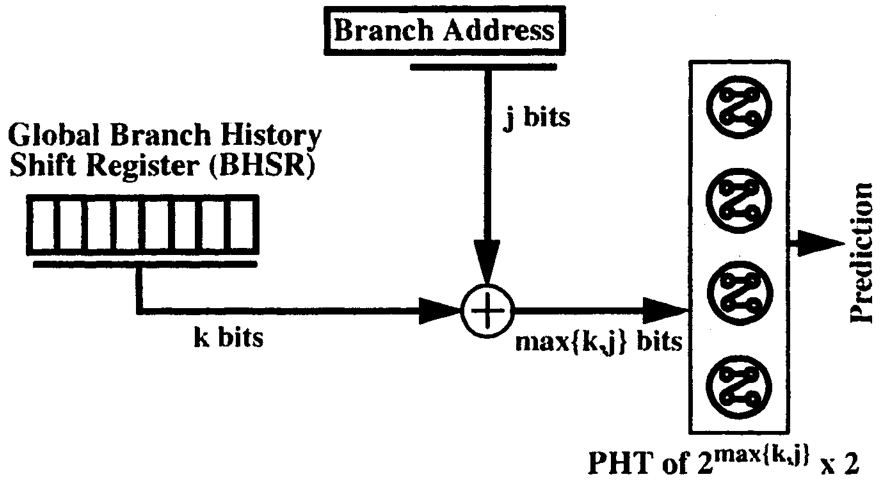

Branch Address
i bits
j bits
Individual Branch History Shift Registers (BHSR)
$2^i \times k$
k bits
Prediction
PHT of $2^k \times 2^j \times 2$

Figure 33 illustrates a correlated branch predictor with individual, or "per-branch," BHSR's (P) and the same shared PHT (s). Similar to the GAs scheme, the PAs scheme also uses j bits of the branch address to select one of the 2^j columns of the PHT. However, i bits of the branch address, which can overlap with the j bits used to access the PHT, are used to index into a set of BHSR's. Depending on the branch address, one of the 2^i BHSR's is selected. Hence each BHSR is associated with one particular branch address, or a set of branch addresses if there is aliasing. Essentially, instead of using a single BHSR to provide the dynamic control-flow context for all static branches, multiple BHSR's are used to provide distinct dynamic control-flow contexts for different subsets of static branches. This adds additional flexibility in tracking and exploiting correlations between different branch instructions. Each BHSR tracks the directions of the last k dynamic branches belonging to the same subset of static branches. Both the GAs and the PAs schemes require a PHT of size $2^k \times 2^j \times 2$ bits. The GAs scheme has only one k-bit BHSR whereas the PAs scheme requires 2^i k-bit BHSR's.

A fairly efficient correlated branch predictor called *gshare* has been proposed by McFarling [McFarling93]. In this scheme, j bits from the branch address are "hashed" (via bit-wise XOR function) with the k bits from a global BHSR; see Figure 34. The resultant max{k,j} bits are used to index into a PHT of size $2^{\max\{k,j\}} \times 2$ bits to select one of the $2^{\max\{k,j\}}$ 2-bit branch predictors. The *gshare* scheme requires only one k-bit BHSR and a much smaller PHT, and yet achieves comparable prediction accuracy as other correlated branch predictors. This scheme is used in the DEC Alpha 21264 4-way superscalar microprocessor [???].

FIGURE 34 The *gshare* correlated branch predictor [McFarling93].



3.2.1.6 Other Instruction Flow Techniques

The primary objective for instruction-flow techniques is to supply as many useful instructions as possible to the execution core of the processor. The two major challenges deal with conditional branches and taken branches. For a wide superscalar processor, to provide adequate conditional branch throughput, the processor must very accurately predict the outcomes and targets of multiple conditional branches in every machine cycle. For example, in a fetch group of four instructions, it is possible that all four instructions are conditional branches. Ideally one would like to use the addresses of all four instructions to index into a 4-ported BTB to retrieve the history bits and target addresses of all four branches. A complex predictor can then make an overall prediction based on all the history bits. Speculative fetching can then proceed based on this prediction. Techniques for predicting multiple branches in every cycle have recently been proposed [conte95, rotenberg96]. It is also important to ensure high accuracy in such predictions. Global branch history can be used in conjunction with per-branch history to achieve very accurate predictions [superflow97]. For those branches or sequences of branches that do not exhibit strongly biased branching behavior and therefore are not predictable, *dynamic eager execution* (DEE) has been proposed. DEE [uht95] employs multiple PC's to simultaneously fetch from multiple addresses. Essentially the fetch stage pursues down multiple control flow paths until some branches are resolved, at which time, some of the wrong paths are dynamically pruned by invalidating the instructions on those paths.

Taken branches are the second major obstacle to supplying enough useful instructions to the execution core. In a wide machine the fetch unit must be able to correctly process more than one taken branch per cycle, which involves predicting each branch's direction and target, and fetching, aligning, and merging instructions from multiple branch targets. A promising approach in alleviating this problem called the *trace cache* [rotenberg96] has recently been proposed. Trace cache is a history-based fetch mechanism that stores dynamic instruction traces in a cache indexed by the fetch address and branch outcomes. These traces are assembled dynamically based on the dynamic branching behavior and can contain multiple non-consecutive basic blocks. Whenever the fetch address hits in the trace cache, instructions are fetched from the trace cache rather than the instruction cache. Since a dynamic sequence of instructions in the trace cache can contain multiple taken branches but is stored sequentially, there is no need to fetch from multiple targets, and no need for a multiported instruction cache or complex merging and aligning logic in the fetch stage. The trace cache can be viewed as doing dynamic basic block reordering according to the dominant execution paths taken by a program. The merging and aligning is done at completion time when non-consecutive basic blocks on a dominant path are first executed to assemble a trace, which is then stored in one line of the trace cache. The goal is that once the trace cache is "warmed up" most of the fetching will come from the trace cache instead of the instruction cache. Since the reordered basic blocks in the trace cache better match the dynamic execution order, there will be fewer fetches from non-consecutive locations in the trace cache, and there will be an increase in the overall throughput of taken branches.

3.2.2 Register Data Flow Techniques

Register data-flow techniques concern the effective execution of ALU (or register-register) type instructions in the execution core of the processor. It can be viewed that ALU instructions perform the "real" work specified by the program, with control-flow and load-store instructions playing the supportive roles of providing the necessary instructions and the required data, respectively. In the most ideal machine, branch and load/store instructions, being "overhead" instructions, should take no time to execute and the computation latency should be strictly determined by the processing of ALU instructions. The effective processing of these instructions is foundational to achieving high performance.

Assuming a load-store architecture, ALU instructions specify operations to be performed on source operands stored in registers. Typically an ALU instruction specifies a binary operation, two source registers where operands are to be retrieved and a destination register where the result is to be placed. $R_i \leftarrow F_n(R_j,R_k)$ specifies a typical ALU instruction, the execution of which requires the availability of: 1) F_n, the functional unit; 2) R_j and R_k, the two source operand registers; and 3) R_i, the destination register. If the functional unit F_n is not available, then a structural dependence exists that can result in a structural hazard. If one or both of the source operands in R_j and R_k is not available then a hazard due to true data dependence can occur. If the destination register R_i is not available then a hazard due to anti and output dependences can occur.

3.2.2.1 Register Reuse and False Data Dependences

The occurrence of anti and output dependences, or false data dependences, is due to the reuse of registers. If registers are never reused to store operands, then such false data dependences will not occur. The reuse of registers is commonly referred to as *register recycling*. Register recycling occurs in two different forms, one static and one dynamic. The static form is due to optimization performed by the compiler and is presented first. In a typical compiler, towards the back-end of the compilation process two tasks are performed: *code generation* and *register allocation*. Code generation task is responsible for the actual emitting of machine instructions. Typically the code generator assumes the availability of an unlimited number of symbolic registers in which it stores all the temporary data. Each symbolic register is used to store one value and is only written once, producing what is commonly referred to as "single-assignment" code [ ]. However an ISA has a limited number of architected registers and hence the register allocation tool is used to map the unlimited number of symbolic registers to the limited and fixed number of architected registers. The register allocator attempts to keep as many of the temporary values in registers as possible to avoid having to move the data out to memory locations and reloading them later on. It accomplishes this by reusing registers. A register is written with a new value when the old value stored there is no longer needed; effectively each register is recycled to hold multiple values.

Writing of a register is referred to as the *definition* of a register and the reading of a register as the *use* of a register. After each definition there can be one or more uses of that definition. The duration between the definition and the last use of a value is referred to as the *live range* of that value. After the last use of a live range, that register can be assigned to store another value and begin another live range. Register allocation procedures attempt to map non-overlapping live ranges into the same architected register and maximize register reuse. In single-assignment code there is a one-to-one correspondence between symbolic registers and values. After register allocation each architected register can receive multiple assignments and the register becomes a variable that can take on multiple values. Consequently the one-to-one correspondence between registers and values is lost.

If the instructions are executed sequentially and a redefinition is never allowed to precede the previous definition or the last use of the previous definition, then the live ranges that share the same register will never overlap during execution and the recycling of registers does not induce any problem. Effectively, the one-to-one correspondence between values and registers can be maintained implicitly if all the instructions are processed in the original program order. However, in a superscalar machine, especially with out-of-order processing of instructions, registers reading and writing operations can occur in an order different from the program order. Consequently the one-to-one correspondence between values and registers can potentially be perturbed; in order to ensure semantic correctness all anti and output dependences must be detected and enforced. Out-of-order reading (writing) of registers can be permitted as long as all the anti (output) dependences are enforced.

The dynamic form of register recycling occurs when a loop of instructions is repeatedly executed. With an aggressive superscalar machine capable of supporting many instructions in flight and a relatively small loop body being executed, multiple iterations of the loop can be simultaneously in flight in a machine. Hence, multiple copies of a register defining instruction from the multiple iterations can be simultaneously present in the machine, inducing the dynamic form of register recycling. Consequently anti and output dependences can be induced among these dynamic instructions from the multiple iterations of a loop, and must be detected and enforced to ensure semantic correctness of program execution.

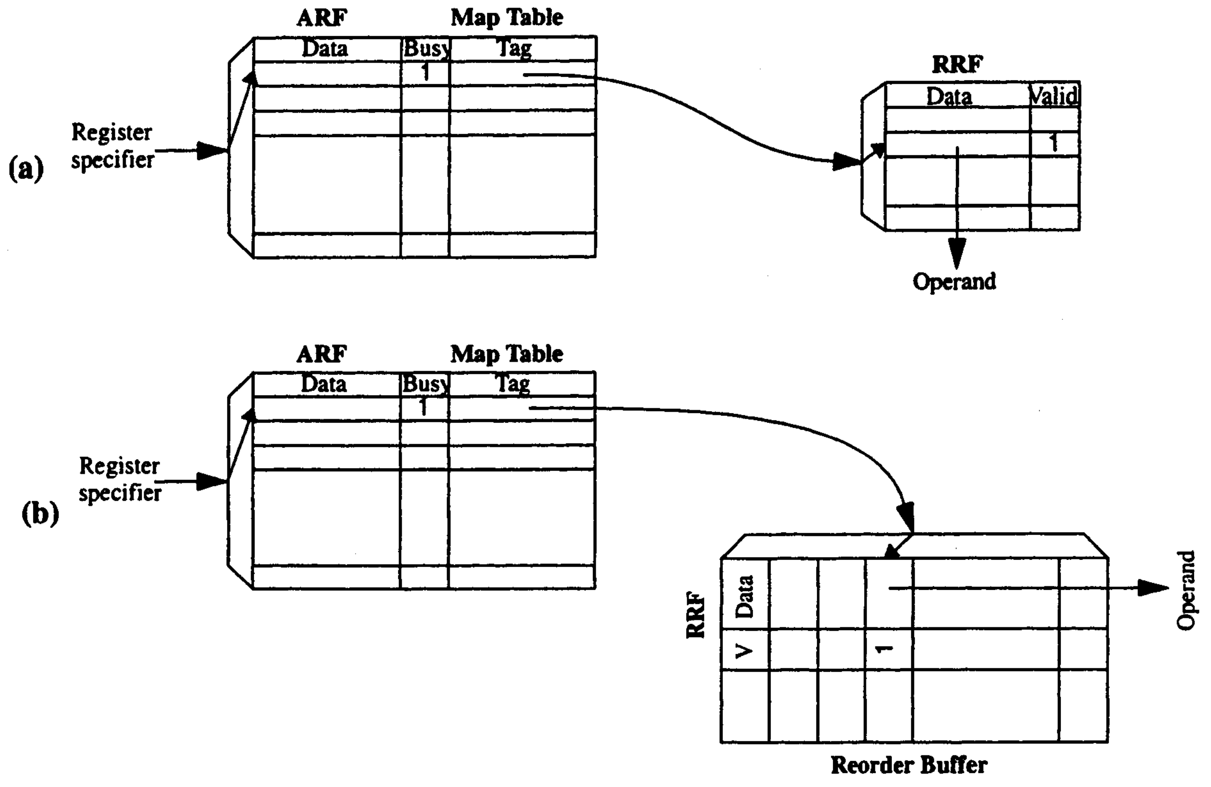
One way to enforce anti and output dependences is to simply stall the dependent instruction until the leading instruction has finished accessing the dependent register. If an anti (WAR) dependence exists between a pair of instructions, the trailing instruction (register updating instruction) must be stalled until the leading instruction has read the dependent register. If an output (WAW) dependence exists between a pair of instructions, the trailing instruction (register updating instruction) must be stalled until the leading instruction has first updated the register. Such stalling of anti and output dependent instructions can lead to significant performance lost and is not necessary. Recall that such false data dependences are induced by the recycling of the architected registers and are not intrinsic to the program semantics.

3.2.2.2 Register Renaming Techniques

A more aggressive way to deal with false data dependences is to dynamically assign different "names" to the multiple definitions of an architected register, and as a result eliminate the presence of such false dependences. This is called *register renaming* and requires the use of hardware mechanism at run-time to undo the effects of register recycling by reproducing the one-to-one correspondence between registers and values for all the instructions that might be simultaneously in flight. By performing register renaming, single-assignment is effectively recovered for the instructions that are in flight, and no anti and output dependences can exist among these instructions. This will allow the instructions that originally had false dependences between them to be executed in parallel.

A common way to implement register renaming is to use a separate rename register file (RRF) in addition to the architected register file (ARF). A straight forward way to implement the RRF is to simply duplicate the ARF and use the RRF as a shadow version of the ARF. This will allow each architected register to be renamed once. However this is not a very efficient way to use the registers in the RRF. Most current designs implement an RRF with fewer entries (such as eight instead of 32) than the ARF and allow each of the registers in the RRF to be flexibly used to rename any one of the architected registers. This facilitates the efficient use of the rename registers, but does require a mapping table to store the pointers to the entries in the RRF. The use of a separate RRF in conjunction with a mapping table to perform renaming of the ARF is illustrated in Figure 35.

FIGURE 35 Rename register file (RRF) implementations: (a) stand alone; (b) attached to the reorder buffer.

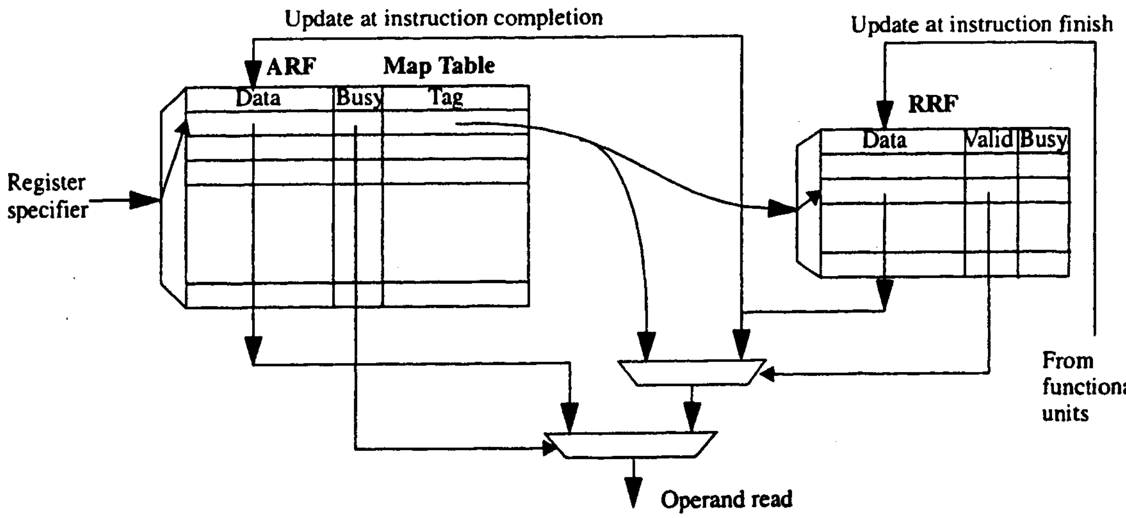


When a separate RRF is used for register renaming, there are implementation choices in terms of where to place the RRF. One option is to implement a separate stand-alone structure similar to the ARF and perhaps adjacent to the ARF. This is shown in Figure 35a. An alternative is to incorporate the RRF as part of the reorder buffer, as shown in Figure 35b. In both options a busy field is added to the ARF along with a mapping table. If the busy bit of a selected entry of the ARF is set, indicating the architected register has been renamed, the corresponding entry of the map table is accessed to obtain the tag or the pointer to an RRF entry. In the former option, the tag specifies a rename register and is used to index into the RRF; whereas in the latter option, the tag specifies a reorder buffer entry and is used to index into the reorder buffer.

Based on the diagrams in Figure 35, the difference between the two options may seem artificial; however, there are important subtle differences. If the RRF is incorporated as part of the reorder buffer, every entry of the reorder buffer contains an additional field that functions as a rename register and hence there is a rename register allocated for every instruction in flight. This is a design based on worst case scenario and may be wasteful since not every instruction defines a register. For example branch instructions do not update any architected register. On the other hand, a reorder buffer already contains ports to received data from the functional units and to update the ARF at instruction completion time. When a separate stand-alone RRF is used, it introduces an additional structure that requires ports for receiving data from the functional units and for updating the ARF. The choice of which of the two options to implement involves design trade-offs, and both options have been employed in real designs. We now focus on the stand-alone option to get a better feel of how register renaming actually works.

Register renaming involves three tasks: 1) *source read*; 2) *destination allocate*; and 3) *register update*. The first task of source read typically occurs during the decode (or possibly dispatch) stage and is for the purpose of fetching the register operands. When an instruction is decoded, its source register specifiers are used to index into a multi-ported ARF in order to fetch the register operands. Three possibilities can occur for each register operand fetch. First, if the busy bit is not set, indicating there is no pending write to the specified register and that the architected register contains the specified operand, the operand is fetched from the ARF. If the busy bit is set, indicating there is a pending write to that register and that the content of the architected register is stale, the corresponding entry of the map table is accessed to retrieve the rename tag. This rename tag specifies a rename register and is used to index into the RRF. Two possibilities can occur when indexing into the RRF. If the valid bit of the indexed entry is set, it indicates that the register-updating instruction has already finished execution although it is still waiting to be completed. In this case, the source operand is available in the rename register and is retrieved from the indexed RRF entry. If the valid bit is not set, it indicates that the register-updating instruction still has not been executed and that the rename register has a pending update. In this case the tag, or the rename register specifier, from the map table is forwarded to the reservation station instead of the source operand. This tag will be used later by the reservation station to obtain the operand when it becomes available. These three possibilities for source read are shown in Figure 36.

FIGURE 36 Register renaming tasks: source read, destination allocate, and register update.



