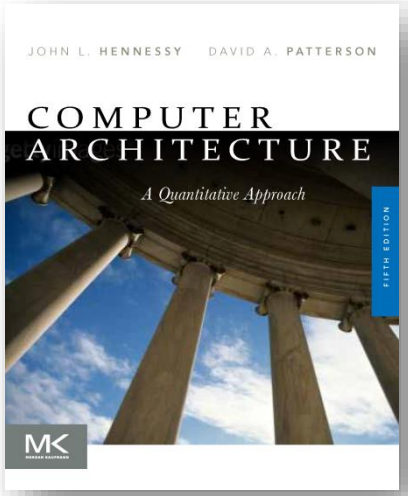


Chapter 3

Instruction-Level Parallelism and Its Exploitation

Outline

- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions (ILP)
  - Exploits “Instruction Level Parallelism”
- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMD processors
    - In Intel Core series
  - Compiler-based static approaches
    - Not as successful outside of scientific applications
    - In ARM Cortex-A8

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to minimize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited
  - Typical size of basic block = 4-7 instructions
    - A straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - The average dynamic branch frequency is often between 15% and 25%
    - Must optimize across branches

Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs), in Chapter 4
- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

Data Dependence

- 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];
```

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall

- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism

- Dependencies that flow through memory locations are difficult to detect

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

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

  2. or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$

dependence chain

- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence ⇒ data dependence in source code ⇒ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, called a Read After Write (RAW) hazard

這種相依性所產生的**hazard**為**RAW hazard**

Data Dependence

- The dependence chain can be as long as the entire program.
- ADD.D F0, F2, F4 ?

浮點數指令

- Example

```
Loop:           L.D    F0, 0(R1)
                ADD.D  F4, F0, F2
                S.D    F4, 0(R1)
                DADDUI    R1, R1, #8
                BNE    R1, R2, Loop
```

Dependence in floating-point data

```
    Loop:       L.D    F0, 0(R1)
                ADD.D  F4, F0, F2
                S.D    F4, 0(R1)
```

Dependence in integer data

```
                DADDUIR1, R1, #8
                BNE    R1, R2, Loop
```

Data Dependence and Hazards

- Executing two instruction with data dependence simultaneously will cause *a processor with pipeline interlocks* to detect a hazard and stall, thereby reducing or eliminating the overlap.
- In a processor without interlocks that relies on compiler scheduling, the compiler can not schedule dependent instructions in such a way that they completely overlap, *since the program will not execute correctly*.

Data Dependence and Hazards

- A dependence can be overcome in two different ways
  1. Maintaining the dependence but avoiding a hazard
     - Scheduling code by the compiler or the hardware
  2. Eliminating a dependence by transforming the code

Hardness

- A data value may flow between instructions either through registers or through memory locations
- It gets more complicated when branches intervene.
- It also is difficult when dependence through memory location since
  - two addresses may refer to the same location but look different
    - 100(R4) and 20(R6)
  - or may look the same but locate different addresses.
    - 20(R4) and 20(R4)

Other Factors

- Data Hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)

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

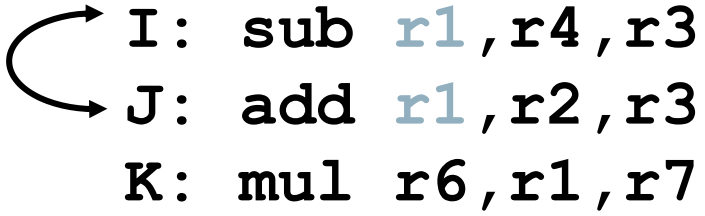
  2. or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence $\Rightarrow$ data dependence in source code $\Rightarrow$ effect of original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

Example

```
Loop:   L.D   F0, 0(R1)
        ADD.DF4, F0, F2
        S.D   F4, 0(R1)
        DADDUI    R1, R1, #8
        BNE   R1, R2, Loop
```

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 output-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

Name Dependence

- 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

Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, *but is a problem when reordering instructions*
  - *Antidependence*: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - *Output dependence*: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use renaming techniques

Dependence and Hazard

- A hazard is created whenever
  - There is a dependence between instructions
  - And they are close enough that overlap during execution would change the order of access to the operand involved in the dependence.

Other Factors

- Control Dependence
  - Ordering of instruction i with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

無相依性的不能變成有相依性，
有相依性的不能變成無相依性，
否則程式邏輯將產生問題。

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 may 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`?

在改變完指令的執行順序後，
不能產生額外的例外

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

Data Flow

- Example:

```
        DADDU     R1, R2, R3
        BEQZ      R12, skip
        DSUBU     R4, R5, R6
        DADDU     R5, R4, R9
skip:OR           R7,R8,R9
```

Suppose we know that the register destination of the DSUB instruction (R4) was unused after the instruction labeled skip. We can move the DSUBU instruction before the branch since the data flow can not be affect by the change.

Outline

- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction

調整指令位置，以達到在**pipeline**中盡量不要出現**stall**的情況

Pipeline Scheduling

- To keep a pipeline full,
  - Parallelism among instructions must be exploited by finding *sequences of unrelated instructions* that can be overlapped in the pipeline.
- To avoid a pipeline stall,
  - A dependent instruction must be separated from the source instruction by *a distance in clock cycles equal to the pipeline latency* of that instruction latency.

Compiler Techniques for Exposing ILP

- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

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
Load double	Store double	0

Pipeline Stalls

```
Loop:   L.D      F0,0(R1)
        stall
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,0(R1)
        DADDUI R1,R1,#-8
        stall (assume integer load latency is 1)
        BNE R1,R2,Loop
```

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
Load double	Store double	0

Pipeline Scheduling

Scheduled code:

```
Loop:   L.D      F0,0(R1)
        DADDUI R1,R1,#-8
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,8(R1)
        BNE R1,R2,Loop
```

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
Load double	Store double	0

Unroll Loop

- A scheme for increasing the number instructions relative to the *branch and overhead instructions*
- Unrolling replicates the loop body multiple times, *adjusting the loop termination code*.
- Because unrolling eliminates the branch, it allows *instructions from different iterations* to be scheduled together.
- The resulting use of the same registers could prevent us from effectively scheduling the loop.
  - Use different registers for different iterations-> *increasing the required number of registers*

Loop Unrolling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1) ;drop DADDUI & BNE
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1) ;drop DADDUI & BNE
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1) ;drop DADDUI & BNE
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```

- note: number of live registers vs. original loop

Unroll Loop Four Times (straightforward way)

Rewrite loop to minimize stalls?

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

1 cycle stall

2 cycles stall

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)

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
Load double	Store double	0

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   L.D F0,0(R1)
        L.D F6,-8(R1)
        L.D F10,-16(R1)
        L.D F14,-24(R1)
        ADD.D F4,F0,F2
        ADD.D F8,F6,F2
        ADD.D F12,F10,F2
        ADD.D F16,F14,F2
        S.D F4,0(R1)
        S.D F8,-8(R1)
        DADDUI R1,R1,#-32
        S.D F12,16(R1)
        S.D F16,8(R1)
        BNE R1,R2,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      DADDUI  R1,R1,#-32
12      S.D     F12,16(R1)
13      S.D     F16, 8(R1)
14      BNE     R1, R2, ,LOOP
```

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

14 clock cycles, or 3.5 per iteration

Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = *n*
  - Goal: make *k* copies of the loop body
  - Generate pair of loops:
    - First executes *n* mod *k* times
    - Second executes *n* / *k* times
    - “Strip mining”

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: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation

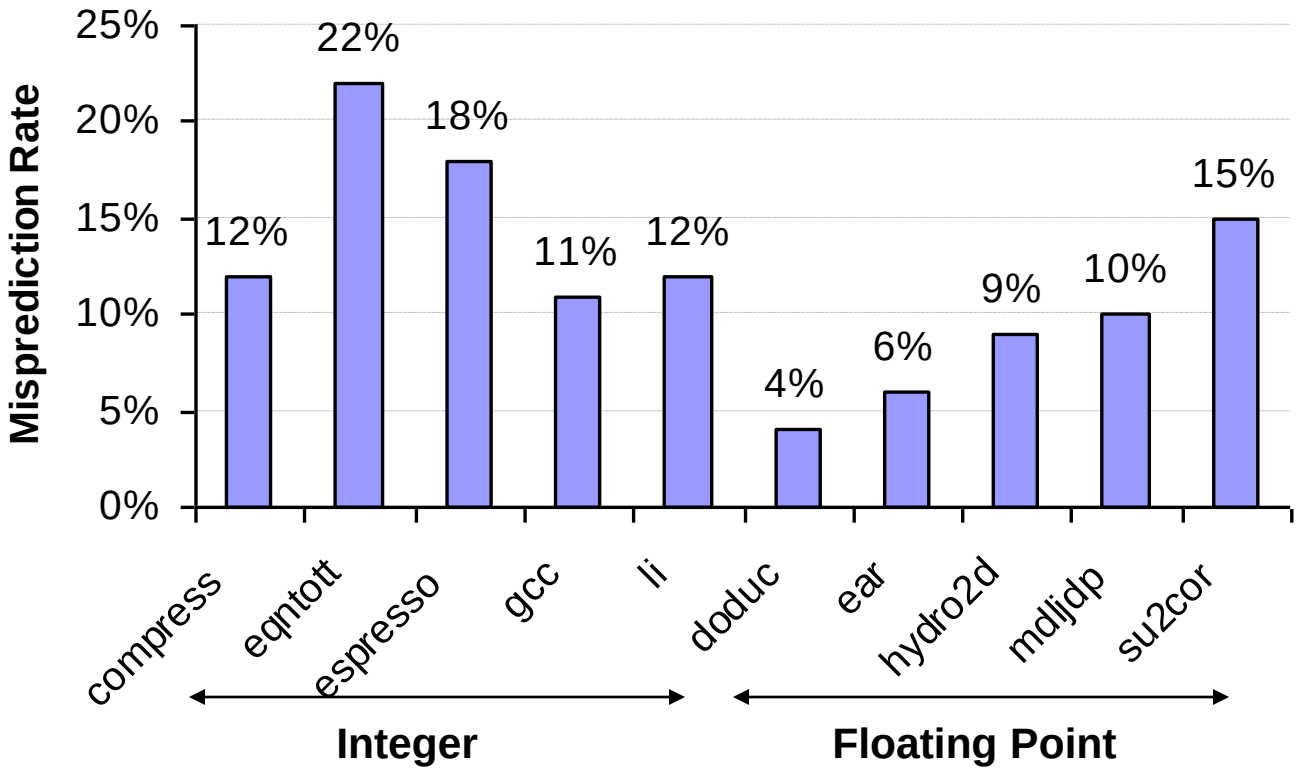
Control Hazards

- 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 (Compiler)

- Previous lecture showed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically when compiling
- 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:



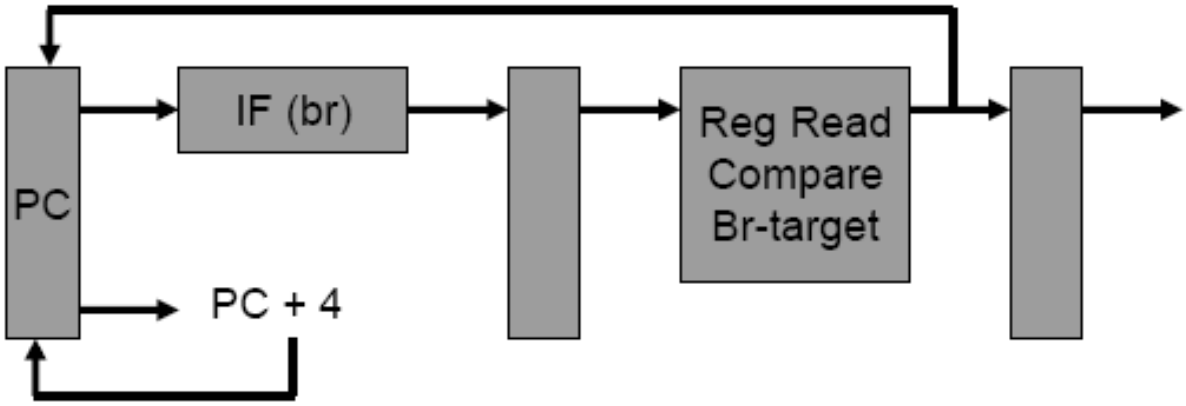
Dynamic Hardware Prediction

- 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
  - 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

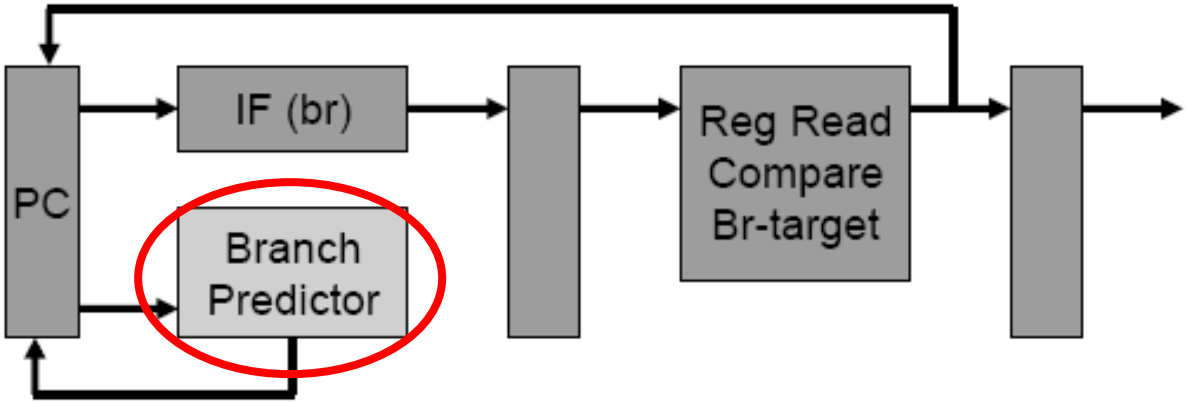
- Why does prediction work?
  - Underlying algorithm has *regularities*
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - 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 instruction is fetched
- *One stall cycle* per incorrect branch

