Chapter 2 – Instruction Level Parallelism and Its Exploitation

Outline

- Instruction Level Parallelism (2.1)
- Compiler techniques for Exposing ILP (2.2)
- Reducing Branch Costs with Prediction (2.3)
- Overcoming Data Hazards with Dynamic Scheduling (2.4)
- Dynamic Scheduling: Examples and the Algorithm (2.5)
- Hardware-Based Speculation (2.6)
- Exploiting ILP using Multiple Issue and Static Scheduling (2.7)
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation (2.8)

Recall from Pipelining Review

- **Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls**
  - **Ideal pipeline CPI: measure of the maximum performance attainable by the implementation**
  - **Structural hazards: HW cannot support this combination of instructions**
  - **Data hazards: Instruction depends on result of prior instruction still in the pipeline**
  - **Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)**

Instruction Level Parallelism

- **Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance**
- **2 approaches to exploit ILP:**
  1) **Rely on hardware to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power) , and**
  2) **Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2)**

Instruction-Level Parallelism (ILP)

- **Basic Block (BB) ILP is quite small**
  - **BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit**
  - **average dynamic branch frequency 15% to 25% => 4 to 7 instructions execute between a pair of branches**
  - **Plus instructions in BB likely to depend on each other**
- **To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks**
- **Simplest: loop-level parallelism to exploit parallelism among iterations of a loop. E.g.,**

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

Loop-Level Parallelism

- Exploit loop-level parallelism to parallelism by “unrolling loop” either by

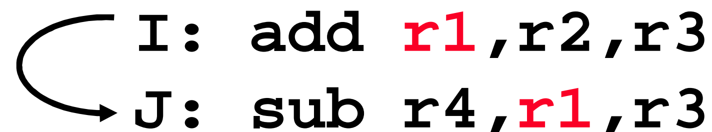
1. dynamic via branch prediction or
2. static via loop unrolling by compiler

   (Another way is vectors, to be covered later)

- Determining instruction dependence is critical to Loop Level Parallelism
- If 2 instructions are
  - parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
  - dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

Data Dependence and Hazards

- $Instr_J$ is data dependent (aka true dependence) on $Instr_I$:
  1. $Instr_J$ tries to read operand before $Instr_I$ writes it



  2. or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence $\Rightarrow$ data dependence in source code $\Rightarrow$ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, called a Read After Write (RAW) hazard

ILP and Data Dependencies,Hazards

- HW/SW must preserve program order: order instructions would execute in if executed sequentially as determined by original source program
  - Dependences are a property of programs
- Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies

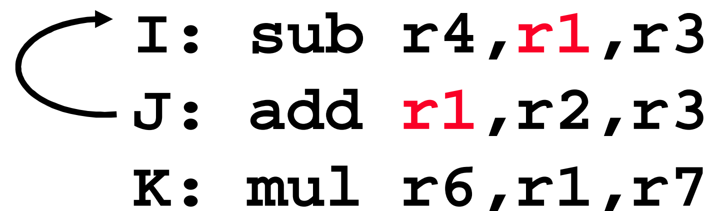
  1) indicates the possibility of a hazard

  2) determines order in which results must be calculated

  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Name Dependence #1: Anti-dependence

- **Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence**
- **Instr_J writes operand *before* Instr_I reads it**

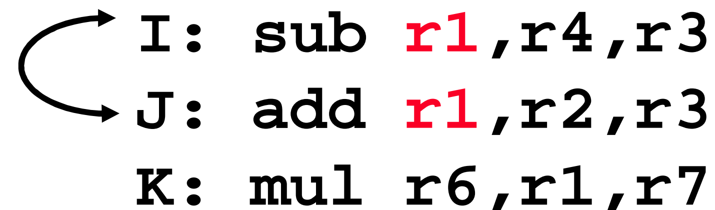


Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1"

- **If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard**

Name Dependence #2: Output dependence

- $Instr_J$ writes operand ***before*** $Instr_I$ writes it.



- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
  - Register renaming resolves name dependence for regs
  - Either by compiler or by HW

Control Dependencies

- **Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order**

```
if p1 {
  S1;
};
if p2 {
  S2;
}
```

- **`S1` is control dependent on `p1`, and `S2` is control dependent on `p2` but not on `p1`.**

Control Dependence Ignored

- **Control dependence need not be preserved**
  - **willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program**
- **Instead, 2 properties critical to program correctness are**

  1) exception behavior and

  2) data flow

Exception Behavior

- **Preserving exception behavior**
  ⇒ any changes in instruction execution order must not change how exceptions are raised in program
  (⇒ no new exceptions)
- **Example:**

```
        DADDU     R2,R3,R4
        BEQZ      R2,L1
        LW        R1,0(R2)
L1:
```

  - **(Assume branches not delayed)**
- **Problem with moving `LW` before `BEQZ`?**

Data Flow

- **Data flow: actual flow of data values among instructions that produce results and those that consume them**
  - **branches make flow dynamic, determine which instruction is supplier of data**
- **Example:**

```
DADDU    R1,R2,R3
BEQZ     R4,L
DSUBU    R1,R5,R6
L: …
OR       R7,R1,R8
```

- **OR depends on DADDU or DSUBU?**
  Must preserve data flow on execution

Outline

- Instruction Level Parallelism (2.1)
- Compiler techniques for Exposing ILP (2.2)
- Reducing Branch Costs with Prediction (2.3)
- Overcoming Data Hazards with Dynamic Scheduling (2.4)
- Dynamic Scheduling: Examples and the Algorithm (2.5)
- Hardware-Based Speculation (2.6)
- Exploiting ILP using Multiple Issue and Static Scheduling (2.7)
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation (2.8)

Software Techniques - Example

- This code, add a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Assume following latencies for all examples
  - Ignore delayed branch in these examples

Instruction producing result	*Instruction using result*	*Latency in cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

FP Loop: Where are the Hazards?

- **First translate into MIPS code:**
  - -To simplify, assume 8 is lowest address

```
Loop:  L.D     F0,0(R1)  ;F0=vector element
       ADD.D   F4,F0,F2  ;add scalar from F2
       S.D     F4, 0(R1);store result
       DADDUI  R1,R1,#-8;decrement pointer 8B (DW)
       BNEZ    R1,Loop   ;branch R1!=zero
```

FP Loop Showing Stalls

```
1 Loop: L.D    F0,0(R1)  ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2  ;add scalar in F2
4       stall
5       stall
6       S.D    F4 0(R1)  ;store result
7       DADDUI R1,R1,-8  ;decrement pointer 8B (DW)
8       stall            ;assumes can't forward to branch
9       BNE    R1,R2,Loop ;branch R1!=zero
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 clock cycles: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: L.D    F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4  ;altered offset when move DSUBUI
7       BNE    R1, R2, Loop
```

Swap DADDUI and S.D by changing address of S.D

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How make faster?

Unroll Loop Four Times (straightforward way)

```
1  Loop:L.D    F0,0(R1)
3       ADD.D  F4,F0,F2
6       S.D    0(R1),F4       ;drop DSUBUI & BNEZ
7       L.D    F6,-8(R1)
9       ADD.D  F8,F6,F2
12      S.D    -8(R1),F8      ;drop DSUBUI & BNEZ
13      L.D    F10,-16(R1)
15      ADD.D  F12,F10,F2
18      S.D    -16(R1),F12    ;drop DSUBUI & BNEZ
19      L.D    F14,-24(R1)
21      ADD.D  F16,F14,F2
24      S.D    -24(R1),F16
25      DADDUI R1,R1,#-32     ;alter to 4*8
26      BNE    R1, R2, LOOP
```

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n, most of the execution time will be spent in the unrolled loop

Unrolled Loop That Minimizes Stalls

```
1 Loop:L.D    F0,0(R1)
2       L.D    F6,-8(R1)
3       L.D    F10,-16(R1)
4       L.D    F14,-24(R1)
5       ADD.D  F4,F0,F2
6       ADD.D  F8,F6,F2
7       ADD.D  F12,F10,F2
8       ADD.D  F16,F14,F2
9       S.D    F4,0(R1)
10      S.D    F8,-8(R1)
11      DSUBUI R1,R1,#32
12      S.D    F12,16(R1)
13      S.D    F16, 8(R1)
14      BNE    R1, R2, ,LOOP
```

14 clock cycles, or 3.5 per iteration

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

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

Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl's Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Compiler
4. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not be possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

Outline

- Instruction Level Parallelism (2.1)
- Compiler techniques for Exposing ILP (2.2)
- Reducing Branch Costs with Prediction (2.3)
- Overcoming Data Hazards with Dynamic Scheduling (2.4)
- Dynamic Scheduling: Examples and the Algorithm (2.5)
- Hardware-Based Speculation (2.6)
- Exploiting ILP using Multiple Issue and Static Scheduling (2.7)
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation (2.8)