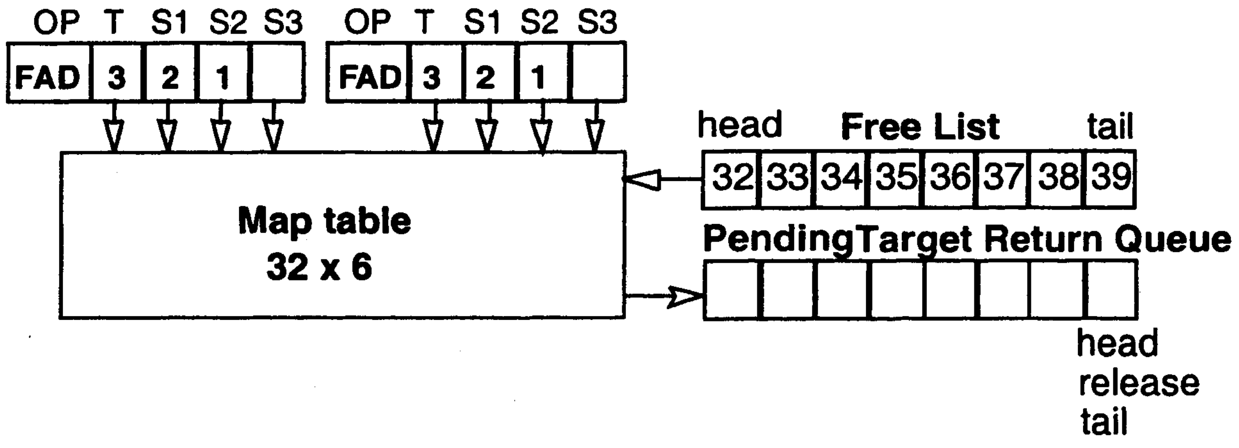
The task of destination allocate also occurs during the decode (or possibly dispatch) stage and has three subtasks, namely *set busy bit*, *assign tag*, and *update map table*. When an instruction is decoded, its destination register specifier is used to index into the ARF. The selected architected register now has a pending write and its busy bit must be set. The specified destination register must be mapped to a rename register. A particular unused (indicated by the busy bit) rename register must be selected. The busy bit of the selected RRF entry must be set, and the index of the selected RRF entry is used as a tag. This tag must then be written into the corresponding entry in the map table, to be used by subsequent dependent instructions for fetching their source operands.

While the task of register update takes place in the backend of the machine and is not part of the actual renaming activity of the decode/dispatch stage, it does have direct impact on the operation of the RRF. Register update can occur in two separate steps; see Figure 36. When a register-updating instruction finishes execution, its result is written into the entry of the RRF indicated by the tag. Later on when this instruction is completed, its result is then copied from the RRF into the ARF. Hence, register update involves first updating an entry in the RRF and then an entry in the ARF. These two steps can occur in back to back cycles if the register-updating instruction is at the head of the reorder buffer, or can be separated by many cycles if there are other unfinished instructions in the reorder buffer ahead of this instruction. Once a rename register is copied to its corresponding architected register, its busy bit is reset and it can be used again to rename another architected register.

So far we have assumed that register renaming implementation requires the use of two separate physical register files, namely the ARF and the RRF. This is not necessary. The architected registers and the rename registers can be pooled together and implemented as a single physical register file with its number of entries equal to the sum of the ARF and RRF entry counts. Such a pooled register file does not rigidly designate some of the registers as architected registers and others as rename registers. Each physical register can be flexibly assigned to be an architected register or a rename register. Unlike a separate ARF and RRF implementation which must physically copy a result from the RRF to the ARF at instruction completion, the pooled register file only needs to change the designation of a register from being a rename register to an architected register. This will save the data transfer interconnect between the RRF and the ARF. The key disadvantage of the pooled register file is its hardware complexity. A secondary disadvantage is that at context swap time, when the machine state must be saved, the subset of registers constituting the architected state of the machine must be explicitly identified first before state saving can begin.

The pooled register file approach is used in the floating-point unit of the original IBM RS/6000 design and is illustrated in Figure 37. In this design, 40 physical registers are implemented for supporting an ISA that specifies 32 architected registers. A mapping table is implemented, based on whose content any subset of 32 of the 40 physical registers can be designated as the architected registers. The mapping table contains 32 entries indexed by the 5-bit architected register specifier. Each entry when indexed returns a 6-bit specifier indicating the physical register to which the architected register has been mapped.

FIGURE 37 Floating-point unit (FPU) register renaming in the IBM RS/6000 [ ].



The FPU of the RS/6000 is a pipelined functional unit with the Rename pipe stage preceding the Decode pipe stage. The Rename pipe stage contains the map table, two circular queues, and the associated control logic. The first queue is called the Free List (FL) and contain physical registers that are available for new renaming. The second queue is called the Pending Target Return Queue (PTRQ) and contains those physical registers that have been used to rename architected registers that have been subsequently re-renamed in the map table. Physical registers in the PTRQ can be returned to the FL once last use of that register has occurred. Two instructions can traverse the Rename stage in every machine cycle. Due to the possibility of FMA (Fused Multiply-Add) instructions that have three sources and one destination, each of the two instructions can contain up to four register specifiers. Hence, the map table must be 8-ported to support the simultaneous translation of the eight architected register specifiers. The map table is initialized with the identity mapping, i.e. architected register *i* is mapped to physical register *i* for *i*=0, 1, ..., 31. At initialization, physical registers 32-39 are placed in the FL and the PTRQ is empty.

When an instruction traverses the Rename stage, its architected register specifiers are used to index into the map table to obtain their translated physical register specifiers. The 8-ported map table has 32 entries, indexed by the 5-bit architected register specifier, with each entry containing 6 bits indicating the physical register to which the architected register is mapped. The content of the map table represents the latest mapping of architected registers to physical registers and specifies the subset of physical registers that currently represents the architected registers.

In the FPU of the RS/6000, only load instructions can trigger a new renaming. Such register renaming prevents the FPU from stalling while waiting for loads to execute in order to enforce anti and output dependences. When a load instruction traverses the Rename stage, its destination register specifier is used to index into the map table. The current content of that entry of the map table is push out to the PTRQ and the next physical register in the FL is loaded into the map table. This effectively renames the redefinition of that destination register to a different physical register. All subsequent instructions that specifies this architected register as a source operand will received the new physical register specifier as the source register. Beyond the Rename stage, i.e. in the Decode and Execute stages, the FPU uses only physical register specifiers, and all true register dependences are enforced using the physical register specifiers.

The map table approach represents the most aggressive and versatile implementation of register renaming. Every physical register can be used to represent any redefinition of any architected register. There is significant hardware complexity required to implement the multi-ported map table and the logic to control the two circular queues. The return of a register in PTRQ to FL is especially troublesome due to the difficulty in identifying the last-use instruction of a register. However, unlike approaches based on the use of rename buffers, at instruction completion time no copying of the content of the rename buffers to the architected registers is necessary. On the other hand, when interrupts occur and as part of context swap the subset of physical registers that constitute the current architected machine state must be explicitly determined based on the map table contents.

Most contemporary superscalar microprocessors implement some form of register renaming to avoid having to stall for anti and output register data dependences induced by the reuse of registers. Typically register renaming occurs during the instruction decoding time and its implementation can become quite complex, especially for wide superscalar machines in which many register specifiers for multiple instructions must be simultaneously renamed. It's possible that multiple redefinitions of a register can occur within a fetch group. Implementing register renaming mechanism for wide superscalars without seriously impacting machine cycle time is a real challenge. To achieve high performance the serialization constraints imposed by false register data dependences must be eliminated; hence dynamic register renaming is absolutely essential.

3.2.2.3 True Data Dependences and the Data-Flow Limit

A read-after-write (RAW) dependence between two instructions is called a true data dependence due to the producer-consumer relationship between these two instructions. The trailing consumer instruction cannot obtain its source operand until the leading producer instruction produces its result. A true data dependence imposes a serialization constraint between the two dependent instructions; the leading instruction must finish execution before the trailing instruction can begin execution. Such true data dependences result from the semantics of the program and are usually represented by a data-flow graph (DFG) or data-dependence graph (DDG).

FIGURE 38 FFT code fragment: (a) original source statements; (b) compiled assembly instructions.

```
w[i+k].ip = z[i].rp + z[m+i].rp;
w[i+j].rp = c[k+1].rp* (z[i].rp -z[m+i].rp) - e[k+1].ip *(z[i].ip - z[m+i].ip);
```

(a)

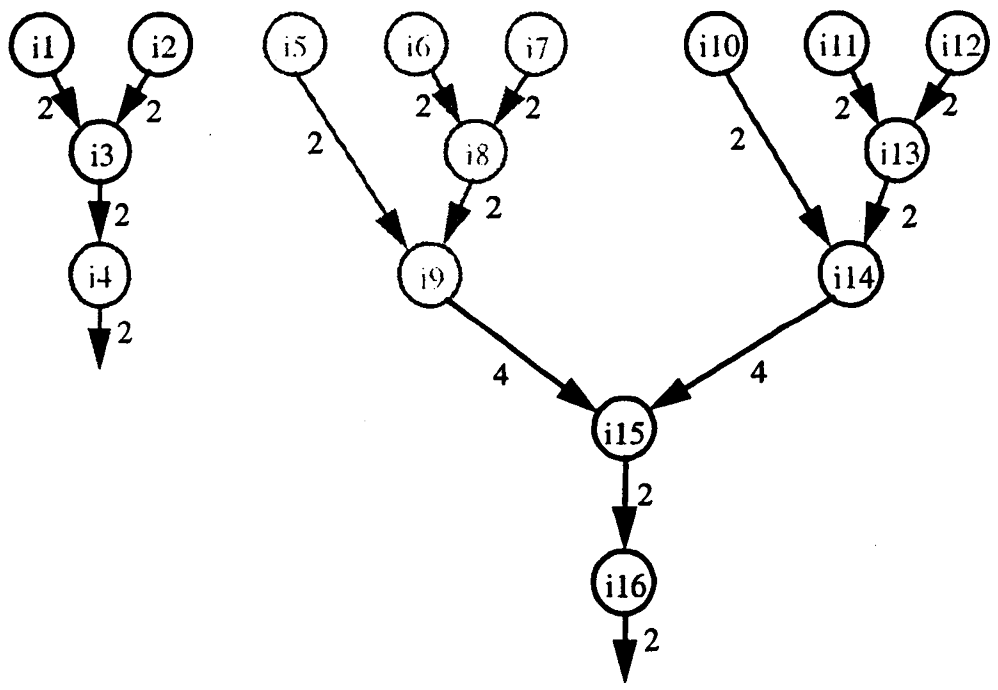
```
i1: l.s f2, 4(r2)
i2: l.s f0, 4(r5)
i3: fadd.s f0, f2, f0
i4: s.s f0, 4(r6)
i5: l.s f14, 8(r7)
i6: l.s f6, 0(r2)
i7: l.s f5, 0(r3)
i8: fsub.s f5, f6, f5
i9: fmul.s f4, f14, f5
i10: l.s f15, 12(r7)
i11: l.s f7, 4(r2)
i12: l.s f8, 4(r3)
i13: fsub.s f8, f7, f8
i14: fmul.s f8, f15, f8
i15: fsub.s f8, f4, f8
i16: s.s f8, 0(r8)
```

(b)

Figure 38 illustrates a code fragment for an FFT implementation. Two source-level statements are compiled into 16 assembly instructions, including load and store instructions. The floating-point array variables are stored in memory and must be first loaded before operations can be performed. After the computation the results are stored back out to memory. Integer registers are used to hold addresses of arrays. Floating point registers are used to hold temporary data. The DFG induced by the writing and reading of floating-point registers by the 16 instructions of Figure 38b is shown in Figure 39.

Each node in Figure 39 represents an instruction in Figure 38b. A directed edge exists between two instructions if there exists a true data dependence between the two instructions. A dependent register can be identified for each of the dependence edges in the DFG. A latency can also be associated with each dependence edge. In Figure 39, each edge is labeled with the execution latency of the producer instruction. In this example, load, store, addition and subtraction instructions are assumed to have 2-cycle execution latency, while multiplication instructions require 4 cycles.

FIGURE 39 Data-flow graph (DFG) of the code fragment in Figure 38b.



The latencies associated with dependence edges are cumulative. The longest dependence chain, measured in terms of total cumulative latency, is identified as the critical path of a DFG. Even assuming unlimited machine resources, a code fragment cannot be executed any faster than the length of its critical path. This is commonly referred to as the *data-flow limit* to program execution and represents the best performance that can possibly be achieved. For the code fragment of Figure 39 the data-flow limit is 12 cycles. The data-flow limit is dictated by the true data dependences in the program. Traditionally, the *data-flow execution model* stipulates that every instruction in a program begins execution immediately in the cycle following when all its operands become available. In effect, all existing register data-flow techniques are attempts to approach the data-flow limit.

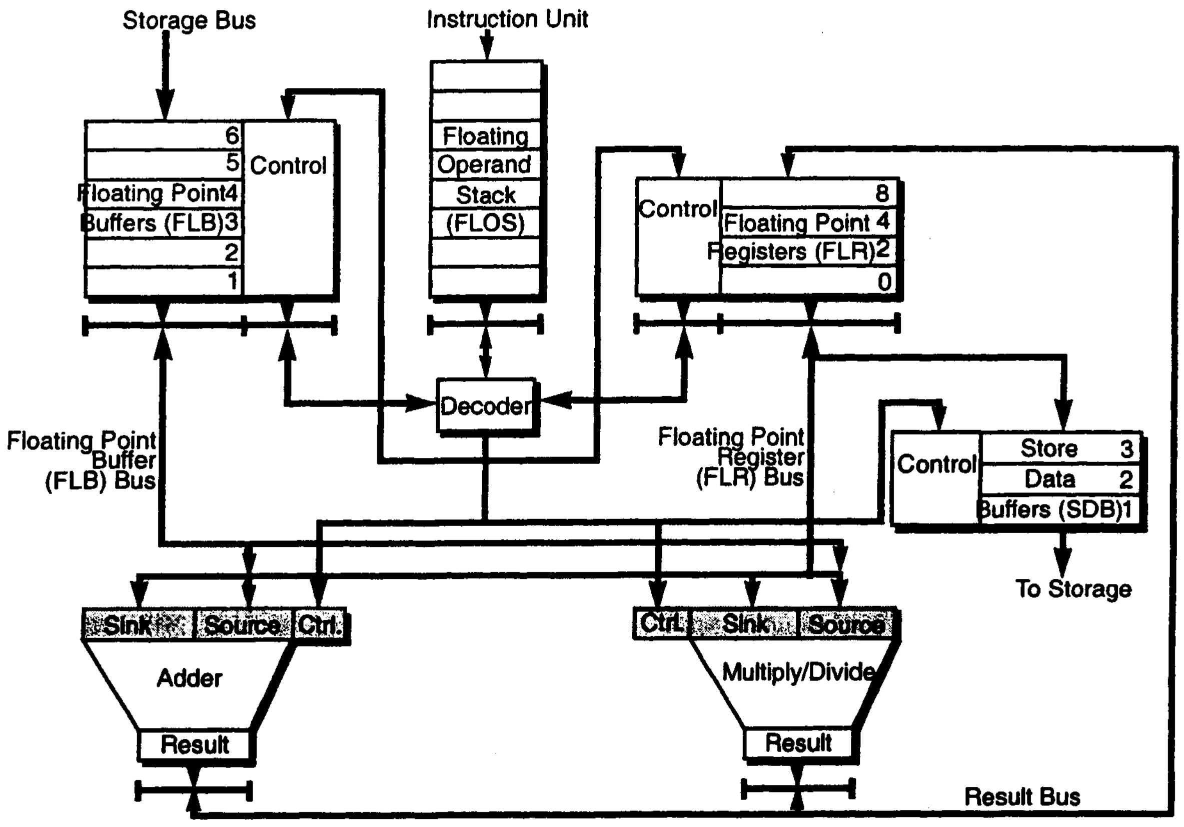
3.2.2.4 The Classic Tomasulo's Algorithm

The design of the IBM 360/91's floating-point unit, incorporating what has come to be known as the "Tomasulo's algorithm," laid the ground work for modern superscalar processor designs. Key attributes of most current register data-flow techniques can be found in the classic Tomasulo's algorithm, which deserves an in-depth examination. We first introduce the original design of the floating-point unit (FPU) of the IBM 360, then describe in detail the modified design of the FPU in the IBM 360/91 that incorporated the Tomasulo's algorithm, and then illustrate its operation and effectiveness in processing an example code sequence.

The original design of the IBM 360 Floating-Point Unit (FPU) is shown in Figure 40. The FPU contains two functional units: one floating-point add unit and one floating-point multiply/divide unit. There are three register files in the FPU: the floating-point registers (FLR), the floating-point buffers (FLB), and the store data buffers (SDB). There are four FLR registers; these are the architected floating-point registers. Floating-point instructions with storage-register or storage-storage addressing modes are preprocessed. Address generation and memory accessing are performed outside of the FPU. When the data is retrieved from the memory it is loaded into one of the six FLB registers. Similarly if the destination of an instruction is a memory location, the result to be stored is placed in one of the three SDB registers and a separate unit accesses the SDB to complete the storing of the result to a memory location. Using these two additional register files, i.e. FLB and SDB, to support storage-register and storage-storage instructions, the FPU effectively functions as a register-register machine.

In the IBM 360, the Instruction Unit (IU) decodes all the instructions and passes all floating-point instructions (in order) to the floating-point operation stack (FLOS). In the FPU, floating-point instructions are then further decoded and issued in order from the FLOS to the two functional units. The two functional units are not pipelined and incur multiple-cycle latencies. The adder incurs two cycles for add instructions, while the multiply/divide unit incurs three cycles and 12 cycles for performing multiply and divide instructions, respectively.

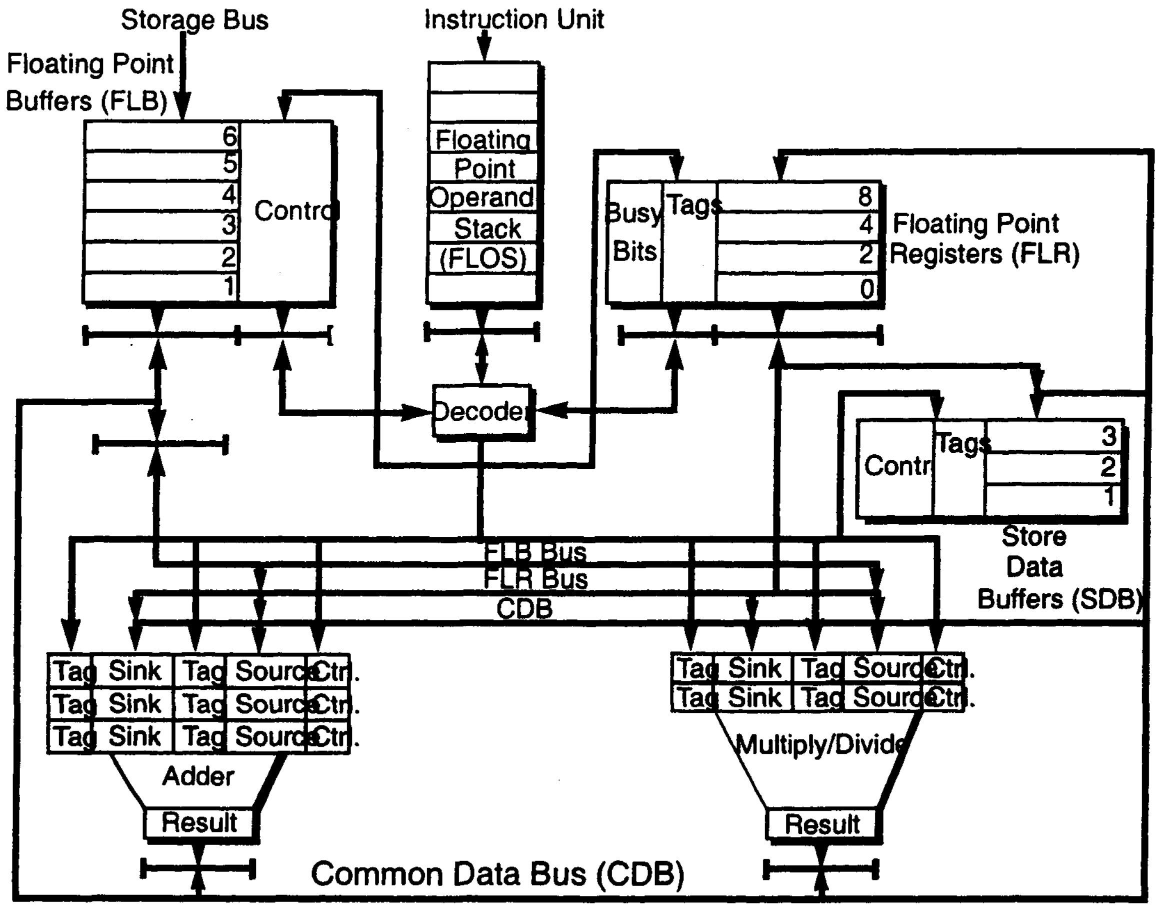
FIGURE 40 The original design of the IBM 360 floating-point unit (FPU).



In the mid 1960's, IBM began developing what eventually became Model 91 of the Systems 360 family. One of the goals was to achieve concurrent execution of multiple floating-point instructions and to sustain a throughput of one instruction per cycle in the instruction pipeline. This is quite aggressive considering the complex addressing modes of the 360 ISA and the multi-cycle latencies of the execution units. The end result is a modified FPU in the 360/91 that incorporated the Tomasulo's algorithm; see Figure 41.