Pipeline with Branch Predictor

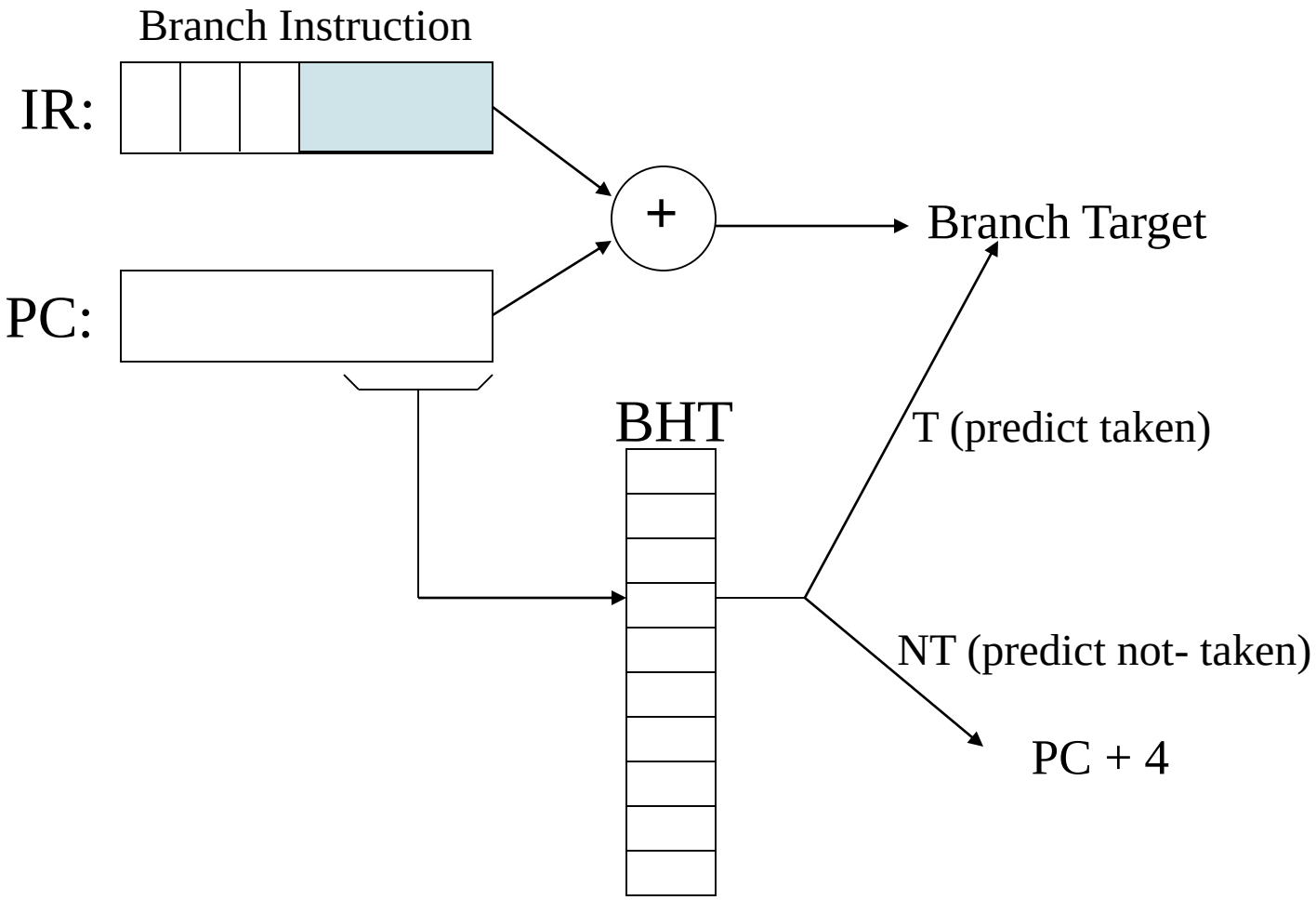


Basic Branch Prediction Buffers

- A branch-prediction buffer (branch history table) is a small cache indexed by *the lower potion of the address of the branch instruction*.
  - The cache 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.

Basic Branch Prediction Buffers

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



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 = f(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

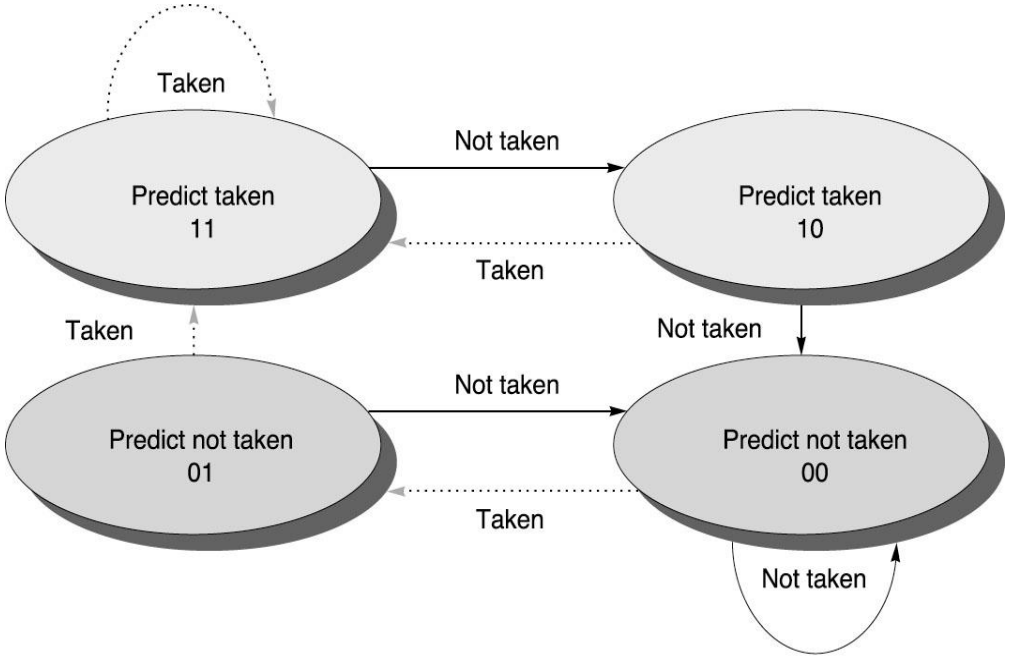
- For each branch, maintain a 2-bit saturating counter:
  if the branch is taken: counter = min(3,counter+1)
  if the branch is not taken: counter = max(0,counter-1)
- If (counter >= 2), predict taken, else predict not taken
- Advantage: a few atypical branches will not influence the prediction (a better 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)

2-bit Branch Prediction Buffers

Use an n-bit saturating counter

Only the loop exit causes a misprediction

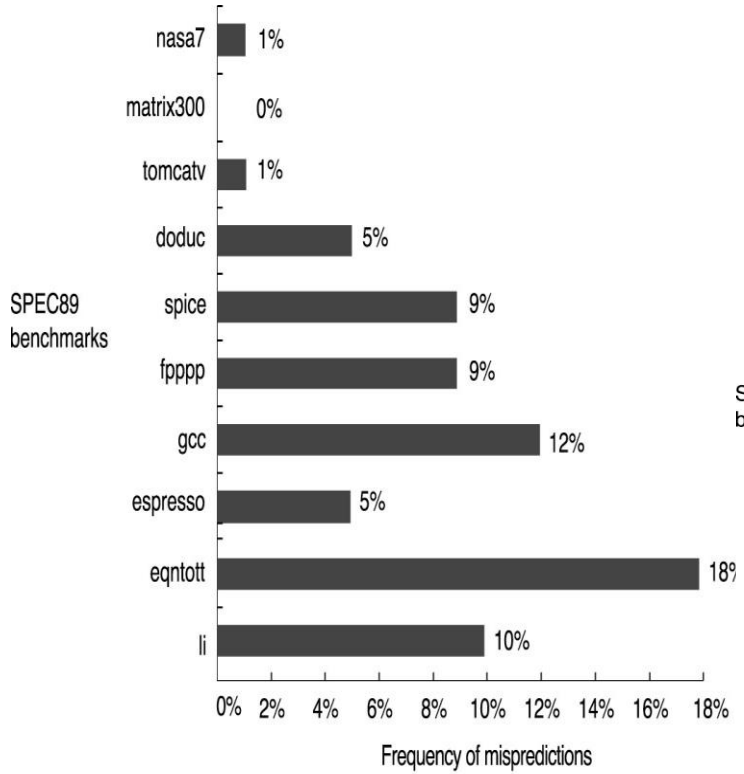
2-bit predictor almost as good as any general n-bit predictor

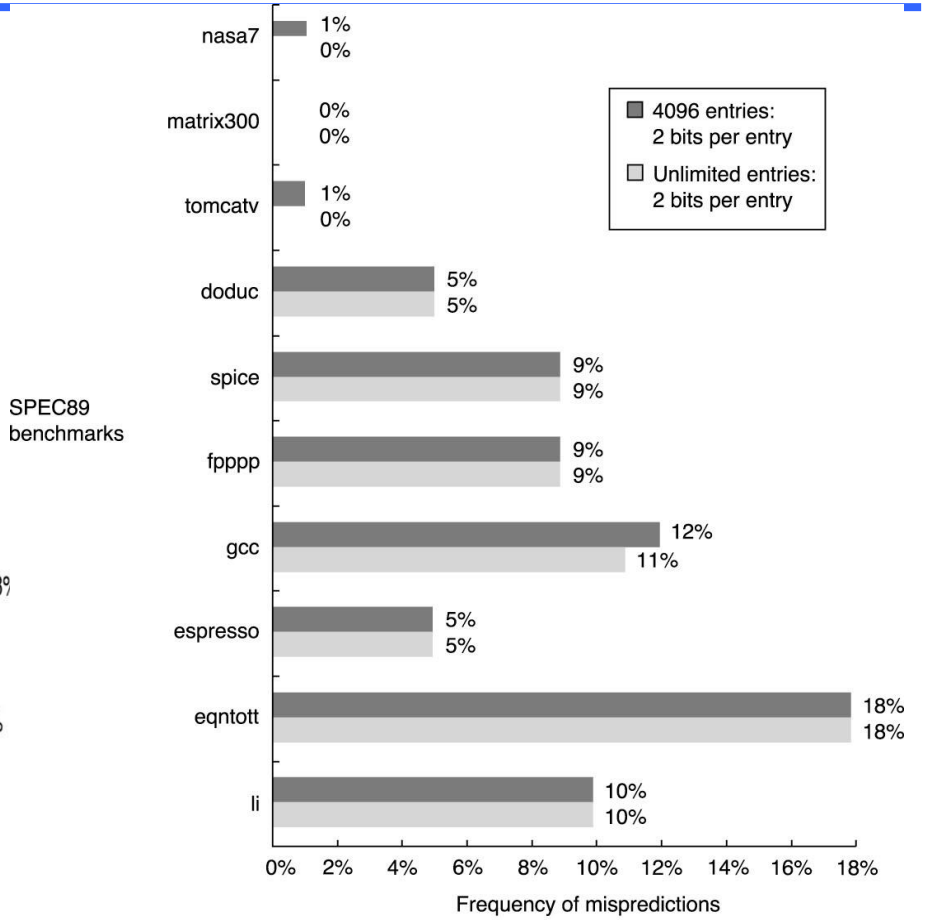


N-bit Branch Prediction Buffers

- 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.

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





The integer programs have higher branch frequencies.

We should focus on the accuracy 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.

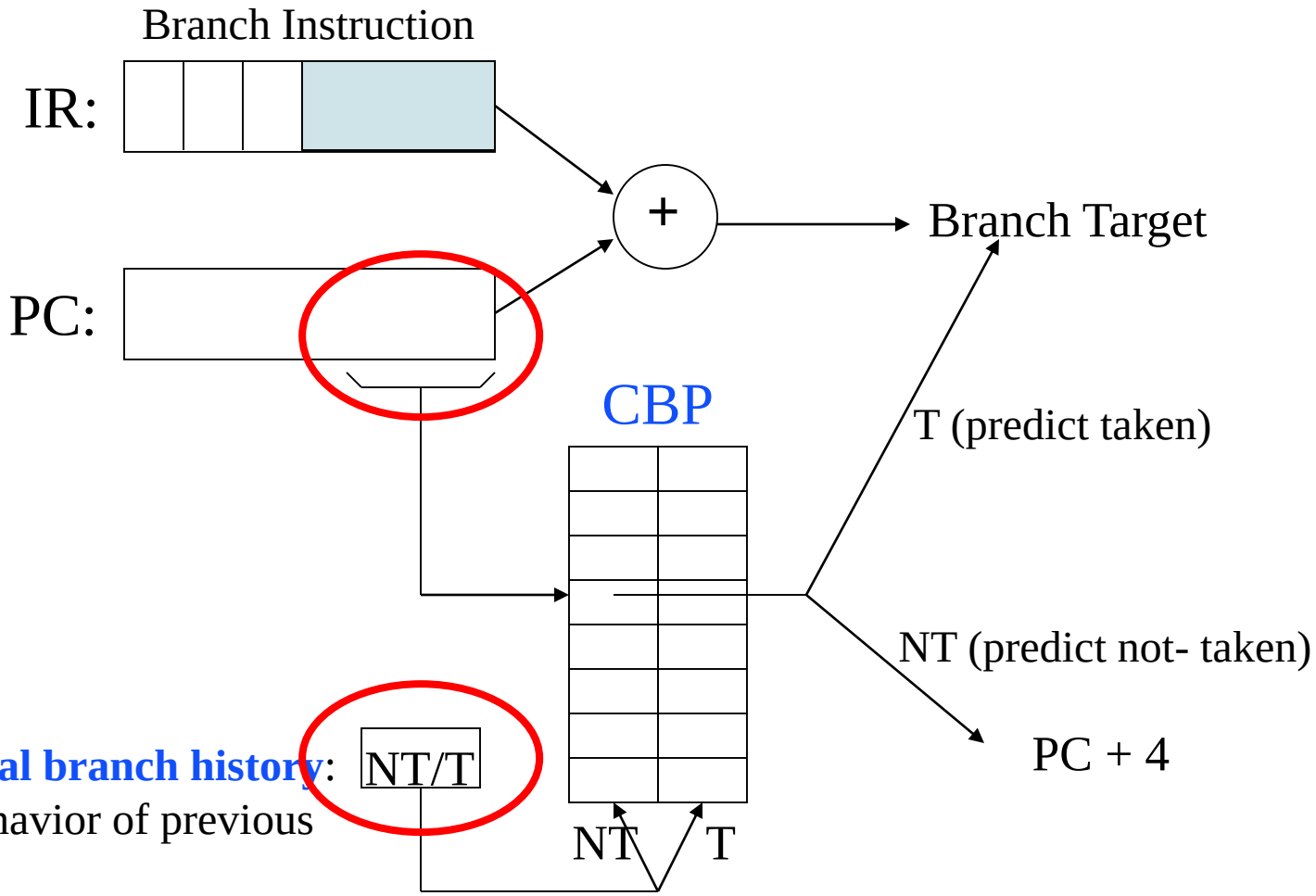
除考慮本身即將執行的**branch**的過去執行行為外，也考慮到鄰近於此**branch**指令的其他**branch**指令的執行結果

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 tables 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.