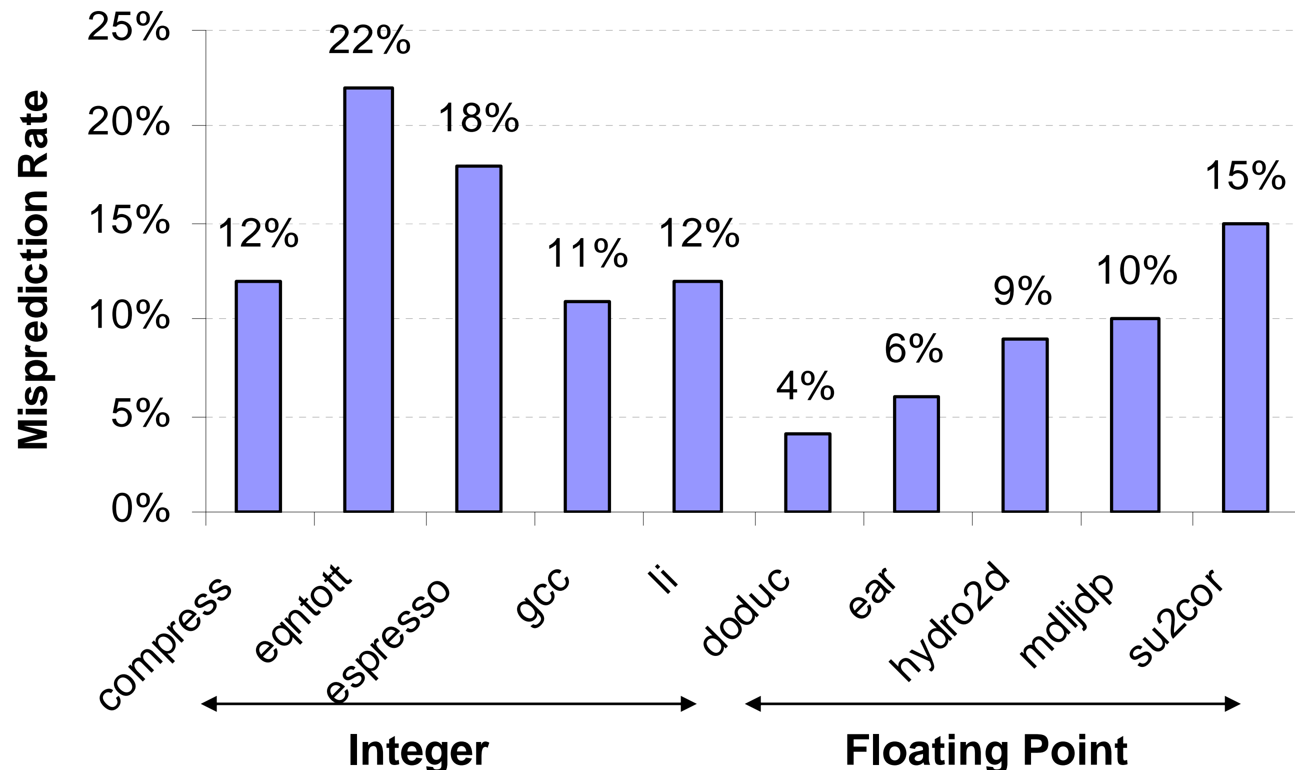
Control Hazards

- In the 5-stage in-order processor: assume always taken or assume always not taken; if the branch goes the other way, squash mis-fetched instructions
- Modern out-of-order processors: dynamic branch prediction
- Branch predictor: a cache of recent branch outcomes

Static Branch Prediction

- Previous lecture showed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC

- More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:



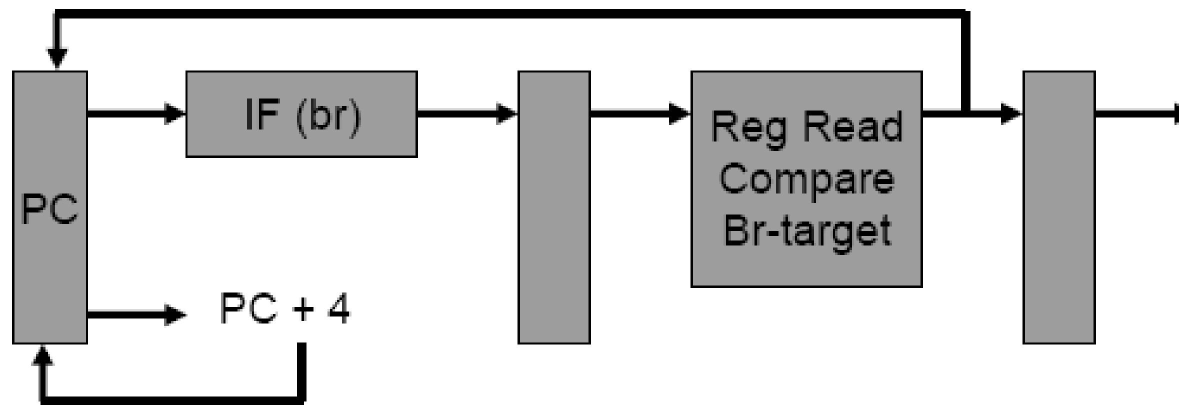
Dynamic Hardware Prediction

- **Importance of control dependences**
  - **Branches and jumps are frequent**
- **Schemes to attack control dependences**
  - **Static**
    - » **Basic (stall the pipeline)**
    - » **Predict-not-taken and predict-taken**
    - » **Delayed branch and canceling branch**
  - Dynamic predictors
- **The prediction will depend on the behavior of the branch at run time and the branch changes its own behavior during execution.**
- **Goal: allow the processor resolve the outcome of a branch early, preventing control dependences.**
- **Effectiveness of dynamic prediction schemes**
  - **Accuracy**
  - **Cost of the branch when the direction is incorrect**

Dynamic Branch Prediction

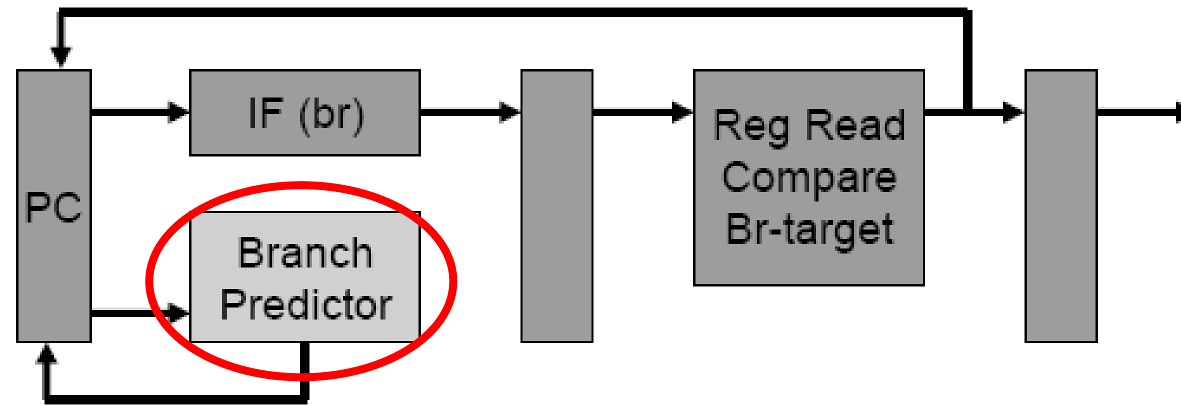
- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior

Pipeline without Branch Predictor



- In the 5-stage pipeline, a branch completes in two cycles
- If the branch went the wrong way, one incorrect instr is fetched
- One stall cycle per incorrect branch

Pipeline with Branch Predictor



Basic Branch Prediction Buffers

- **A branch-prediction buffer (branch history table) is a small memory indexed by the lower potion of the address of the branch instruction.**
  - **The memory contains whether the branch was recently taken or not.**
  - **Any branch with the same low-order address can modify the content.**
- **If the hint turns out to be wrong, the prediction bit is inverted and stored back.**
- **It is effectively a cache.**

Performance Shortcoming

- **If a branch is almost always taken, will predict incorrectly twice, rather than once, when it not taken.**
  - **Mispredict on the first and last loop iterations**

Dynamic Branch Prediction

- **Performance = *ƒ*(accuracy, cost of misprediction)**
- **Branch History Table: Lower bits of PC address index table of 1-bit values**
  - **Says whether or not branch taken last time**
  - **No address check**
- **Problem: in a loop, 1-bit BHT will cause two mispredictions:**
  - **End of loop case, when it exits instead of looping as before**
  - **First time through loop on *next* time through code, when it predicts exit instead of looping**

1-Bit Prediction

- **For each branch, keep track of what happened last time and use that outcome as the prediction**
- **What are prediction accuracies for branches 1 and 2 below:**

```
while (1) {
   for (i=0;i<10;i++) {   branch-1
      …
   }
   for (j=0;j<20;j++) {   branch-2
      …
   }
}
```

2-Bit Prediction

- For each branch, maintain a 2-bit saturating counter:

  if the branch is taken: counter = min(3,counter+1)

  if the branch is not taken: counter = max(0,counter-1)
- If (counter >= 2), predict taken, else predict not taken
- Advantage: a few atypical branches will not influence the prediction (a better measure of "the common case")
- Especially useful when multiple branches share the same counter (some bits of the branch PC are used to index into the branch predictor)
- Can be easily extended to N-bits (in most processors, N=2)

N-bit Branch Prediction Buffers

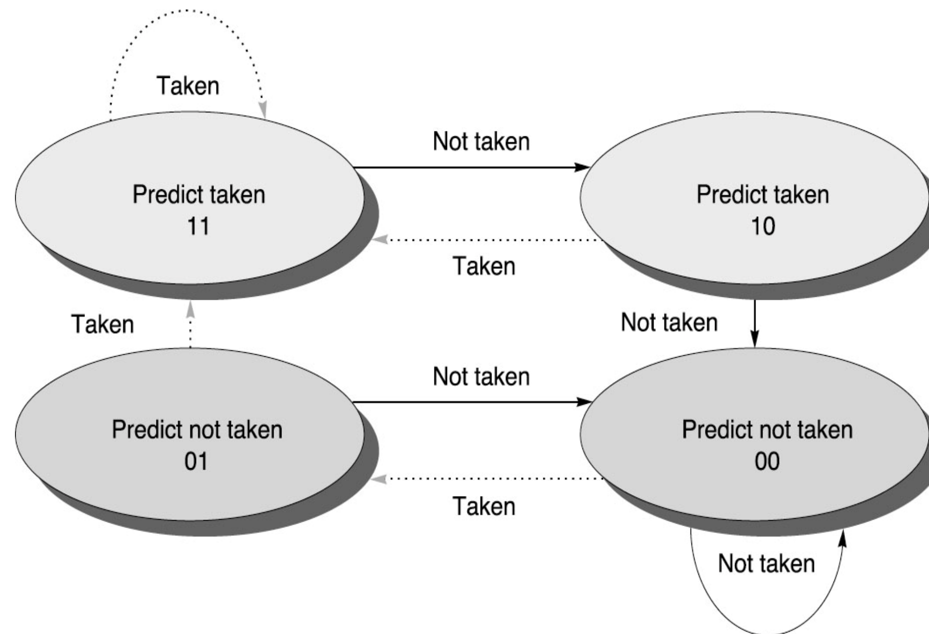
- When the counter is greater than or equal to one-half of its maximum value (2^n-1), the branch is predicted as taken.
- The counter is increased on a taken branch and decremented on an untaken branch.
- A branch buffer can be implemented as a small cache accessed during the IF stage.