Tomasulo's algorithm consists of adding three new mechanisms to the original FPU design, namely reservation stations, the common data bus, and register tags. In the original design, each functional unit has a single buffer on its input side to hold the instruction currently being executed. If a functional unit is busy, issuing of instructions by FLOS will stall whenever the next instruction to be issued requires the same functional unit. To alleviate this structural bottleneck, multiple buffers, called *reservation stations*, are attached to the input side of each functional unit. The adder unit has three reservations stations, while the multiply/divide unit has two. These reservation stations are viewed as virtual functional units; as long as there is a free reservation station, the FLOS can issue an instruction to that functional unit even if it is currently busy executing another instruction. Since FLOS issues instructions in order, this will prevent unnecessary stalling due to unfortunate ordering of different floating-point instruction types.

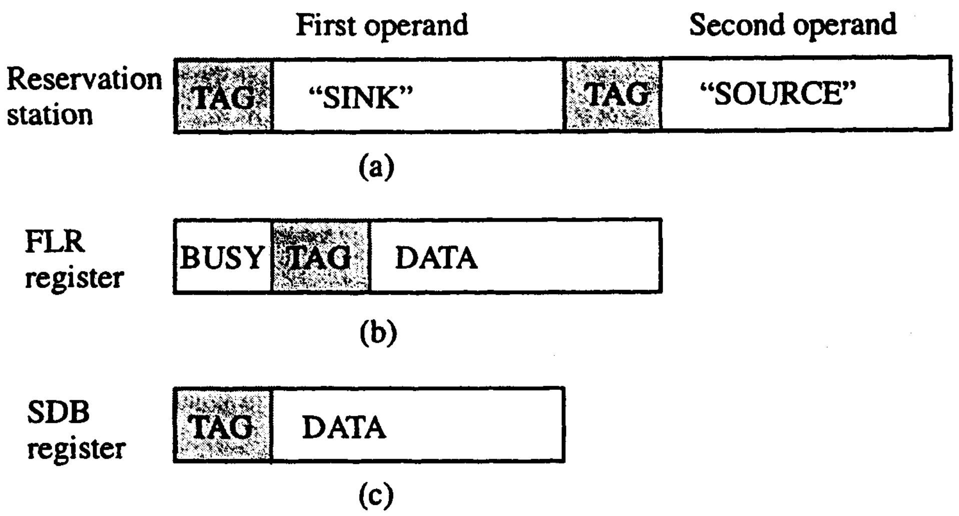
FIGURE 41 The modified design of the IBM 360/91 floating-point unit (FPU) with Tomasulo's algorithm.



With the availability of reservation stations, instructions can also be issued to the functional units by FLOS even though not all of their operands are yet available. These instructions can wait in the reservation station for their operands and only begin execution when they become available. The *common data bus* (CDB) connects the outputs of the two functional units to the reservation stations as well as the FLR and SDB registers. Results produced by the functional units are broadcast unto the CDB. Those instructions in the reservation stations needing the results as their operands will latch in the data from the CDB. Those registers in the FLR and SDB that are the destinations of these results also latch in the same data from the CDB. The CDB facilitates the forwarding of results directly from producer instructions to consumer instructions waiting in the reservation stations without having to go through the registers. Destination registers are updated simultaneously with the forwarding of results to dependent instructions. If an operand is coming from a memory location, it will be loaded into a FLB register once memory accessing is performed. Hence, the FLB can also output onto the CDB, allowing an waiting instruction in a reservation station to latch in its operand. Consequently, the two functional units and the FLB can drive data onto the CDB, and the reservation stations, FLR and SDB can latch in data from the CDB.

When the FLOS is dispatching an instruction to a functional unit, it allocates a reservation station and checks to see if the needed operands are available. If an operand is available in the FLR, then the content of that register in the FLR is copied to the reservation station, otherwise a tag is copied to the reservation station instead. The tag indicates where the pending operand is going to come from. The pending operand can come from a producer instruction currently resident in one of the five reservation stations, or it can come from one of the six FLB registers. In order to uniquely identify one of these eleven possible sources for a pending operand, a 4-bit tag is required. If one of the two operand fields of a reservation station contains a tag instead of the actual operand, it indicates that this instruction is waiting for a pending operand. When that pending operand becomes available, the producer of that operand drives the tag along with the actual operand onto the CDB.

FIGURE 42 The use of tag fields in: (a) a reservation station; (b) a FLR register; and (c) a SDB register.



A waiting instruction in a reservation station uses its tag to monitor the CDB. When it detects a tag match on the CDB, it then latches in the associated operand. Essentially the producer of an operand broadcasts the tag and the operand on the CDB, all consumers of that operand monitors the CDB for that tag and when the broadcasted tag matches their tag, they then latch in the associated operand from the CDB. Hence, all possible destinations of pending operands must carry a tag field and must monitor the CDB for a tag match. Each reservation station contains two operand fields, each of which must carry a tag field since each of the two operands can be pending. The four registers in the FLR and the three registers in the SDB must also carry tag fields. This is a total of 17 tag fields representing 17 places that can monitor and receive operands; see Figure 42. The tag field at each potential consumer site is used in an associative fashion to monitor for possible matching of its content with the tag value being broadcasted on the CDB. When a tag match occurs, the consumer latches in the broadcasted operand.

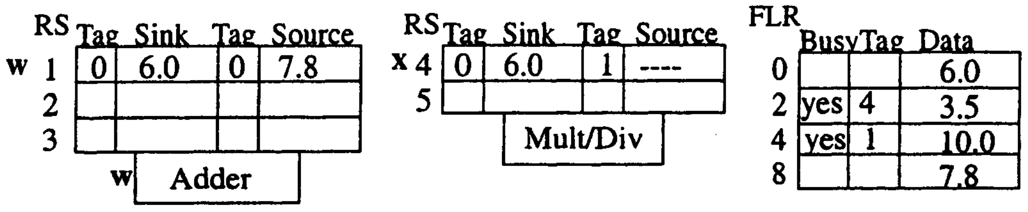
The IBM 360 floating-point instructions use a two-address instruction format. Two source operands can be specified. The first operand specifier is called the "sink" because it also doubles as the destination specifier. The second operand specifier is called the "source." Each reservation station has two operand fields, one for the sink and the other for the source. Each operand field is accompanied by a tag field. If an operand field contains real data, then its tag field is set to zero. Otherwise, its tag field identifies the source where the pending operand will be coming from, and is used to monitor the CBD for the availability of the pending operand. Whenever an instruction is dispatched by FLOS to a reservation station, the data in the FLR register corresponding to the sink operand is retrieved and copied to the reservation station. At the same time, the "busy" bit associated with this FLR register is set, indicating that there is a pending update of that register, and the tag value that identifies the particular reservation station to which the instruction is being dispatched is written into the tag field of the same FLR register. This clearly indicates which of the reservation station will eventually produce the updated data for this FLR register. Subsequently if a trailing instruction specifies this register as one of its source operands, when it is dispatched to a reservation station, only the tag field (called the "pseudo operand") will be copied to the corresponding tag field in the reservation station and not the actual data. When the busy bit is set it indicates the data in the FLR register is stale and the tag represents the source from which the real data will come from. Other than reservation stations and FLR registers, SDB registers can also be destinations of pending operands and hence also require a tag field for each of the three SDB registers.

We now use an example sequence of instructions to illustrate the operation of the Tomasulo's algorithm. We deviate from the actual IBM 360/91 design in several ways to help make the example more clear. First, instead of the 2-address format of the IBM 360 instructions, we will use 3-address instructions to avoid potential confusion. The example sequence contains only register-register instructions. To reduce the number of machine cycles we have to trace, we will allow the FLOS to dispatch (in program order) up to two instructions in every cycle. We also assume that an instruction can begin execution in the same cycle that it is dispatched to a reservation station. We keep the same latencies of 2 cycles and 3 cycles for add and multiply instructions, respectively. However, we allow an instruction to forward its result to dependent instructions during its last execution cycle, and a dependent instruction can begin execution in the next cycle. The tag values of 1, 2 and 3 are used to identify the three reservation stations of the adder functional unit, while 4 and 5 are used to identify the two reservation stations of the multiply/divide functional unit. These tag values are called the "ID's" of the reservation stations. The example sequence consists of the following four register-register instructions.

```
w:  R4 <-- R0 + R8
x:  R2 <-- R0 * R4
y:  R4 <-- R4 + R8
z:  R8 <-- R4 * R2
```

FIGURE 43 Illustration of Tomasulo's algorithm on an example instruction sequence. (Part 1)

CYCLE #1 Dispatched instruction(s): w, x (in order)

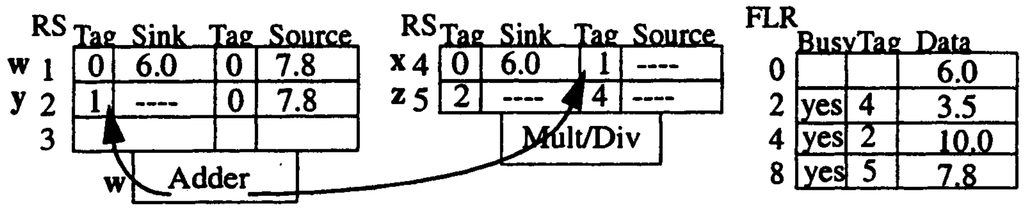


	RS	Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
	2				
	3				

	RS	Tag	Sink	Tag	Source
x	4	0	6.0	1	----
	5				

FLR	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4	yes	1	10.0
8			7.8

CYCLE #2 Dispatched instruction(s): y, z (in order)

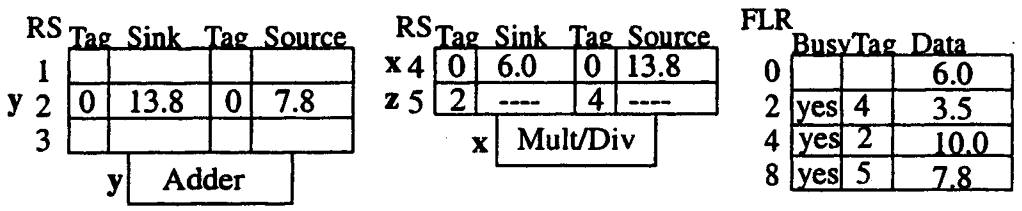


	RS	Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
y	2	1	----	0	7.8
	3				

	RS	Tag	Sink	Tag	Source
x	4	0	6.0	1	----
z	5	2	----	4	----

FLR	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4	yes	2	10.0
8	yes	5	7.8

CYCLE #3 Dispatched instruction(s): ____________



	RS	Tag	Sink	Tag	Source
	1				
y	2	0	13.8	0	7.8
	3				

	RS	Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
z	5	2	----	4	----

FLR	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4	yes	2	10.0
8	yes	5	7.8

Figure 43 illustrates the first three cycles of execution. In Cycle #1, instructions w and x are dispatched (in order) to reservation stations 1 and 4. The destination registers of instructions w and x are R4 and R2 (i.e. FLR registers 4 and 2), respectively. The busy bit of these two registers are set. Since instruction w is dispatched to reservation station 1, the tag value of 1 is entered into the tag field of R4 indicating that the instruction in reservation station 1 will produce the result for updating R4. Similarly the tag value of 4 is entered into the tag field of R2. Both source operands of instruction w are available so it begins execution immediately. Instruction x requires the result (R4) of instruction w for its second ("source") operand. Hence when instruction x is dispatched to reservation station 4, the tag field of the second operand is written the tag value of 1 indicating that the instruction in reservation station 1 will produce the needed operand.

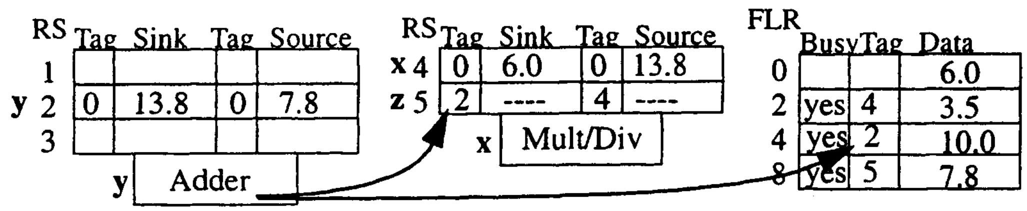
During Cycle #2, instructions y and z are dispatched (in order) to reservation stations 2 and 5, respectively. Because it needs the result of instruction w for its first operand, instruction y when it is dispatched to reservation station 2 receives the tag value of 1 in the tag field of the first operand. Similarly instruction z, dispatched to reservation station 5, receives the tag values of 2 and 4 in its two tag fields, indicating that reservation stations 2 and 4 will eventually produce the two operands it needs. Since R4 is the destination of instruction y, the tag field of R4 is updated with the new tag value of 2, indicating reservation station 2 (i.e. instruction y) is now responsible for the pending update of R4. The busy bit of R4 remains set. The busy bit of R8 is set when instruction z is dispatched to reservation station 5, and the tag field of R8 is set to 5. At the end of Cycle #2, instruction w finishes execution and broadcasts its ID (reservation station 1) and its result onto the CDB. All the tag fields containing the tag value of 1 will trigger a tag match and latches in the broadcasted result. The first tag field of reservation stations 2 (holding instruction y) and the second tag field of reservation station 4 (holding instruction x) have such tag matches. Hence the result of instruction w is forwarded to dependent instructions x and y.

In Cycle #3, instruction y begins execution in the adder unit and instruction x begins execution in the multiply/divide unit. Instruction y finishes execution in Cycle #4 (see Figure 44) and broadcasts its result on the CDB along with the tag value of 2 (its reservation station ID). The first tag field in reservation station 5 (holding instruction z) and the tag field of R4 have tag matches and pull in the result of instruction y. Instruction x finishes execution in Cycle #5 and broadcasts its result on the CDB along with the tag value of 4. The second tag field in reservation station 5 (holding instruction z) and the tag field of R2 have tag matches and pull in the result of instruction x. In Cycle #6, instruction z begins execution and finishes in Cycle #8.

Figure 45a illustrates the data dependence graph of the above example sequence of four instructions. The four solid arcs represent the four true data dependences, while the other three arcs represent the anti and output dependences. Instructions are dispatched in program order. Anti-dependences are resolved by copying an operand at dispatch time to the reservation station. Hence, it is not possible for a trailing instruction to overwrite a register before an earlier instruction has a chance to read that register. If the operand is still pending the dispatched instruction will receive the tag for that operand. When that operand becomes available, the instruction will receive that operand via a tag match in its reservation station.

FIGURE 44 Illustration of Tomasulo's algorithm on an example instruction sequence. (Part 2)

CYCLE #4 Dispatched instruction(s): __________

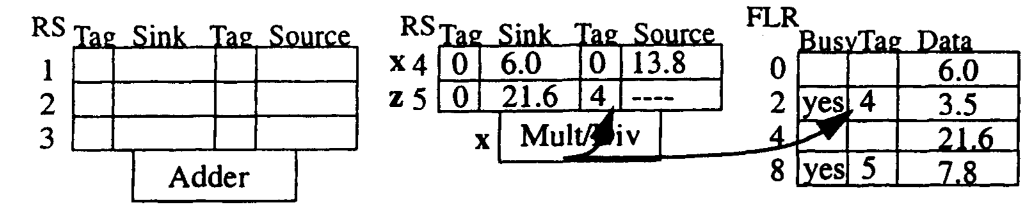


RS	Tag	Sink	Tag	Source
1				
y 2	0	13.8	0	7.8
3				

RS	Tag	Sink	Tag	Source
x 4	0	6.0	0	13.8
z 5	2	----	4	----

FLR	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4	yes	2	10.0
8	yes	5	7.8

CYCLE #5 Dispatched instruction(s): __________

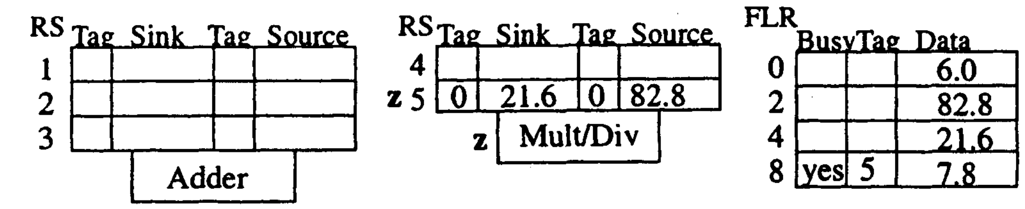


RS	Tag	Sink	Tag	Source
1				
2				
3				

RS	Tag	Sink	Tag	Source
x 4	0	6.0	0	13.8
z 5	0	21.6	4	----

FLR	Busy	Tag	Data
0			6.0
2	yes	4	3.5
4			21.6
8	yes	5	7.8

CYCLE #6 Dispatched instruction(s): __________



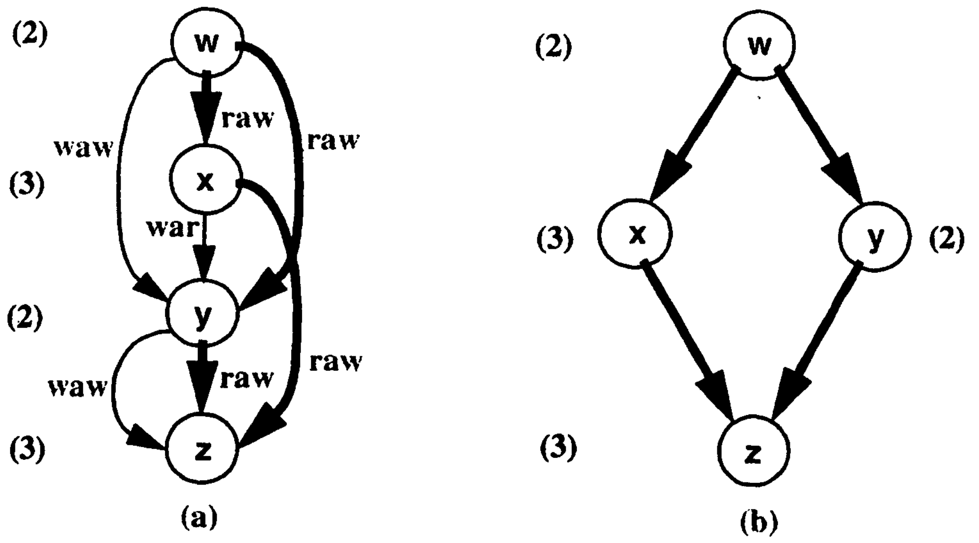
RS	Tag	Sink	Tag	Source
1				
2				
3				

RS	Tag	Sink	Tag	Source
4				
z 5	0	21.6	0	82.8

FLR	Busy	Tag	Data
0			6.0
2			82.8
4			21.6
8	yes	5	7.8

As an instruction is dispatched, the tag field of its destination register is written with the reservation station ID of that instruction. When a subsequent instruction with the same destination register is dispatched, the same tag field will be updated with the reservation station ID of this new instruction. The tag field of a register always contains the reservation station ID of the latest updating instruction. If there are multiple instructions in flight that have the same destination register, only the latest instruction will be able to update that register. Output dependences are implicitly resolved by making it impossible for an earlier instruction to update an register after a later instruction has updated the same register. This does introduce the problem of not being able to support precise exception since the register file does not necessarily evolve through all of its sequential states, i.e. a register can potentially miss an intermediate update. For example in Figure 43, at the end of Cycle #2 instruction w should have updated its destination register R4. However, instruction y has the same destination register and when it was dispatched earlier in that cycle, the tag field of R4 was changed from 1 to 2 anticipating the update of R4 by instruction y. At the end of Cycle #2 when instruction w broadcasts its tag value of 1, the tag field of R4 fails to trigger a tag match and does not pull in the result of instruction w. Eventually R4 will be updated by instruction y. However, if an exception is triggered by instruction x, precise exception will be impossible since the register file does not evolve through all of its sequential states.

FIGURE 45 Data dependence graphs of the example instruction sequence: (a) all data dependences; (b) true data dependences.



The Tomasulo's algorithm resolves anti and output dependences via a form of register renaming. Each definition of an FLR register triggers the renaming of that register to a register tag. This tag is taken from the ID of the reservation station containing the instruction that redefines that register. This effectively removes false dependences from causing pipeline stalls. Hence the dataflow limit is strictly determined by the true data dependences. Figure 45b depicts the data dependence graph involving only true data dependences. As shown in Figure 45a if all four instructions were required to execute sequentially to enforce all the data dependences, including anti and output dependences, the total latency required for executing this sequence of instructions will be 10 cycles given the latencies of 2 cycles and 3 cycles for addition and multiplication instructions, respectively. When only true dependences are considered, Figure 45b reveals that the critical path is only 8 cycles, i.e. the path involving instructions w, x, and z. Hence the dataflow limit for this sequence of four instructions is 8 cycles. This limit is achieved by the Tomasulo's algorithm as demonstrated in Figure 43 and Figure 44.

