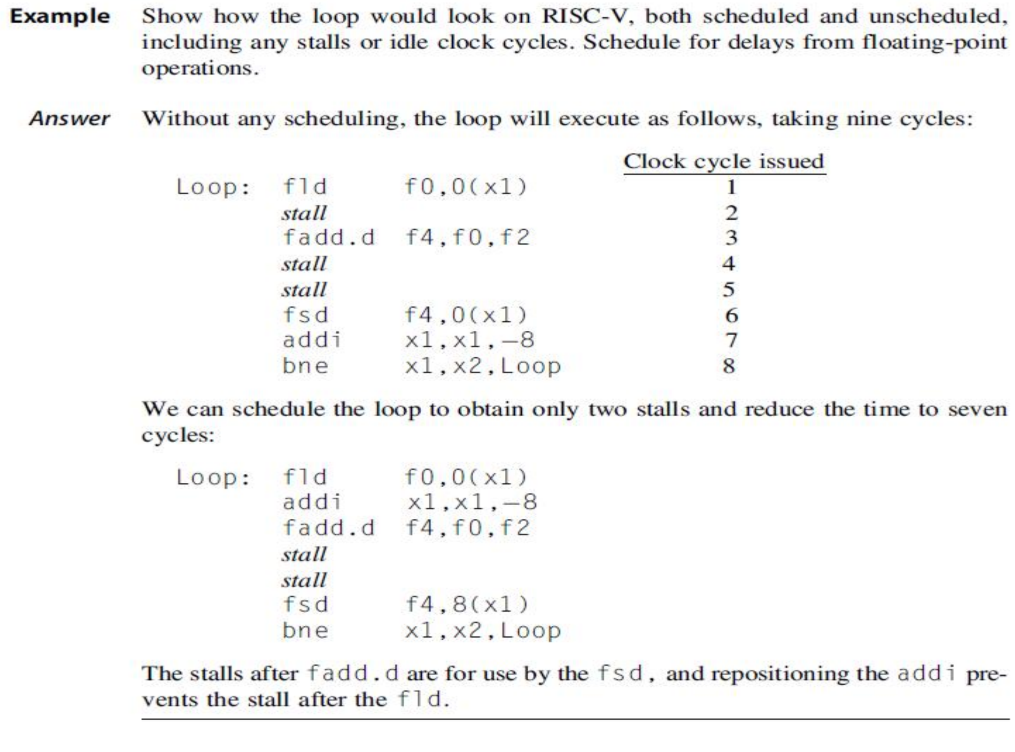
Pipeline : Basic and Intermediate Concepts (Week#13b)

Basic Compiler Techniques for Exposing ILP



Example Show how the loop would look on RISC-V, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations.

Answer Without any scheduling, the loop will execute as follows, taking nine cycles:

```
                                    Clock cycle issued
Loop:   fld      f0,0(x1)                  1
        stall                              2
        fadd.d   f4,f0,f2                  3
        stall                              4
        stall                              5
        fsd      f4,0(x1)                  6
        addi     x1,x1,−8                  7
        bne      x1,x2,Loop                8
```

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

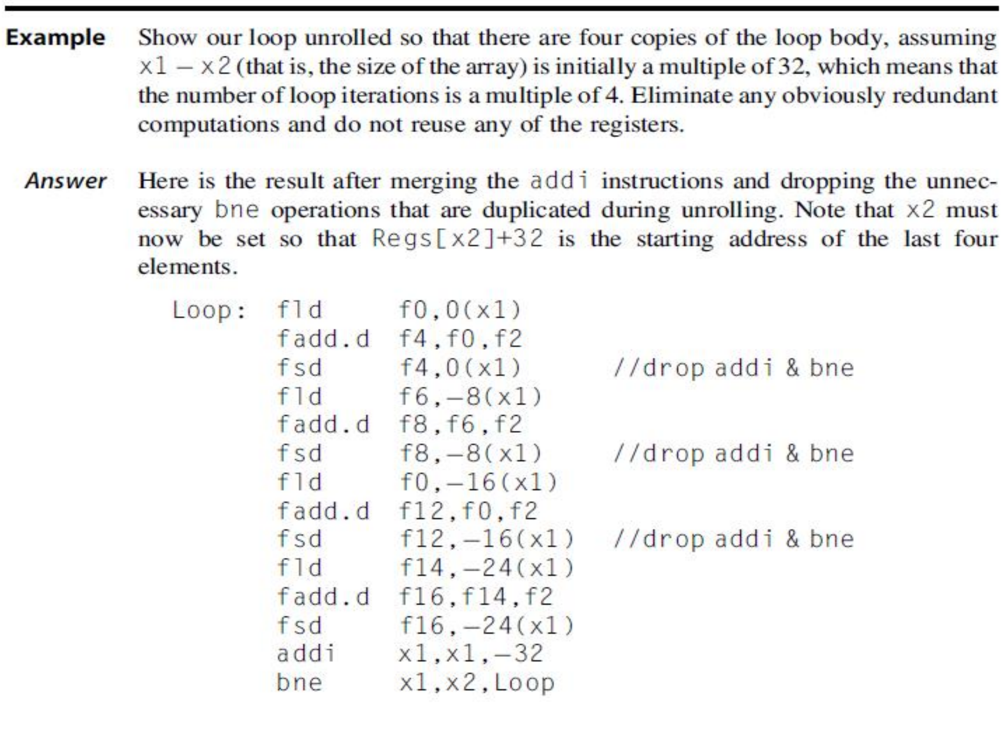
```
Loop:   fld      f0,0(x1)
        addi     x1,x1,−8
        fadd.d   f4,f0,f2
        stall
        stall
        fsd      f4,8(x1)
        bne      x1,x2,Loop
```