N-bit Branch Prediction Buffers

Use an n-bit saturating counter

Only the loop exit causes a misprediction

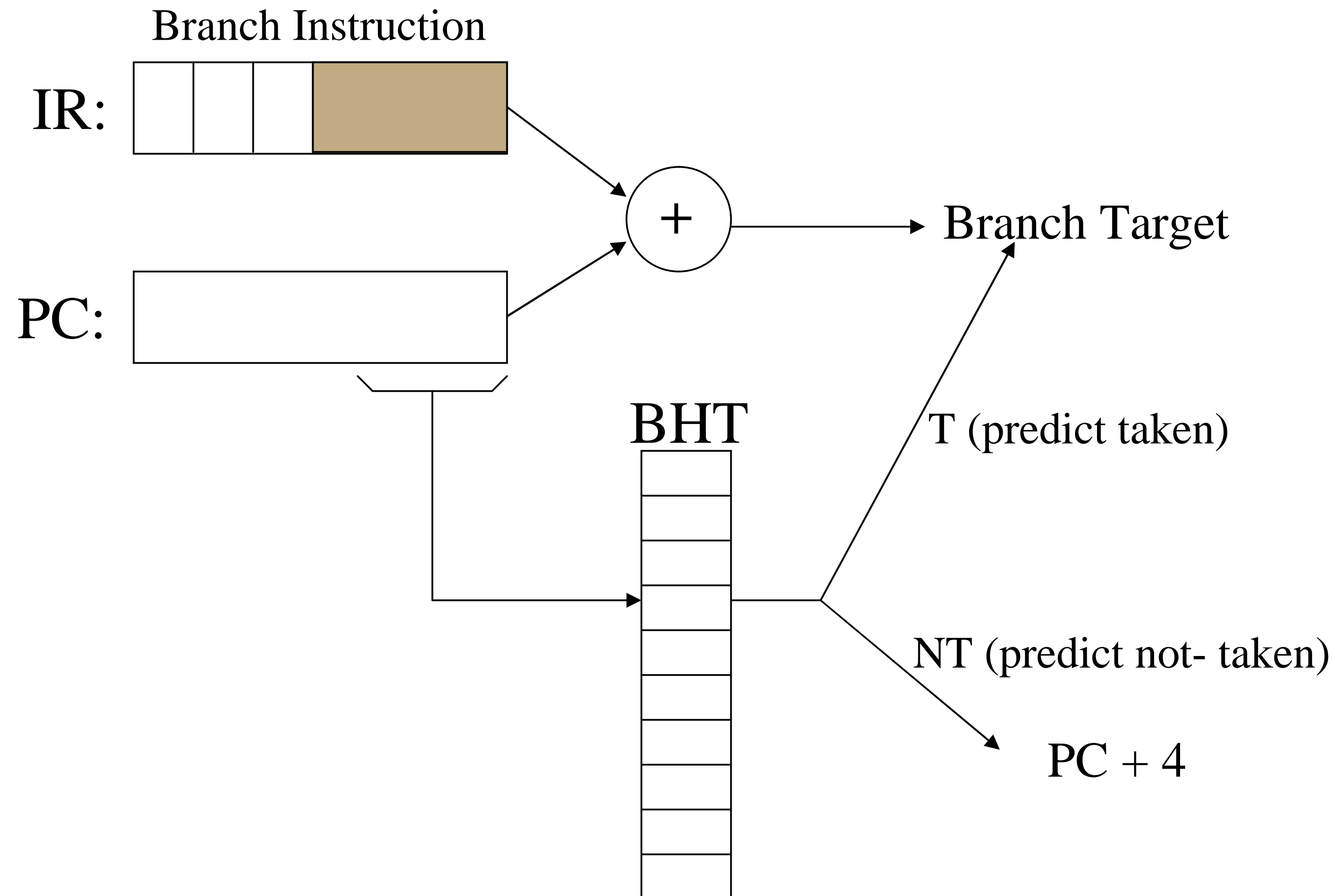
2-bit predictor almost as good as any general n-bit predictor



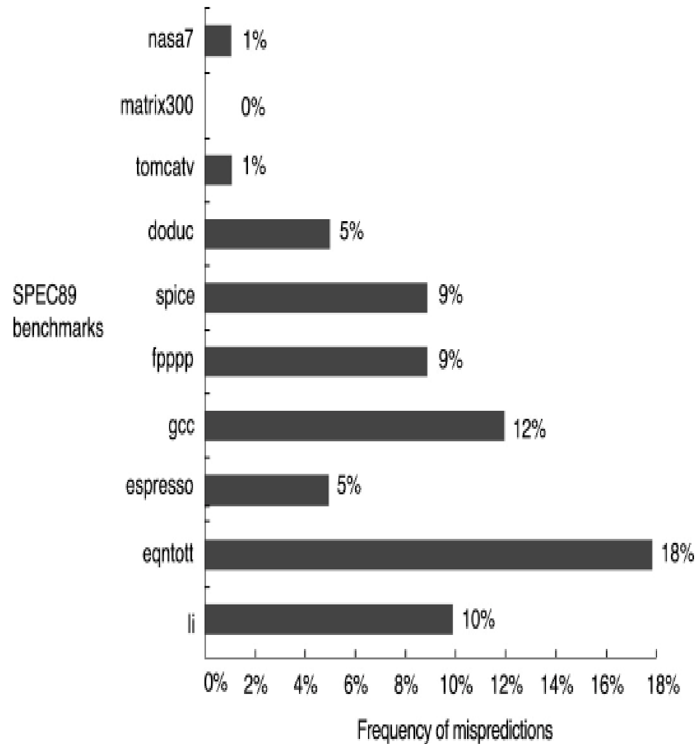
© 2003 Elsevier Science (USA). All rights reserved.

Basic Branch Prediction Buffers

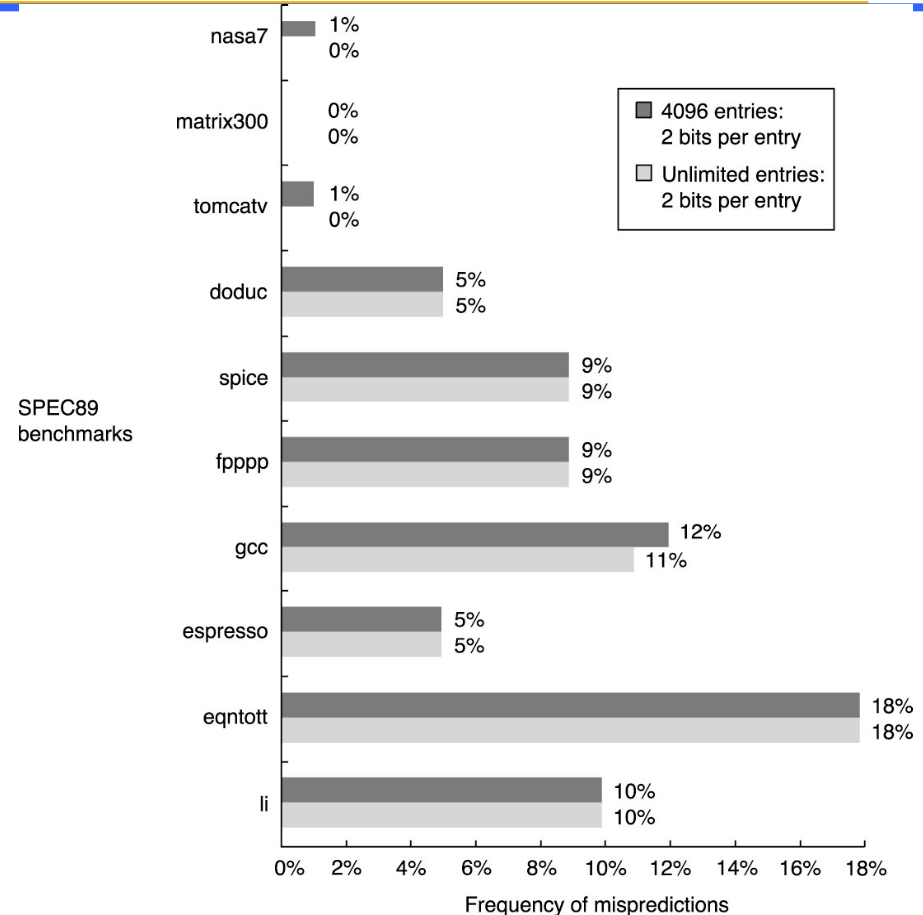
Branch History Table (BHT) - Small direct-mapped cache of T/NT bits



Prediction Accuracy of a 4K-entry 2-bit Prediction Buffer



© 2003 Elsevier Science (USA). All rights reserved.



© 2003 Elsevier Science (USA). All rights reserved.

The integer programs have higher branch frequencies.
We should focus on the accurancy of each predicator.