3.2.2.5 Dynamic Execution Core

Most current state-of-the-art superscalar microprocessors consists of an out-of-order execution core sandwiched between an in-order front-end, that fetches and dispatches instructions in program order, and an in-order back-end, that completes and retires instructions also in program order. The out-of-order execution core (more recently referred to as the dynamic execution core), resembling a refinement of Tomasulo's algorithm, can be viewed as an embedded dataflow engine that attempts to approach the dataflow limit in instruction execution. The operation of such a dynamic execution core can be described according to the three phases in the pipeline, namely *instruction dispatching*, *instruction execution* and *instruction completion*; see Figure 46.

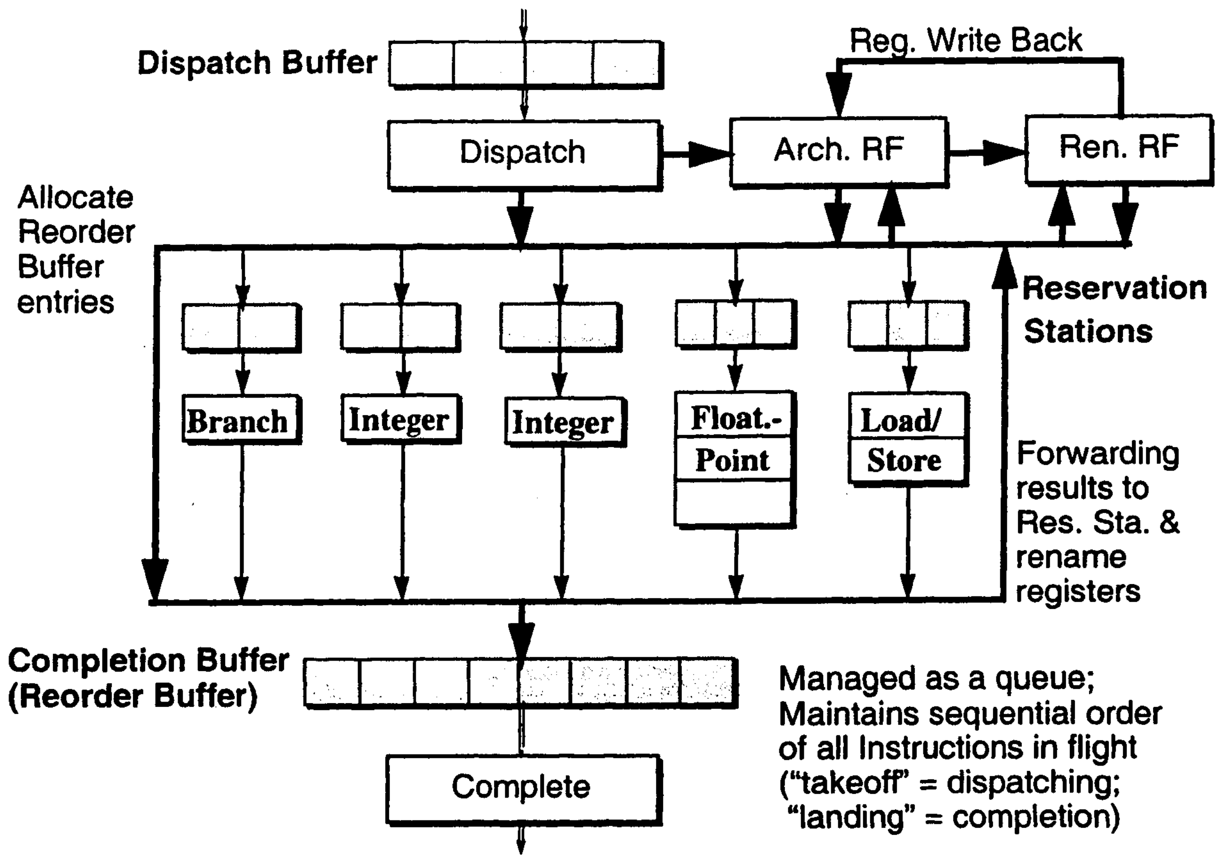
The instruction dispatching phase consists of *renaming* of destination registers, *allocating* of reservation station and reorder buffer entries, and *advancing* instructions from the dispatch buffer to the reservation stations. For ease of presentation, this subsection assumes that register renaming is performed in the dispatch stage. All redefinition of architected registers are renamed to rename registers. Trailing uses of these redefinitions are assigned the corresponding rename register specifiers. This ensures that all producer-consumer relationships are properly identified and all false register dependences are removed.

Instructions in the dispatch buffer are then dispatched to the appropriate reservation stations based on instruction type. Here we assume the use of distributed reservation stations and we use "reservation station" to refer to the (multi-entry) instruction buffer attached to each functional unit and "reservation station entry" to refer to one of the entries of this buffer. Simultaneous with the allocation of reservation station entries for the dispatched instructions is the allocation of entries in the reorder buffer for the same instructions. Reorder buffer entries are allocated according to program order.

Typically, for an instruction to be dispatched there must be the availability of a rename register, a reservation station entry, and a reorder buffer entry. If any one of these three is not available, instruction dispatching is stalled. The actual dispatching of instructions from the dispatch buffer entries to the reservation station entries is via a complex routing network. If the connectivity of this routing network is less than that of a full crossbar (this is frequently the case in real designs) then stalling can also occur due to resource contention in the routing network.

The instruction execution phase consists of *issuing* of ready instructions, *executing* the issued instructions, and *forwarding* of results. Each reservation station is responsible for identifying instructions that are ready to execute and for scheduling their execution. When an instruction is first dispatched to a reservation station, it may not have all of its source operands and therefore must wait in the reservation station. Waiting instructions continually monitor the bus(es) for tag matches. When a tag match is triggered, indicating the availability of the pending operand, the result being broadcasted is latched into the reservation station entry. When an instruction in a reservation station entry has all of its operands, it becomes ready for execution and can be issued into the functional unit. In a given machine cycle if multiple instructions in a reservation station are ready, a scheduling algorithm is used (typically oldest first) to pick one of them for issuing into the functional unit to begin execution. If there is only one functional unit connected to a reservation station (as is the case for distributed reservation stations) then that reservation station can only issue one instruction per cycle.

FIGURE 46 "Dataflow engine" for dynamic execution.


Dispatch Buffer
Reg. Write Back
Dispatch
Arch. RF
Ren. RF
Allocate Reorder Buffer entries
Reservation Stations
Branch
Integer
Integer
Float.-Point
Load/ Store
Forwarding results to Res. Sta. & rename registers
Completion Buffer (Reorder Buffer)
Managed as a queue; Maintains sequential order of all Instructions in flight ("takeoff" = dispatching; "landing" = completion)
Complete

Once issued into a functional unit, an instruction is executed. Functional units can vary in terms of their latency. Some have single-cycle latency, others have fixed multiple-cycle latencies. Certain functional units can require variable number of cycles, depending on the values of the operands and the operation being performed. Typically, even with function units that require multiple-cycle latencies, once an instruction begins execution in a pipelined functional unit, there is no further stalling of that instruction in the middle of the execution pipeline since all data dependences have been resolved prior to issuing and there shouldn't be any resource contention.

When an instruction finishes execution it asserts its destination tag (i.e. the specifier of the rename register assigned for its destination) and the actual result onto a forwarding bus. All dependent instructions waiting in the reservation stations will trigger a tag match and latch in the broadcasted result. This is how an instruction forwards its result to other dependent instructions without requiring the intermediate steps of updating and then reading of the dependent register. Concurrent with result forwarding, the RRF uses the broadcasted tag as an index and loads the broadcasted result into the selected entry of the RRF.

Typically a reservation station entry is deallocated when its instruction is issued in order to allow another trailing instruction to be dispatched into it. Reservation saturation can cause instruction dispatch to stall. Certain instructions whose execution can induce an exceptional condition that may require their rescheduling for execution in a future cycle. Frequently, for these instructions, their reservation station entries are not deallocated until they finish execution without triggering any exceptions. For example a load instruction can potentially trigger a D-cache miss that may require many cycles to service. Instead of stalling the functional unit, such an excepting load can be reissued from the reservation station after the miss has been serviced.

In a dynamic execution core as described above, a producer-consumer relationship is satisfied without having to wait for the writing and then the reading of the dependent register. The dependent operand is directly forwarded from the producer instruction to the consumer instruction to minimize the latency incurred due to the true data dependence. Assuming that an instruction can be issued in the same cycle that it receives its last pending operand via a forwarding bus; if there is no other instruction contending for the same functional unit, then this instruction should be able to begin execution in the cycle immediately following the availability of all of its operands. Hence, if there are adequate resources such that no stalling due to structural dependences occurs, then the dynamic execution core should be able to approach the dataflow limit.

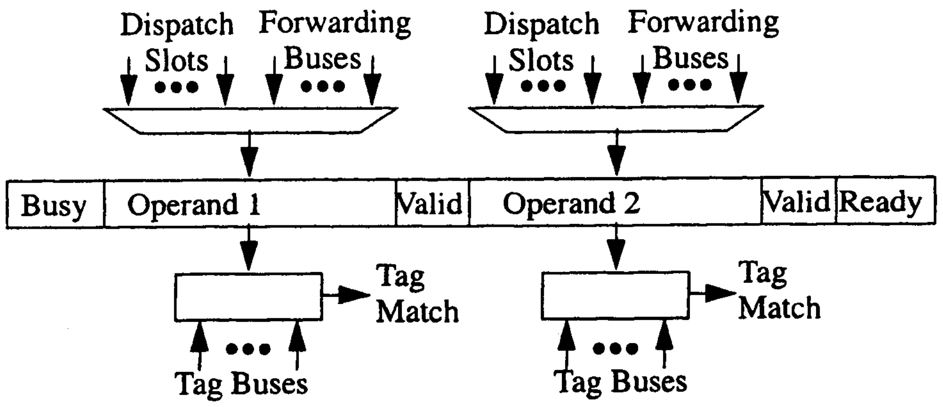
3.2.2.6 Reservation Stations and Reorder Buffer

Other than the functional units, the critical components of the dynamic execution core are the reservation stations and the reorder buffer. The operations of these buffers dictate the function of the dynamic execution core. This subsection presents the issues associated with the implementation of the reservation station and the reorder buffer. We present their organization and behavior with special focus on loading and unloading of an entry of a reservation station and the reorder buffer.

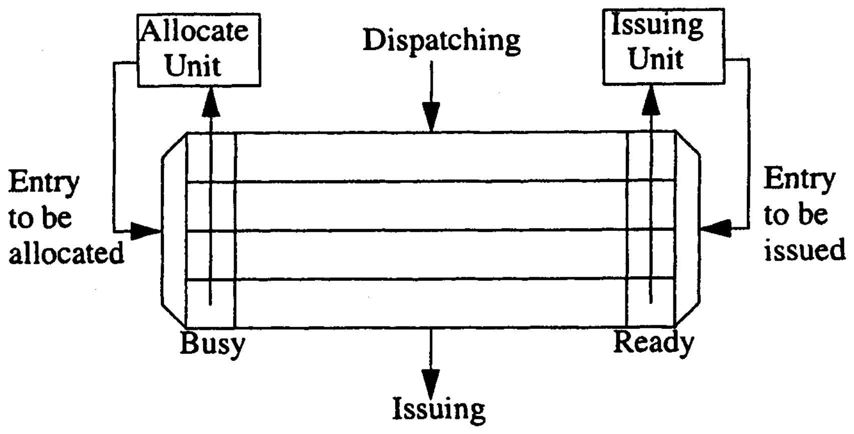
There are three tasks associated with the operation of a reservation station: *dispatching*, *waiting*, and *issuing*. A typical reservation station is shown in Figure 47b and the various fields in an entry of a reservation station is illustrated in Figure 47a. Each entry has a busy bit, indicating that the entry has been allocated, and a ready bit, indicating that the instruction in that entry has all of its source operands. Dispatching involves loading an instruction from the dispatch buffer into an entry of the reservation station. Typically the dispatching of an instruction requires the following three steps: select a free, i.e. not busy, reservation station entry, load operands and/or tags into the selected entry, and set the busy bit of that entry. The selection of a free entry is based on the busy bits and is performed by the allocate unit. The allocate unit examines all the busy bits and selects one of the nonbusy entries to be the allocated entry. This can be implemented using a priority encoder. Once an entry is allocated the operands/tags of the instruction are loaded into the entry. Each entry has two operand fields, each of which has an associated valid bit. If the operand field contains the actual operand, then the valid bit is set. If the field contains a tag, indicating a pending operand, then its valid bit is reset and it must wait for the operand to be forwarded. Once an entry is allocated, its busy bit must be set.

FIGURE 47 Reservation station mechanisms: (a) a reservation station entry; (b) dispatching into and issuing from a reservation station.

(a)



(b)



An instruction with a pending operand, must wait in the reservation station. When a reservation station entry is waiting for a pending operand, it must continuously monitor the tag bus(es). When a tag match occurs, the operand field latches in the forwarded result and sets its valid bit.

When both operand fields are valid, the ready bit is set indicating that the instruction has all of its source operands and ready to be issued.

The issuing step is responsible for selecting a ready instruction in the reservation station and issues it into the functional unit. All the ready instructions are identified by their ready bits being set. The selecting of a ready instruction is performed by the issuing unit based on a scheduling heuristic; see Figure 47b. The heuristic can be based on program order or how long each ready instruction has been waiting in the reservation station. Typically when an instruction is issued into the functional unit, its reservation station entry is deallocated by resetting the busy bit.

A large reservation station can be quite complex to implement. On its input side, it must support many possible sources, including all the dispatch slots and forwarding buses; see Figure 47a. The data routing network on its input side can be quite complex. During the waiting step, all operand fields of a reservation station with pending operands must continuously compare its tag against potentially multiple tag buses. This is comparable to doing associative search across all the reservation station entries involving multiple keys (tag buses). If the number of entries is small, this is quite feasible. However as the number of entries increases, the complexity increase is quite significant. This portion of the hardware is commonly referred to as the "wake up logic" [ ]. When the entry count increases, it also complicates the issuing unit and the scheduling heuristic in selecting the best ready instruction to issue.

The reorder buffer contains all the instructions that are "in flight," i.e. all the instructions that have been dispatched but not yet completed architecturally. These include all the instructions waiting in the reservation stations, executing in the functional units, and those that have finished execution but are waiting to be completed in program order. The status of each instruction in the reorder buffer can be tracked using several bits in each entry of the reorder buffer. Each instruction can be in one of several states, i.e. waiting execution, in execution, and finished execution. These status bits are updated as an instruction traverses from one state to the next. An additional bit can also be used to indicate whether an instruction is speculative (in the predicted path) or not. If speculation can cross multiple branches, additional bits can be employed to identify which speculative basic block an instruction belongs to. When a branch is resolved a speculative instruction can become nonspeculative (if the prediction is correct) or invalid (if the prediction is incorrect). Only finished and nonspeculative instructions can be completed. An instruction marked invalid is not architecturally completed when exiting the reorder buffer. Figure 48a illustrates the fields typically found in a reorder buffer entry; in this figure the rename register field is also included.

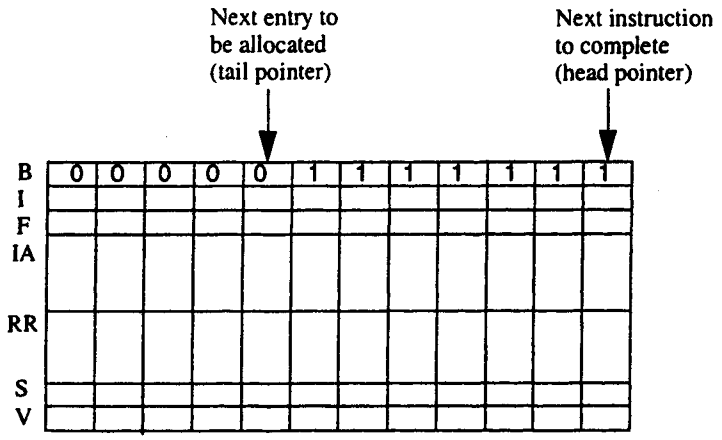
The reorder buffer is managed as a circular queue using a head pointer and a tail pointer; see Figure 48b. Tail pointer is advanced when reorder buffer entries are allocated at instruction dispatch. The number of entries that can be allocated per cycle is limited by the dispatch bandwidth. Instructions are completed from the head of the queue. From the head of the queue as many instructions that have finished execution can be completed as the completion bandwidth allows. The completion bandwidth is determined by the capacity of another routing network. One of the critical issues is the number of write ports to the architected register file that are needed to support the transferring of data from the rename registers (or the reorder buffer entries if they are used as rename registers) to the architected registers. When an instruction is completed, its rename register as well as its reorder buffer entry are deallocated. The head pointer to the reorder buffer is also appropriately updated. In a way the reorder buffer can be viewed as the heart or the central control of the dynamic execution core because the status of all in-flight instructions are tracked by the reorder buffer.

FIGURE 48 (a) Reorder buffer entry; (b) reorder buffer organization.

(a)

Busy	Issued	Finished	Instruction Address	Rename Register	Speculative	Valid

(b)



Reorder Buffer

It is possible to combine the reservation stations and the reorder buffer as one structure, called the instruction window, that manages all the instructions in flight. Since at dispatch an entry in the reservation station and an entry in the reorder buffer must be allocated for each instruction, they can be combined as one entry in the instruction window. Hence, instructions are dispatched into the instruction window, entries of the instruction window monitor the tag buses for pending operands, results are forwarded into the instruction window, and instructions are completed from the instruction window.

3.2.2.7 Other Register Data-Flow Techniques

For many years the dataflow limit has been assumed to be an absolute theoretical limit and the ultimate performance goal. Extensive research efforts on dataflow architectures and dataflow machines have been ongoing for over three decades. The dataflow limit assumes that true data dependences are absolute and cannot possibly be overcome. Interestingly, in the late 1960's and the early 1970's similar assumption was made concerning control dependences [,]. It was generally thought that control dependences are absolute and that when encountering a conditional branch instruction there is no choice but to wait for that conditional branch to be executed before proceeding to the next instruction due to the uncertainty of actual control flow. Since then we have witnessed tremendous strides made in the area of branch prediction techniques. Conditional branches and associated control dependences are no longer absolute barriers and can frequently be overcome by speculating on the direction and the target address of the branch [,]. The basis that made such speculation possible is that frequently the outcome of a branch instruction is quite predictable. It wasn't until 1995 that researchers began to also question the absoluteness of true data dependences.

In 1996 several research papers appeared that proposed the concept of value prediction [, ,]. The first paper focused on predicting load values [ ] based on the observation that frequently the values being loaded by a particular static load instruction are quite predictable. The second paper generalized the same basic idea for predicting the result of ALU instructions [ ]. Experimental data based on real input data sets indicate that the results produced by many instructions are actually quite predictable. The notion of "value locality" that indicates that certain instructions tend to repeatedly produce the same small set (sometimes one) of result values. By tracking the results produced by these instructions, future values can become predictable based on the historical values. Since these seminal papers, numerous papers have been published in recent years proposing various designs of value predictors [, , ,]. In a recent study, it was shown that a hybrid value predictor can achieve prediction rates of up to 80% [ ] and a realistic design incorporating value prediction can achieve IPC improvements in the range of 8.6% to 23% for the SPEC benchmarks.

When the result of an instruction is correctly predicted via value prediction, typically performed during the fetch stage, a subsequent dependent instruction can begin execution using this speculative result without having to wait for the actual decoding and execution of the leading instruction. This effectively removes the serialization constraint imposed by the data dependence between these two instructions. In a way this particular dependence edge in the data dependence graph is effectively removed when correct value prediction is performed. Hence, value prediction provides the potential to exceed the classical dataflow limit. Of course validation is still required to ensure that the prediction is correct, and becomes the new limit on instruction execution throughput. Value prediction becomes effective in increasing machine performance if misprediction rarely occurs and the misprediction penalty is small (e.g. zero or one cycle), and if the validation latency is less than the average instruction execution latency. Clearly efficient implementation of value prediction is crucial in ensuring its efficacy in improving performance.

Another recently proposed idea is call dynamic instruction reuse [,]. Similar to the concept of value locality, it has been observed through experiments with real programs that frequently a same sequence of instructions is repeatedly executed using the same set of input data. This results in redundant computation being performed by the machine. Dynamic instruction reuse techniques attempt to track such redundant computations and when they are detected the previous result is used without performing the redundant computation. These techniques are nonspeculative, hence no validation is required. While value prediction can be viewed as the elimination of certain dependence edges in the data dependence graph, dynamic instruction reuse techniques attempt to remove both nodes and edges of a subgraph from the data dependence graph. A much earlier research effort had shown that such elimination of redundant computations can yield significant performance gains for programs written in functional languages [ ]. A more recent study also yields similar data on the presence of redundant computations in real programs [ ]. This is an area that is currently being actively researched, new insightful results can be expected.