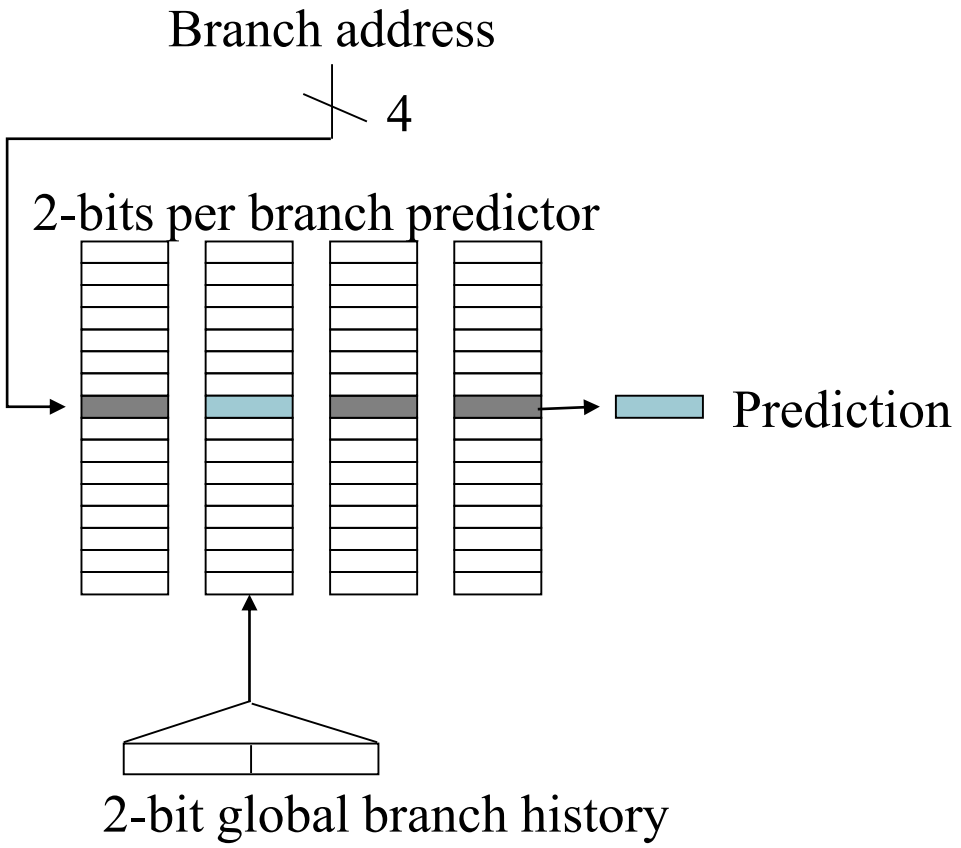


1-bit global branch history: (stores behavior of previous branch)

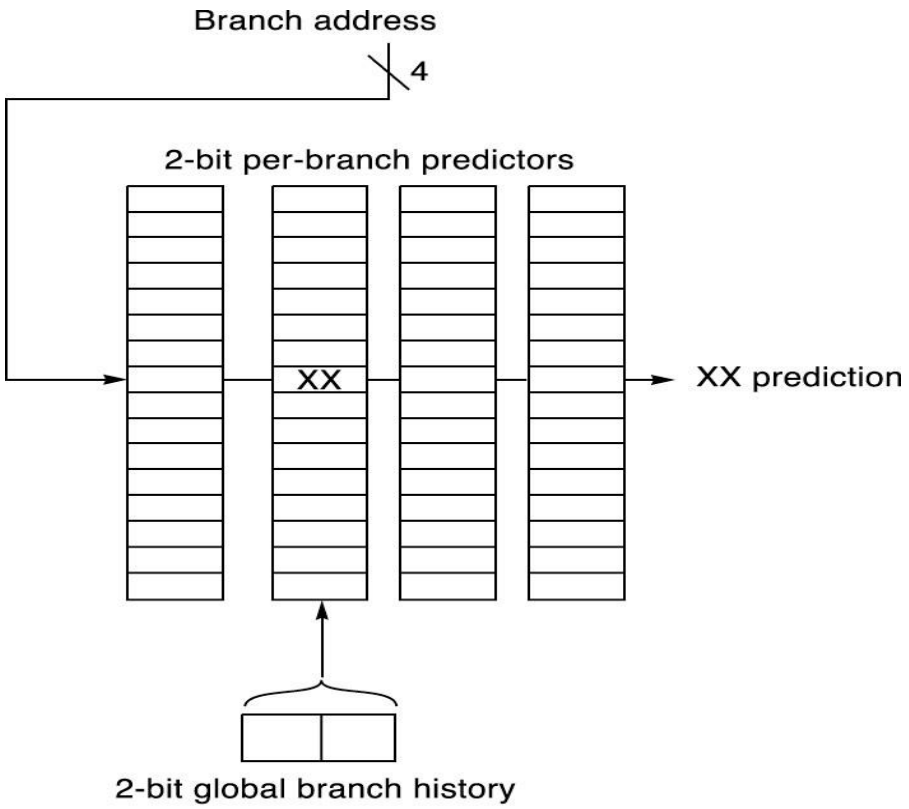
Correlating Branches

(2,2) predictor

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



(2,2) Branch Prediction Buffer



$2^4*2^2=64$ cells

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	
false	if (aa == 2)		DADDIU R3, R1, #-2;	
			BNEZ R3, L1	**Not taken**
false	aa = 0;		DADD R1, R0, R0	
	if (bb == 2)	L1:	DADDIU R3, R2, #2;	
			BNEZ R3, L2	**Not taken**
	bb = 0;		ADD R2, R0, R0	
true	if (aa != bb)	L2:	DSUBI R3, R1, R2;	
			BEQZ R3, L3	**Taken**

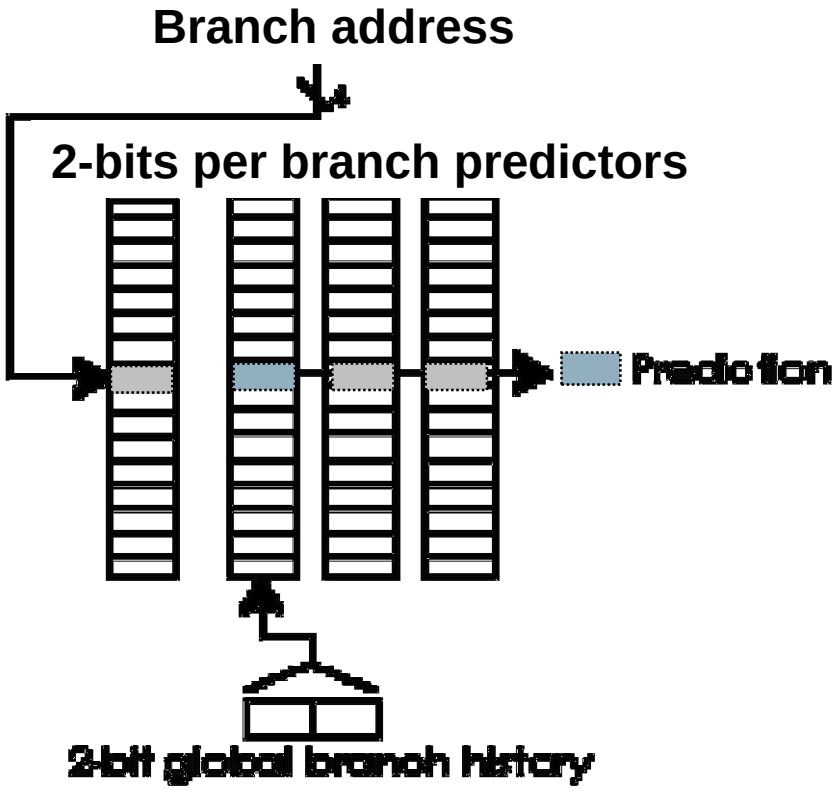
If the first two branches are not taken, the third one will be.

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

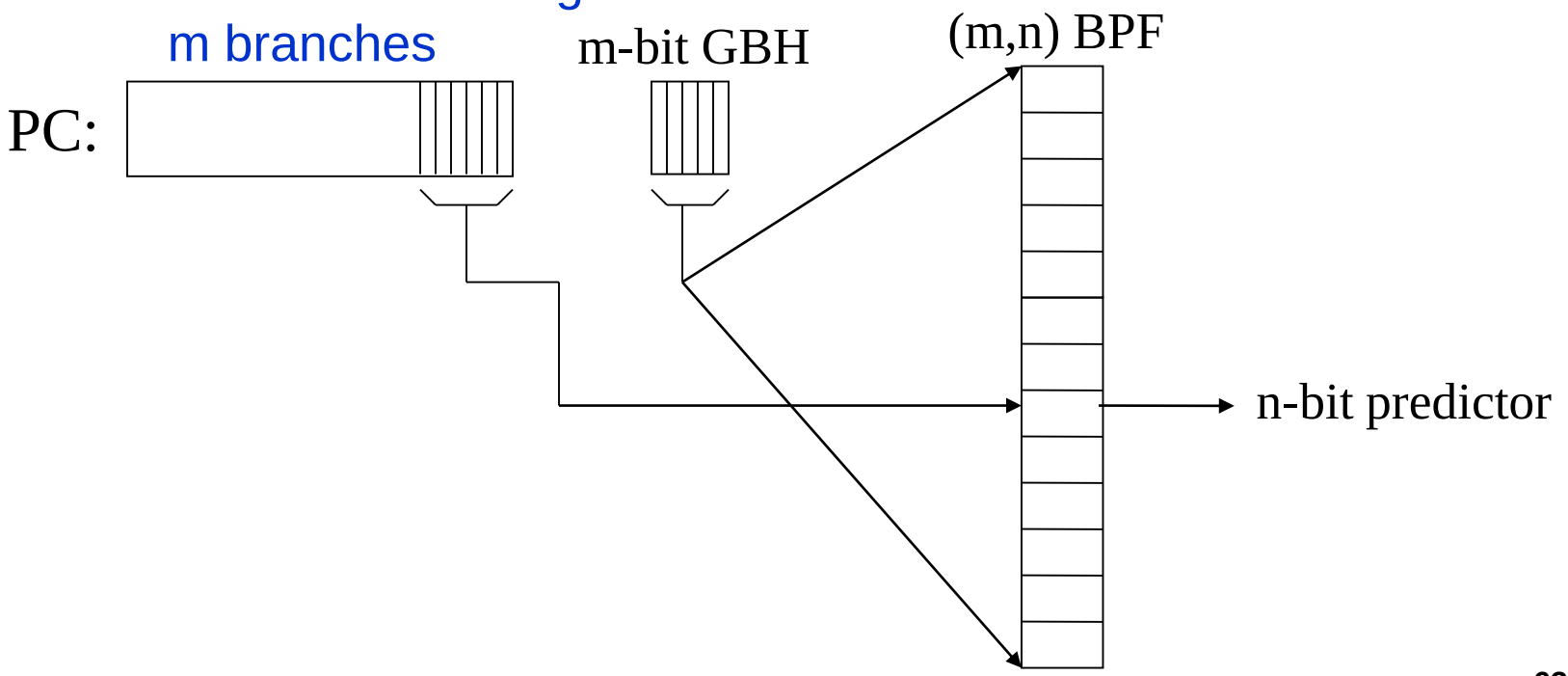
    $2^2 \times 2 \times 2^4 = 128$ bits



(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:

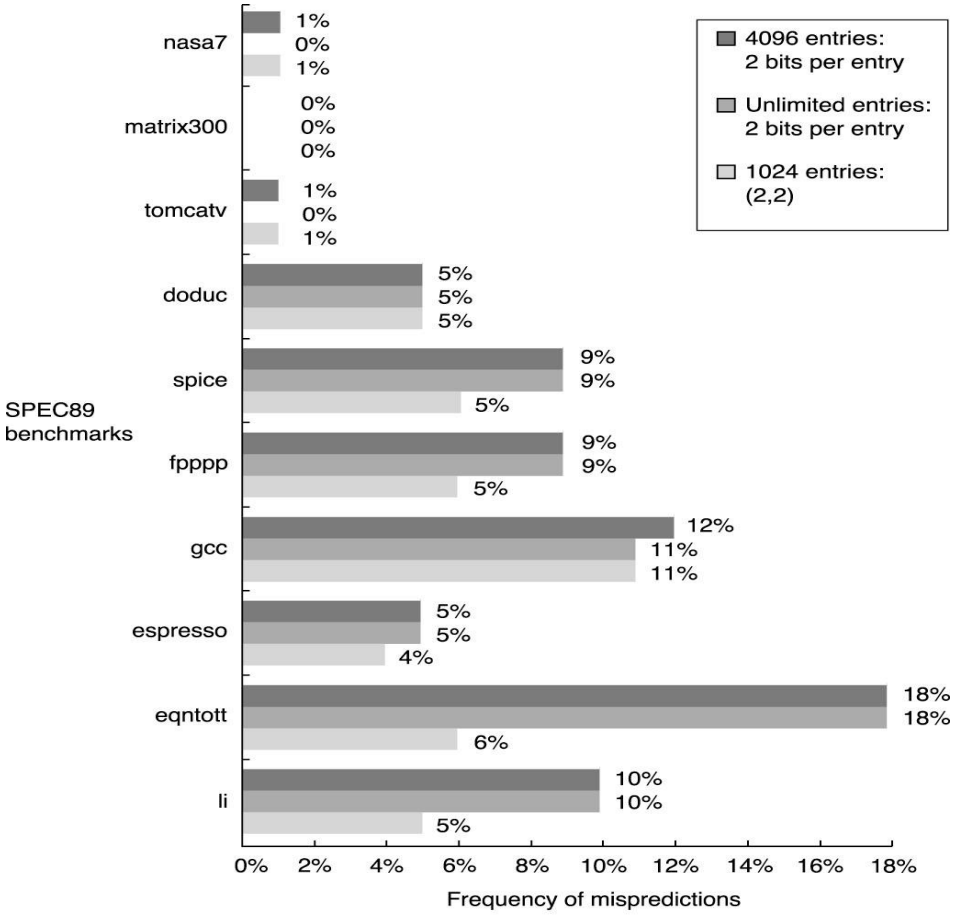
$$2^m \times n \times 2^p = \text{Total cache bits required}$$

  - 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



Tournament Predictors - Alpha *21264*

- 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
    - 在欲執行branch之前的其他branch指令之執行狀況，T or NT
  - 1 based on local information,
    - 欲執行之branch在前幾次的執行狀況
  - and combine with a selector
- Hopes to select right predictor for right branch