The stalls after `fadd.d` are for use by the `fsd`, and repositioning the `addi` prevents the stall after the `fld`.

Basic Compiler Techniques for Exposing ILP

- ✓ Previous example shows complete one loop iteration and store back one array element every seven clock cycles, but work of operating on array element takes three (load, add, and store) of those seven clock cycles.
- ✓ Remaining four clock cycles consist of loop overhead, addi and bne and two stalls. To eliminate these four clock cycles, get more operations relative to the number of overhead instructions.
- ✓ simple scheme for increasing number of instructions relative to the branch and overhead instructions is loop unrolling, replicates the loop body multiple times, adjusting the loop termination code.
- ✓ Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together.
- ✓ eliminate data use stalls by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. use different registers for each iteration, increasing the required number of registers.

Basic Compiler Techniques for Exposing ILP



Example Show our loop unrolled so that there are four copies of the loop body, assuming x1 − x2 (that is, the size of the array) is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the registers.

Answer Here is the result after merging the addi instructions and dropping the unnecessary bne operations that are duplicated during unrolling. Note that x2 must now be set so that Regs[x2]+32 is the starting address of the last four elements.

```
Loop:  fld      f0,0(x1)
       fadd.d   f4,f0,f2
       fsd      f4,0(x1)        //drop addi & bne
       fld      f6,−8(x1)
       fadd.d   f8,f6,f2
       fsd      f8,−8(x1)       //drop addi & bne
       fld      f0,−16(x1)
       fadd.d   f12,f0,f2
       fsd      f12,−16(x1)     //drop addi & bne
       fld      f14,−24(x1)
       fadd.d   f16,f14,f2
       fsd      f16,−24(x1)
       addi     x1,x1,−32
       bne      x1,x2,Loop
```

Summary of the Loop Unrolling and Scheduling

- ✓ Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
- ✓ Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations (e.g., name dependences).
- ✓ Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
- ✓ Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent.
- ✓ This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.
- ✓ Schedule the code, preserving any dependences needed to yield the same result as the original code

Summary of the Loop Unrolling and Scheduling

- ✓ Three different effects limit the gains from loop unrolling:

  (1) a decrease in the amount of overhead amortized with each unroll,

  (2) code size limitations, and

  (3) compiler limitations.
- ✓ Consider loop overhead first. When we unrolled the loop four times, it generated sufficient parallelism among the instructions that the loop could be scheduled with no stall cycles.
  - i. In 14 clock cycles, only 2 cycles were loop overhead: the addi, which maintains the index value, and the bne, which terminates the loop.
  - ii. If the loop is unrolled eight times, the overhead is reduced from 1/2 cycle per element to 1/4.
- ✓ A second limit to unrolling is the resulting growth in code size. For larger loops, the code size growth may be a concern if it causes an increase in instruction cache miss rate.

Summary of the Loop Unrolling and Scheduling

Register Pressure:

- ✓ Code size in registers created by aggressive unrolling and scheduling. This secondary effect that results from instruction scheduling in large code segments is called register pressure.
- ✓ It arises because scheduling code to increase ILP causes number of live values to increase.
- ✓ Without unrolling, scheduling is sufficiently limited by branches so that register pressure is a rare problem.
- ✓ Register pressure problem becomes challenging in multiple-issue processors that require the exposure of more independent instruction sequences whose execution can be overlapped.
- ✓ Use of high-level transformations, whose potential improvements are difficult to measure before detailed code generation led to significant increases in the complexity of modern compilers.

Reducing Branch Costs With Advanced Branch Prediction

- ✓ Loop unrolling is to reduce the number of branch hazards; we can also reduce the performance losses of branches by predicting how they will behave.
- ✓ **Correlating Branch Predictors:** Two-bit predictor schemes use only recent behavior of a single branch to predict future behavior of that branch.
- ✓ Consider a small code fragment from the eqntott benchmark, a member of SPEC benchmark suites that displayed particularly bad branch prediction behavior:

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

Here is the RISC-V code that we would typically generate for this code fragment assuming that aa and bb are assigned to registers x1 and x2:

```
     addi  x3,x1,-2
     bnez  x3,L1       //branch b1  (aa!=2)
     add   x1,x0,x0    //aa=0
L1:  addi  x3,x2,-2
     bnez  x3,L2       //branch b2  (bb!=2)
     add   x2,x0,x0    //bb=0
L2:  sub   x3,x1,x2    //x3=aa-bb
     beqz  x3,L3       //branch b3  (aa==bb)
```