3.2.3 Memory Data Flow Techniques

Memory instructions are responsible for moving data between the main memory and the register file, and are essential for supporting the execution of ALU instructions. Register operands needed by ALU instructions must first be loaded from memory. With limited number of registers, during the execution of a program not all of the operands can be kept in the register file. The compiler generates "spill code" to temporarily place certain operands out to the main memory and to reload them when they are needed. Such spill code are implemented using store and load instructions. Typically, the compiler only allocates scalar variables into registers. Complex data structures, such as arrays and linked lists, that far exceed the size of the register file are usually kept in the main memory. In order to perform operations on such data structures, load and store instructions are required. The effective processing of load/store instructions can minimize the overhead of moving data between the main memory and the register file.

The processing of load/store instructions and the resultant memory data flow can become a bottleneck to overall machine performance due to the potential long latency for executing memory instructions. The long latency of load/store instructions results from the need to compute a memory address and the need to access a memory location. In order to support virtual memory, the computed memory address (called the virtual address) also needs to be translated into a physical address before the physical memory can be accessed. Cache memories are very effective in reducing the effective latency for accessing the main memory. Furthermore, various techniques have been developed to reduce the overall latency and increase the overall throughput for processing load/store instructions.

3.2.3.1 Memory Accessing Instructions

The execution of memory data flow instructions occurs in three steps: memory address generation, memory address translation, and data memory accessing. We first state the basis for these three steps and then describe the processing of load/store instructions in a superscalar pipeline.

The register file and the main memory are defined by the instruction set architecture for data storage. The main memory as defined in an instruction set architecture is a collection of 2^n memory locations with random access capability, i.e. every memory location is identified by an n-bit address and can be directly accessed with the same latency. Just like the architected register file, the main memory is an architected entity and is visible to the software instructions. However, unlike the register file, the address that identifies a particular memory location is usually not explicitly stored as part of the instruction format. Instead, a memory address is usually generated based on a register and an offset specified in the instruction. Hence address generation is required and involves the accessing of the specified register and the adding of the offset value.

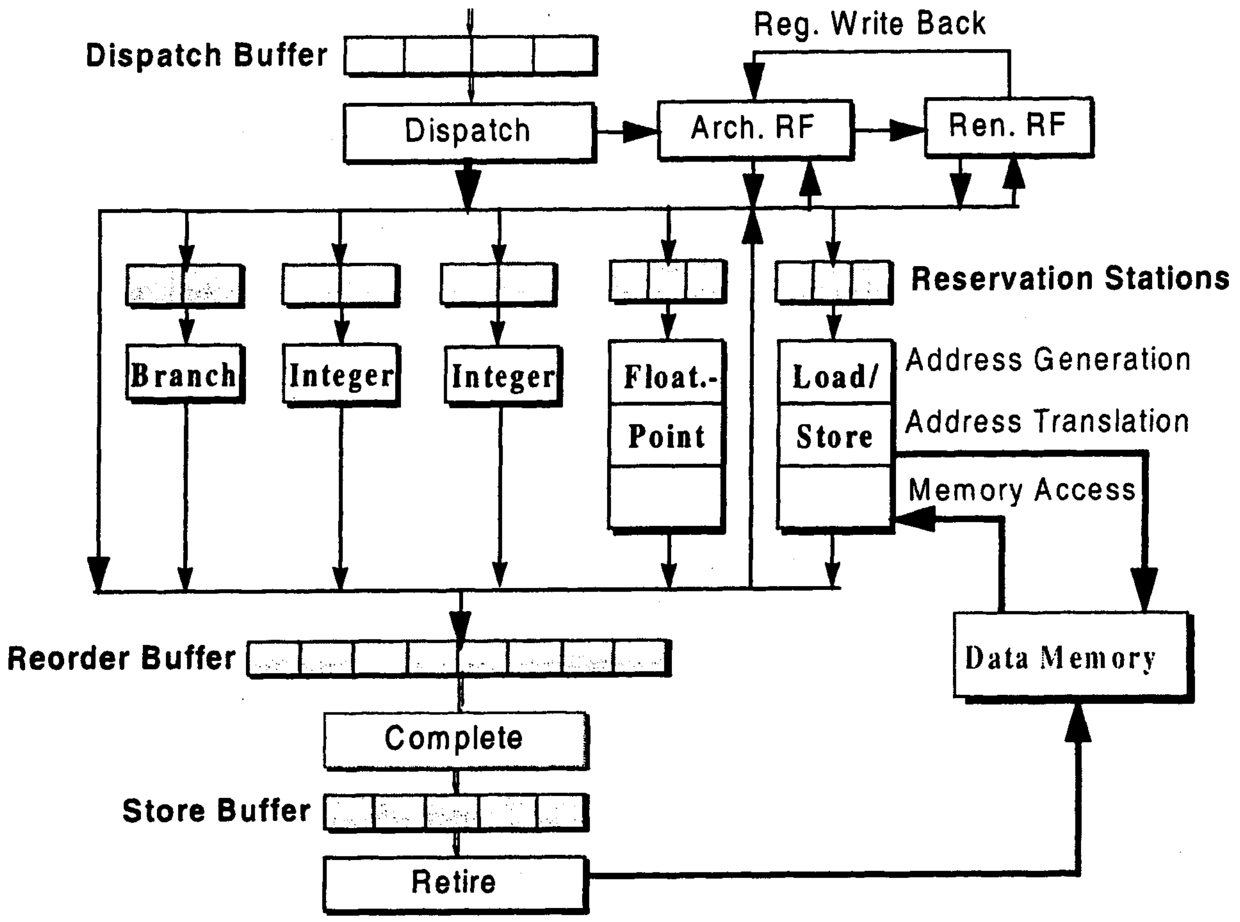
In addition to address generation, address translation is also required when virtual memory is implemented in a system. The architected main memory constitutes the virtual address space of the program and is viewed by each program as its private address space. The physical memory that is implemented in a machine constitutes the physical address space, which may be smaller than the virtual address space and may even be shared by multiple programs. Virtual memory is a mechanism that maps the virtual address space of a program to the physical address space of the machine. With such address mapping, virtual memory is able to support the execution of a program with a virtual address space that is larger than the physical address space, and the multiprogramming paradigm by mapping multiple virtual address spaces to the same physical address space. This mapping mechanism involves the translation of the computed effective address, i.e. the virtual address, into a physical address that can be used to access the physical memory. This mechanism is usually implemented using a mapping table, and address translation is performed via a table lookup.

The third step in processing a load/store instruction is memory accessing. For load instructions data is read from a memory location and stored into a register, while for store instructions a register value is stored into a memory location. While the first two steps of address generation and address translation are performed in identical fashion for both loads and stores, the third step is performed differently for loads and stores by a superscalar pipeline.

In Figure 49, we illustrate these three steps as occurring in three pipeline stages. The first pipe stage performs effective address generation. We assume the typical addressing mode of register indirect with an offset for both load and store instructions. For a load instruction, soon as the address register operand is available, it is issued into the pipelined functional unit and the effective address is generated by the first pipe stage. A store instruction must wait for the availability of both the address register and the data register operands before it is issued.

After the first pipe stage generates the effective address, the second pipe stage translates this virtual address into a physical address. Typically, this is done by accessing the translation lookaside buffer (TLB), which is a hardware controlled table containing the mapping of virtual to physical addresses. The TLB is essentially a cache of the page table which is stored in the main memory. (Subsection 3.2.3.2 provides more background material on the page table and the TLB.) It is possible that the virtual address being translated belongs to a page, the mapping of which is not currently resident in the TLB. This is called a TLB miss. If the particular mapping is present in the page table, then it can be retrieved by accessing the page table in the main memory. Once the missing mapping is retrieved and loaded into the TLB, the translation can be completed. It is also possible that the mapping is not resident even in the page table, meaning that the particular page being referenced has not been mapped and is not resident in the main memory. This will induce a *page fault* and require accessing disk storage to retrieve the missing page. This constitutes a program exception and will necessitate the suspension of the execution of the current program.

FIGURE 49 Processing of load/store instructions.



After successful address translation in the second pipe stage, a load instruction accesses the data memory during the third pipe stage. At the end of this machine cycle, the data is retrieved from the data memory and written into either the register rename buffer or the reorder buffer. At this point the load instruction finishes execution. The updating of the architected register is not performed until when this load instruction is completed from the reorder buffer. Here we assume that data memory access can be done in one machine cycle in the third pipe stage. This is possible if a data cache is employed. (Subsection 3.2.3.2 provides more background material on caches.) With a data cache, it is possible that the data being loaded is not resident in the data cache. This will result in a data *cache miss* and require the filling of the data cache from the main memory. Such cache misses can necessitate the stalling of the load/store pipeline.

Store instructions are processed some what differently than load instructions. Unlike a load instruction, a store instruction is considered *finished* with execution at the end of the second pipe stage when there is a successful translation of the address. The register data to be stored to memory is kept in the reorder buffer. At the time when the store is being completed, this data is then written out to memory. The reason for this delayed access to memory is to prevent the premature and potentially erroneous update of the memory in case the store instruction may have to be flushed due to the occurrence of an exception or a branch misprediction. Since load instructions only read the memory, their flushing will not result in unwanted side effects affecting the memory state.

For a store instruction, instead of updating the memory at completion, it is also possible to move the data to the store buffer at completion. The store buffer is a FIFO queue that buffers architecturally completed store instructions. Each of these store instructions is then retired, i.e. updates the memory, when the memory bus is available. The purpose of the store buffer is to allow stores to be retired when the memory bus is not busy, and thus giving priority to loads that need to access the memory bus. We use the term *completion* to refer to the updating of the CPU state and the term *retiring* to the updating of the memory state. With the store buffer, a store instruction can be architecturally complete but not yet retired to memory. When a program exception occurs, the instructions that follow the excepting instruction and may have finished out of order, must be flushed from the reorder buffer; however the store buffer must be drained, i.e. the store instructions in the store buffer must be retired, before the excepting program can be suspended.

We have assumed that both address translation and memory accessing can be done in one machine cycle. Such latency usually requires the use of TLBs and caches. A brief review of cache memory and virtual memory implementation is provided in the next subsection.

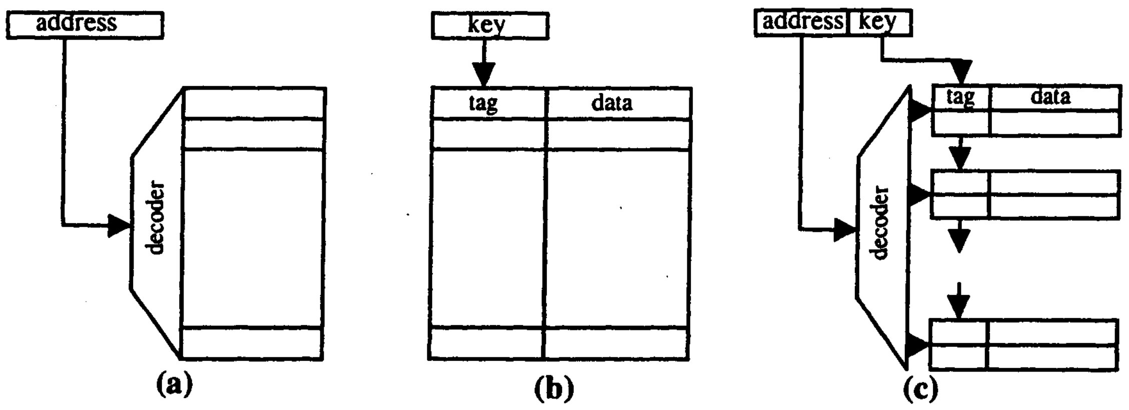
3.2.3.2 Memory Hierarchy Revisited

Before discussing specific memory data flow techniques, we first review the fundamentals of cache memory and virtual memory from the implementation perspective. Four topics are covered: memory accessing mechanisms, cache memory implementations, TLB implementations, and interaction between cache memory and TLB.

There are two fundamental ways to access a multi-entry memory: indexing via an address or associative search via a tag. An *indexed memory* uses an address to index into the memory to select a particular entry; see Figure 50a. A decoder is used to decode the n-bit address in order to enable one of the 2^n entries for reading or writing that entry. There is a rigid mapping of an address to the data which requires the data to be stored in a fixed entry in the memory. Indexed memory is rigid in this mapping but less complex to implement. In contrast, an *associative mem-*

ory uses a key to search through the memory to select a particular entry; see Figure 50b. Each entry of the memory has a tag field and a comparator that compares the content of its tag field to the key. When a match occurs that entry is selected. Using this form of associative search allows the data to be flexibly stored in any location of the memory. This flexibility comes at the cost of implementation complexity. A compromise between the indexed memory and the associative memory is the *set associative memory* which uses both indexing and associative search; see Figure 50c. An address is used to index into one of the sets while the multiple entries within a set are searched with a key to identify one particular entry. This compromise provides some flexibility in the placement of data without incurring the complexity of a fully associative memory.

FIGURE 50 Memory accessing mechanisms: (a) indexed memory; (b) (fully) associative memory; (c) set associative memory.



Memory hierarchy is used to provide an overall memory subsystem with both high capacity and low latency. The lower level memory is usually larger but slower, and the higher level memory is smaller but much faster. By leveraging locality of memory references, such a memory hierarchy can provide performance close to a very large and fast memory without the cost and complexity. Cache memory constitutes the higher level memory to the lower level main memory.

The main memory is normally implemented as a large indexed memory. However a cache memory can be implemented using any one of the three memory accessing schemes shown in Figure 50. When a cache memory is implemented as an indexed memory it is referred to as a *direct-mapped cache* (illustrated in). Since the direct-mapped cache is smaller and has fewer entries than the main memory, it requires fewer address bits and its smaller decoder can only decode a subset of the main memory address bits. Consequently, many main memory addresses can be mapped to the same entry in the cache. In order to ensure the selected entry contains the correct data, the remaining, i.e. not decoded, address bits must be used to identify the selected entry. Hence in addition the data field, each entry has an additional tag field for storing these undecoded bits. When an entry of the cache is selected, its tag field is accessed and compared with the undecoded bits of the original address to ensure that the entry contains the data being addressed.

FIGURE 51 Direct mapped caches: (a) single-word per block; (b) multi-word per block.

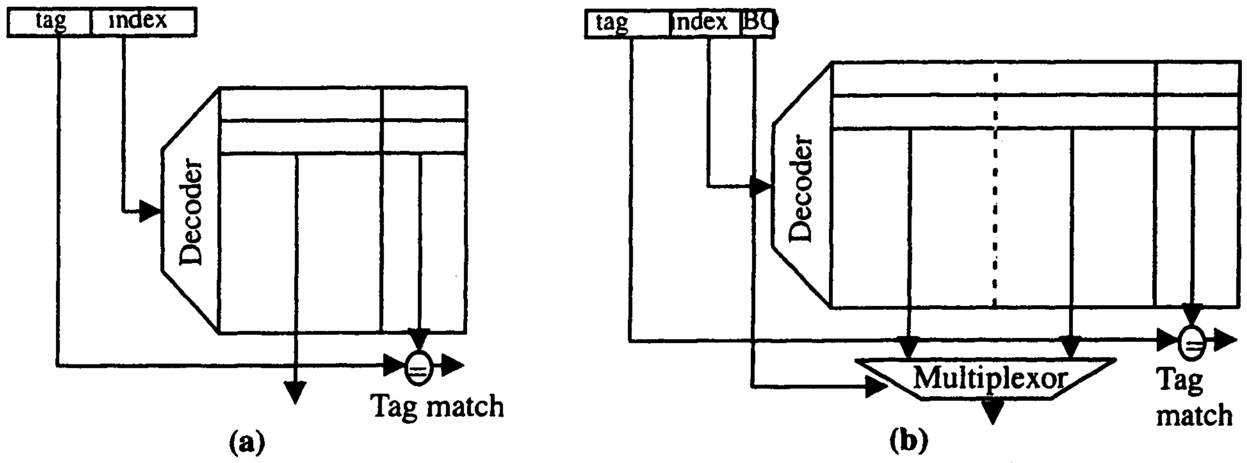


Figure 51a illustrates a direct mapped cache with each entry, or block, containing one word. In order to take advantage of spatial locality, the block size of a cache can contain multiple words as shown in Figure 51b. With a multi-word block, some of the bits from the original address are used to select the particular word being referenced. Hence the original address is now partitioned into three portions: the index bits are used to select an entry; the block offset bits are used to select a word within a selected block, and the tag bits are used to do tag match against the tag stored in the tag field of the selected entry.

Cache memory can also be implemented as a fully associative or a set-associative memory as shown in Figure 52 and Figure 53, respectively. Fully associative caches have the greatest flexibility in terms of the placement of data in the entries of the cache. Other than the block offset bits, all the other address bits are used as a key for associatively searching all the entries of the cache. This full associativity facilitates the most efficient use of all the entries of the cache, but incurs the greatest implementation complexity. Set associative caches permits the flexible placement of data among all the entries of a set. The index bits select a particular set, the tag bits select an entry within the set, and the block offset bits selects the word within the selected entry. The partitioning of the original address bits into these three categories is a result of careful trade-offs.

FIGURE 52 Fully associative cache.

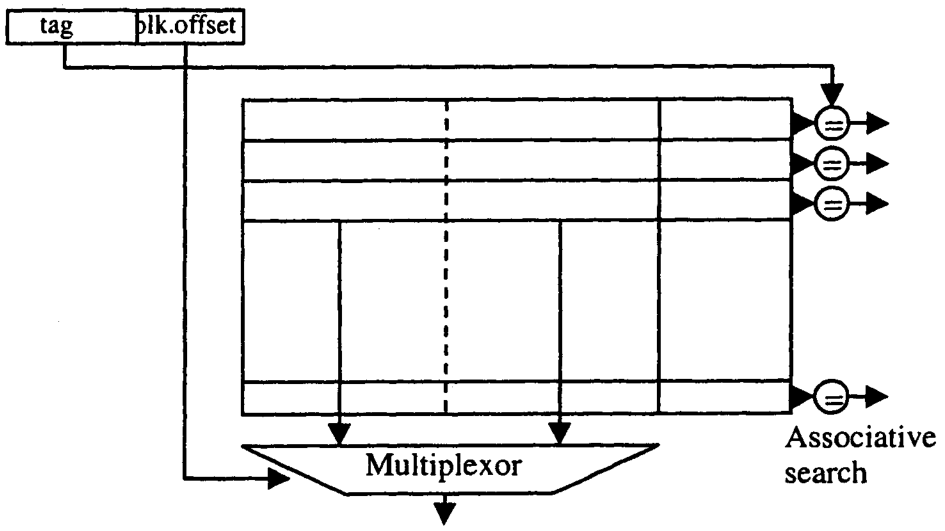
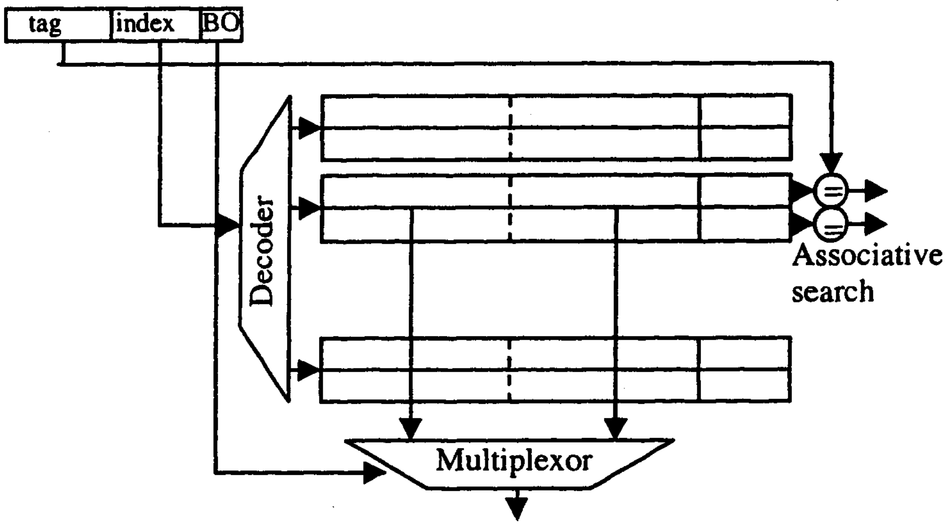


FIGURE 53 Set-associative cache.



Inherent to virtual memory is the mapping of the virtual address space to the physical address space. This requires the translation of the virtual address into the physical address. Instead of directly accessing the main memory with the address generated by the processor, the virtual address generated by the processor must first be translated into a physical address. The physical address is then used to access the physical main memory as shown in Figure 54.