Tournament Predictors

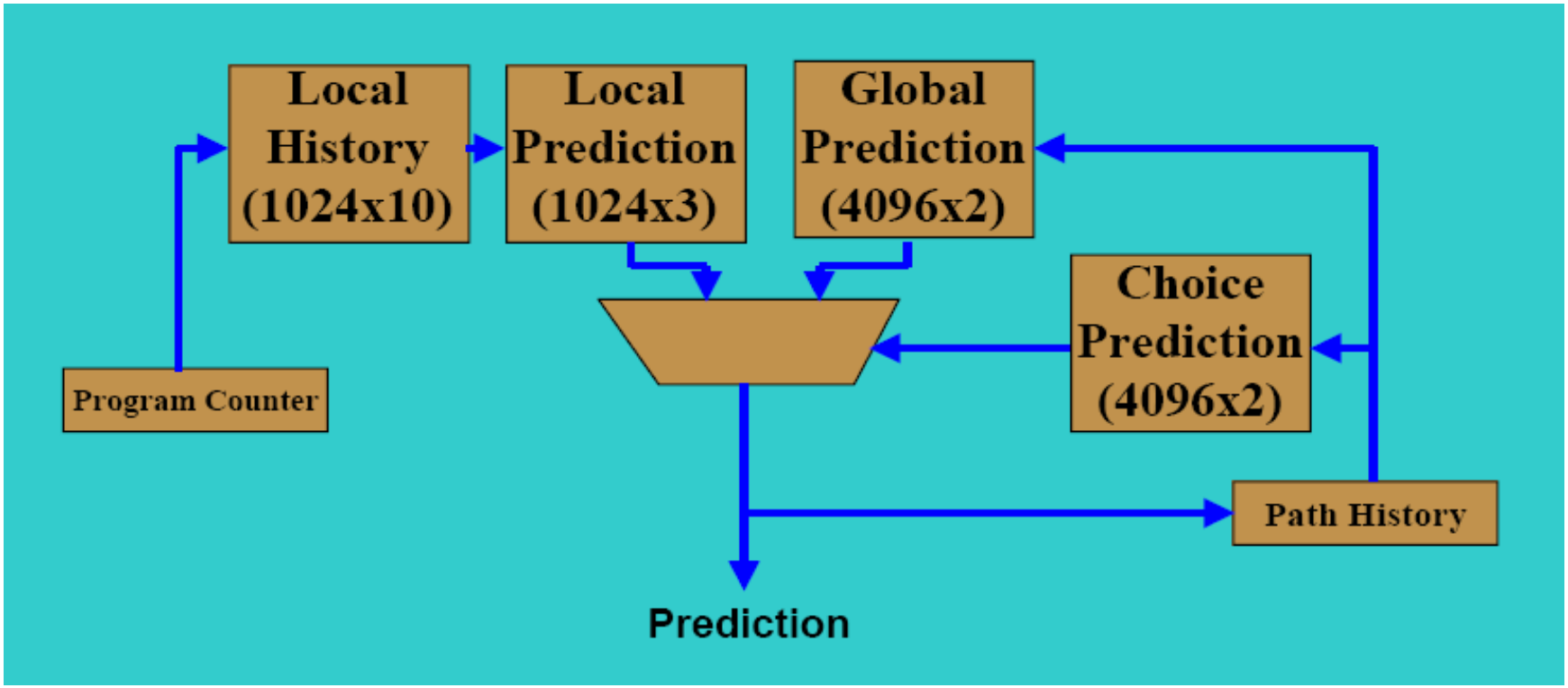
Tournament predictor using, say, 4K 2-bit counters indexed by local branch address. Chooses between:

- 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 the last 10 outcomes for the branch, 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 + 1K*10 + 1K*3 = 21K bits!

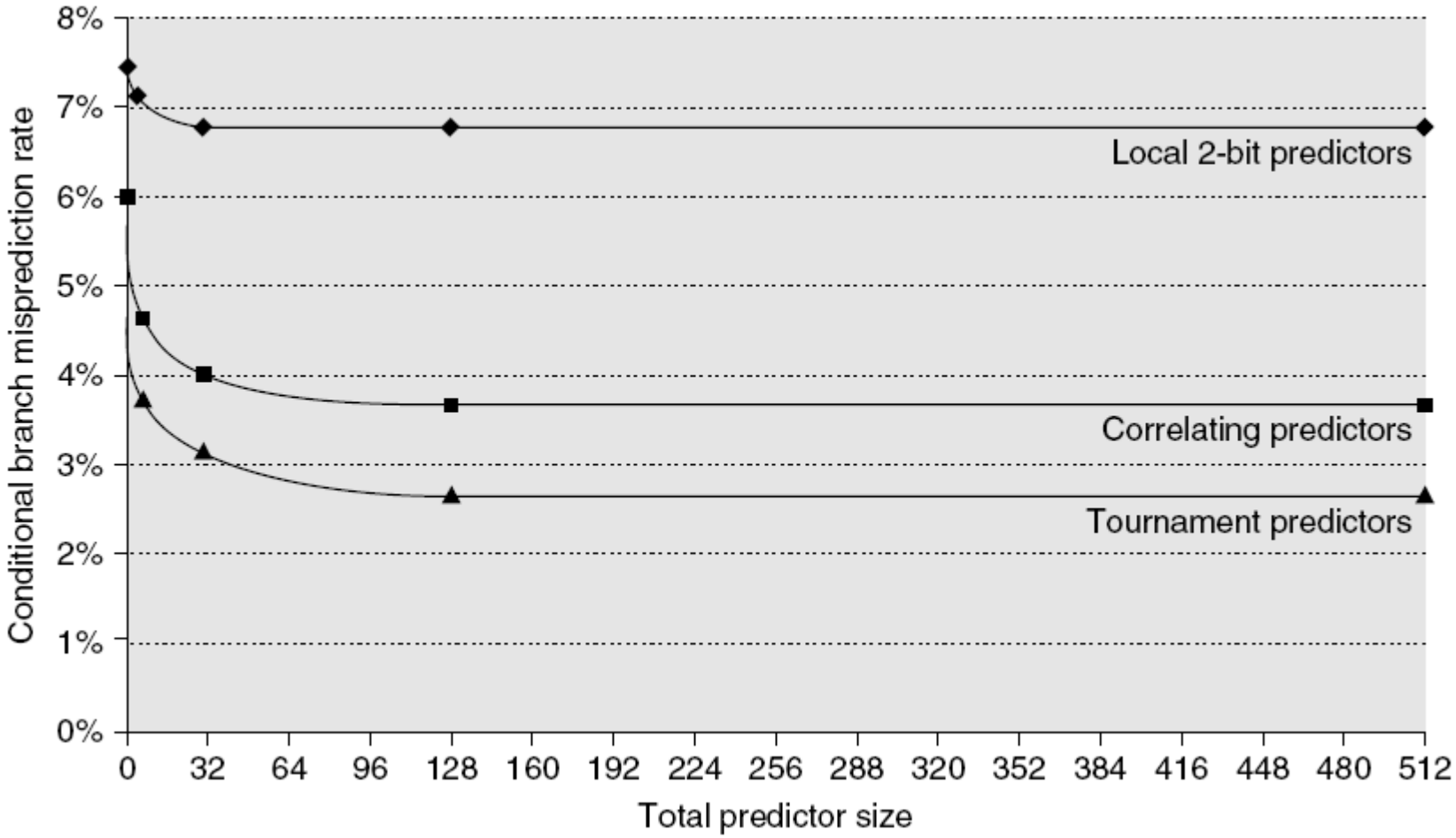
Tournament Predictor in Alpha 21264



Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Branch Prediction Performance



Branch predictor performance

Outline

- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation

Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow

- Advantages:
  - Compiler doesn't need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time

- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions

Dynamic Scheduling

- Dynamic scheduling implies:
  - Out-of-order execution
  - Out-of-order completion

- Creates the possibility for WAR and WAW hazards

由於原先後執行的指令可以提早執行，因此原不會產生的這兩種**hazard**將可能產生

- Tomasulo's Approach
  - Tracks when operands are available
  - Introduces register renaming in hardware
    - Minimizes WAW and WAR hazards

將以某些方法讓**Hazard**的發生機會降低

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed

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

因為**ADDD**在**DIVD**未完成之前不可執行，且**DIVD**將花較長的執行時間，因此允許之後的**SUBD**先執行

- 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

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

- 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 …..
ADDD F10, F0, F8	IF ID stall …..
SUBD F12, F8, F14	IF ID A1 A2 A3 A4 …

Key idea: Allow instructions behind stall to proceed.

Idea

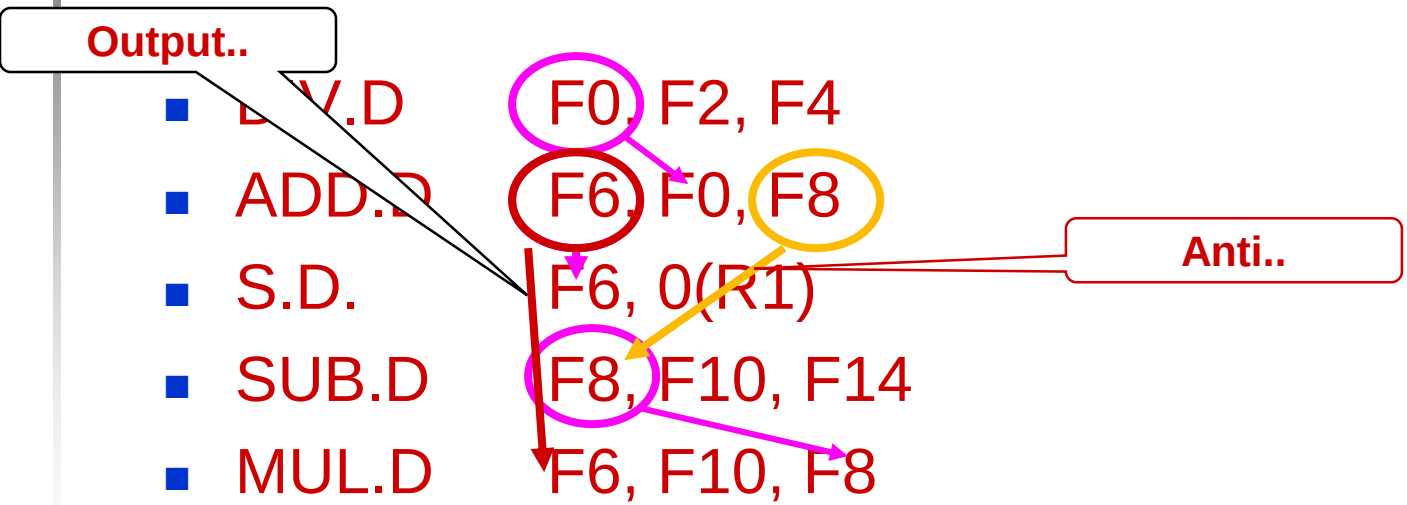
- The scheme of Tomasulo's approach
  - tracks when operands for instructions are available, to minimize RAW hazards, and 先讀取現有可用的**operands**，然後儲存，之後再利用
  - introduces register renaming, to minimize WAW and WAR hazards.

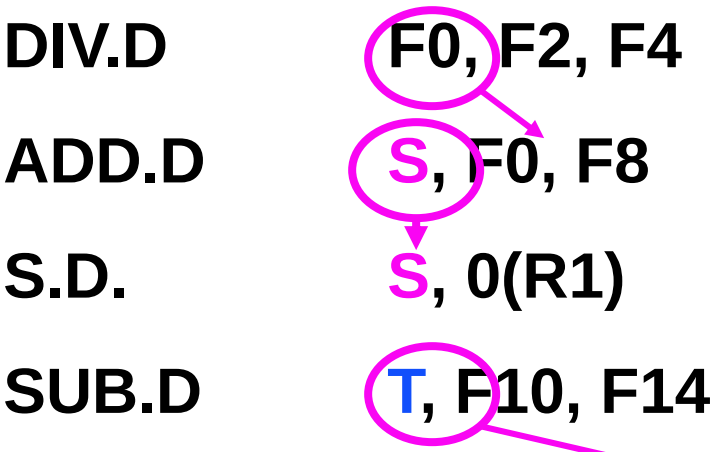
利用其它空間儲存執行完畢欲寫到暫存器的值，之前未讀取或是未寫入的指令就不會受影響

Register Renaming

- Register renaming eliminates these hazards by renaming *all destination registers*, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand.
- See the next slide

Hazard and Register Renaming





- Using **S** and **T**, the sequence can be without any dependences.
- Any subsequent uses of **F8** must be replaced by the register **T**.

看起來是使用不同暫存器儲存資料，而實際在實作上是用別的空間紀錄這些資料

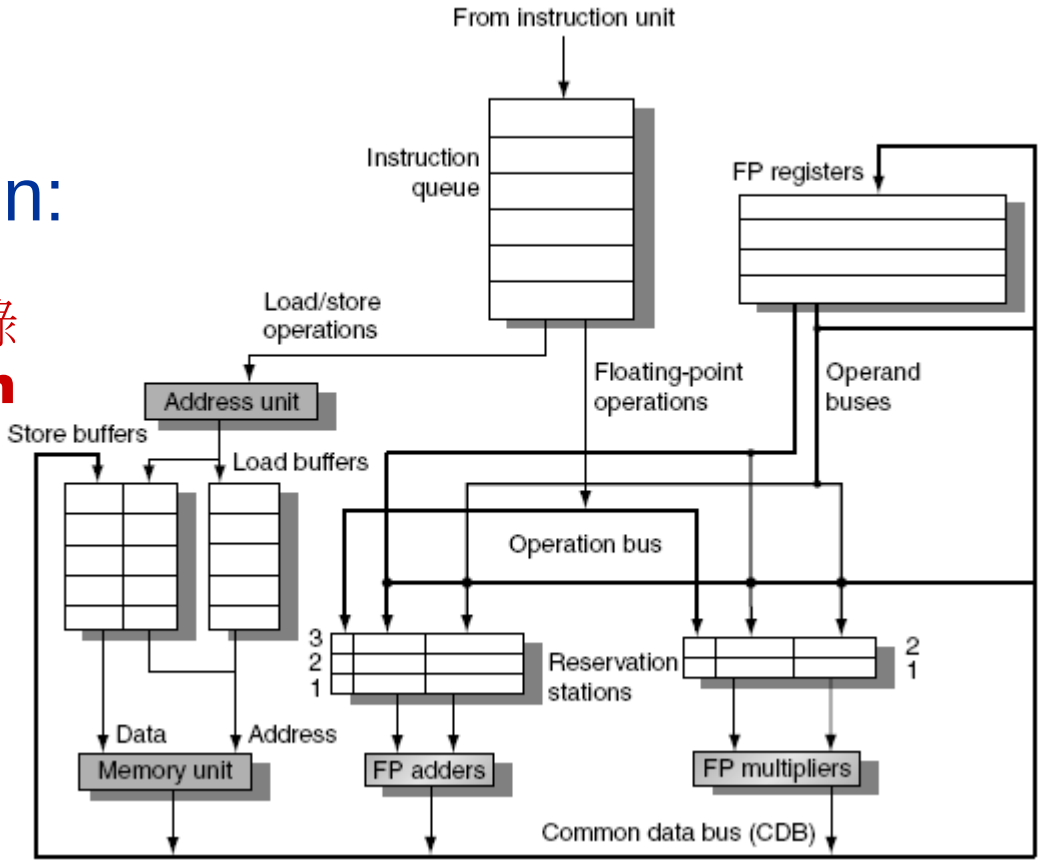
Register Renaming

- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number of instruction providing the operand values
  - RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
  - Pending instructions designate the RS to which they will send their output
    - Result values broadcast on a result bus, called the common data bus (CDB)
  - Only the last output updates the register file
  - As instructions are issued, the register specifiers are renamed with the reservation station
  - May be more reservation stations than registers

Tomasulo's Algorithm

- Load and store buffers
  - Contain data and addresses, act like reservation stations
- Top-level design:

Reservation Station是記錄相關參數的元件，而**Function Unit**是執行的元件



Seven Fields in Reservation Station

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

Vj, Vk: Values of Source operands

- Store buffers has V field, result to be stored

記錄已確定的暫存器資料內容

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

記錄未確定資料內容的暫存器尚在等待哪個**function unit**的產生資料

Busy: Indicates reservation station or FU is busy

A: memory address. Initially, the immediate value

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

Three Steps of Tomasulo Algorithm (1/2)