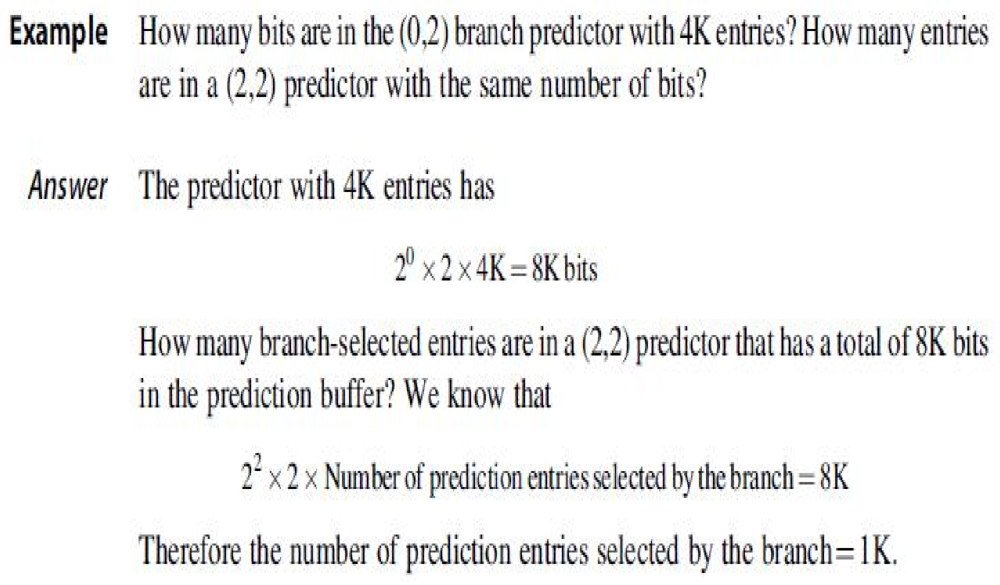
Reducing Branch Costs With Advanced Branch Prediction

- ✓ Behavior of branch b3 is correlated with the behavior of branches b1 and b2.
- ✓ if neither branches b1 nor b2 are taken then b3 will be taken, because aa and bb are clearly equal.
- ✓ Predictor that uses behavior of only a single branch to predict the outcome of that branch can never capture this behavior.
- ✓ Branch predictors that use the behavior of other branches to make a prediction are called correlating predictors or two-level predictors.
- ✓ Existing correlating predictors add information about the behavior of most recent branches to decide how to predict a given branch.
- ✓ For example, (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors.
- ✓ How much better do correlating branch predictors work when compared with the standard 2-bit scheme? To compare them fairly, number of bits in an (m,n) predictor is :

$$2^m \times n \times \text{Number of prediction entries selected by the branch address}$$

- ✓ A 2-bit predictor with no global history is simply a (0,2) predictor.

Reducing Branch Costs With Advanced Branch Prediction



Example How many bits are in the (0,2) branch predictor with 4K entries? How many entries are in a (2,2) predictor with the same number of bits?

Answer The predictor with 4K entries has

$$2^0 \times 2 \times 4\text{K} = 8\text{K bits}$$

How many branch-selected entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer? We know that

$$2^2 \times 2 \times \text{Number of prediction entries selected by the branch} = 8\text{K}$$

Therefore the number of prediction entries selected by the branch = 1K.

Reducing Branch Costs With Advanced Branch Prediction

- ✓ Figure 3.3 compares misprediction rates of the earlier (0,2) predictor with 4K entries and a (2,2) predictor with 1K entries.
- ✓ Correlating predictor not only outperforms a simple 2-bit predictor with the same total number of state bits, but it outperforms a 2-bit predictor with an unlimited number of entries.

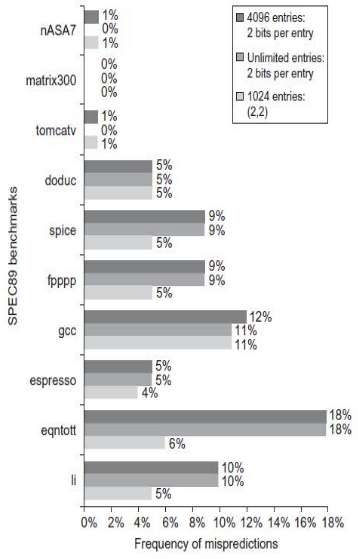


Figure 3.3 **Comparison of 2-bit predictors.** A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

Reducing Branch Costs With Advanced Branch Prediction

- ✓ Example of a correlating predictor is McFarling's gshare predictor.

McFarling's gshare Predictor: index is formed by combining address of the branch and the most recent conditional branch outcomes using an XOR, which acts as a hash of the branch address and the branch history.

- ✓ The hashed result is used to index a prediction array of 2-bit counters, as shown in Figure 3.4.