Correlating Branch Predictors

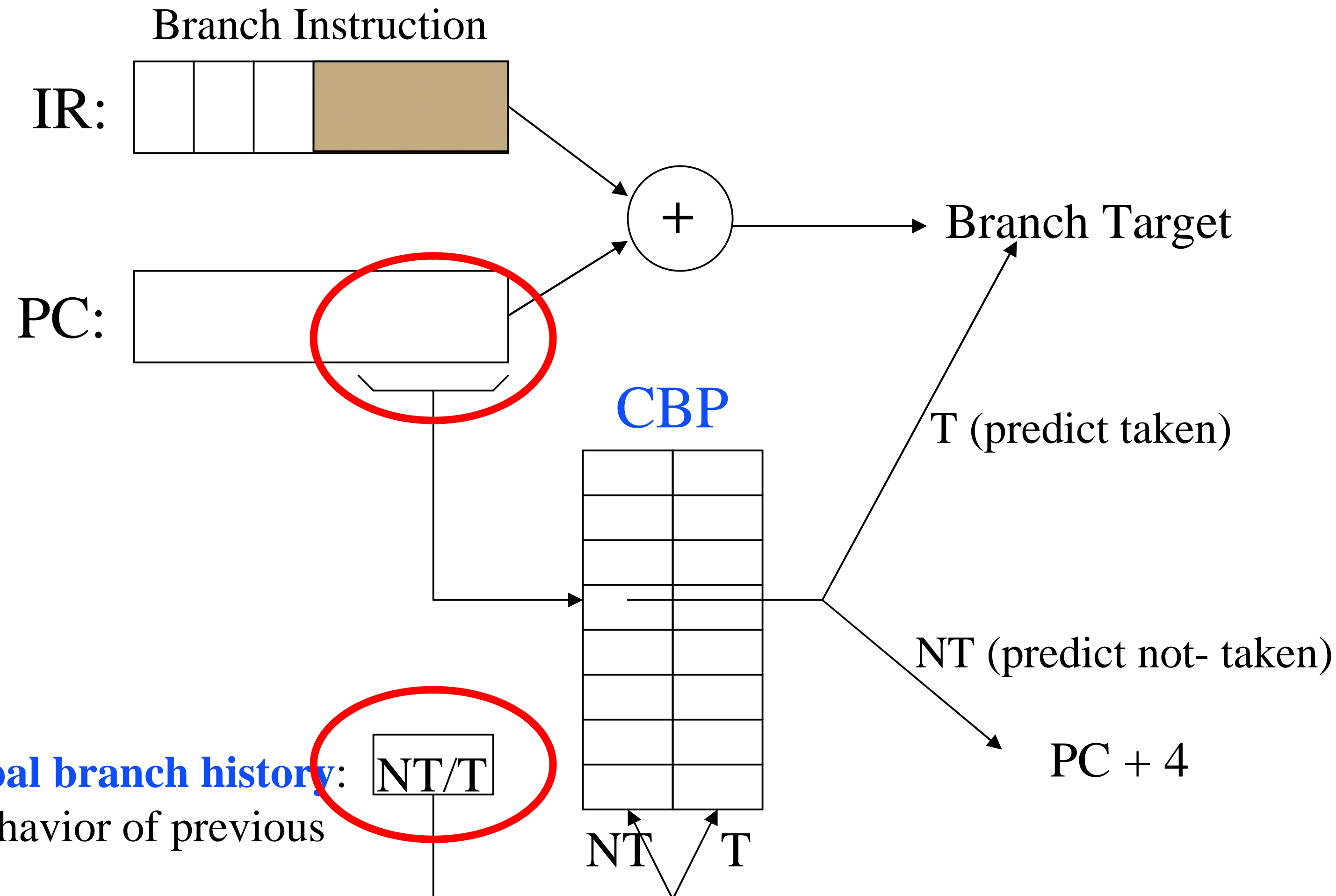
- **The previous schemes use only** the recent behavior of a signal branch **to predict the future behavior of the branch.**
- **Correlating Branch Predicator**
  - **Branch predictors that use the behavior of other branches to make a predication.**

Correlating Branches

- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table
- In general, (m,n) predictor means record last m branches to select between 2^m history talbes each with n-bit counters
  - Old 2-bit BHT is then a (0,2) predictor

Correlating Branch Predictors

Two-level Predictors – Use recent behavior of other (previous) branches, 1 bit of correction.

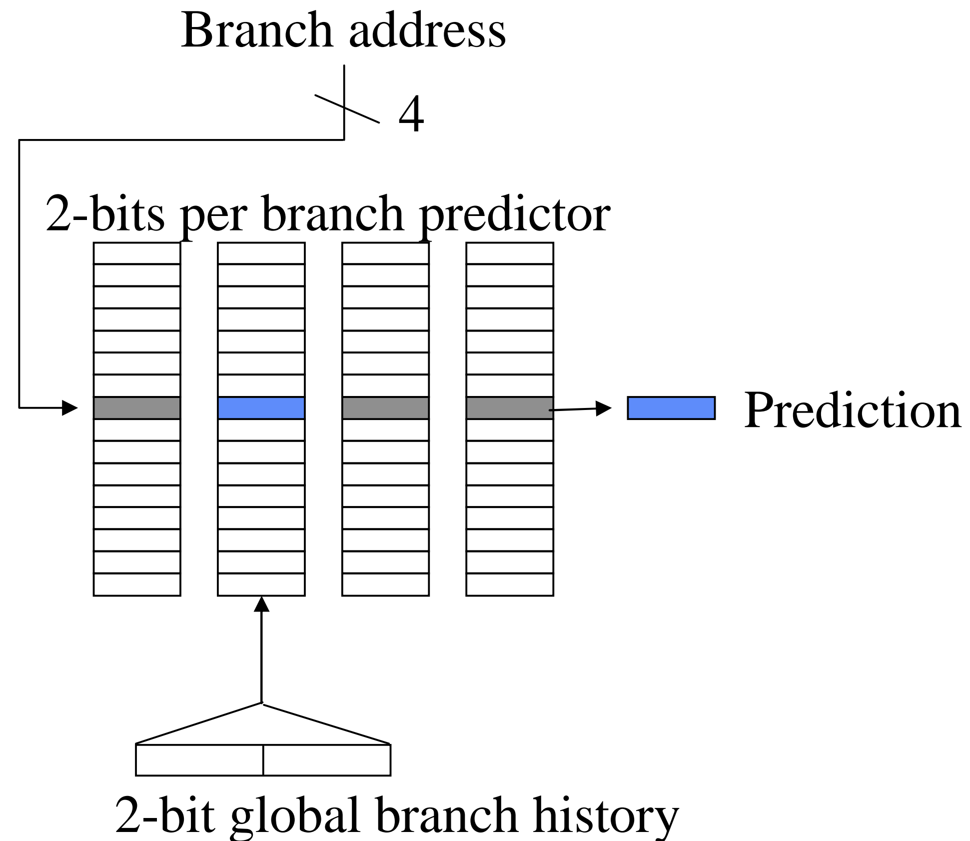


1-bit global branch history: (stores behavior of previous branch)

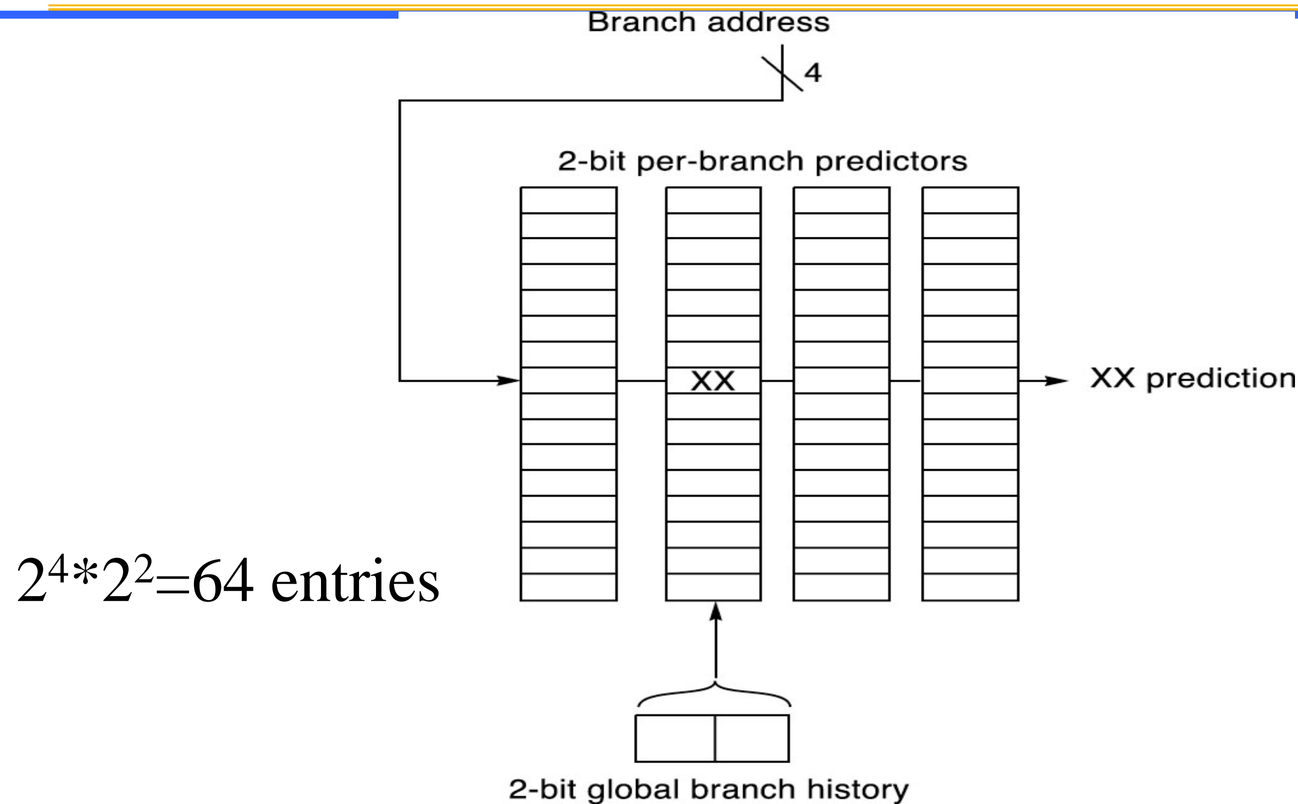
Correlating Branches

(2,2) predictor

– Behavior of recent branches selects between four predictions of next branch, updating just that prediction



(2,2) Branch Prediction Buffer



$2^4 * 2^2 = 64$ entries

© 2003 Elsevier Science (USA). All rights reserved.

The indexing is done by concatenating the global history bits and the number of required bits from the branch address.