- Issue **(or** Dispatch**) – Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to *ensure the maintenance of correct data flow*.**
  - ***If there is no empty RS (structural hazard),* the instruction stalls until a station or buffer is freed.**
  - ***If there is a matching RS that is empty*,**
    - ***If the operands are currently in the registers*, issue the instruction to the station with the operand values,**
    - ***If the operands are not in the registers,* keep track of the functional units that will produce the operands. This step renames registers, eliminating WAR and WAW hazards.**

後面指令**write**暫存器不會有任何影響，
因為**operand**已經讀到**RS**中

Three Steps of Tomasulo Algorithm (2/2)

- Execute
  - ***If one or more of the operands is not yet available,*** **monitor the CDB while waiting for it to be computed. When an operand becomes available, it is placed into any RS awaiting it.**
  - ***When all the operands are available,*** **the operation can be executed at the corresponding functional unit.**
  - **By delaying instruction execution until the operands are available, *RAW hazards are avoided.***
  - **Loads and stores require *a two-step execution process***
    - **a) Compute the effective address when the base register is available**
    - **b) Effective address is then placed in the load or store buffer**
  - **To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede the instruction in program order have completed.**
- Write Result **– When the result is available, write it on the CDB and from there into the registers and into any RS waiting for this result.**

Example

		Instruction status		
Instruction		Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

	Register status								
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Outline

- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation

Tomasulo's Algorithm: Issue

Instruction state	Wait until	Action or bookkeeping

FP operation	Station r empty	`if (RegisterStat[rs].Qi≠0)`
		`  {RS[r].Qj←RegisterStat[rs].Qi}`
		`else {RS[r].Vj←Regs[rs]; RS[r].Qj←0};`
		`if (RegisterStat[rt].Qi≠0)`
		`  {RS[r].Qk←RegisterStat[rt].Qi}`
		`else {RS[r].Vk←Regs[rt]; RS[r].Qj←0};`
		`RS[r].Busy←yes; RegisterStat[rd].Qi←r;`

Load or store
ld rt, imm(rs)
sd rt, imm(rs)
	Buffer r empty	`if (RegisterStat[rs].Qi≠0)`
		`  {RS[r].Qj←RegisterStat[rs].Qi}`
		`else {RS[r].Vj←Regs[rs]; RS[r].Qj←0};`
		`RS[r].A←imm; RS[r].Busy←yes;`

Load only		`RegisterStat[rt].Qi←r;`

Store only		`if (RegisterStat[rt].Qi≠0)`
		`  {RS[r].Qk←RegisterStat[rt].Qi}`
		`else {RS[r].Vk←Regs[rt]; RS[r].Qk←0};`

說明

- Qi不為0時，代表暫存器等待function unit所算出的結果。若為0，代表沒有在等待。
- Qj和Qk不為0，代表等待function unit所算出的結果。若為0，代表Vj和Vk中已有資料。

Tomasulo's Algorithm: Execute

Instruction state	Wait until	Action or bookkeeping
FP operation	`(RS[r].Qj=0)` and `(RS[r].Qk=0)`	Compute result: operands are in `Vj` and `Vk`
Load-store step 1	`RS[r].Qj=0` & `r` is head of load-store queue	`RS[r].A←RS[r].Vj+RS[r].A;`
Load step 2	Load step 1 complete	Read from `Mem[RS[r].A]`

Tomasulo's Algorithm: Write Result

Instruction state	Wait until	Action or bookkeeping
FP operation or load	Execution complete at r & CDB available	∀x(if (RegisterStat[x].Qi=r) {Regs[x]←result; RegisterStat[x].Qi←0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj←result; RS[x].Qj←0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk←result; RS[x].Qk←0}); RS[r].Busy←no;
Store	Execution complete at r & RS[r].Qk=0	Mem[RS[r].A]←RS[r].Vk; RS[r].Busy←no;

Notice that all writes occur in Write Result, whether the destination is a register or memory.

Execution Cycle

- Dynamic DLX (much simpler)
  - 1 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

Qi記錄哪個Function Unit未來將更新暫存器的內容

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
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
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					

R()代表從暫存器中讀出的值

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	

F6

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					

M(?)代表從記憶體中**load**的值

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2			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				Add1	Mult2			

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

Add1與Mult1在此cycle獲得值，但是尚未開始執行execute的步驟

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			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			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			Add2		Mult2			

Add2在此cycle獲得值，但是尚未開始執行execute的步驟

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			Add2		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			Add2		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					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					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					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					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					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						Mult2		

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

Mult2在此**cycle**獲得值，但是尚未開始執行**execute**的步驟

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*						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*						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	No	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
57	FU									

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

Elimination of WAR Hazard

```
L.D     F6,32(R2)
L.D     F2,44(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F2,F6
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

1. If the **L.D** has completed:
   - Vk will store the result, allowing DIV.D to execute independent of the ADD.D.
2. If the **L.D** has not completed:
   - Qk would point to the Load1 reservation station, and the DIV.D instruction would be independent of the ADD.D.

Elimination of WAW Hazard

- By register status
- Ex:

```
L.D     F6, 32(R2)
.....
.....
ADD.D   F6, F8, F2
```

The corresponding register status for F6 will be updated when ADD.D instruction is issued.

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 ⇒high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - Multiple CDBs ⇒ 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
- 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

DADDIU R1, R1, #-8

BNEZ R1, Loop

Instruction		Instruction status: 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

- If we predict that branches are taken, using reservation stations will allow multiple executions of the loop to proceed at once.
  - 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
     DADDUI  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 DADDUI, 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	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

```
LD      F0   0     R1
MULTD   F4   F0    F2
SD      F4   0     R1
DADDU   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	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
DADDU  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
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
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	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

```
LD     F0  0   R1
MULTD  F4  F0  F2
SD     F4  0   R1
DADDU  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	0	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 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
DADDU	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*	*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
DADDU  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 DADDUI 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	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
DADDU  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	0	
Load3	No		
Store1	Yes	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
DADDU  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 R1+0

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]+0	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S1 Vk	S2 Qj	S2 Qk	RS
	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
DADDU  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]+0	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	0	Mult2
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	Yes	Multd		R(F2)	Load2	

Code:

```
LD     F0  0  R1
MULTD  F4  F0 F2
SD     F4  0  R1
DADDU  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]+0	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]+0	Mult2
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	Yes	Multd		R(F2)	Load2	

Code:

```
LD     F0   0    R1
MULTD  F4   F0   F2
SD     F4   0    R1
DADDUI 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 DADDUI

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]+0	
Load3	No		
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]+0	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S1 Vk	S2 Qj	S2 Qk	RS
	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
DADDU  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 BNE

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	0	
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]+0	Mult2
Store3	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[80]	R(F2)		
4	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

```
LD     F0   0    R1
MULTD  F4   F0   F2
SD     F4   0    R1
DADDU  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]+0	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS 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
DADDU  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]+0	
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]+0	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
DADDU  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]+0	
Store1	Yes	Regs[R1]+0	Mult1
Store2	Yes	Regs[R1]+0	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 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
DADDU  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]+0	
Store1	Yes	Regs[R1]+0	[80]*R2
Store2	Yes	Regs[R1]+0	Mult2
Store3	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	Multd	M[72]	R(F2)		

Code:

```
LD     F0   0    R1
MULTD  F4   F0   F2
SD     F4   0    R1
DADDU  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]+0	
Store1	Yes	Regs[R1]+0	[80]*R2
Store2	Yes	Regs[R1]+0	[72]*R2
Store3	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(F2)	Load3	
	Mult2	No					

Code:

```
LD     F0   0    R1
MULTD  F4   F0   F2
SD     F4   0    R1
DADDU  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	Instruction		j	k	Issue	Comp	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]+0	
Store1	Yes	Regs[R1]+0	[80]*R2
Store2	Yes	Regs[R1]+0	[72]*R2
Store3	Yes	Regs[R1]+0	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
DADDU   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]+0	
Store1	Yes	Regs[R1]+0	[80]*R2
Store2	Yes	Regs[R1]+0	[72]*R2
Store3	Yes	Regs[R1]+0	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
DADDU  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]+0	
Store1	No		
Store2	Yes	Regs[R1]+0	[72]*R2
Store3	Yes	Regs[R1]+0	Mult1