Alloyed or Hybrid Predictors: Predictors that combine local branch information and global branch history are also called alloyed predictors or hybrid predictors.

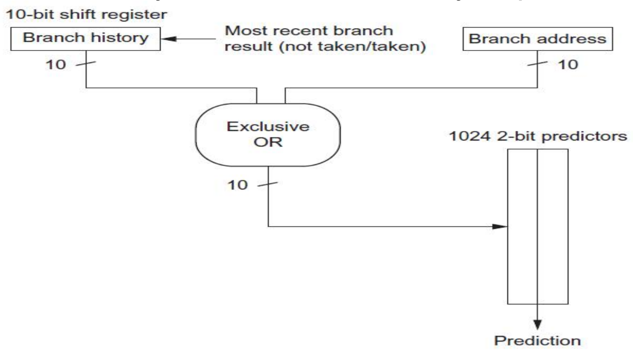


Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

Tournament Predictors: Adaptively Combining Local and Global Predictors

Tournament predictors: Multiple predictors usually a global predictor and a local predictor and choosing between them with a selector, as shown in Figure 3.5.

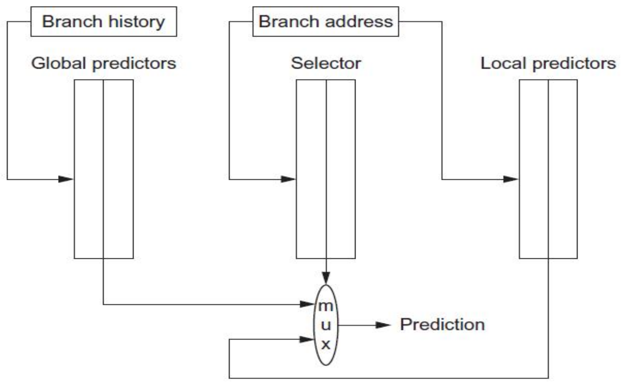


Figure 3.5 A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor. In this case, the index to the selector table is the current branch address. The two tables are also 2-bit predictors that are indexed by the global history and branch address, respectively. The selector acts like a 2-bit predictor, changing the preferred predictor for a branch address when two mispredicts occur in a row. The number of bits of the branch address used to index the selector table and the local predictor table is equal to the length of the global branch history used to index the global prediction table. Note that misprediction is a bit tricky because we need to change both the selector table and either the global or local predictor.

Tournament Predictors: Adaptively Combining Local and Global Predictors

Global predictor: uses the most recent branch history to index the predictor,

Local predictor: uses the address of the branch as the index. Tournament predictors are another form of hybrid or alloyed predictors.

- ✓ Tournament predictors can achieve better accuracy at medium sizes (8K–32K bits) and large numbers of prediction bits.
- ✓ Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor (local, global) was most effective in recent predictions.
- ✓ In 2-bit predictor, saturating counter requires two mispredictions before changing the identity of the preferred predictor.
- ✓ Advantage of a tournament predictor is its ability to select right predictor for a particular branch, crucial for the integer benchmarks.

Tournament Predictors: Adaptively Combining Local and Global Predictors

- ✓ Figure 3.6 shows performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark.
- ✓ Local predictor reaches its limit first. Correlating predictor shows a significant improvement, and the tournament predictor generates a slightly better performance.

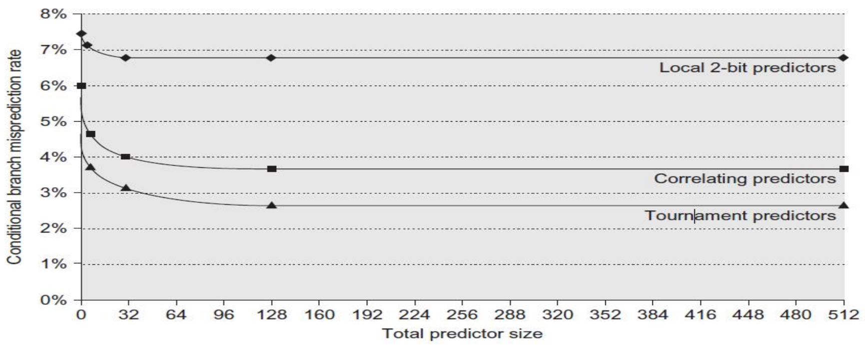


Figure 3.6 The misprediction rate for three different predictors on SPEC89 versus the size of the predictor in kilobits. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

Tournament Predictors: Adaptively Combining Local and Global Predictors

- ✓ Local predictor consists of a two-level predictor.
- ✓ Top level is a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to most recent 10 branch outcomes for the entry.
- ✓ If the branch is taken 10 or more times in a row, entry in the local history table will be all 1s.
- ✓ If the branch is alternately taken and untaken, history entry consists of alternating 0s and 1s.
- ✓ 10-bit history allows patterns of up to 10 branches to be discovered and predicted.
- ✓ branch prediction schemes involve combining multiple predictors that track whether a prediction is associated with current branch.
- ✓ One important class of predictors is loosely based on an algorithm for statistical compression called PPM (Prediction by Partial Matching).
- ✓ PPM attempts to predict future behavior based on history. This class of branch predictors employs a series of global predictors indexed with different length histories.