Correlating Branches

- Often the behavior of one branch is correlated with the behavior of other branches.
- For example

C CODE		MIPS CODE
if (aa == 2)		DADDIU R3, R1, #-2;
		BNEZ R3, L1
aa = 0;		DADD R1, R0, R0
if (bb == 2)	L1:	DADDIU R3, R2, #2;
		BNEZ R3, L2
bb = 0;		ADD R2, R0, R0
if (aa != bb)	L2:	DSUBI R3, R1, R2;
		BEQZ R3, L3

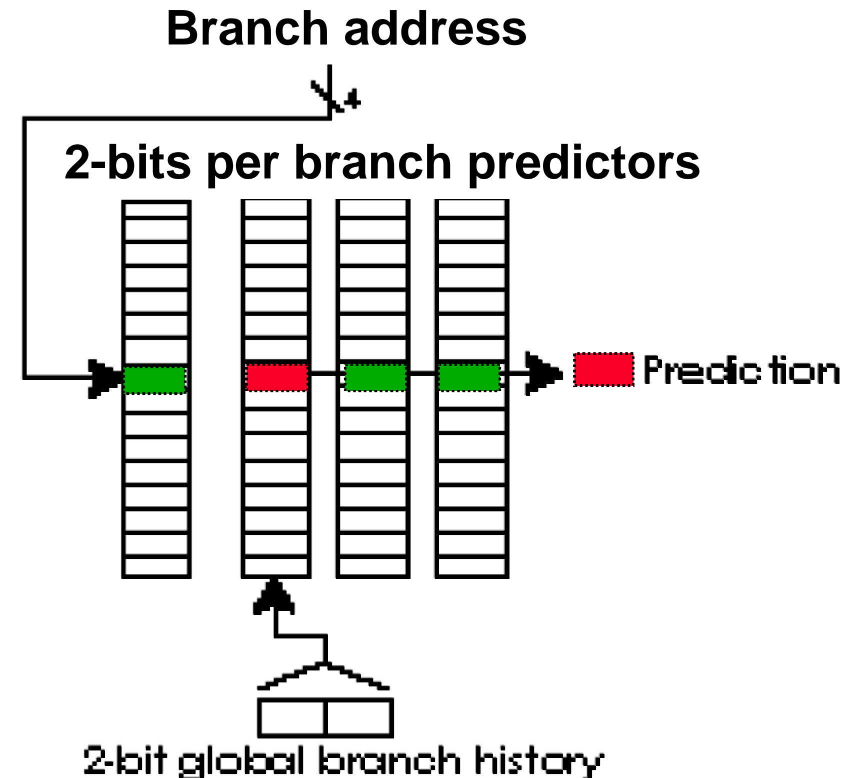
If the first two branches are not taken, the third one will be.

Correlating Predicators

- **Correlating predicators** or **two-level predictors** use the behavior of other branches to predict if the branch is taken.
  - An (m, n) predictor uses the behavior of the last m branches to chose from (2^m) n-bit predictors.
  - The branch predictor is accessed using the low order p bits of the branch address and the m-bit global history.
  - The number of bits needed to implement an (m, n) predictor, which uses p bits of the branch address is

    $2^m \times n \times 2^p$
  - In the figure, we have m = 2, n = 2, p=4

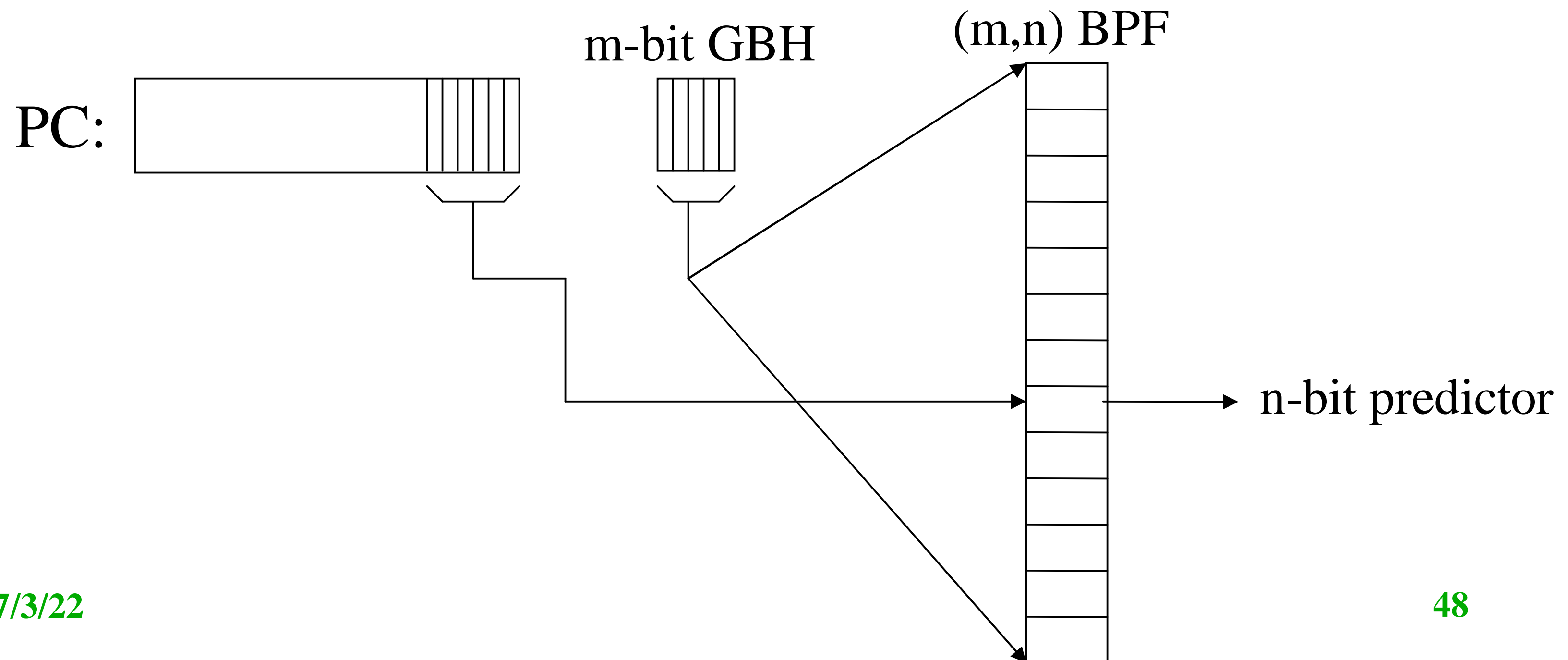
    $2^2 \times 2 \times 2^4 = 128$ bits



(2, 2) predictor

(m, n) Predictors

- Use behavior of the last m branches
  - Use last m branches = global branch history
  - Use n bit predictor
- 2^m n-bit predictors for each branch
- Simple implementation
  - Use m-bit shift register to record the behavior of the last m branches



(m,n) BHT (or prediction buffer)

- ***p*** **bits of buffer index = 2^p bit BHT**
- **Total bits:**

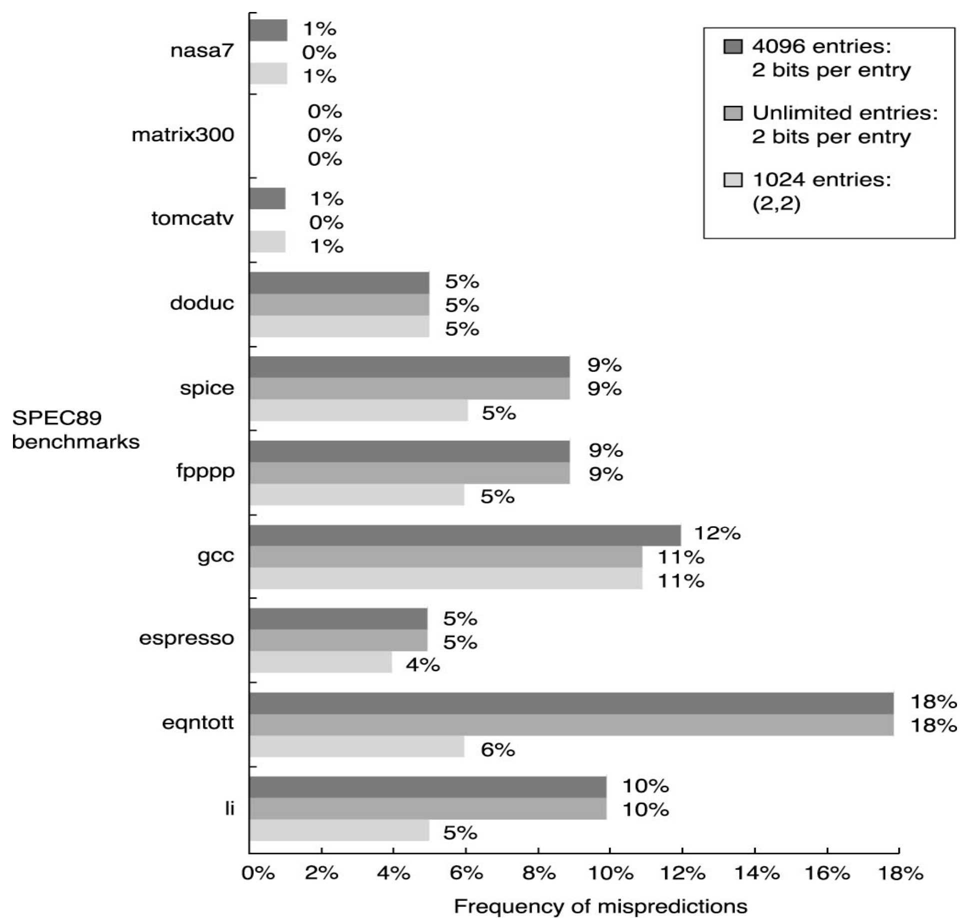
$$2^m \times n \times 2^p = \text{Total memory bits required}$$

  - **2^m banks of memory selected by the global branch history (which is just a shift register)**
  - **Use *p* bits of the branch address to select row**
  - **Get the n predictor bits in the entry to make the decision**

Size of the Buffers

- **Number of bits in a (m,n) predictor**
  - 2^m *x n x* Number of entries in the table
    - » *x*: the nubmer of predication entries selected by the branch address
- **Example – assume 8K bits in the BHT**
  - (0,1): 8K entries
  - (0,2): 4K entries
  - (2,2): 1K entries
  - (12,2): 1 entry!
    - » Does not use the branch address
    - » Relies only on the global branch history

Performance Comparison of 2-bit Predictors



© 2003 Elsevier Science (USA). All rights reserved.