FIGURE 54 Virtual to physical address translation.

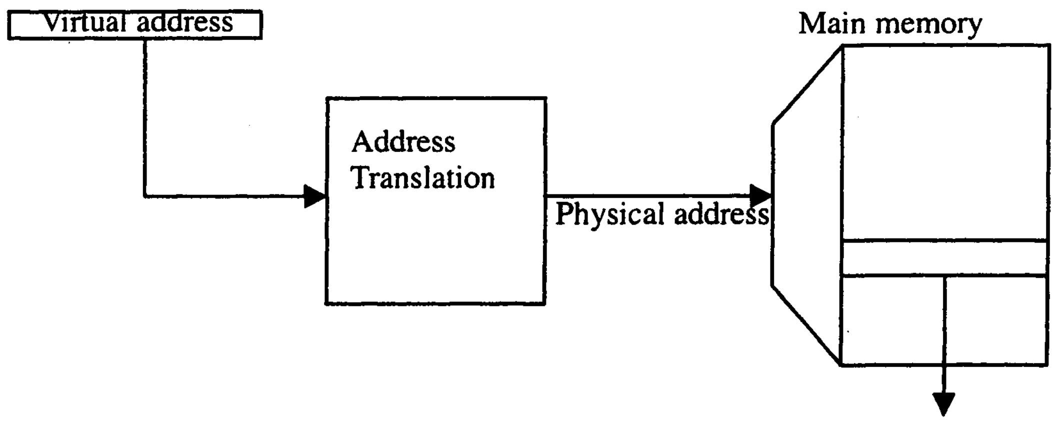
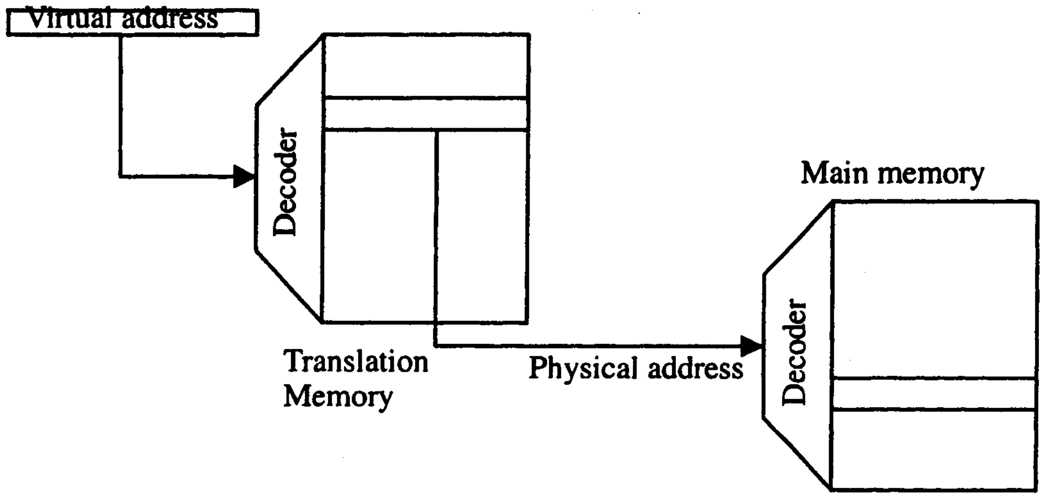


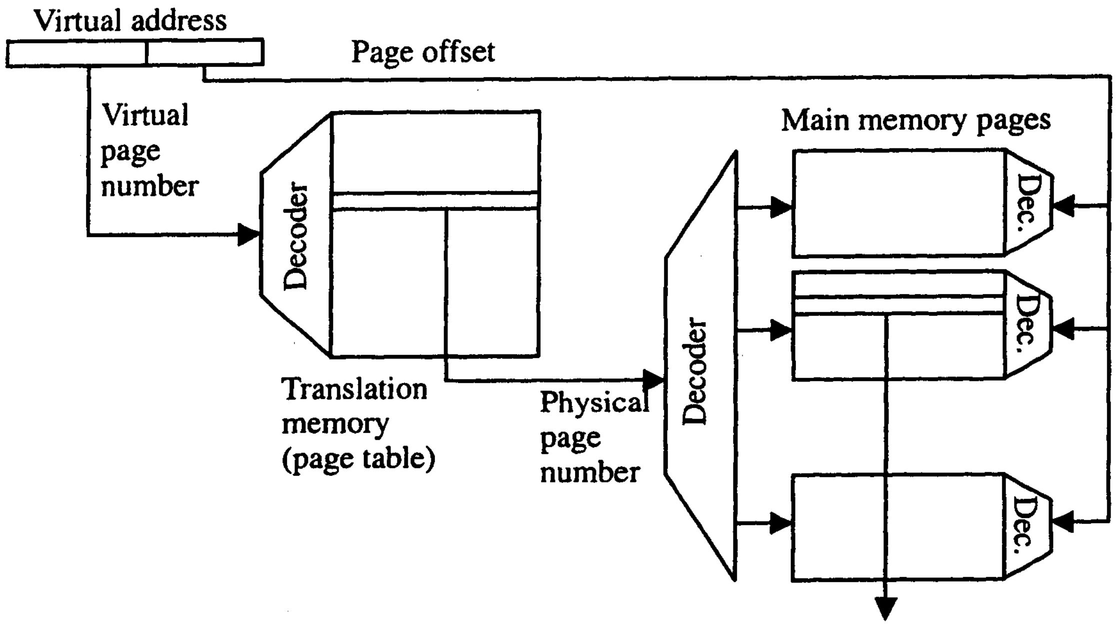
FIGURE 55 Translation of virtual word address to physical word address using a translation memory.



Address translation can be done using a translation memory. The virtual address is used to index into the translation memory. The data retrieved from the selected entry of the translation memory is then used as the physical address to index the main memory. Hence physical addresses that correspond to the virtual addresses are stored in the corresponding entries of the translation memory. Figure 55 illustrates the use of a translation memory to translate word addresses, i.e. it maps a virtual address of a word in the virtual address space into a physical address of a word in the physical main memory.

There are two weaknesses to the translation memory scheme shown in Figure 55. First, translation of word addresses will require a translation memory with the same number of entries as the main memory. This can result in doubling the size of the physical main memory. Translation can be done at coarser granularity. Multiple (usually in powers of 2) words in the main memory can be grouped together into a *page*, and only addresses to each page need to be translated. Within the page, words can be selected using the lower order bits of the virtual address, i.e. page offset bits, directly without requiring translating them. This is illustrated in Figure 56. With a paging system, the translation memory is called the *page table*.

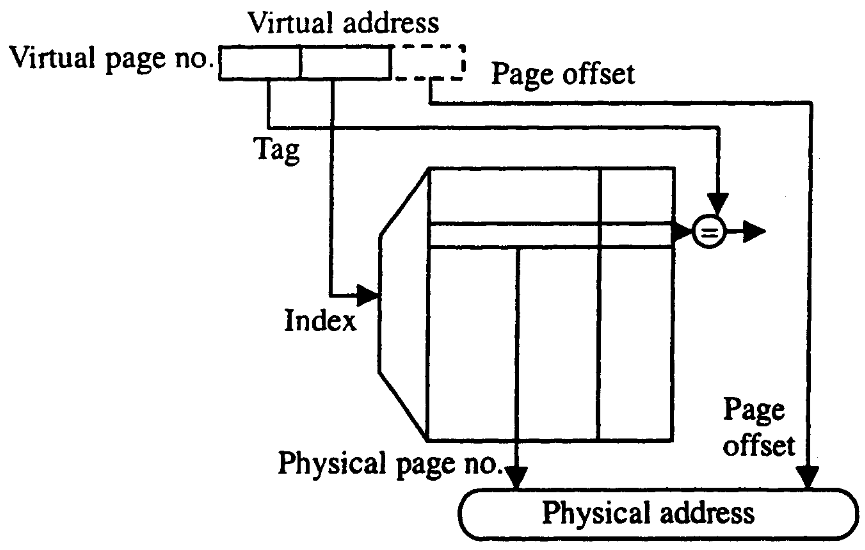
FIGURE 56 Translation of virtual page address to physical page address using a translation memory.



The second weakness of the translation memory scheme is the fact that two memory accesses are required for every main memory reference by an instruction. First the page table must be accessed to obtain the physical page number, then the physical main memory can be accessed using the translated physical page number along with the page offset. In actual implementations the page table is typically stored in the main memory (usually in the portion of main memory allocated to the operating system), hence every reference to memory by an instruction requires two sequential accesses to the physical main memory. This can become a serious bottleneck to performance. The solution is to implement the page table using a fast cache memory.

Translation lookaside buffer (TLB) is essentially a cache memory for the page table. Just like any other cache memory, the TLB can be implemented using any one of the three memory accessing schemes of Figure 50. A direct mapped TLB is simply a smaller (and faster) version of the page table. The virtual page number is partitioned into an index for the TLB and a tag; see Figure 57. The virtual page number is translated into the physical page number, which is concatenated with the page offset to form the physical address.

FIGURE 57 Direct mapped TLB.

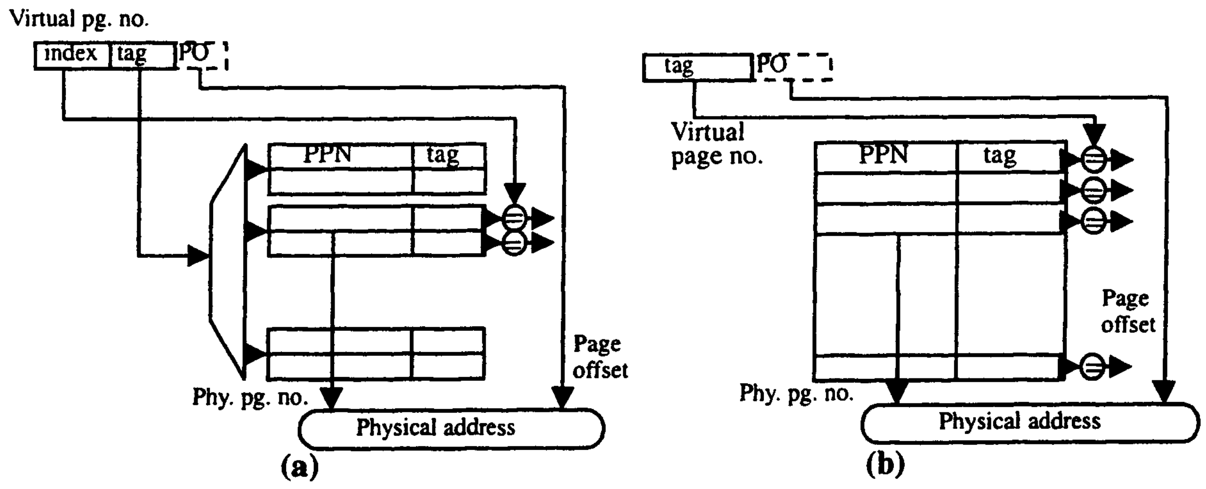


To ensure more flexible and efficient use of the TLB entries, associativity is usually added to the TLB implementation. Figure 58 illustrates the set associative and fully associative TLBs. For the set associative TLB, the virtual address bits are partitioned into three fields: index, tab and page offset. The size of the page offset field is dictated by the page size which is specified by the architecture and the operating system. The remaining fields, i.e. index and tag, constitute the virtual page number. For the fully associative TLB, the index field is missing, and the tag field contains the virtual page number.

Caching a portion of the page table into the TLB allows fast address translation; however, *TLB misses* can occur. All of the virtual page to physical page mappings in the page table cannot be simultaneously present in the TLB. When accessing the TLB a cache miss can occur, in which case the TLB must be filled from the page table sitting in the main memory. This can incur a number of stall cycles in the pipeline. It is also possible that a TLB miss can lead to a *page fault*.

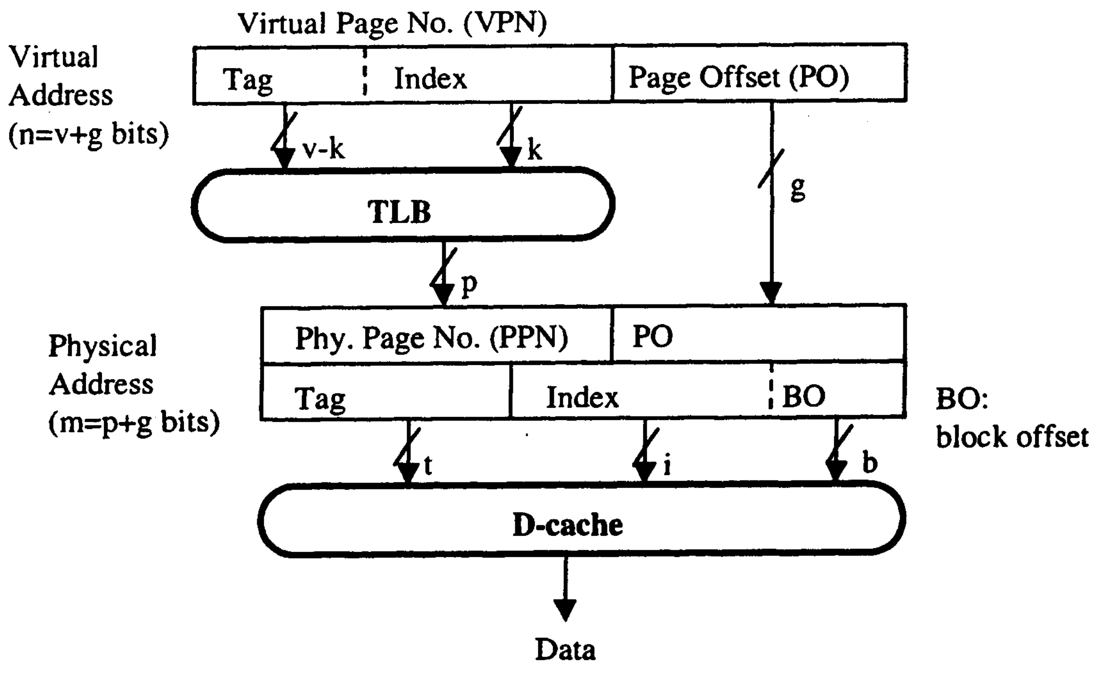
A page fault occurs when the virtual page to physical page mapping does not even exist in the page table. This means that the particular page being referenced is not resident in the main memory and must be fetched from secondary storage. To service a page fault requires accessing the disk storage and can require potentially tens of thousands of machine cycles. Hence, when a page fault is triggered by a program, that program is suspended from execution until the page fault is serviced by the operating system.

FIGURE 58 Associative TLBs: (a) set associative TLB; (b) fully associative TLB.



A data cache is used to cache a portion of the main memory; a TLB is used to cache a portion of the page table. The interaction between the TLB and the data cache is illustrated in Figure 59. Correlating back to the load/store unit pipeline of Figure 49, the n-bit virtual address shown in Figure 59 is the effective address generated by the first pipe stage. This virtual address consists of a virtual page number (v bits) and a page offset (g bits). If the TLB is a set associative cache, the v bits of the virtual page number is further split into a k-bit index and a (v-k)-bit tag. The second pipe stage of the load/store unit corresponds to the accessing of the TLB using the virtual page number. Assuming there is no TLB miss, the TLB will output the physical page number (p bits), which is then concatenated with the g-bit page offset to produce the m-bit physical address where $m = p+g$ and not necessarily equal to n. During the third pipe stage the m-bit physical address is used to access the data cache. The exact interpretation of the m-bit physical address depends on the design of the data cache. If the data cache block contains multiple words, then the lower order b bits are used as a block offset to select the referenced word from the selected block. The selected block is determined by the remaining (m-b) bits. If the data cache is a set associative cache, then the remaining (m-b) bits are split into an t-bit tag and an i-bit index. The value of i is determined by the total size of the cache and the set associativity, i.e. there should be i sets in the set associative data cache. If there is no cache miss, then at the end of the third pipe stage (assuming the data cache can be accessed in a single cycle) the data will be available from the data cache (assuming a load instruction is being executed).

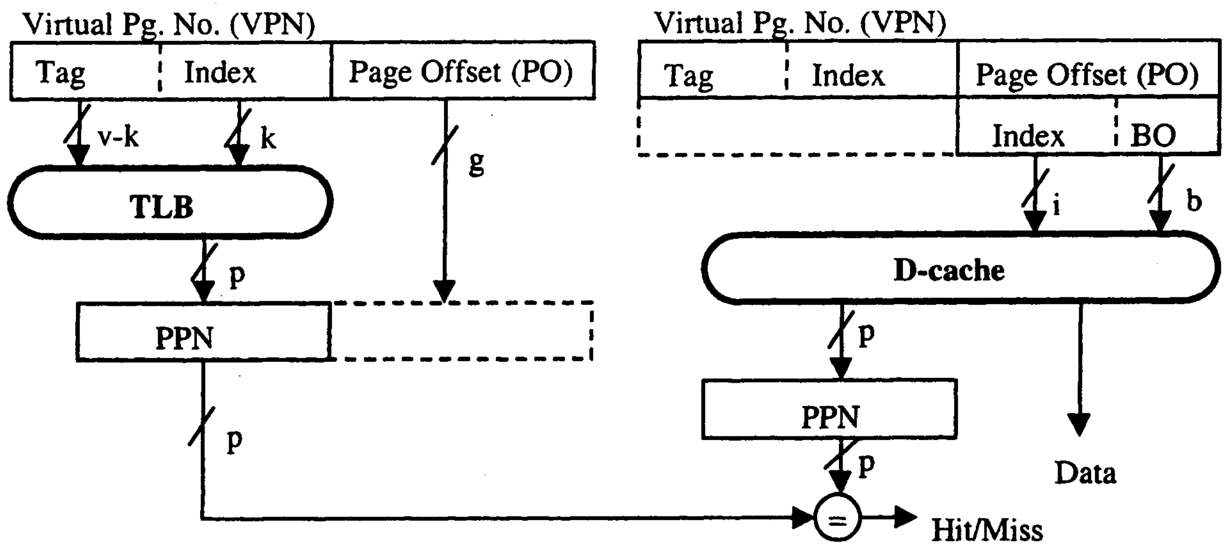
FIGURE 59 Interaction between the TLB and the data cache.



The organization shown in Figure 59 has a disadvantage because the TLB must be accessed before the data cache can be accessed. Serializing of the TLB and data cache accesses introduces an overall latency that is the sum of the two latencies. Hence, the reason for assuming that address translation and memory access are done in two separate pipe stages in Figure 49. The solution to this problem is to use a *virtually indexed* data cache that allows the accessing of the TLB and the data cache to be performed in parallel. Figure 60 illustrates such a scheme.

A straight-forward way to implement a virtually indexed data cache is to use only the page offset bits to access the data cache. Since the page offset bits do not require translation, they can be used without translation. The g bits of the page offset can be used as the block offset (b bits) and the index (i bits) fields in accessing the data cache. For simplicity, let's assume that the data cache is a direct mapped cache of 2^i entries with each entry, or block, containing 2^b words. Instead of storing the remaining bit of the virtual address, i.e. the virtual page number, as its tag field, the data cache can store the translated physical page number in its tag field. This is done at the time when a data cache line is filled. At the same time when the page offset bits are being used to access the data cache, the remaining bit of the virtual address, i.e. the virtual page number, are used to access the TLB. Assuming the TLB and data cache access latencies are comparable, at the time when the physical page number from the TLB becomes available, the tag field (also containing the physical page number) of the data cache will also be available. The two p-bit physical page numbers can then be compared to determine whether there is a hit in the data cache or not. With a virtually indexed data cache, address translation and data cache access can be overlapped to reduce the overall latency.

FIGURE 60 Virtually indexed data cache.



3.2.3.3 Ordering of Memory Accesses

A memory data dependence exists between two load/store instructions if they both reference the same memory location, i.e. there exists an *aliasing*, or collision, of the two memory addresses. A load instruction performs a read from a memory location, while a store instruction performs a write to a memory location. Similar to register data dependences, read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependences can exist between load and store instructions. A store (load) instruction followed by a load (store) instruction involving the same memory location will induce a RAW (WAR) memory data dependence. Two stores to the same memory location will induce a WAW dependence. These memory data dependences must be enforced in order to preserve the correct semantics of the program.