Tagged Hybrid Predictors

- ✓ Figure 3.7 shows a five-component tagged hybrid predictor has five prediction tables: P(0), P(1), . . . P(4), where P(i) is accessed using a hash of the PC and the history of most recent i branches (kept in a shift register).
- ✓ Use of multiple history lengths to index separate predictors is first critical difference.

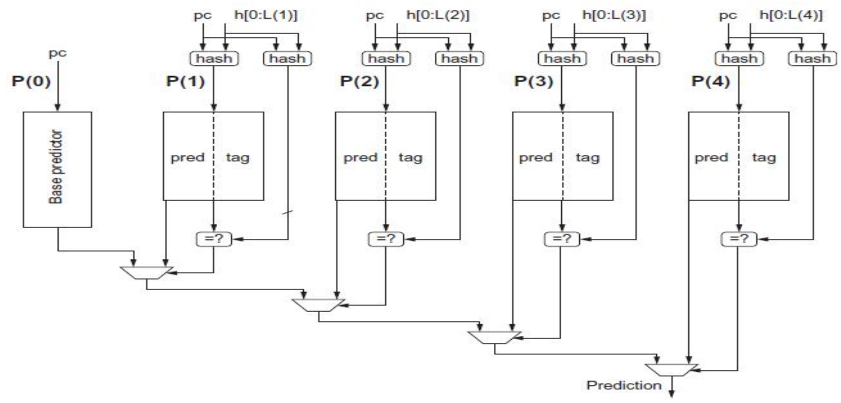


Figure 3.7 A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h" in this figure. The hash can be as simple as an exclusive-OR, as in gshare. Each predictor is a 2-bit (or possibly 3-bit) predictor. The tags are typically 4–8 bits. The chosen prediction is the one with the longest history where the tags also match.

Tagged Hybrid Predictors

- ✓ Second critical difference is the use of tags in tables P(1) through P(4). The tags can be short because 100% matches are not required:
- ✓ a small tag of 4–8 bits appears to gain most of the advantage. A prediction from P(1), . . . P(4) is used only if tags match hash of branch address and global branch history.
- ✓ Each of the predictors in P(0…n) can be a standard 2-bit predictor.
- ✓ Three-bit counter requires three mispredictions to change a prediction, gives slightly better results than a 2-bit counter.

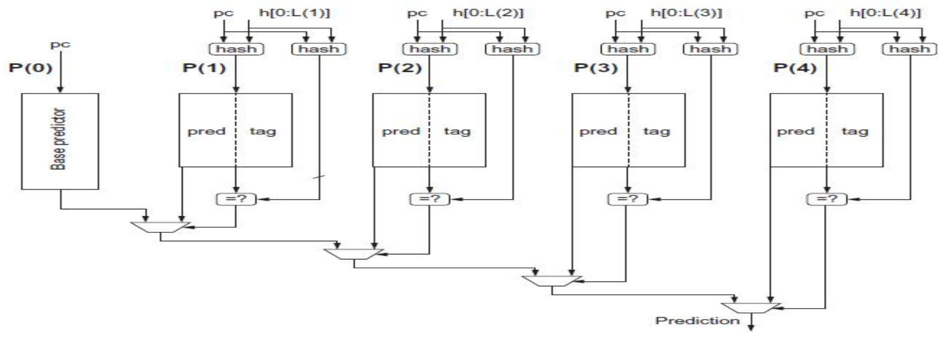


Figure 3.7 A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h" in this figure. The hash can be as simple as an exclusive-OR, as in gshare. Each predictor is a 2-bit (or possibly 3-bit) predictor. The tags are typically 4–8 bits. The chosen prediction is the one with the longest history where the tags also match.

Tagged Hybrid Predictors

- ✓ The prediction for a given branch is the predictor with the longest branch history that also has matching tags.
- ✓ P(0) always matches because it uses no tags and becomes the default prediction if none of P(1) through P(n) match.
- ✓ Tagged hybrid version of this predictor also includes a 2-bit use field in each of the history-indexed predictors.
- ✓ Tagged hybrid predictors (TAGE—TAgged GEometic predictors) and the earlier PPM-based predictors have been the winners in recent annual international branch-prediction competitions.
- ✓ TAGE predictors outperform gshare and the tournament predictors with modest amounts of memory (32–64 KiB)
- ✓ TAGE predictors seems to effectively use larger prediction caches to deliver improved prediction accuracy.

Tagged Hybrid Predictors

- ✓ Figure 3.8 shows that a hybrid tagged predictor significantly outperforms gshare for less predictable programs like SPECint and server applications.