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS 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
DADDU  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]+0	
Store1	No		
Store2	No		
Store3	Yes	Regs[R1]+0	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
DADDU  R1   R1   #-8
BNEZ   R1   Loop
```

Register result status

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

Load and Store the same address

- If a load and a store access *the same memory address*
  - The load is before the store in program order and interchanging them results in a WAR hazard.
    - Ex:

      ld F0, 0(R1)

      sd F2, 0(R1)
  - The store is before in program order and interchanging them results in a RAW hazard.
    - Ex:

      sd F4, 8(R2)

      ld F6, 8(R2)
  - Interchanging two stores to the same address results in a WAW hazard.

Load and Store the same address

- To determine if a load can be executed at a given time, the processor can check whether *any uncompleted store that precedes the load* in program order shares the same data memory address
- A store must wait until *there are no unexecuted loads or stores that are earlier* in program order and share the same data memory address.

Outline

- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation

Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an execution

Reorder Buffer

- Reorder buffer – holds the result of instruction between completion and commit
- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?
- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit

Reorder Buffer

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in ROB are cleared
- Exceptions:
  - Not recognized until it is ready to commit

Speculation to greater ILP

- 3 components of HW-based speculation:

1. *Dynamic branch prediction* to choose which instructions to execute
2. *Speculation* to allow execution of instructions before control dependences are resolved

   + ability to *undo effects* of incorrectly speculated sequence
3. *Dynamic scheduling* to deal with scheduling of different combinations of basic blocks

Adding Speculation to Tomasulo

- Must separate execution from allowing instruction to finish or "commit"
- This additional step called *instruction commit*
- When an instruction is *no longer speculative*, allow it to update the register file or memory
- Requires additional set of buffers to hold results of instructions that have *finished execution but have not committed*
- This *reorder buffer (ROB)* is also used to pass results among instructions that may be speculated

Reorder Buffer (ROB)

- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the *instruction commits* (the instruction is no longer speculative)
  - (we know definitively that the instruction should execute)
- Thus, the ROB supplies operands in interval between *completion of instruction execution* and *instruction commit*
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo's algorithm
  - ROB extends architectured registers like RS

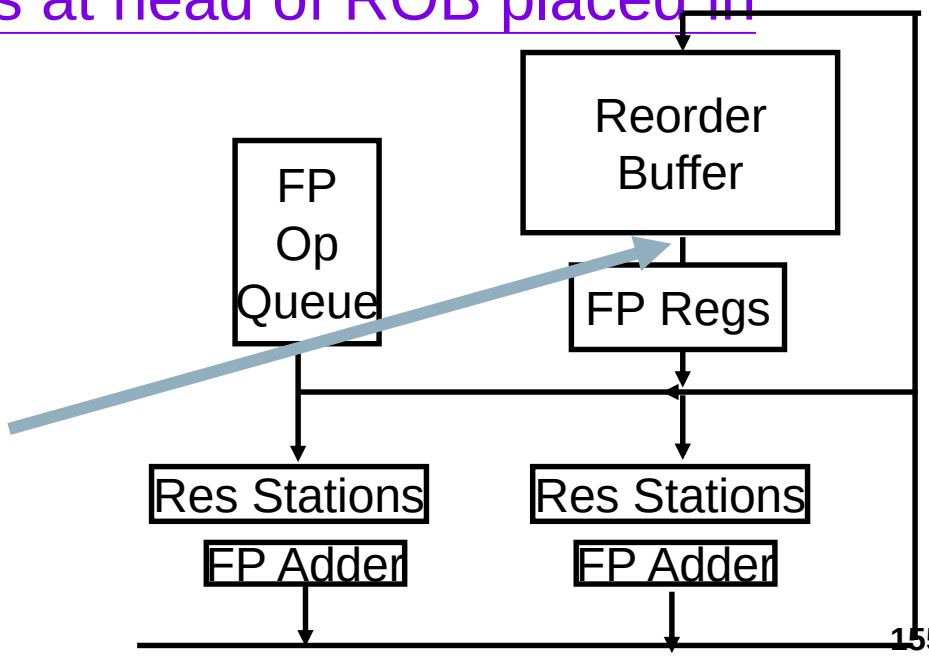
Reorder Buffer Entry

- Each entry in the ROB contains four fields:

1. Instruction type
   - a branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations)
2. Destination
   - Register number (for loads and ALU operations) or memory address (for stores) where the instruction result should be written
3. Value
   - Value of instruction result until the instruction commits
4. Ready
   - Indicates that instruction has completed execution, and the value is ready

Reorder Buffer operation

- Holds instructions in FIFO order, exactly as issued
  - *Commit in order*
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions *commit* ⇒values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on *mispredicted branches or on exceptions*



Recall: 4 Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)

2. Execution—operate on operands (EX)

   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)

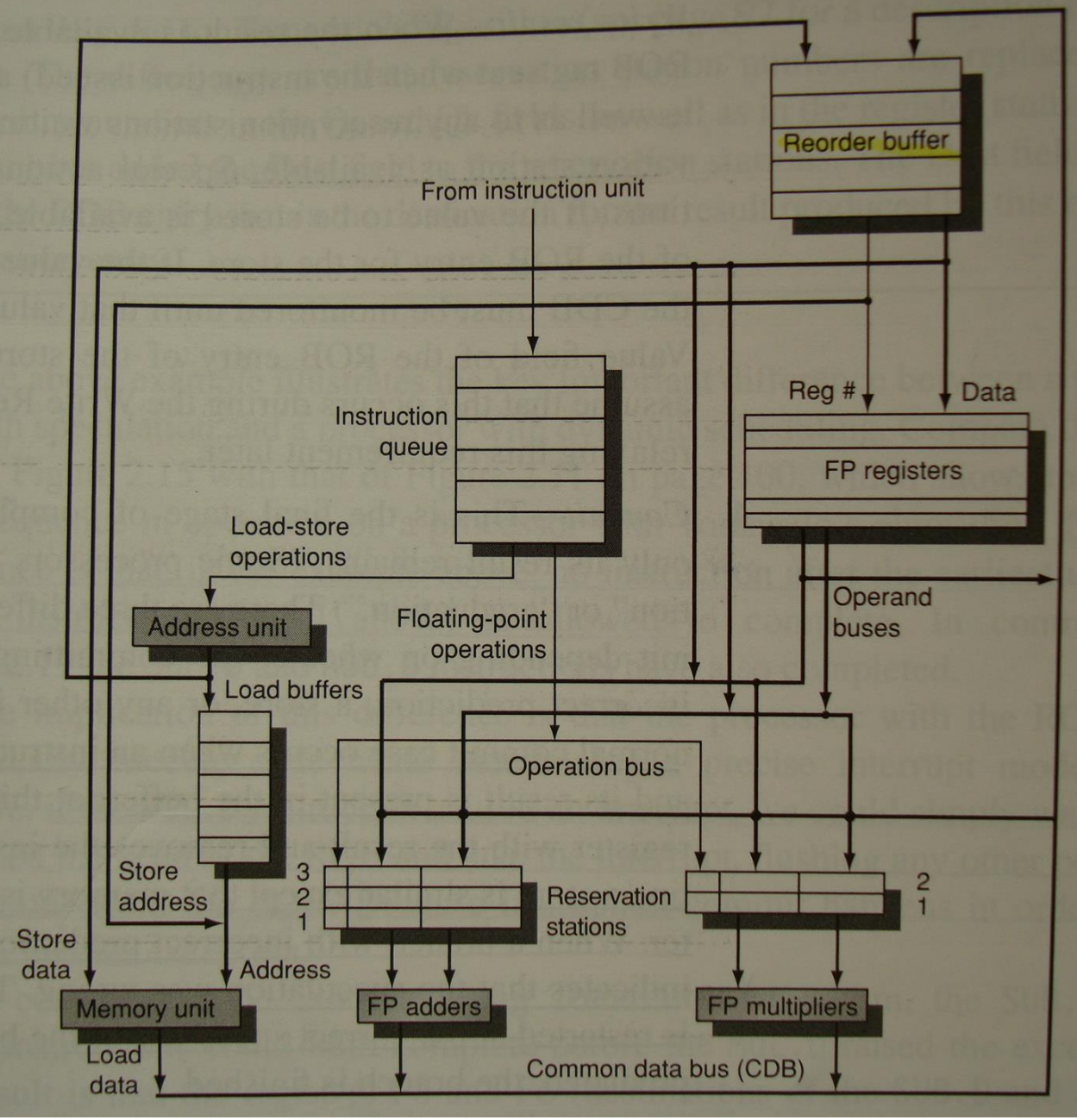
3. Write result—finish execution (WB)

   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

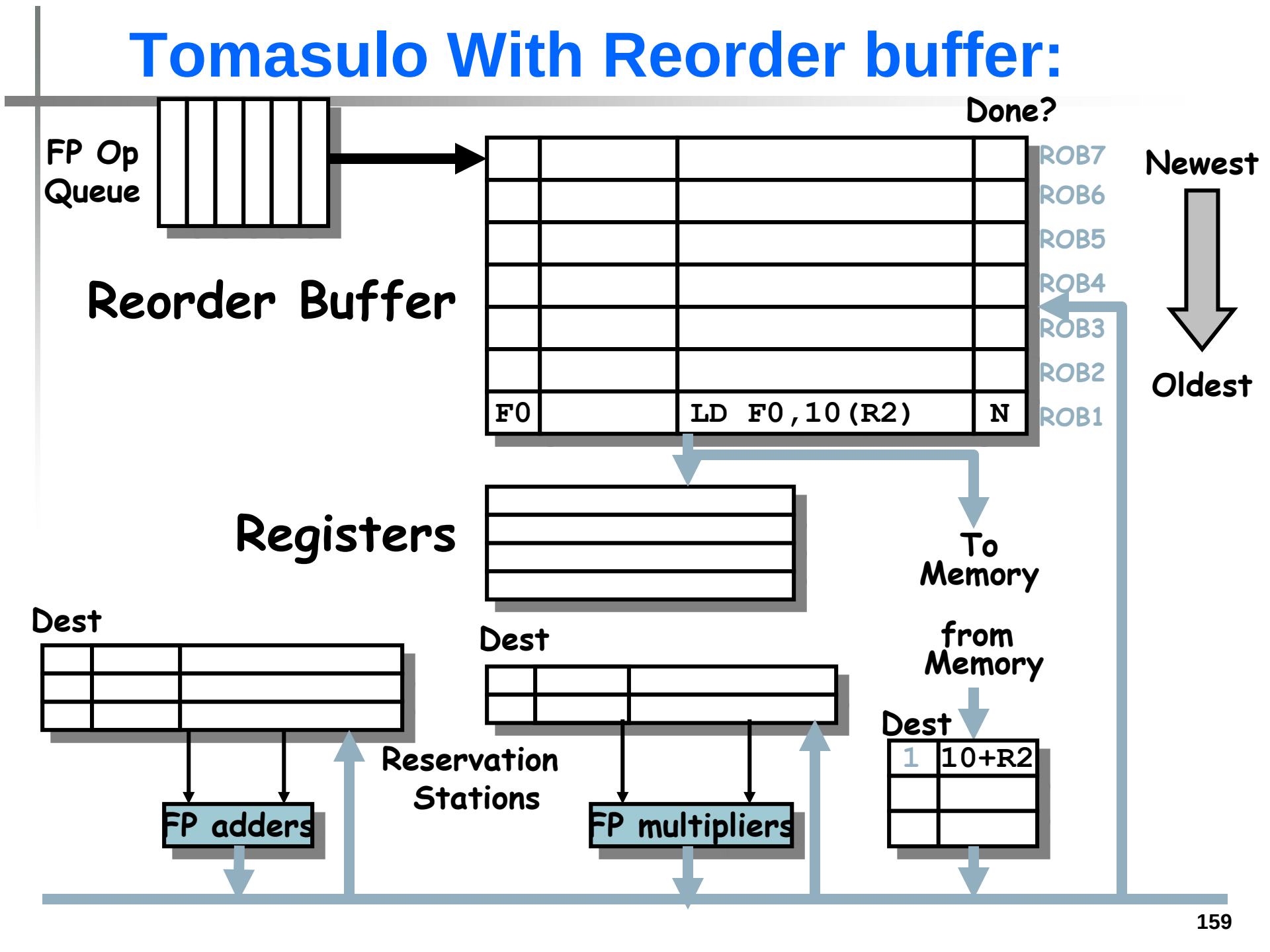
4. Commit—update register with reorder result

   When *instr. at head of reorder buffer & result present*, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called “graduation”)

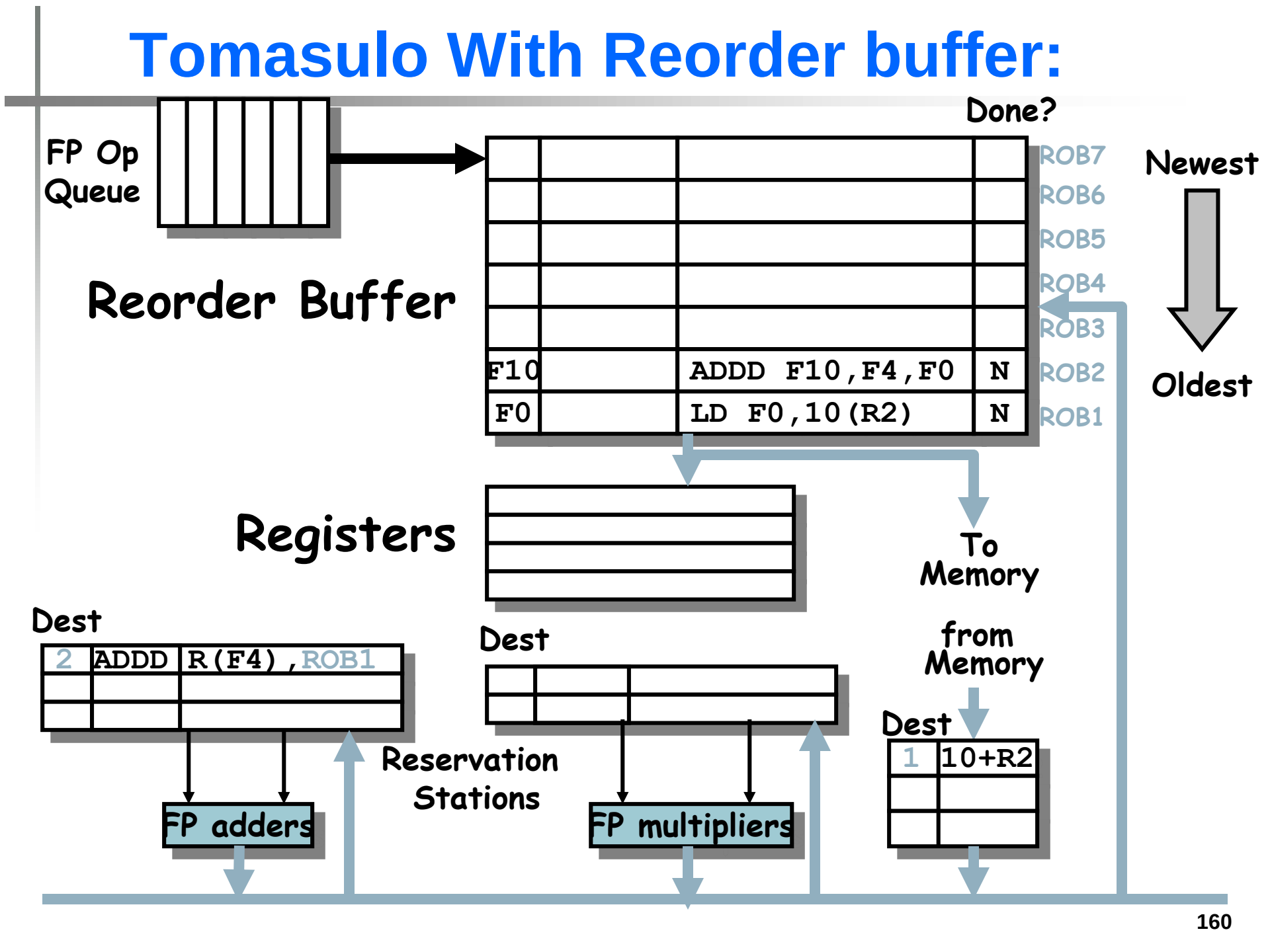
Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;`
FP operations and stores		`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};`
FP operations		`RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;`
Loads		`RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;`
Stores		`RS[r].A ← imm;`
Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj == 0) and store at queue head	`ROB[h].Address ← RS[r].Vj + RS[r].A;`
Write result all but store	Execution done at r and CDB available	`b ← RS[r].Dest; RS[r].Busy ← no; ∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes;`
Store	Execution done at r and (RS[r].Qk == 0)	`ROB[h].Value ← RS[r].Vk;`
Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	`d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction==Branch) {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;};} else if (ROB[h].Instruction==Store) {Mem[ROB[h].Destination] ← ROB[h].Value;} else /* put the result in the register destination */ {Regs[d] ← ROB[h].Value;}; ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};`


Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load-store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)

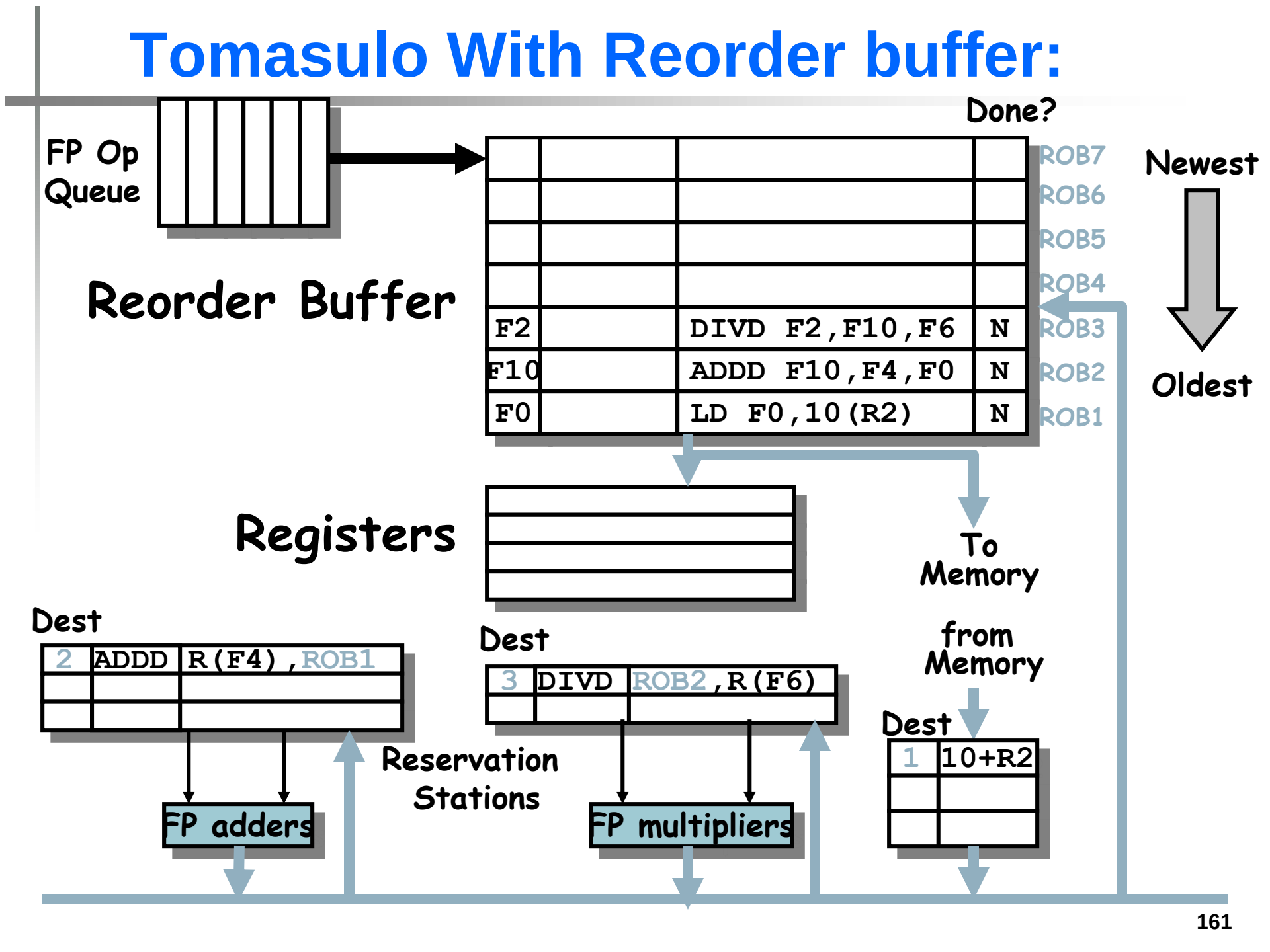
Tomasulo With Reorder buffer:



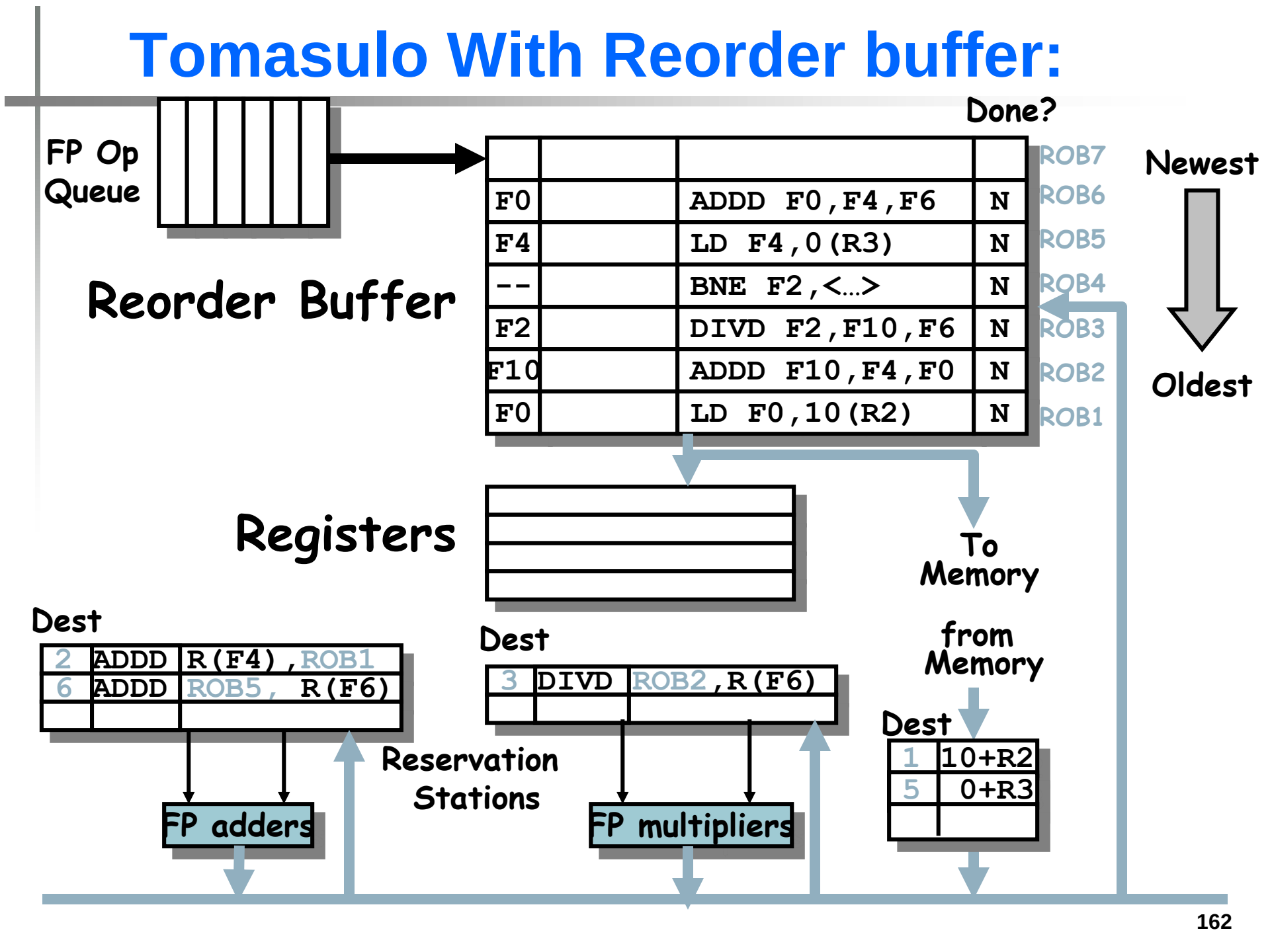
Tomasulo With Reorder buffer:



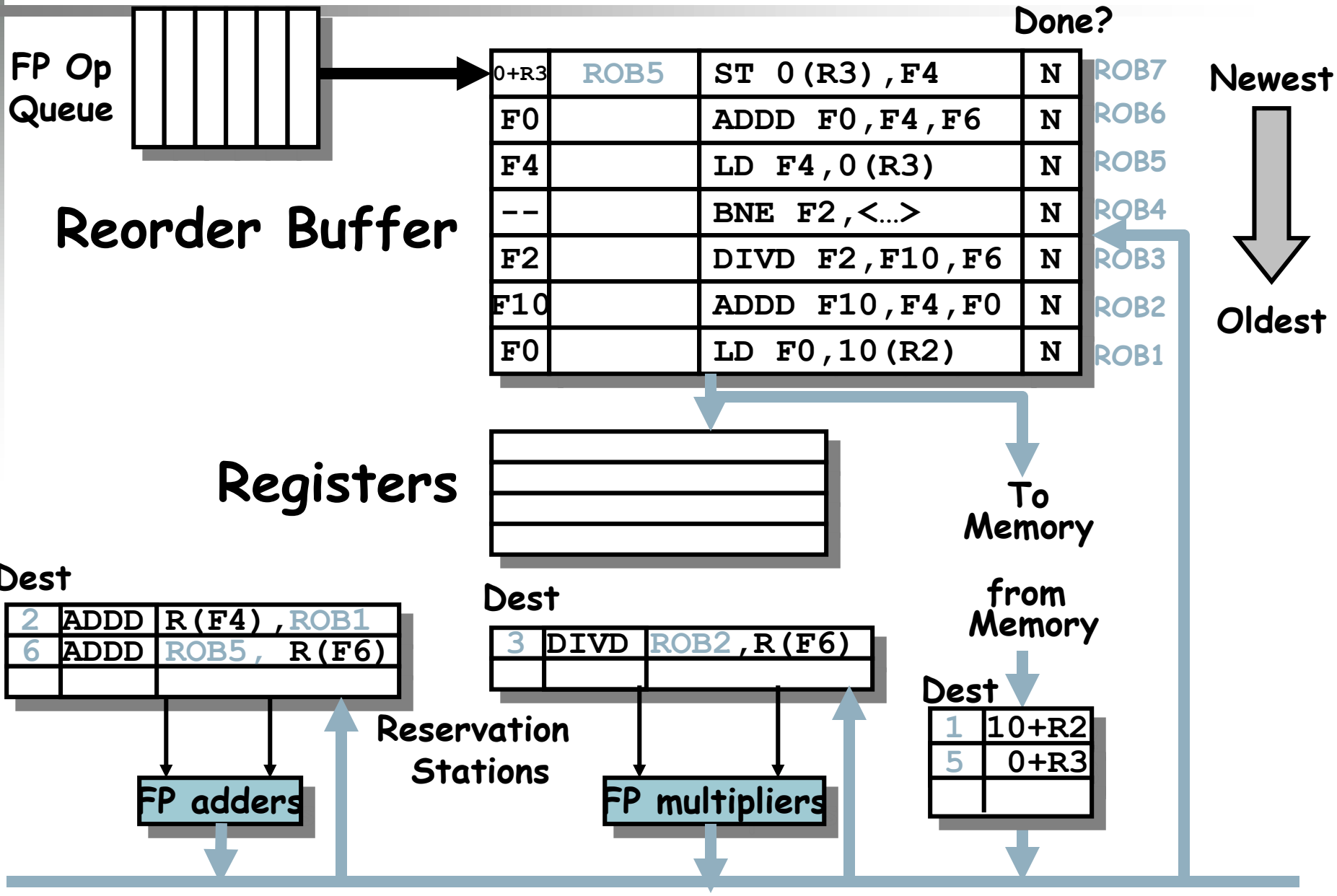
Tomasulo With Reorder buffer:



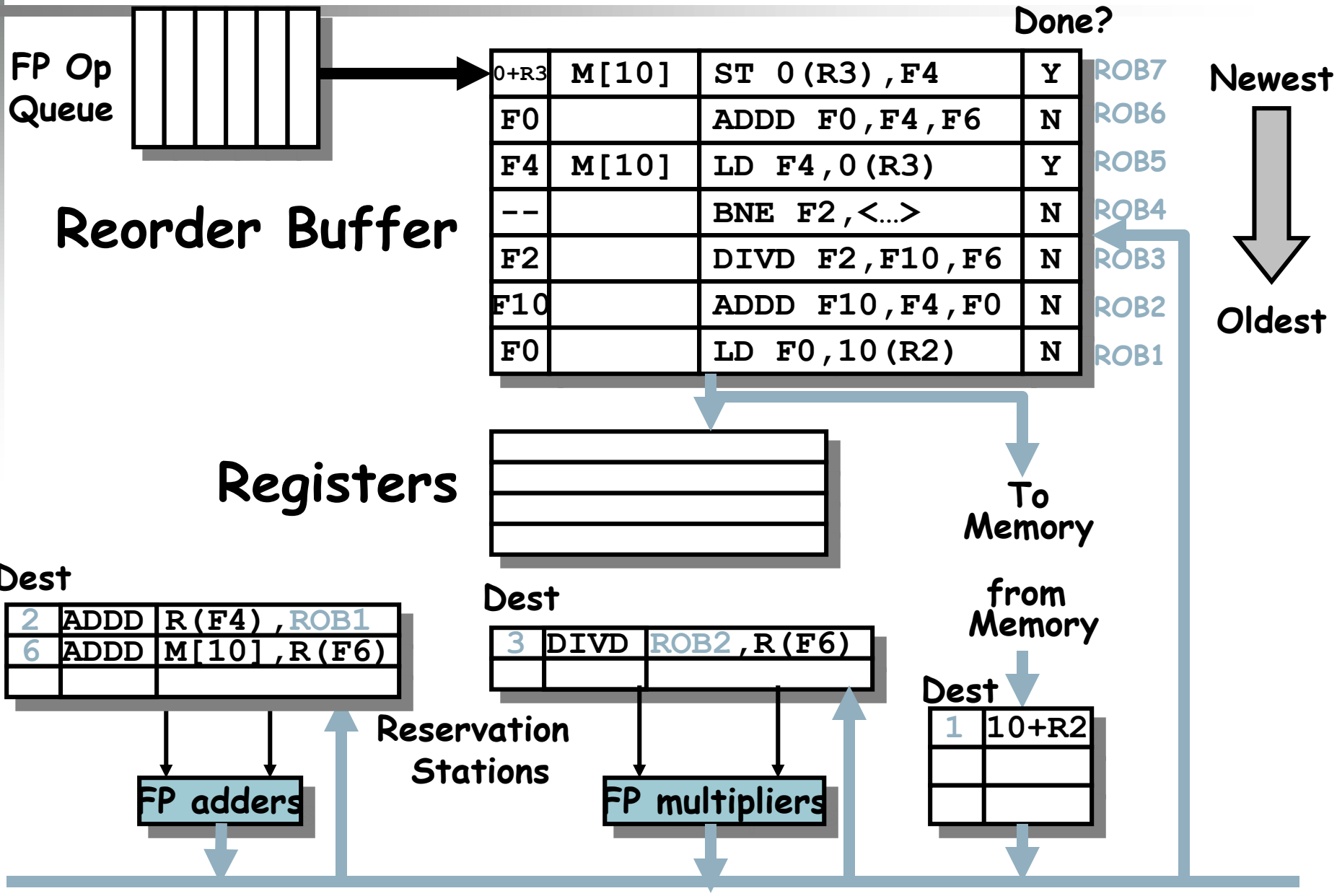
Tomasulo With Reorder buffer:



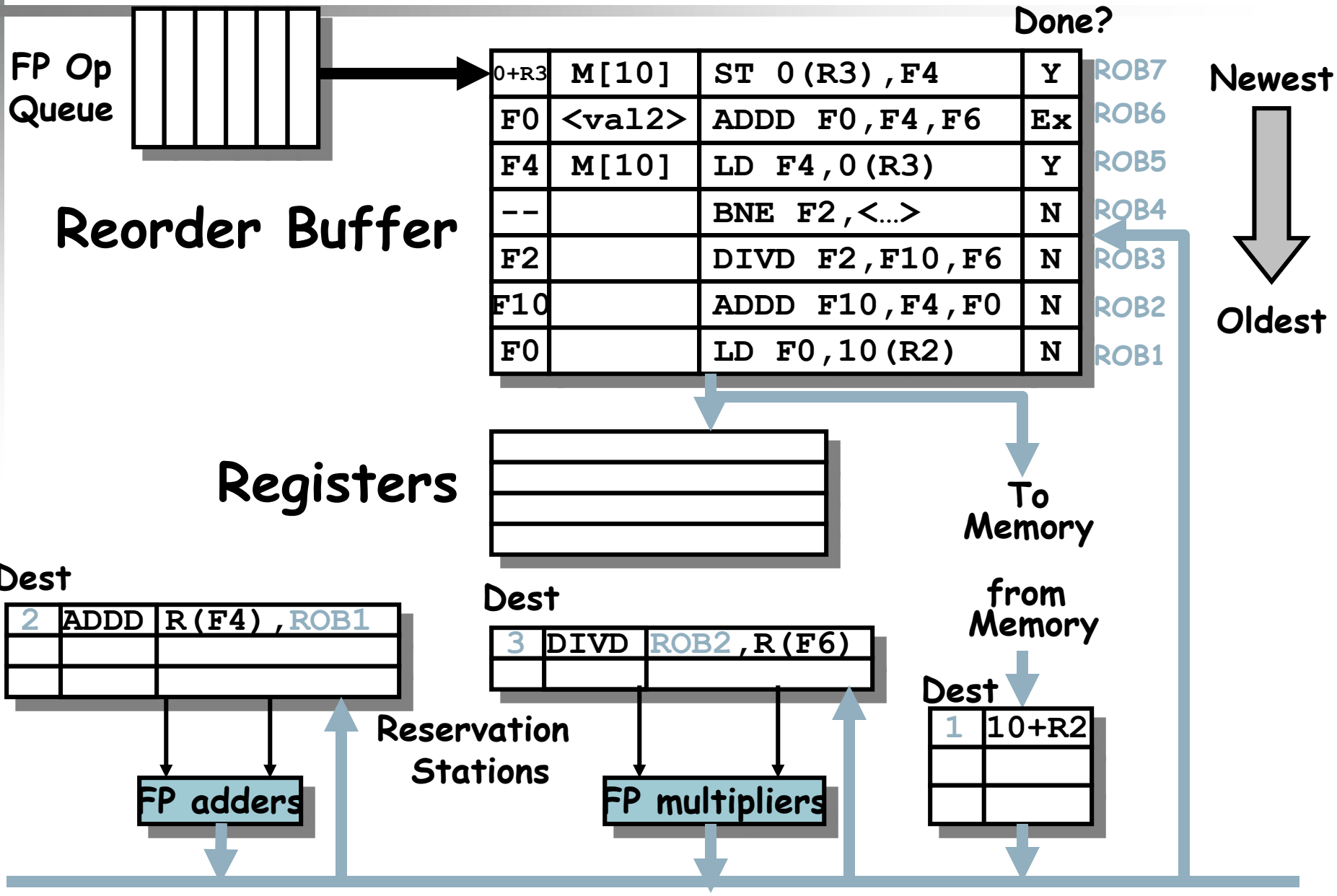
Tomasulo With Reorder buffer:



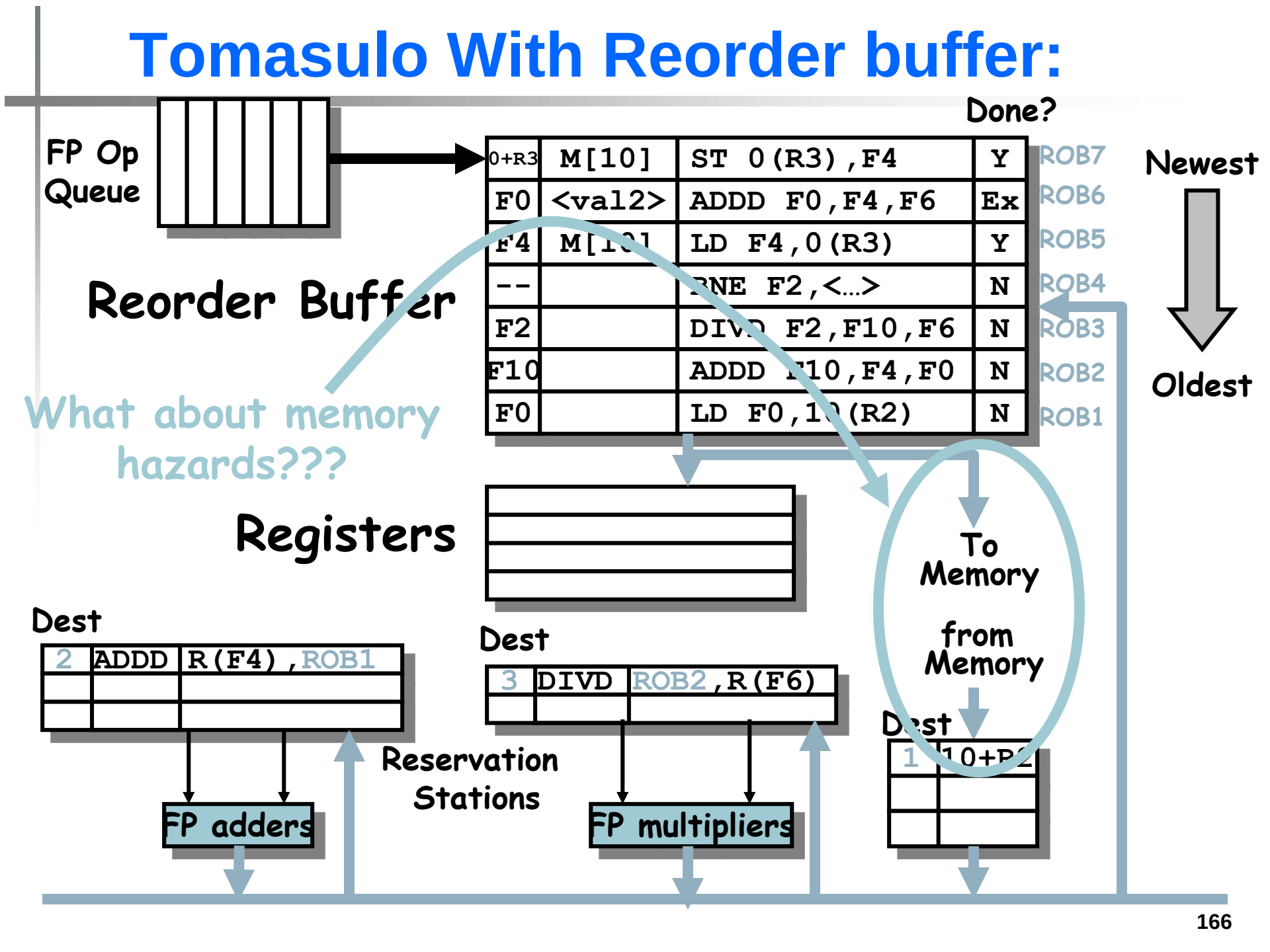
Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:



Implication (Example in Text Book)

- **The processor with the ROB can dynamically execute code while maintaining a precise interrupt model.**
  - **For example, if the MUL.D instruction caused an interrupt, we could simply wait until it reached the head of the ROB and take the interrupt, flushing any other pending instructions from the ROB. Because instruction commit happens in order, this yields a precise exception.**
  - **In the example using Tomasulo's algorithm, the SUB.D and ADD.D instructions could both complete before the MUL.D raised the exception.**

L.D	F6, 32(R2)
L.D	F2, 44(R3)
MUL.D	F0, F2, F4
SUB.D	F8, F6, F2
DIV.D	F10, F0, F6
ADD.D	F6, F8, F2

Avoiding Memory Hazards

- WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending
- RAW hazards through memory are maintained by two restrictions:
  1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and
  2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.
- these restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data

Outline

- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using dynamic scheduling, multiple issue, and speculation

Multiple Issue and Static Scheduling

- CPI ≥ 1 if issue only 1 instruction every clock cycle
- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors

Multiple Issue

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

VLIW Processors

- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Must be enough parallelism in code to fill the available slots

VLIW Processors

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility

VLIW

- Each “instruction” has explicit coding for multiple operations
  - In IA-64, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Recall: Unrolled Loop that Minimizes Stalls for Scalar