Tournament Predictors

- **Motivation for correlating branch predictors is 2-bit predictor failed on important branches; by adding global information, performance improved**
- **Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information, and combine with a selector**
- **Hopes to select right predictor for right branch**

Tournament Predictors

Tournament predictor using, say, 4K 2-bit counters indexed by local branch address. Chooses between:

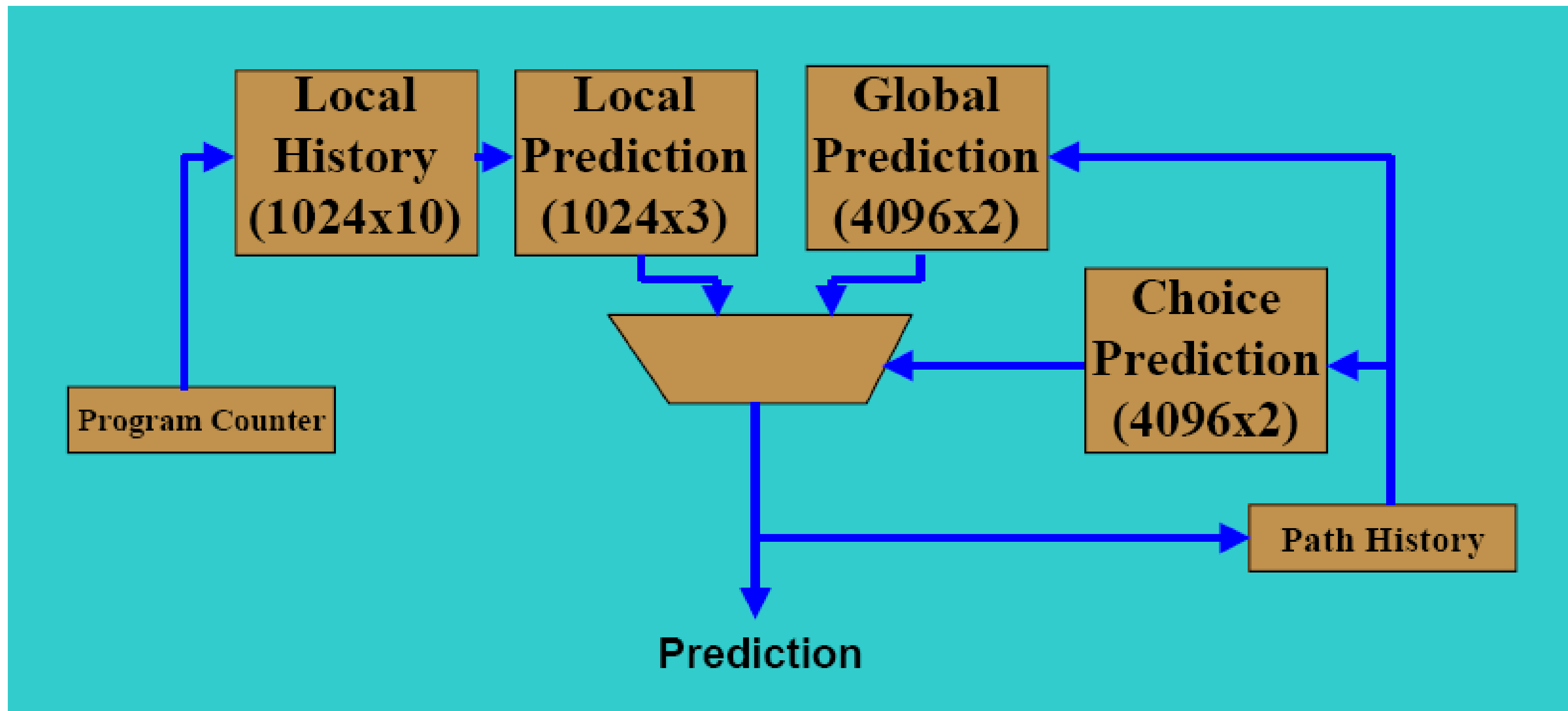
- Global predictor
  - 4K entries index by history of last 12 branches ($2^{12} = 4K$)
  - Each entry is a standard 2-bit predictor
- Local predictor
  - Local history table: 1024 10-bit entries recording last 10 branches, index by branch address
  - The pattern of the last 10 occurrences of that particular branch used to index table of 1K entries with 3-bit saturating counters

Tournament Predictor in Alpha 21264

- 4K 2-bit counters to choose from among a global predictor and a local predictor
- **Global predictor** also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - 12-bit pattern: ith bit 0 => ith prior branch not taken; ith bit 1 => ith prior branch taken;
- **Local predictor** consists of a 2-level predictor:
  - **Top level** a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
  - **Next level** Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction
- Total size: 4K*2 + 4K*2 + 1K*10 + 1K*3 = **29K bits!**

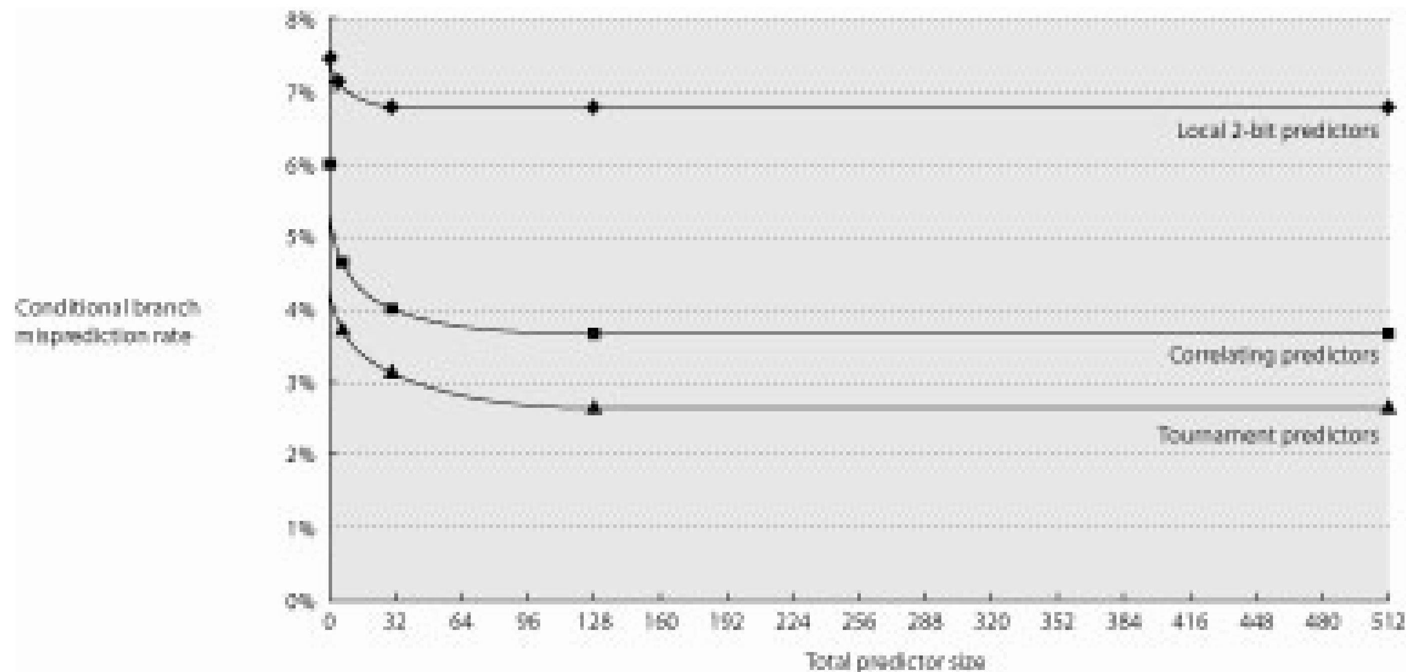
(~180,000 transistors)

Tournament Predictor in Alpha 21264



Comparing Predictors (Fig. 2.8)

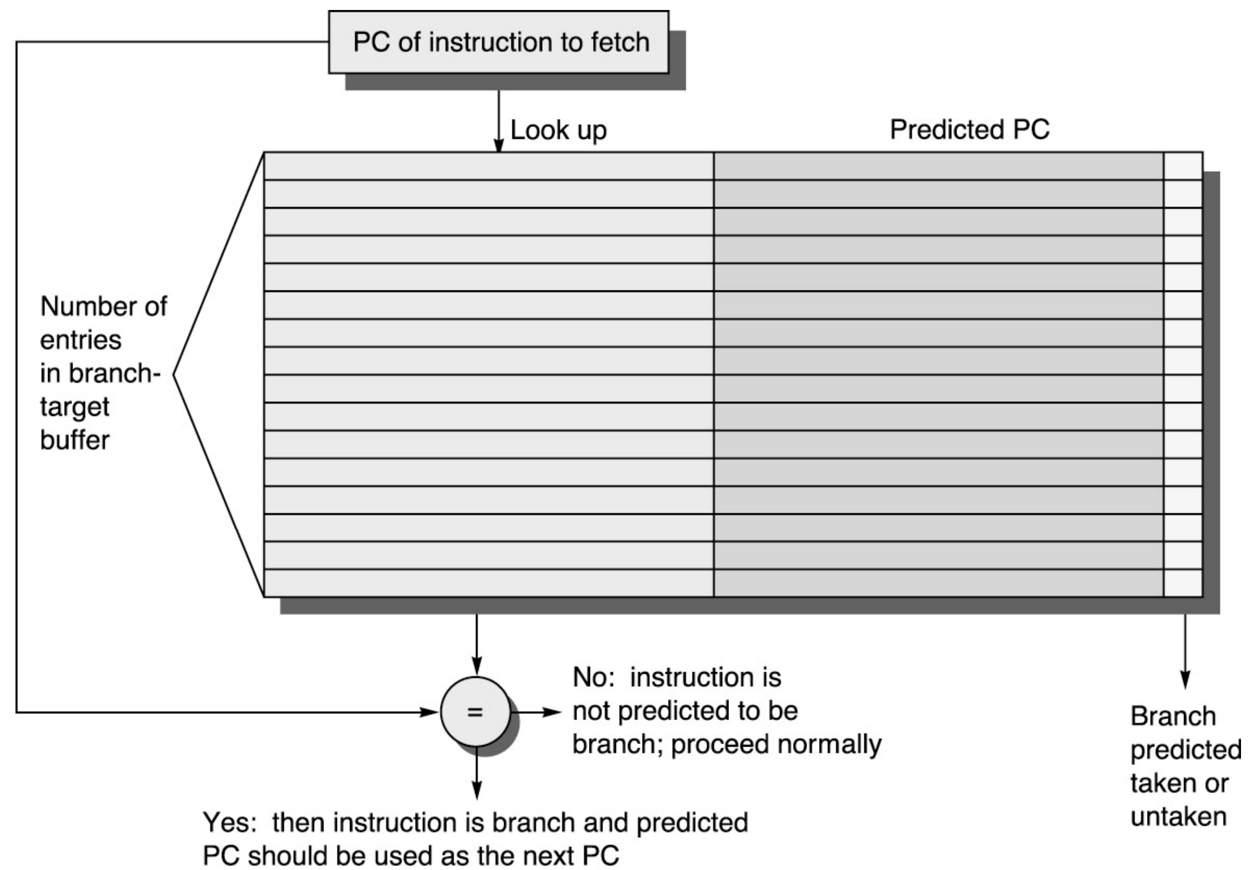
- **Advantage of tournament predictor is ability to select the right predictor for a particular branch**
  - **Particularly crucial for integer benchmarks.**
  - **A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks**



Branch Target Buffers (BTB)

- Branch target calculation is costly and stalls the instruction fetch.
- BTB stores PCs the same way as caches
- The PC of a branch is sent to the BTB
- When a match is found the corresponding Predicted PC is returned
- If the branch was predicted taken, instruction fetch continues at the returned predicted PC

Branch Target Buffers



© 2003 Elsevier Science (USA). All rights reserved.