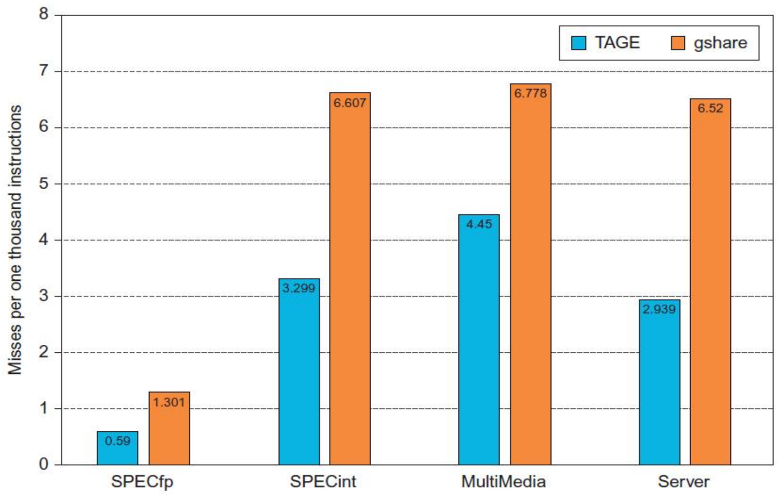


Figure 3.8 A comparison of the misprediction rate (measured as mispredicts per 1000 instructions executed) for tagged hybrid versus gshare. Both predictors use the same total number of bits, although tagged hybrid uses some of that storage for tags, while gshare contains no tags. The benchmarks consist of traces from SPECfp and SPECint, a series of multimedia and server benchmarks. The latter two behave more like SPECint.

Evolution of Core i7 Predictor

- ✓ The Core i7 920 used a two-level predictor that has a smaller first-level predictor designed to meet the cycle constraints of predicting a branch every clock cycle, and a larger second-level predictor as a backup. Each predictor combines three different predictors:
  - i. simple 2-bit predictor used in the preceding tournament predictor
  - ii. a global history predictor
  - iii. a loop exit predictor.

Loop exit predictor: uses a counter to predict the exact number of taken branches (which is the number of loop iterations)for a branch that is detected as a loop branch.

- ✓ For each branch, the best prediction is chosen from among the three predictors by tracking the accuracy of each prediction, like a tournament predictor.
- ✓ In addition to this multilevel main predictor, a separate unit predicts target addresses for indirect branches, and stack is used to predict return addresses. **E**

Overcoming Data Hazards With Dynamic Scheduling

Static scheduling: minimize stalls by separating dependent instructions so that they will not lead to hazards.

Dynamic scheduling: a technique by which hardware reorders instruction execution to reduce stalls while maintaining data flow and exception behavior.

Dynamic scheduling offers several advantages.

i. Allows code that was compiled with one pipeline to run efficiently on a different pipeline, eliminating the need to have multiple binaries and recompile for a different microarchitecture.

ii. Enables handling when dependences are unknown at compile time; for example: involve a memory reference or a data dependent branch or a result from a modern programming environment that uses dynamic linking or dispatching.

iii. Allows the processor to tolerate unpredictable delays such as cache misses by executing other code while waiting for the miss to resolve.

Dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences are present.

Overcoming Data Hazards With Dynamic Scheduling

- ✓ Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in five-stage integer pipeline and its logical extension to in-order floating-point pipeline.
- ✓ Consider the following RISC-V floating-point code sequence:

```
fdiv.d f0,f2,f4
fmul.d f6,f0,f8
fadd.d f0,f10,f14
```

- ✓ There is an antidependence between the fmul.d and the fadd.d (for the register f0), and if the pipeline executes the fadd.d before the fmul.d (which is waiting for the fdiv.d), it will violate the antidependence, yielding a WAR hazard.
- ✓ To avoid violating output dependences, such as the write of f0 by fadd.d before fdiv.d completes, WAW hazards must be handled.
- ✓ Both these hazards are avoided by the use of register renaming.
- ✓ Dynamically scheduled processors preserve exception behavior by delaying the notification of an associated exception until processor knows that instruction should be the next one completed.

Overcoming Data Hazards With Dynamic Scheduling

- ✓ Exception behavior must be preserved, dynamically scheduled processors could generate imprecise exceptions. Imprecise exceptions can occur because of two possibilities:

i. Pipeline may have already completed instructions that are later in program order than the instruction causing the exception.

ii. Pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception.

Tomasulo's algorithm:

- ✓ handles antidependences and output dependences by renaming the registers dynamically.
- ✓ Tomasulo's algorithm can be extended to handle speculation, a technique to reduce the effect of control dependences by predicting the outcome of a branch, executing instructions at the predicted destination address, and taking corrective actions when the prediction was wrong.
- ✓ Use of scoreboarding is probably sufficient to support simpler processors, higher performance processors make use of speculation.