```
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    0(R1),F4
10      S.D    -8(R1),F8
11      S.D    -16(R1),F12
12      DSUBUI R1,R1,#32
13      BNEZ   R1,LOOP
14      S.D    8(R1),F16     ; 8-32 = -24
```

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

14 clock cycles, or 3.5 per iteration
(9 registers: F0 –F16)

Loop Unrolling in VLIW

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

- Unrolled 7 times to avoid delays
- 7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
- Average: 2.5 ops per clock, 50% efficiency
- Note: Need more registers in VLIW (15 registers: F0-F28)

Problems with 1st Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding

To combat this code size increase, clever encodings are sometimes used.

Another technique is to compress the instructions in main memory and expand them when they are read into the cache or are decoded.

Problems with 1st Generation VLIW

- Operated in lock-step; no hazard detection HW
  - A stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might predict function units, but caches hard to predict
- Binary code compatibility
  - Pure VLIW → different numbers of functional units and unit latencies require different versions of the code

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- IA-64: instruction set architecture
- 128 64-bit integer registers + 128 82-bit floating point registers
- Hardware checks dependencies
  (interlocks => binary compatibility over time)
- Extension for more aggressive software speculation
- Preserving binary compatibility
- Predicated execution (select 1 out of 64 1-bit flags)
  => 40% fewer mispredictions?
- Itanium™ was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process
- Itanium 2™ is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3

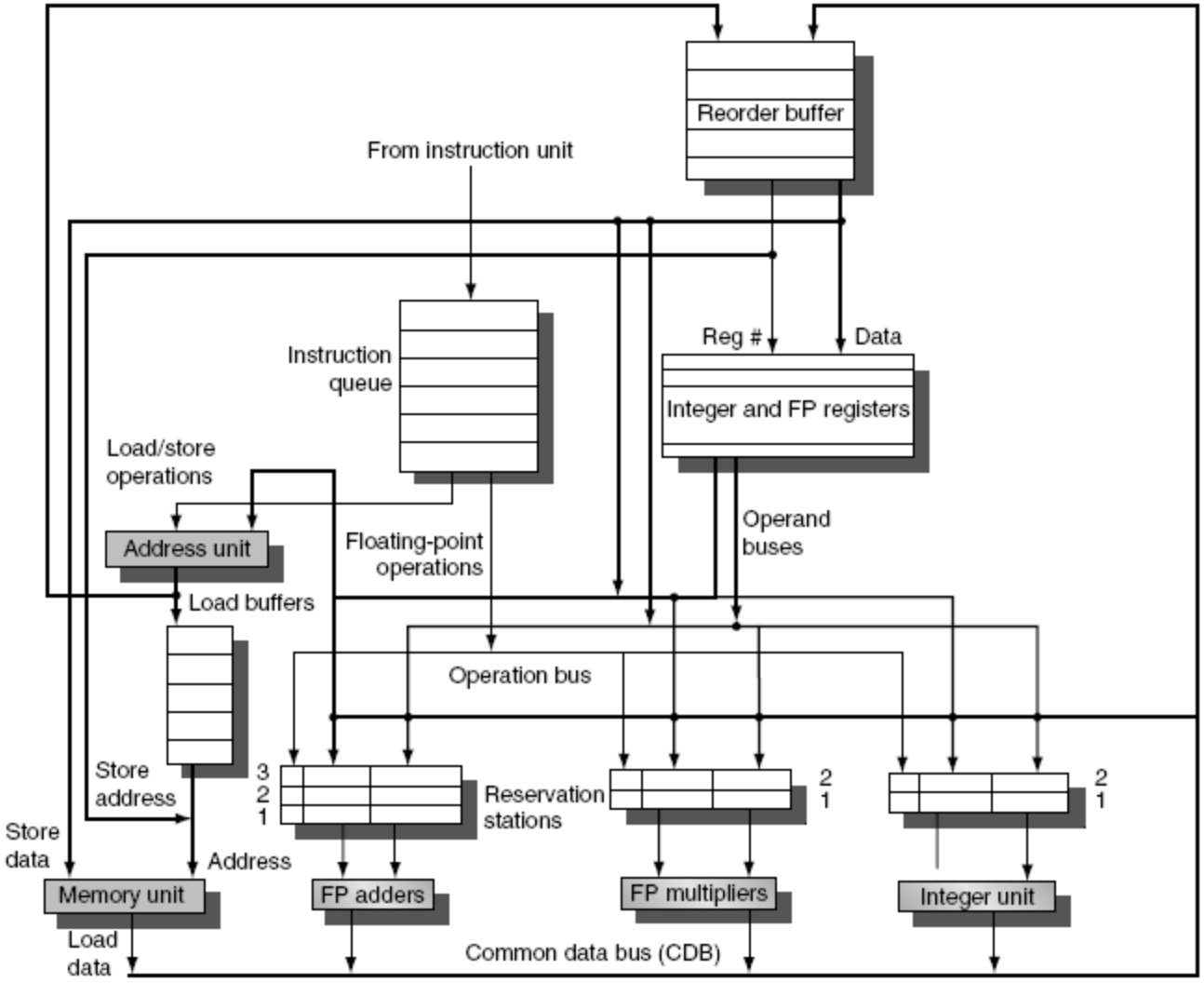
Outline

- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using dynamic scheduling, multiple issue, and speculation

Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
  - Hybrid approaches
- Issue logic can become bottleneck

Overview of Design



Multiple Issue

- Limit the number of instructions of a given class that can be issued in a “bundle”
  - I.e. on FP, one integer, one load, one store
- Examine all the dependencies among the instructions in the bundle
- If dependencies exist in bundle, encode them in reservation stations
- Also need multiple completion/commit

Example