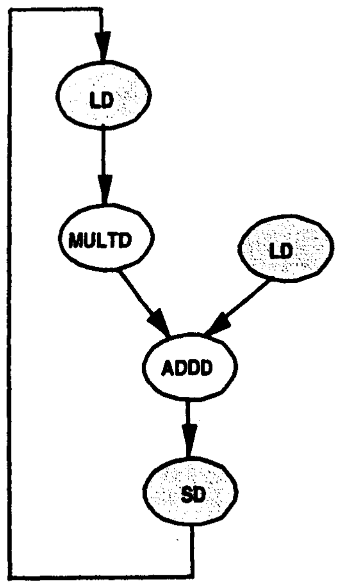
One way to enforce memory data dependences is to execute all load/store instructions in program order. Such total ordering of memory instructions is sufficient for enforcing memory data dependences but not necessary. It is conservative and can impose an unnecessary limitation on the performance of a program. We use the example in Figure 61 to illustrate this point. DAXPY is the name of a piece of code that multiplies an array by a coefficient and then add the array to another array. DAXPY (derived from "Double precision A times X Plus Y") is a kernel in the LINPAC routines <<??>> and is commonly found in many numerical programs. Notice that all the iterations of this loop are data independent and can be executed in parallel. However, if we impose the constraint that all load/store instructions must be executed in total order, then the first load instruction of the second iteration cannot begin until the store instruction of the first iteration is performed. This constraint will effectively serialize the execution of all the iterations of this loop.

FIGURE 61 The "DAXPY" example [H&P pg. 357]

Y(i) = A * X(i) + Y(i)

```
    LD      F0, a
    ADDI    R4, Rx, #512        ; last address

Loop:
    LD      F2, 0(Rx)           ; load X(I)
    MULTD   F2, F0, F2          ; A*X(I)
    LD      F4, 0(Ry)           ; load Y(I)
    ADDD    F4, F2, F4          ; A*X(I) + Y(I)
    SD      F4, 0(Ry)           ; store into Y(I)
    ADDI    Rx, Rx, #8          ; inc. index to X
    ADDI    Ry, Ry, #8          ; inc. index to Y
    SUB     R20, R4, Rx         ; compute bound
    BNZ     R20, loop           ; check if done
```



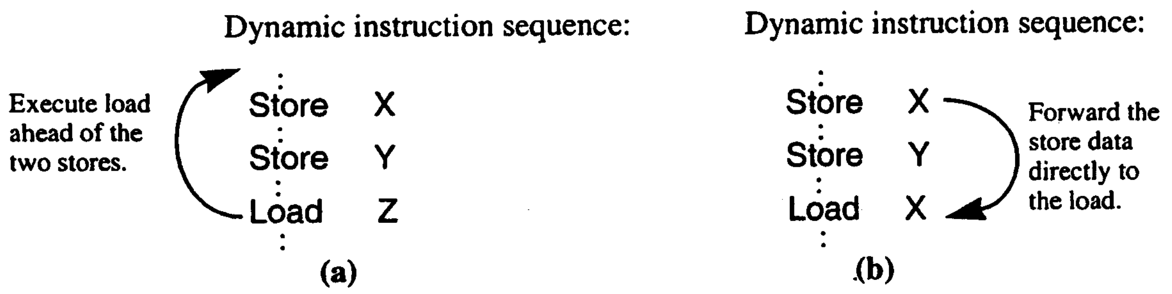
By allowing load/store instructions to execute out of order, without violating memory data dependences, performance gain can be achieved. Take the example of the DAXPY loop. The graph in Figure 61 represents the true data dependences involving the core instructions of the loop body. These dependences exist among the instructions of the same iteration of the loop. There are no data dependences between multiple iterations of the loop. The loop closing branch instruction is highly predictable; hence the fetching of subsequent iterations can be done very quickly. The same architected registers specified by instructions from subsequent iterations are dynamically renamed by register renaming mechanisms; hence there is no register dependences between the iterations due to the dynamic reuse of the same architected registers. Consequently if load/store instructions are allowed to execute out of order, the load instructions from a trailing iteration can begin before the execution of the store instruction from an earlier iteration. By overlapping the execution of multiple iterations of the loop, performance gain is achieved for the execution of this loop.

Memory models impose certain limitation on the out of order execution of load/store instructions by a processor. First, in order to facilitate recovery from exceptions, the sequential state of the memory must be preserved. In other words, the memory state must evolve according to the sequential execution of load/store instructions. Second, most shared memory multiprocessor systems assume the sequential consistency memory model, which requires that the accessing of the shared memory by each processor be done according to program order. Both of these reasons effectively require that store instructions must be executed in program order, or at least the memory must be updated as if stores are performed in program order. If stores are required to execute in program order, WAW and WAR memory data dependences are implicitly enforced and are not an issue. Hence, only RAW memory data dependences must be enforced.

3.2.3.4 Load Bypassing and Load Forwarding

Out-of-order execution of load instructions is the primary source for potential performance gain. As can be seen in the DAXPY example, load instructions are frequently at the beginning of dependence chains and their early execution can facilitate the early execution of other dependent instructions. While relative to memory data dependences, load instructions are viewed as performing read operations on the memory locations, they are actually performing write operations to their destination registers. With loads being register-defining (DEF) instructions, they are typically followed immediately by other dependent register-use (USE) instructions. The goal is to allow load instructions to begin execution as early as possible, possibly jumping ahead of other preceding store instructions, as long as RAW memory data dependences are not violated and that memory is updated according to the sequential memory consistency model.

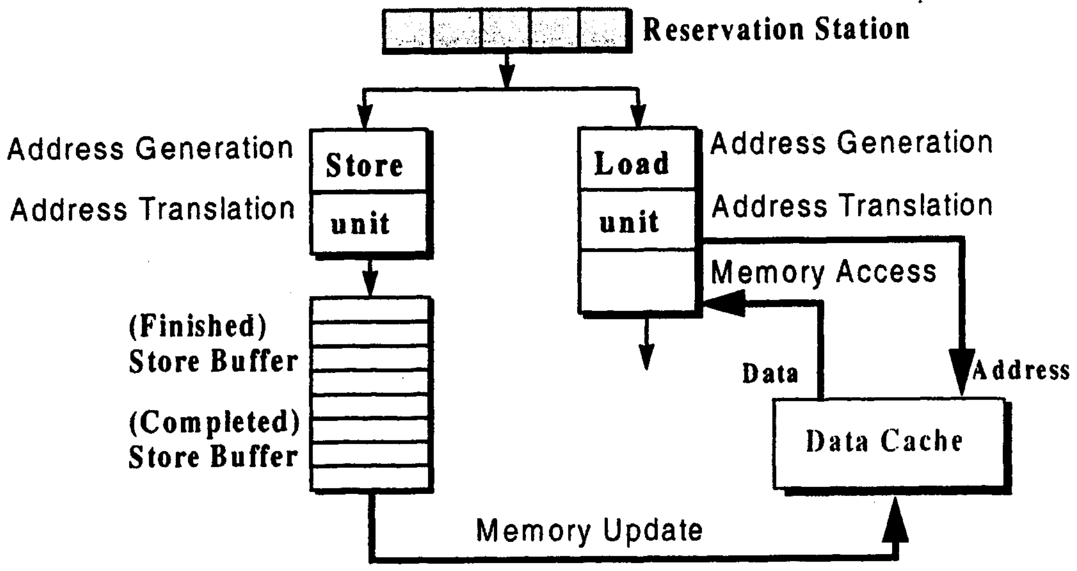
FIGURE 62 Early execution of load instructions: (a) load bypassing; (b) load forwarding.


Dynamic instruction sequence:
Execute load ahead of the two stores.
Store X
Store Y
Load Z
(a)
Dynamic instruction sequence:
Store X
Store Y
Load X
Forward the store data directly to the load.
(b)

Two specific techniques for early out-of-order execution of loads are: *load bypassing* and *load forwarding*. As shown in Figure 62a, load bypassing allows a trailing load to be executed earlier than preceding stores if the load address does not alias with the preceding stores, i.e. there is no memory data dependence between the stores and the load. On the other hand, if a trailing load aliases with a preceding store, i.e. there is a RAW dependence from the store to the load, load forwarding allows the load to receive its data directly from the store without having to access the data memory; see Figure 62b. In both of these cases, earlier execution of a load instruction is achieved.

Before we discuss the issues that must be addressed in order to implement load bypassing and load forwarding, we first present the organization of the portion of the execution core responsible for processing load/store instructions. This organization, shown in Figure 63, is used as the vehicle for our discussion on load bypassing and load forwarding. There is one store unit (2 pipe stages) and one load unit (3 pipe stages); both are fed by a common reservation station. For now we assume that load and store instructions are issued from this shared reservation station in program order. The store unit is supported by a store buffer. The load unit and the store buffer can access the data cache.

FIGURE 63 **Mechanisms for load/store processing: separate load and store units with in order issuing from a common reservation station.**

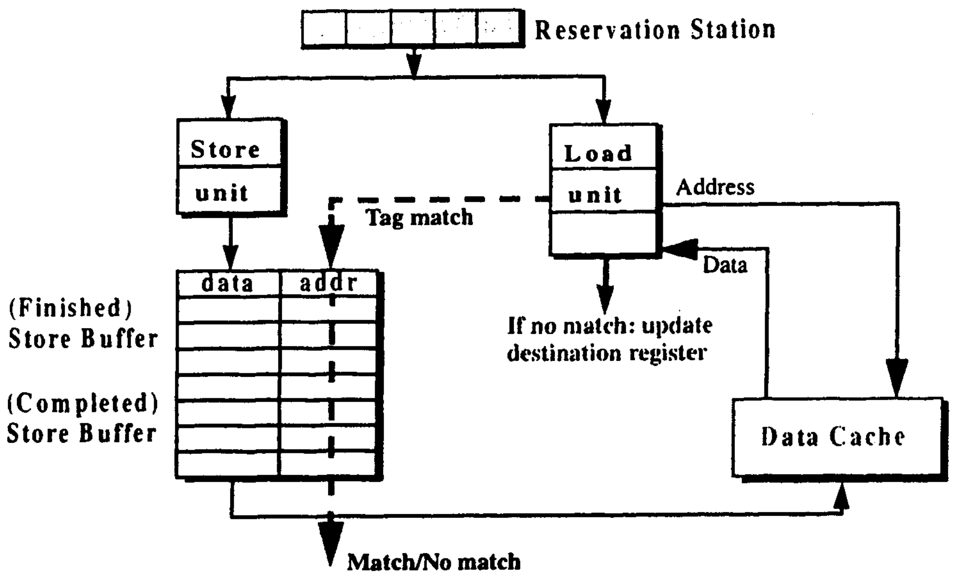


Given the organization of Figure 63, a store instruction can be in one of several states while it is in flight. When a store instruction is dispatched to the reservation station, an entry in the reorder buffer is allocated for it. It remains in the reservation station until all of its source operands become available and it is issued into the pipelined execution unit. Once the memory address is generated and successfully translated, it is considered to have finished execution and is placed into the finished portion of the store buffer (the reorder buffer is also updated). The store buffer operates as a queue and has two portions, *finished* and *completed.* The finished portion contains those stores that have finished execution but are not yet architecturally completed. The completed portion of the store buffer contains those stores that are completed architecturally but waiting to updated the memory. The identification of the two portions of the store buffer can be done via a pointer to the store buffer or a status bit in the store buffer entries. A store in the finished portion of the store buffer can potentially be speculative and when a misspeculation is detected it will need to be flushed from the store buffer. When a finished store is completed by the reorder buffer, it changes from the finished state to the completed state. This can be done by updating the store buffer pointer or flipping the status bit. When a completed store finally exits the store buffer and updates the memory it is considered retired. Viewing from the perspective of the memory state, a store does not really finish its execution until it is retired. When an exception occurs, the stores in the completed portion of the store buffer must be drained in order to appropriately update the memory. So between being dispatched and retired, a store instruction can be in one of three states: *issued* (in the execution unit), *finished* (in the finished portion of the store buffer), and *completed* (in the completed portion of the store buffer).

One key issue in implementing load bypassing is the need to check for possible aliasing with preceding stores, i.e. those stores being bypassed. A load is considered to bypass a preceding store, if the load reads from the memory before the store writes to the memory. Hence before such a load is allowed to execute or read from the memory, it must be determined that it does not alias with all the preceding stores that are still in flight, i.e. those that have been issued but not retired. Assuming in-order issuing of load/store instructions from the load/store reservation station, all such stores should be sitting in the store buffer, in both the finished and the completed portions. This alias checking for possible dependence between the load and the preceding store can be done using the store buffer. A tag field containing the memory address of the store is incorporated with each entry of the store buffer. Once the memory address of the load is available, this address can be used to perform an associative search on the tag field of the store buffer entries. If a match occurs, then aliasing exists and the load is not allowed to execute out of order. Otherwise, the load is independent of the preceding stores in the store buffer and can be executed ahead of them. This associative search can be performed in the third pipe stage of the load unit concurrent with the accessing of the data cache. If no aliasing is detected the load is allowed to finish and the corresponding renamed destination register is updated with the data returned from the data cache. If aliasing is detected the data returned by the data cache is discarded and the load is held back in the reservation station for future reissue.

Most of the complexity in implementing load bypassing is the store buffer and the associated associative search mechanism. To reduce the complexity, the tag field used for associative search can be reduced to contain only a subset of the address bits. Using only a subset of the address bits can reduce the width of the comparators needed for associative search. However, the result can be pessimistic. Potentially, aliasing can be indicated by the narrower comparators when it really doesn't exist if the full length address bits were used. Some of the load bypassing opportunities can be lost due to this optimization of the implementation. In general, the degradation of performance is minimal if enough address bits are used.

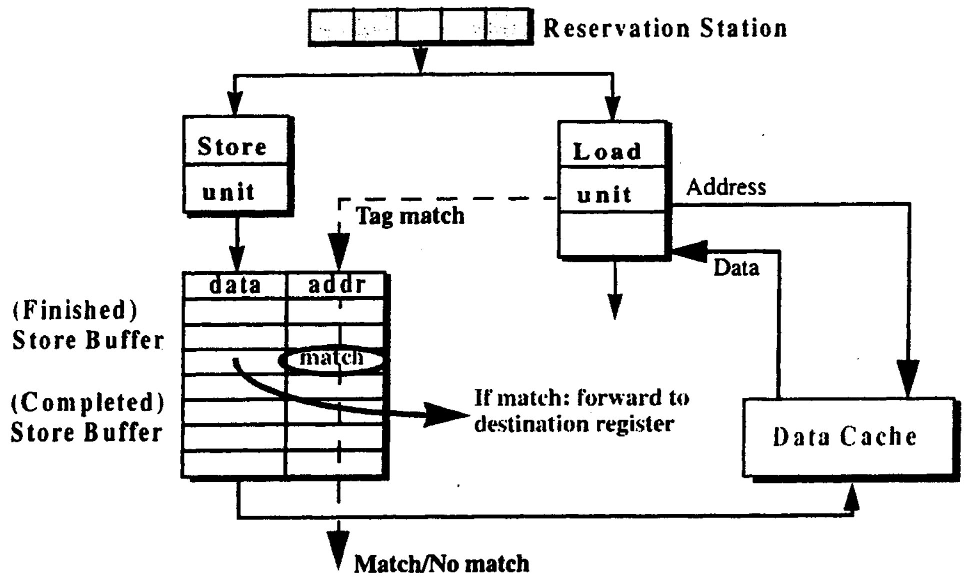
FIGURE 64 Illustration of load bypassing.



Load forwarding technique further enhances and complements the load bypassing technique. When a load is allowed to jump ahead of preceding stores, if it is determined that the load aliases with a preceding store, there is the potential to satisfy that load by forwarding the data directly from the aliased store. Essentially a memory RAW dependence exists between the leading store and the trailing load. The same associative search of the store buffer is needed. When aliasing is detected, instead of holding the load back for future reissue, the data from the aliased entry of the store buffer is forwarded to the renamed destination register of the load instruction. This technique not only allows the load to be executed early, it also eliminates the need for the load to access the data cache. This can reduce the bandwidth pressure on the bus to the data cache.

To support load forwarding added complexity to the store buffer is required. First, the full length address bits must be used for performing the associative search. When a subset of the bits is used for supporting load bypassing, the only negative consequence is lost opportunity. For load forwarding the alias detection must be exact before forwarding of data can be performed, otherwise it will lead to semantic incorrectness. Second, there can be multiple preceding stores in the store buffer that will alias with the load. When such multiple matches occur during the associative search, there must be logic added to determine which of the aliased stores is the most recent. This will require additional priority encoding logic to identify the latest store to which the load is dependent on before forwarding is performed. Third, an additional read port may be required for the store buffer. Prior to the incorporation of load forwarding, the store buffer has one write port that interfaces with the store unit and one read port that interfaces with the data cache. A new read port is now required that interfaces with the load unit.

FIGURE 65 Illustration of load forwarding.

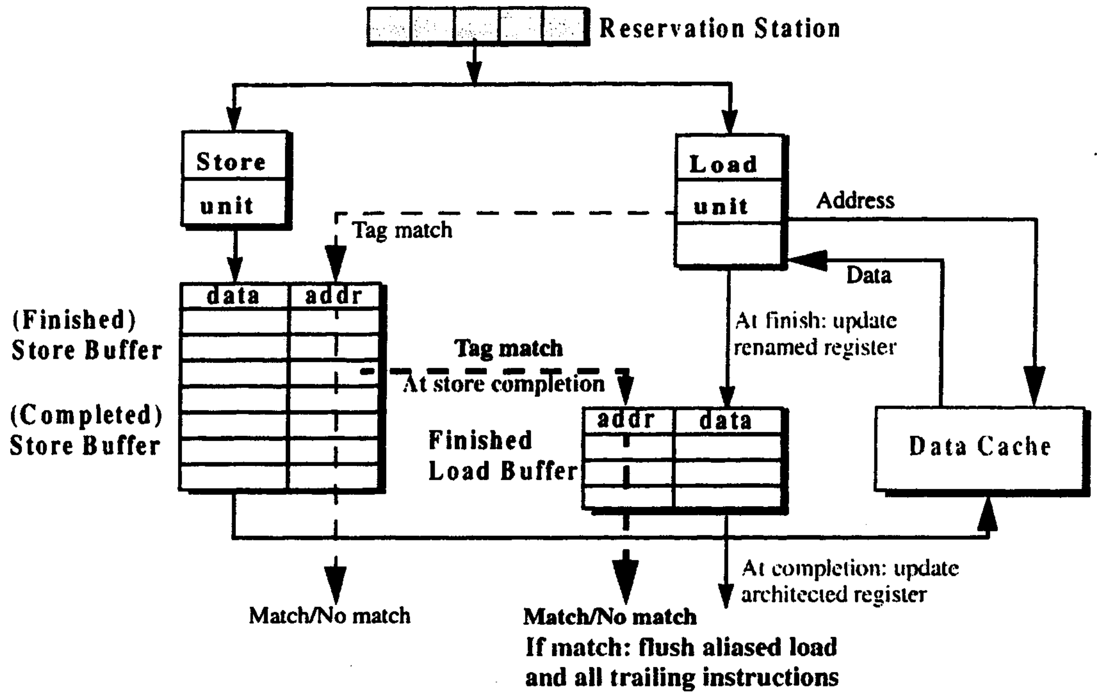


Significant performance improvement can be obtained with load bypassing and load forwarding. According to Mike Johnson [ ], typically load bypassing can yield 11%-19% performance gain and load forwarding can yield another 1%-4% of additional performance improvement.

So far we have assumed that loads and stores share a common reservation station with instructions being issued from the reservation station to the store and the load units in program order. This in-order issuing assumption ensures that all the preceding stores to a load will be in the store buffer when the load is executed. This simplifies memory dependence checking; only an associative search of the store buffer is necessary. However this in-order issuing assumption introduces a unnecessary limitation on the out-of-order execution of loads. A load instruction can be ready to be issued, however a preceding store can hold up the issuing of the load even though the two memory instructions do not alias. Hence allowing out-of-order issuing of loads and stores from the load/store reservation station can permit a greater degree of out-of-order and early execution of loads. This is especially beneficial if these loads are at the beginnings of critical dependence chains and their early execution can remove critical performance bottlenecks.

If out-of-order issuing from the load/store reservation station is allowed, a new problem must be solved. If a load is allowed to be issued out of order, then it is possible for some of the stores that precede it to still be in the reservation or in the execution pipe stages, and not yet in the store buffer. Hence, simply performing an associative search on the entries of the store buffer is not adequate for checking for potential aliasing between the load and all its preceding stores. Worse yet, the memory addresses for these preceding stores that are still in the reservation station or in the execution pipe stages may not be available yet.

FIGURE 66 Fully out-of-order issuing and execution of load and store instructions.