Dynamic Scheduling Using Tomasulo's Approach

- ✓ IBM 360/91 had long memory accesses and long floating-point delay. To overcome these problems Tomasulo's algorithm was designed.
- ✓ Tomasulos algorithm focuses on floating-point unit and loadstore unit of RISC-V instruction set.
- ✓ Algorithm uses a load functional unit so no significant changes are needed to add register-memory addressing modes.
- ✓ RAW hazards are avoided by executing an instruction only when its operands are available, with scoreboarding.
- ✓ Register renaming eliminates WAR and WAW hazards by renaming all destination registers.
- ✓ Compiler implement such renaming, if there were enough registers available in the ISA.
- ✓ Original 360/91 had only four floating-point registers and modern processors have 32–64 floating point and integer registers. Number of renaming registers available is in hundreds.

Dynamic Scheduling Using Tomasulo's Approach

Register Renaming EXample: consider the following that includes WAR and WAW hazards:

fdiv.d f0,f2,f4

fadd.d f6,f0,f8

fsd f6,0(x1)

fsub.d f8,f10,f14

fmul.d f6,f10,f8

- ✓ There are two antidependences:
  - i. between the fadd.d and the fsub.d
  - ii. between the fsd and the fmul.d.
- ✓ There is also an output dependence between the fadd.d and the fmul.d, leading to three possible hazards:
  - i. WAR hazards on the use of f8 by fadd.d
  - ii. f8 use by the fsub.d
  - iii. a WAW hazard because the fadd.d may finish later than the fmul.d.
- ✓ There are also three true data dependences:
  - i. between the fdiv.d and the fadd.d,
  - ii. between the fsub.d and the fmul.d,
  - iii. between the fadd.d and the fsd.

Dynamic Scheduling Using Tomasulo’s Approach

These three name dependences can be eliminated by register renaming. Assume the existence of two temporary registers **S** and **T**.

fdiv.d f0,f2,f4

fadd.d S,f0,f8

fsd S,0(x1)

fsub.d T,f10,f14

fmul.d f6,f10,T

- ✓ Use of f8 must be replaced by register **T**. Renaming can be done by compiler.
- ✓ Use of f8 later in code requires compiler analysis. Tomasulo’s algorithm can handle renaming across branches.

Dynamic Scheduling Using Tomasulo’s Approach

Reservation Stations:

- ✓ In Tomasulo’s scheme, register renaming is provided by reservation stations, which buffer the operands of instructions waiting to issue and are associated with functional units.
- ✓ Reservation station fetches and buffers an operand, eliminating the need to get the operand from a register.
- ✓ Register specifiers for pending operands are renamed to names of reservation station.
- ✓ Reservation station can eliminate hazards arising from name dependences.
- ✓ Use of reservation stations rather than a centralized register file, leads to two other important properties.
    - i. hazard detection and execution control are distributed: information held in reservation stations at each functional unit determines when an instruction can begin execution.
    - ii. results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (called common data bus, or CDB).

Dynamic Scheduling Using Tomasulo's Approach

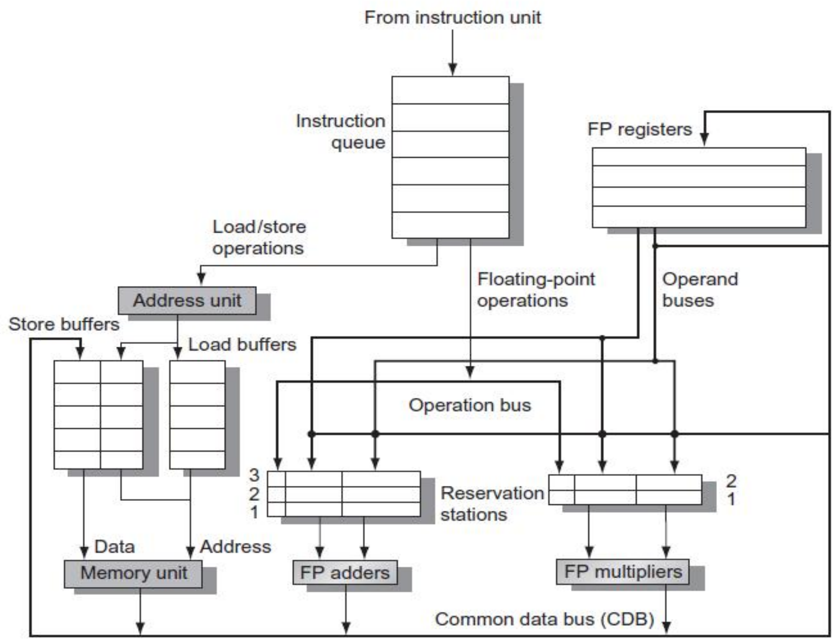


Figure 3.10 The basic structure of a RISC-V floating-point unit using Tomasulo's algorithm. Instructions are sent

Dynamic Scheduling Using Tomasulo's Approach

Structure of a RISC-V floating-point unit using Tomasulo's algorithm

- ✓ Instructions are sent from the instruction unit into the instruction queue from which they are issued in first-in, first-out (FIFO) order.
- ✓ Reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions:
  - i. hold the components of the effective address until it is computed,
  - ii. track outstanding loads that are waiting on the memory,
  - iii. hold the results of completed loads that are waiting for the CDB(commom data bus).
- ✓ Similarly, store buffers have three functions:
  - i. hold the components of the effective address until it is computed,
  - ii. hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and
  - iii. hold the address and value to store until the memory unit is available.
- ✓ All results from FP units or load unit are put on CDB, which goes to the FP register file as well as to the reservation stations and store buffers.
- ✓ FP adders implement addition and subtraction and the FP multipliers for multiplication and division.