Dynamic Branch Prediction Summary

- **Prediction becoming important part of execution**
- **Branch History Table: 2 bits for loop accuracy**
- **Correlation: Recently executed branches correlated with next branch**
  - Either different branches (GA)
  - Or different executions of same branches (PA)
- **Tournament predictors take insight to next level, by using multiple predictors**
  - usually one based on global information and one based on local information, and combining them with a selector
  - In 2006, tournament predictors using ≈ 30K bits are in processors like the Power5 and Pentium 4
- **Branch Target Buffer: include branch address & prediction**

Outline

- Instruction Level Parallelism (2.1)
- Compiler techniques for Exposing ILP (2.2)
- Reducing Branch Costs with Prediction (2.3)
- **Overcoming Data Hazards with Dynamic Scheduling (2.4)**
- **Dynamic Scheduling: Examples and the Algorithm (2.5)**
- **Hardware-Based Speculation (2.6)**
- **Exploiting ILP using Multiple Issue and Static Scheduling (2.7)**
- **Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation (2.8)**

Dynamic Scheduling

- Scheduling separates dependent instructions
  - Static – performed by the compiler
  - Dynamic – performed by the hardware
- Advantages of dynamic scheduling
  - Handles dependences unknown at compile time
  - Simplifies the compiler
  - Optimization is done at run time
- Disadvantages
  - Can not eliminate true data dependences

Advantages of Dynamic Scheduling

- Dynamic scheduling - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior
- It handles cases when dependences unknown at compile time
  - it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
- It allows code that compiled for one pipeline to run efficiently on a different pipeline
- It simplifies the compiler
- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling (next lecture)

Dynamic vs. Static Scheduling

- **Data hazards in a program to cause a processor to stall.**
- **With** static scheduling **the compiler tries to reorder these instructions during compile time to reduce pipeline stalls.**
  - **Uses less hardware**
  - **Can use more powerful algorithms**
- **With** dynamic scheduling **the hardware tries to rearrange the instructions during run-time to reduce pipeline stalls.**
  - **Simpler compiler**
  - **Handles dependencies not known at compile time**
  - **Allows code compiled for a different machine to run efficiently.**

HW Schemes: Instruction Parallelism

- **Key idea: Allow instructions behind stall to proceed**

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F12,F8,F14
```

- **Enables out-of-order execution and allows out-of-order completion (e.g., `SUBD`)**
  - **In a dynamically scheduled pipeline, all instructions still pass through issue stage in order (in-order issue)**
- **Will distinguish when an instruction *begins execution* and when it *completes execution*; between 2 times, the instruction is *in execution***
- **Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder**

Out-of-order Execution

- Central idea of dynamic scheduling
  - In-order execution:

DIVD F0, F2, F4	IF ID DIV …..
ADDD F10, F0, F8	IF ID stall stall stall …
SUBD F12, F8, F14	IF stall stall …..

  - Out-of-order execution:

DIVD F0, F2, F4	IF ID DIV …..
SUBD F12, F8, F14	IF ID A1 A2 A3 A4 …
ADDD F10, F0, F8	IF ID stall …..

Dynamic Scheduling Step 1

- **Simple pipeline had 1 stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue**
- **Split the ID pipe stage of simple 5-stage pipeline into 2 stages:**
- ***Issue*—Decode instructions, check for structural hazards**
- ***Read operands*—Wait until no data hazards, then read operands**

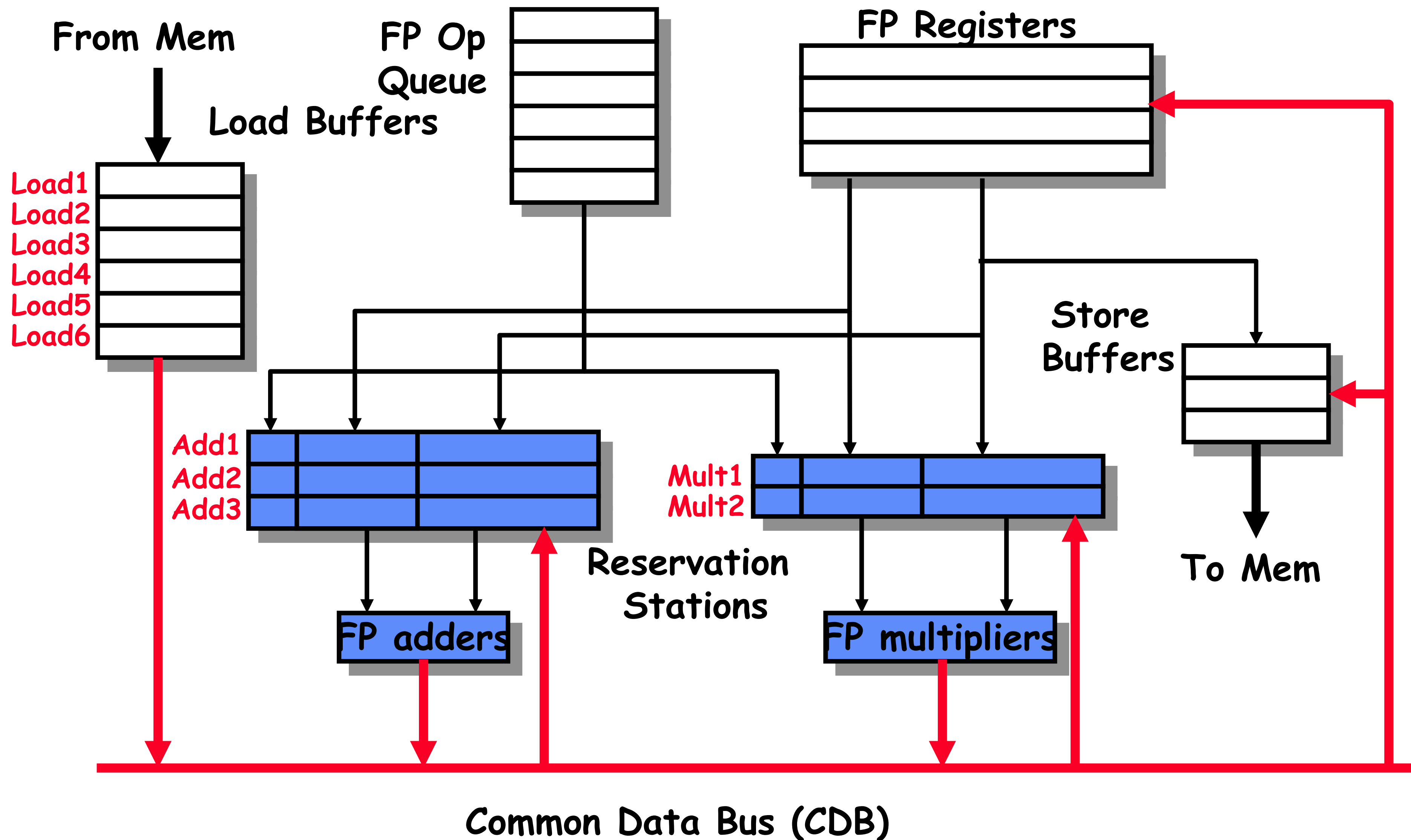
A Dynamic Algorithm: Tomasulo's

- For IBM 360/91 (before caches!)
  - $\Rightarrow$ Long memory latency
- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations
  - This led Tomasulo to try to figure out how to get more effective registers — renaming in hardware!
- Why Study 1966 Computer?
- The descendants of this have flourished!
  - Alpha 21264, Pentium 4, AMD Opteron, Power 5, …

Tomasulo Algorithm

- Control & buffers distributed with Function Units (FU)
  - FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
  - Renaming avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
  - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as FUs with RSs as well

Tomasulo Organization



Reservation Station Components

Op: Operation to perform in the unit (e.g., + or –)

Vj, Vk: Value of Source operands

– Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)

– Note: Qj,Qk=0 => ready

– Store buffers only have Qi for RS producing result

Busy: Indicates reservation station or FU is busy

A: memory address

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. **Issue—get instruction from FP Op Queue**

   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).

2. **Execute—operate on operands (EX)**

   When both operands ready then execute; if not ready, watch Common Data Bus for result

3. **Write result—finish execution (WB)**

   Write on Common Data Bus to all awaiting units; mark reservation station available

- Normal data bus: data + destination ("go to" bus)
- Common data bus: data + source ("come from" bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast
- Example speed: 2 clocks for Fl .pt. +,-; 10 for * ; 40 clks for /

Tomasulo Summary

- Advantages
  - Prevents register from being the bottleneck
  - Eliminates WAR, WAW hazards
  - Allows loop unrolling in HW
- Common Data Bus
  - Broadcasts results to multiple instructions
  - Central bottleneck
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming

Execution Cycle

- Dynamic DLX (much simpler)
  - 2 FP multiply (10 EX cycles)
  - 1 FP add (2 EX cycles)
  - 1 FP divide (40 EX cycles)
  - 1 integer unit (1 EX cycle)

Tomasulo Example

Instruction stream

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

FU count down

3 FP Adder R.S.
2 FP Mult R.S.

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	*FU*									

Clock cycle counter

Tomasulo Example Cycle 1

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU				Load1					

Tomasulo Example Cycle 2

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	*FU*		Load2		Load1					

Note: Can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- **Note: registers names are removed ("renamed") in Reservation Stations; MULT issued**
- **Load1 completing; what is waiting for Load1?**

Tomasulo Example Cycle 4

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1				

- **Load2 completing; what is waiting for Load2?**

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	*FU*	Mult1	M(A2)		M(A1)	Add1	Mult2			

- **Timer starts down for Add1, Mult1**

Tomasulo Example Cycle 6

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4		
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- **Issue ADDD here despite name dependency on F6?**

Tomasulo Example Cycle 7

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- **Add1 (SUBD) completing; what is waiting for it?**

Tomasulo Example Cycle 8

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 9

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- **Add2 (ADDD) completing; what is waiting for it?**

Tomasulo Example Cycle 11

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- **Write result of ADDD here?**
- **All quick instructions complete in this cycle!**

Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 13

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 14

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- **Mult1 (MULTD) completing; what is waiting for it?**

Tomasulo Example Cycle 16

Instruction status:

Instruction	j	k	*Issue*	*Exec Comp*	*Write Result*
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3	15	16
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- **Just waiting for Mult2 (DIVD) to complete**

Faster than light computation (skip a couple of cycles)

Tomasulo Example Cycle 55

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
55	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 56

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
56	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- **Mult2 (DIVD) is completing; what is waiting for it?**

Tomasulo Example Cycle 57

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Result			

- **Once again: In-order issue, out-of-order execution and out-of-order completion.**

Why can Tomasulo overlap iterations of loops?

- **Register renaming**
  - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- **Reservation stations**
  - Permit instruction issue to advance past integer control flow operations
  - Also buffer old values of registers - totally avoiding the WAR stall
- **Other perspective: Tomasulo building data flow dependency graph on the fly**

Tomasulo's scheme offers 2 major advantages

1. Distribution of the hazard detection logic
   - **distributed reservation stations and the CDB**
   - **If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB**
   - **If a centralized register file were used, the units would have to read their results from the registers when register buses are available**
2. Elimination of stalls for WAW and WAR hazards

Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA:AQA 2/e, but not in silicon!
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units $\Rightarrow$high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - » Multiple CDBs $\Rightarrow$ more FU logic for parallel assoc stores
- Non-precise interrupts!
  - We will address this later

And In Conclusion … #1

- **Leverage Implicit Parallelism for Performance: Instruction Level Parallelism**
- **Loop unrolling by compiler to increase ILP**
- **Branch prediction to increase ILP**
- **Dynamic HW exploiting ILP**
  - **Works when can't know dependence at compile time**
  - **Can hide L1 cache misses**
  - **Code for one machine runs well on another**

And In Conclusion … #2

- Reservations stations: *renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards
  - Allows loop unrolling in HW
- Not limited to basic blocks (integer units gets ahead, beyond branches)
- Helps cache misses as well
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are Intel Pentium 4, IBM Power 5, AMD Athlon/Opteron, …

Loop Iterations