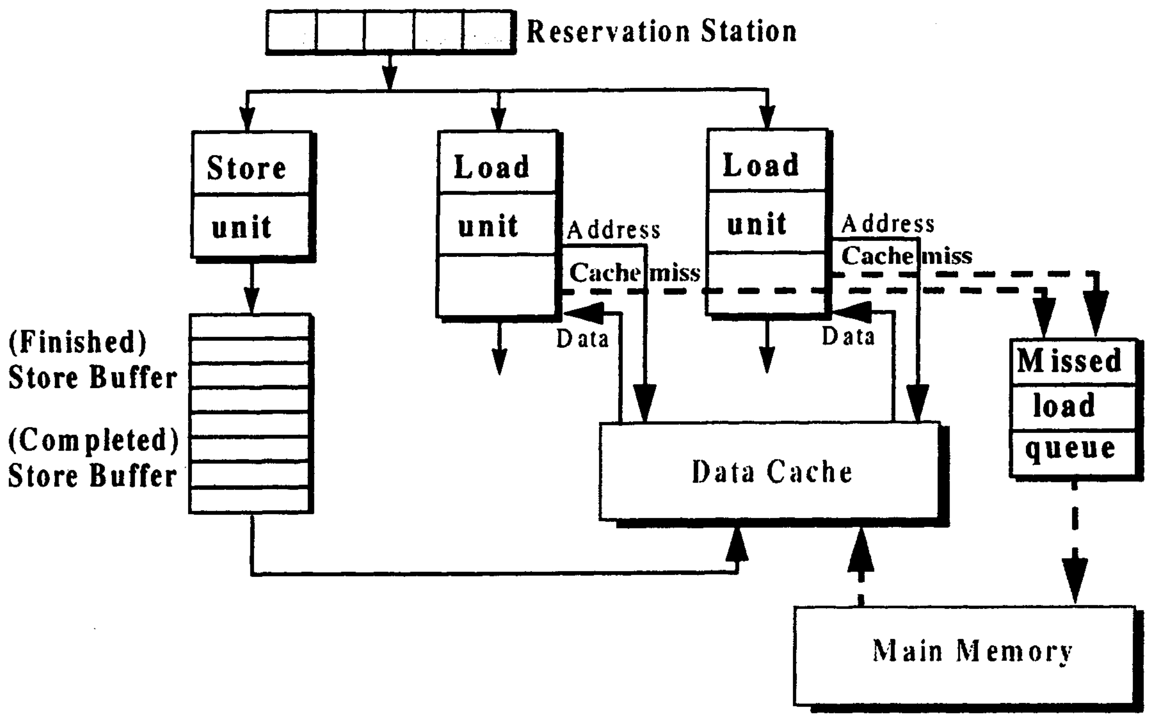


One approach is to allow the load to proceed assuming no aliasing with the preceding stores that are not yet in the store buffer and then validate this assumption later. With this approach a load is allowed to issue out of order and be executed speculatively. If it does not alias with any of the stores in the store buffer, the load is allowed to finish execution. However, this load must be put into a new buffer called the finished load buffer; see Figure 66. The finished load buffer is managed in a similar fashion as the finished store buffer. A load is only resident in the finished load buffer after it finishes execution and before it is completed. Whenever a store instruction is being completed, it must perform alias checking against the loads in the finished load buffer. If no aliasing is detected, the store is allowed to complete. If aliasing is detected, then it means that there is a trailing load that is dependent on the store and that load has already finished execution. This implies that the speculative execution of that load must be invalidated and corrected by reissuing, or even refetching, that load and all subsequent instructions. This can require significant hardware complexity and performance penalty.

3.2.3.5 Other Memory Data Flow Techniques

Other than load bypassing and load forwarding, there are other memory data flow techniques. These techniques all have the objective of increasing the memory bandwidth and/or reducing the memory latency. As superscalar processors get wider, greater memory bandwidth capable of supporting multiple load/store instructions per cycle will be needed. As the disparity between processor speed and memory speed continues to increase, the latency of accessing memory, especially when cache misses occur, will become a serious bottleneck to machine performance.

FIGURE 67 Dual-ported and non-blocking data cache.



One way to increase memory bandwidth is to employ multiple load/store units in the execution core supported by a multi-ported data cache. In Subsection 3.2.3.4 we have assumed the presence of one store unit and one load unit supported by a single-ported data cache. The load unit has priority in accessing the data cache. Store instructions are queued in the store buffer and are retired from the store buffer to the data cache whenever the memory bus is not busy and the store buffer can gain access to the data cache. The overall data memory bandwidth is limited to one load/store instruction per cycle. This is a serious limitation, especially when there are bursts of load instructions. One way to alleviate this bottleneck is to provide two load units as shown in Figure 67 and a *dual-ported data cache.* A dual-ported data cache is able to support two simultaneous cache accesses in every cycle. This will double the potential memory bandwidth. However it comes with the cost of hardware complexity; a dual-ported cache can require doubling of the cache hardware. One way to alleviate this hardware cost, is to implement interleaved data cache banks. With the data cache being implemented as multiple banks of memory, two simultaneous accesses to different banks can be supported in on cycle. If two accesses need to access the same bank, a bank conflict occurs and the two accesses must be serialized. From practical experience, a cache with eight banks can keep the frequency of bank conflicts down to acceptable levels.

The most common way to reduce memory latency is the use of a cache. Caches are now widely employed. As the gap between processor speed and memory speed widens, multiple levels of caches are required. Most high performance superscalar processes incorporate at least two levels of caches. The first level (L1) cache can usually keep up with the processor speed with access latency of one or very few cycles. Typically there are separate L1 caches for storing instructions and data. The second level (L2) cache, typically supports the storing of both instructions and data, can be either on chip or off chip, and can be accessed in series (in case of a miss in the L1) or in parallel with the L1 cache. Other than the use of a cache or a hierarchy of caches, there are two other techniques for reducing the effective memory latency, namely *nonblocking cache* [Kroft81, Sohi&Franklin91] and *prefetching cache* [Jouppi90,Chen&Baer91].

A nonblocking cache can reduce the effective memory latency by reducing the performance penalty due to cache misses. Traditionally, when a load instruction encounters a cache miss, it will stall the load unit pipeline and any further issuing of load instructions, until the cache miss is serviced. Such form of stalling is overly conservative and prevents subsequent and independent loads that may hit in the data cache from being issued. A nonblocking data cache alleviates this unnecessary penalty by putting aside a load that has missed in the cache into a *missed load queue* and allow subsequent load instructions to issue; see Figure 67. A missed load sits in the missed load queue while the cache miss is serviced. When the missing block is fetched from the main memory, the missed load exits the missed load queue and finishes execution.

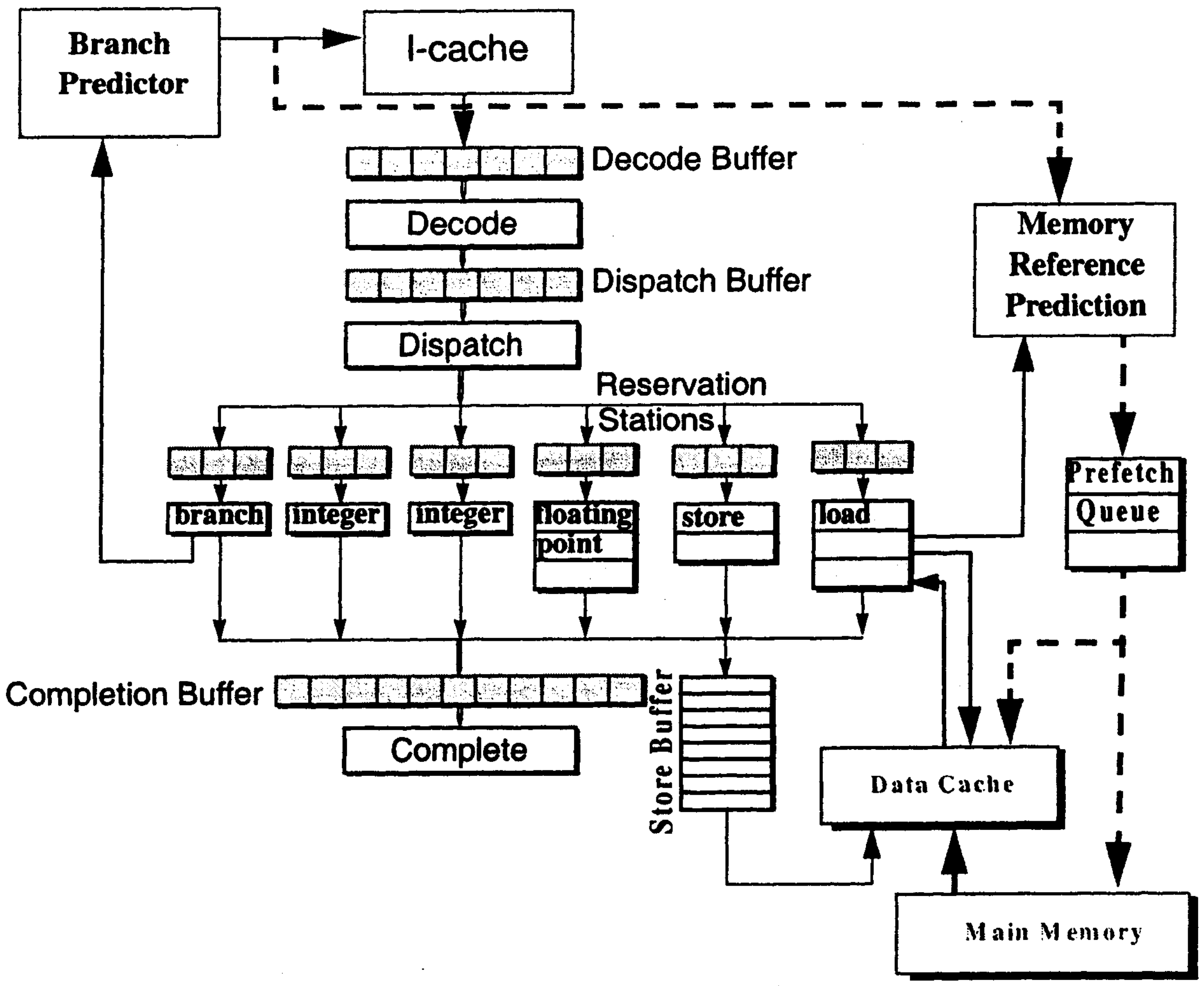
Essentially the cache miss penalty cycles are overlapped with, and masked by, the processing of subsequent independent instructions. Of course, if a subsequent instruction depends on the missed load, the issuing of that instruction is stalled. The number of penalty cycles that can be masked depends on the number of independent instructions following the missed load. A missed load queue can contain multiple entries allowing multiple missed loads to be serviced concurrently. Potentially the cache penalty cycles of multiple missed loads can be overlapped to result in fewer total penalty cycles.

A number of issues must be considered when implementing nonblocking caches. Load misses can occur in bursts. The ability to support multiple misses and overlap their servicing is important. The interface to main memory, or a lower level cache, must be able to support the overlapping or pipelining of multiple accesses. The filling of the cache triggered by the missed load may need to contend with the store buffer for the write port to the cache. There is one complication that can emerge with nonblocking caches. If the missed load is on a speculative path, i.e. the predicted path, there is the possibility that the speculation, i.e. branch prediction, will turn out to be incorrect. If a missed load is on a mispredicted path, the question is whether the cache miss should be serviced. In a machine with very aggressive branch prediction, the number of loads on the mispredicted path can be significant; servicing their misses speculatively can require significant memory bandwidth. Studies [Chen&Baer92] have shown that a nonblocking cache can reduce the amount of load miss penalty by about 15%.

Another way to reduce or mask the cache miss penalty is through the use of a prefetching cache. A prefetching cache anticipates future misses and triggers these misses early so as to overlap the miss penalty with the processing of instructions preceding the missing load. Figure 68 illustrates a prefetching data cache. Two structures are needed to implement a prefetching cache, namely a memory reference prediction table and a prefetch queue. The memory reference prediction table stores information about previously executed loads in three different fields. The first field contains the instruction address of the load and is used as a tag field for selecting an entry of the table. The second field contains the previous address of the load, while the third field contains a stride value that indicates the difference between the previous two addresses used by that load. The memory reference prediction table is accessed via associative search using the fetch address produced by the branch predictor and the first filed of the table. When there is a tag match, indicating a hit in the memory reference prediction table, the previous address is added to the stride value to produce a predicted memory address. This predicted address is then loaded into the prefetch queue. Entries in the prefetch queue are retrieved to speculatively access the data cache and if a cache miss is triggered the main memory or the next lower level cache is accessed. The access to the data cache is in reality a cache touch operation, i.e. access to the cache is attempted in order to trigger a potential cache miss and not to actually retrieve the data in the cache.

The goal of a prefetching cache is to try to anticipate forthcoming cache misses and to trigger those misses early so as to hide the cache miss penalty by overlapping cache refill time with processing of instructions preceding the missing load. When the anticipated missing load is executed the data will be resident in the cache; hence no cache miss is triggered and no cache miss penalty is incurred. The actual effectiveness of prefetching depends on a number of factors. The prefetching distance, i.e. how far in advance is the prefetching being triggered, must be large enough in order to fully mask the miss penalty. This is the reason that the predicted fetch address is used to access the memory reference prediction table, with the hope that the prefetch will occur far enough in advance of the load. However, this makes prefetching effectiveness subject to the effectiveness of branch prediction. Furthermore, there is the potential of polluting the data cache with prefetches that are on the mispredicted path. Status or confidence bits can be added to each entry of the memory reference prediction table to modulate the aggressiveness of prefetching. A converse problem can occur when the prefetching is performed too early so as to evict a useful block from the cache and induce a unnecessary miss. One more factor is the actual memory reference prediction algorithm used. Load address prediction based on stride is quite effective for loads that are stepping through an array. For other loads that are traversing linked list data structures, the stride prediction will not work very well. Prefetching for such memory references will require much more sophisticated prediction algorithms.

FIGURE 68 Prefetching data cache.



To enhance memory data flow load instructions must be executed early and fast. Store instructions are less important because experimental data indicate that they occur less frequent than load instructions and they usually are not on the performance critical path. To speed up the execution of loads we must reduce the latency for processing load instructions. The overall latency for processing a load instruction include four components: 1) pipeline frontend latency for fetching, decoding and dispatching the load instruction; 2) reservation station latency for waiting for register data dependence to resolve; 3) execution pipeline latency for address generation and translation; and 4) the cache/memory access latency for retrieving the data from memory. Both nonblocking and prefetching caches address only the fourth component, which is a crucial component due to the slow memory speed. In order to achieve higher clocking rates, superscalar pipelines are quickly becoming deeper and deeper. Consequently the latencies, in terms of number of machine cycles, of the first three components are also becoming quite significant. A number of speculative techniques have been proposed to address the reduction of these latencies; they include: load address prediction, load value prediction and memory dependence prediction [ ].

Recently *load address prediction* technique has been proposed to address the latencies associated with the first three components. To deal with the latency associated with the first component, a load prediction table, similar to the memory reference prediction table, is proposed. This table is indexed with the predicted fetch address, and a hit in this table indicates the presence of a load instruction in the current fetch group. Hence the prediction of the presence of a load instruction in the next fetch group is performed during the fetch stage and without requiring the decode and dispatch stages. Each entry of this table contains the predicted effective address which is retrieved during the fetch stage, in effect eliminating the need for waiting in the reservation station for the availability of the base register value and the address generation stage of the execution pipeline. Consequently, data cache access can begin in the next cycle and potentially data can be retrieved from the cache at the end of the decode stage. Such form of load address prediction can effectively collapse the latencies of the first three components down to just two cycles, i.e. fetch and decode stages, if the address prediction is correct and that there is a hit in the data cache.

While the hardware structures needed to support load address prediction are quite similar to that needed for memory prefetching, the two mechanisms have significant differences. Load address prediction is actually executing, though speculatively, the load instruction early; whereas memory prefetching is mainly trying to prefetch the needed data into the cache and not actually execute a load instruction. With load address prediction, instructions that depend on the load can also execute earlier because their dependent data is available earlier. Since load address prediction is a speculative technique, it must be validated and if misprediction is detected recovery must be performed. The validation is performed by allowing the actual load instruction to be fetched from the instruction cache and executed in a normal fashion. The result from the speculative version is compared with the normal version. If the result concurs then the speculative result becomes nonspeculative and all the dependent instructions that were executed speculatively are also declared as nonspeculative. If the results do not agree, then the nonspeculative result is used and all dependent instructions must be reexecuted. If the load address prediction mechanism is quite accurate, mispredictions only occur infrequently and the misprediction penalty is minimal and overall performance gain can be achieved.

Even more aggressive than load address prediction is the technique of *load value prediction*. Unlike load address prediction which attempts to predict the effective address of a load instruction, load value prediction actually attempts to predict the value of the data to be retrieved from memory. This is accomplished by extending the load prediction table to contain not just the predicted address, but also the predicted value for the destination register. Experimental studies have shown that many load instructions' destination values are quite predictable. For example, many loads actually load the same value as last time. Hence, by storing the last value loaded by a static load instruction in the load prediction table, this value can be used as the predicted value when the same static load is encountered again. As a result, the load prediction table can be accessed during the fetch stage and at the end of that cycle, the actual destination value of a predicted load instruction can be available and used in the next cycle by a dependent instruction. This significantly reduces the latency required for processing a load instruction if the load value prediction is correct. Again, validation is required and at times misprediction penalty must be paid.

Other than load address prediction and load value prediction, a third speculative technique has been proposed called *memory dependence prediction*. Recall that in order to perform load bypassing and load forwarding, memory dependence checking is required. For load bypassing, it must be determined that the load does not alias with any of the stores being bypassed. For load forwarding, the most recent aliased store must be identified. Memory dependence checking can become quite complex if a larger number of load/store instructions are involved and can potentially require an entire pipe stage. It would be nice to eliminate this latency. Experimental data have shown that the memory dependence relationship is quite predictable. It is possible to track the memory dependence relationship when load/store instructions are executed and use this information to make memory dependence prediction when the same load/store instructions are encountered again. Such memory dependence prediction can facilitate earlier execution of load bypassing and load forwarding. As with all speculative techniques, validation is needed and recovery mechanism for misprediction must be provided.

3.3 THE POWERPC 620 MICROPROCESSOR

The PowerPC family of microprocessors includes the 64-bit PowerPC 620 microprocessor. The 620 is the first 64-bit superscalar processor to employ true out-of-order execution, aggressive branch prediction, distributed multi-entry reservation stations, dynamic renaming for all register files, six pipelined execution units, and a completion buffer to ensure precise exceptions. Most of these features have not been previously implemented in a single-chip microprocessor. Their actual effectiveness is of great interest to both academic researchers as well as industry designers. This chapter presents an instruction-level, or machine-cycle level, performance evaluation of the 620 microarchitecture using a VMW-generated performance simulator of the 620.

3.3.1 The PowerPC 620 Microprocessor

The PowerPC Architecture [46] is the result of the PowerPC alliance among IBM, Motorola, and Apple. It is based on the POWER Architecture [42], designed to facilitate parallel instruction execution and to scale well with advancing technology. The PowerPC alliance has released and announced a number of chips. The first, which provided a transition from the POWER Architecture to the PowerPC Architecture, is the PowerPC 601 microprocessor [47] [48]. The second, a low-power chip, is the PowerPC 603 microprocessor [49]. Recently, a more advanced chip for desktop systems, the PowerPC 604 microprocessor [50] [12], has been shipped. The fourth chip is the 64-bit 620 [51] [37].

The PowerPC Architecture has 32 integer registers (GPRs) and 32 floating-point registers (FPRs). It also has a condition register which can be addressed as one 32-bit register (CR), as a register file of 8 four-bit fields (CRFs), or as 32 single-bit fields. The architecture has a count register (CTR) and a link register (LR), both primarily used for branch instructions, and an integer exception register (XER) and a floating-point status and control register (FPSCR), which are used to record the exception status of the appropriate instruction types. The PowerPC instructions are typical RISC instructions, with the addition of floating-point fused multiply-add (FMA) instructions, load/store instructions with addressing modes that update the effective address, and instructions to set, manipulate, and branch off of the condition register bits.

The 620 is a 4-wide superscalar machine. It uses aggressive branch prediction [35] to fetch instructions as early as possible and a dispatch policy to distribute those instructions to the execution units. The 620 uses six parallel execution units: two simple (single-cycle) integer units, one complex (multi-cycle) integer unit, one floating-point unit (3 stages), one load/store unit (2 stages), and a branch unit. The 620 uses distributed reservation stations [68] and register renaming [39] [18] to implement out-of-order execution. The block diagram of the 620 is shown in Figure 69.

The 620 processes instructions in five major stages, namely the **Fetch**, **Dispatch**, **Execute**, **Complete**, and **Writeback** stages. Some of these stages are separated by buffers to take up slack in the dynamic variation of available parallelism. These buffers are the **Instruction Buffer**, the