Dynamic Scheduling Using Tomasulo's Approach

Tags in the Tomasulo scheme refer to the buffer or unit that will produce a result;

Difference between Tomasulo's Scheme and Scoreboarding: In Tomasulo, register names are discarded when an instruction issues to a reservation station. In scoreboarding, operands stay in the registers and are read only after the producing instruction completes and the consuming instruction is ready to execute.

Each reservation station has seven fields:

Op: Operation to perform on source operands S1 and S2.

Qj, Qk: reservation stations produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.

Vj, Vk: Value of the source operands. V fields or the Q field is valid for each operand. For loads, Vk field is used to hold the offset field.

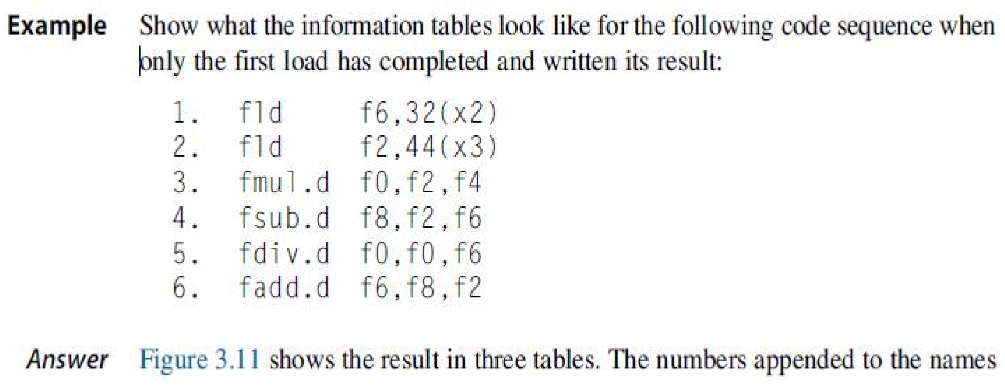
A:Used to hold information for memory address calculation for a load or store. Initially, immediate field is stored; after the address calculation, effective address is stored

Dynamic Scheduling Using Tomasulo's Approach

Busy: Indicates that this reservation station and its accompanying functional unit are occupied.

Register file has a field,

Qi: Number of the reservation station that contains the operation whose currently active instruction is computing a result destined for this register.



Example Show what the information tables look like for the following code sequence when only the first load has completed and written its result:

```
1.   fld     f6,32(x2)
2.   fld     f2,44(x3)
3.   fmul.d  f0,f2,f4
4.   fsub.d  f8,f2,f6
5.   fdiv.d  f0,f0,f6
6.   fadd.d  f6,f8,f2
```

Answer Figure 3.11 shows the result in three tables. The numbers appended to the names

Instruction		Instruction status		
		Issue	Execute	Write result
fld	f6,32(x2)	√	√	√
fld	f2,44(x3)	√	√	
fmul.d	f0,f2,f4	√		
fsub.d	f8,f2,f6	√		
fdiv.d	f0,f0,f6	√		
fadd.d	f6,f8,f2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[x3]
Add1	Yes	SUB		Mem[32 + Regs[x2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[f4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[x2]]	Mult1		

	Register status							
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figure 3.11 Reservation stations and register tags shown when all of the instructions have issued but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation but is waiting on the memory unit. We use the array Regs[ ] to refer to the register file and the array Mem[ ] to refer to the memory. Remember that an operand is specified by either a Q field or a V field at any time. Notice that the fadd.d instruction, which has a WAR hazard at the WB stage, has issued and could complete before the fdiv.d initiates.

Dynamic Scheduling: Examples and the Algorithm

Example Using the same code segment as in the previous example (page 201), show what the status tables look like when the fmul.d is ready to write its result.

Answer The result is shown in the three tables in Figure 3.12. Notice that fadd.d has completed because the operands of fdiv.d were copied, thereby overcoming the WAR hazard. Notice that even if the load of f6 was fdiv.d, the add into f6 could be executed without triggering a WAW hazard.

Instruction		Instruction status		
		Issue	Execute	Write result
fld	f6,32(x2)	√	√	√
fld	f2,44(x3)	√	√	√
fmul.d	f0,f2,f4	√	√	
fsub.d	f8,f2,f6	√	√	√
fdiv.d	f0,f0,f6	√		
fadd.d	f6,f8,f2	√	√	√

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	Mem[44 + Regs[x3]]	Regs[f4]			
Mult2	Yes	DIV		Mem[32 + Regs[x2]]	Mult1		

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1					Mult2			

Figure 3.12 Multiply and divide are the only instructions not finished.