```
Loop: LD    F0, 0(R1)
      MULTD F4,F0,F2
      SD    0(R1), F4
      SUBI  R1, R1, #8
      BNEZ  R1, Loop
```

Instruction		Instruction status		
Instruction	From iteration	Issue	Execute	Write result
LD F0,0(R1)	1	√	√	
MULTD F4,F0,F2	1	√		
SD 0(R1),F4	1	√		
LD F0,0(R1)	2	√	√	
MULTD F4,F0,F2	2	√		
SD 0(R1),F4	2	√		

	Reservation stations					
Name	Busy	Fm	Vj	Vk	Qj	Qk
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULT		Regs[F2]	Load1	
Mult2	Yes	MULT		Regs[F2]	Load2	

	Register status								
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Load2		Mult2						

	Load buffers		
Field	Load 1	Load 2	Load 3
Address	Regs[R1]	Regs[R1]-8	
Busy	Yes	Yes	No

	Store buffers		
Field	Store 1	Store 2	Store 3
Qi	Mult1	Mult2	
Busy	Yes	Yes	No
Address	Regs[R1]	Regs[R1]-8	

Loop Iterations

- **The loop is unrolled dynamically by the hardware, using the reservation station.**

Tomasulo Loop Example

```
Loop: LD        F0    0     R1
      MULTD     F4    F0    F2
      SD        F4    0     R1
      SUBI      R1    R1    #8
      BNEZ      R1    Loop
```

- Assume Multiply takes 4 clocks
- Assume first load takes 8 clocks (cache miss), second load takes 1 clock (hit)
- To be clear, will show clocks for SUBI, BNEZ
- Reality: integer instructions ahead

Loop Example

Instruction status:

ITER	Instruction		*j*	*k*	Issue	Exec Comp	Write Result
1	LD	F0	0	R1			
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	**80**	*Fu*									

Loop Example Cycle 1

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	Regs[R1]+0	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
1	80	Fu	Load1								

Loop Example Cycle 2

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result		Busy	Addr	Fu
1	LD	F0	0	R1	1			Load1	Yes	Regs[R1]+0	
1	MULTD	F4	F0	F2	2			Load2	No		
1	SD	F4	0	R1				Load3	No		
2	LD	F0	0	R1				Store1	No		
2	MULTD	F4	F0	F2				Store2	No		
2	SD	F4	0	R1				Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj (S1)	Vk (S2)	Qj (RS)	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
2	80	Fu	Load1		Mult1						

Loop Example Cycle 3

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	Regs[R1]+0	
Load2	No		
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		F0	F2	F4	F6	F8	F10	F12	...	F30
3	**80**	*Fu*	Load1		Mult1						

- **Implicit renaming sets up "DataFlow" graph**

Loop Example Cycle 4

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	Regs[R1]+0	
Load2	No		
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
4	80	Fu	Load1		Mult1						

- **Dispatching SUBI Instruction**

Loop Example Cycle 5

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	Regs[R1]+0	
Load2	No		
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
5	**72**	*Fu*	Load1		Mult1						

- **And, BNEZ instruction**

Loop Example Cycle 6

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	Regs[R1]+0	
Load2	Yes	Regs[R1]-8	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
6	72	Fu	Load2		Mult1						

- **Notice that F0 never sees Load from location 80**

Loop Example Cycle 7

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	Regs[R1]+0	
Load2	Yes	Regs[R1]-8	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
7	72	Fu	Load2		Mult2						

- **Register file completely detached from computation**
- **First and Second iteration completely overlapped**

Loop Example Cycle 8

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	Yes	Regs[R1]+0	
Load2	Yes	Regs[R1]-8	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]-8	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
8	72	Fu	Load2		Mult2						

Loop Example Cycle 9

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	Yes	Regs[R1]+0	
Load2	Yes	Regs[R1]-8	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]-8	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
9	72	Fu	Load2		Mult2						

- **Load1 completing: who is waiting?**
- **Note: Dispatching SUBI**

Loop Example Cycle 10

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	Yes	Regs[R1]-8	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]-8	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	M[80]	M[]	R(F2)		
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
10	**64**	*Fu*	Load2		Mult2						

- **Load2 completing: who is waiting?**
- **Note: Dispatching BNEZ**

Loop Example Cycle 11

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]-8	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	Multd	M[80]	R(F2)		
4	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
11	64	Fu	Load3		Mult2						

- **Next load in sequence**

Loop Example Cycle 12

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]-8	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	Multd	M[80]	R(F2)		
3	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

```
LD     F0  0    R1
MULTD  F4  F0   F2
SD     F4  0    R1
SUBI   R1  R1   #8
BNEZ   R1  Loop
```

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	**64**	*Fu*	Load3		Mult2						

- **Why not issue third multiply?**

Loop Example Cycle 13

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]-8	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	Multd	M[80]	R(F2)		
2	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

```
LD     F0  0     R1
MULTD  F4  F0    F2
SD     F4  0     R1
SUBI   R1  R1    #8
BNEZ   R1  Loop
```

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
13	**64**	*Fu*	Load3		Mult2						

Loop Example Cycle 14

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]-8	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	Multd	M[80]	R(F2)		
1	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
14	64	Fu	Load3		Mult2						

- **Mult1 completing. Who is waiting?**

Loop Example Cycle 15

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	Yes	Regs[R1]+0	[80]*R2
Store2	Yes	Regs[R1]-8	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	**64**	*Fu*	Load3		Mult2						

- **Mult2 completing. Who is waiting?**

Loop Example Cycle 16

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	Yes	Regs[R1]+0	[80]*R2
Store2	Yes	Regs[R1]-8	[72]*R2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
16	64	Fu	Load3		Mult1						

Loop Example Cycle 17

Iter	Instr	j	k	Issue	Exec Comp	Write Result		Busy	Addr	Fu
1	MULTD F4	F0	F2	2	14	15	Load2	No		
1	SD F4	0	R1	3			Load3	Yes	Regs[R1]-16	
2	LD F0	0	R1	6	10	11	Store1	Yes	Regs[R1]+0	[80]*R2
2	MULTD F4	F0	F2	7	15	16	Store2	Yes	Regs[R1]-8	[72]*R2
2	SD F4	0	R1	8			Store3	Yes	Regs[R1]-16	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk (S1)	Qj (S2)	Qk (RS)
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
17	64	Fu	Load3		Mult1						

Loop Example Cycle 18

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	Yes	Regs[R1]+0	[80]*R2
Store2	Yes	Regs[R1]-8	[72]*R2
Store3	Yes	Regs[R1]-16	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
18	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 19

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	No		
Store2	Yes	Regs[R1]-8	[72]*R2
Store3	Yes	Regs[R1]-16	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
19	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 20

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	20

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	Regs[R1]-16	
Store1	No		
Store2	No		
Store3	Yes	Regs[R1]-16	Mult1

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
20	**64**	*Fu*	Load3		Mult1						