```
Loop:  LD R2,0(R1)        ;R2=array element
       DADDIU R2,R2,#1    ;increment R2
       SD R2,0(R1)        ;store result
       DADDIU R1,R1,#8    ;increment pointer
       BNE R2,R3,LOOP     ;branch if not last element
```

假設**branch**指令需要**4**個**cycle**

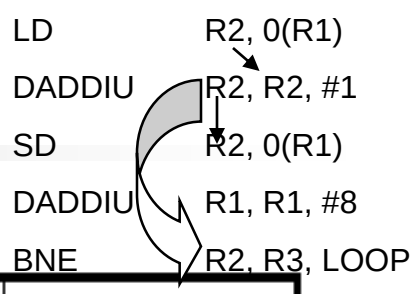
Example (No Speculation)

Iteration number	Instructions		Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD	R2,0(R1)	1	2	3	4	First issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	SD	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	Execute directly
1	BNE	R2,R3,LOOP	3	7			Wait for DADDIU
2	LD	R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU	R2,R2,#1	4	11		12	Wait for LW
2	SD	R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU	R1,R1,#8	5	8		9	Wait for BNE
2	BNE	R2,R3,LOOP	6	13			Wait for DADDIU
3	LD	R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU	R2,R2,#1	7	17		18	Wait for LW
3	SD	R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU	R1,R1,#8	8	14		15	Wait for BNE
3	BNE	R2,R3,LOOP	9	19			Wait for DADDIU

Example

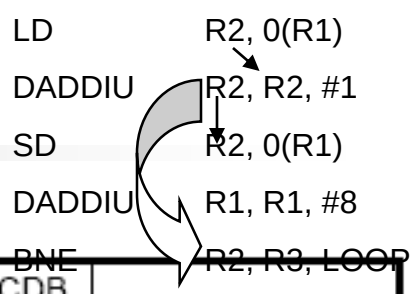
Iteration number	Instructions		Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	8	Commit in order
1	BNE	R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	SD	R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU	R1,R1,#8	5	6		7	11	Commit in order
2	BNE	R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	SD	R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU	R1,R1,#8	8	9		10	14	Executes earlier
3	BNE	R2,R3,LOOP	9	13			14	Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (1)



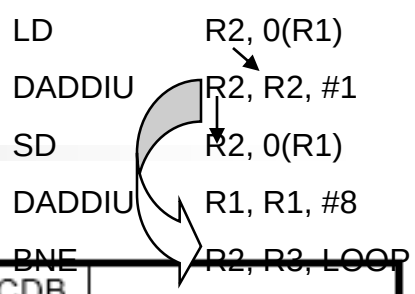
Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1				First issue
1	DADDIU R2,R2,#1	1				
1	SD R2,0(R1)					
1	DADDIU R1,R1,#8					
1	BNE R2,R3,LOOP					
2	LD R2,0(R1)					
2	DADDIU R2,R2,#1					
2	SD R2,0(R1)					
2	DADDIU R1,R1,#8					
2	BNE R2,R3,LOOP					
3	LD R2,0(R1)					
3	DADDIU R2,R2,#1					
3	SD R2,0(R1)					
3	DADDIU R1,R1,#8					
3	BNE R2,R3,LOOP					

Two-Issue Dynamically Scheduled Processor without Speculation (2)



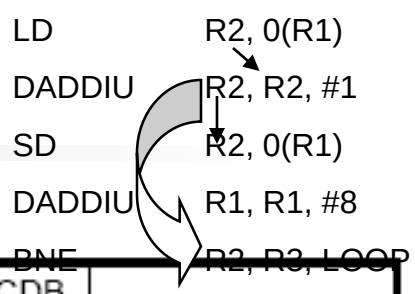
Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2			First issue
1	DADDIU R2,R2,#1	1				Wait for LD
1	SD R2,0(R1)	2				
1	DADDIU R1,R1,#8	2				
1	BNE R2,R3,LOOP					
2	LD R2,0(R1)					
2	DADDIU R2,R2,#1					
2	SD R2,0(R1)					
2	DADDIU R1,R1,#8					
2	BNE R2,R3,LOOP					
3	LD R2,0(R1)					
3	DADDIU R2,R2,#1					
3	SD R2,0(R1)					
3	DADDIU R1,R1,#8					
3	BNE R2,R3,LOOP					

Two-Issue Dynamically Scheduled Processor without Speculation (3)



Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3		First issue
1	DADDIU R2,R2,#1	1				Wait for LD
1	SD R2,0(R1)	2	3			
1	DADDIU R1,R1,#8	2	3			Execute directly
1	BNE R2,R3,LOOP	3				
2	LD R2,0(R1)					
2	DADDIU R2,R2,#1					
2	SD R2,0(R1)					
2	DADDIU R1,R1,#8					
2	BNE R2,R3,LOOP					
3	LD R2,0(R1)					
3	DADDIU R2,R2,#1					
3	SD R2,0(R1)					
3	DADDIU R1,R1,#8					
3	BNE R2,R3,LOOP					

Two-Issue Dynamically Scheduled Processor without Speculation (4)



Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1				Wait for LD
1	SD R2,0(R1)	2	3			Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3				Wait for DADDIU
2	LD R2,0(R1)	4				
2	DADDIU R2,R2,#1	4				
2	SD R2,0(R1)					
2	DADDIU R1,R1,#8					
2	BNE R2,R3,LOOP					
3	LD R2,0(R1)					
3	DADDIU R2,R2,#1					
3	SD R2,0(R1)					
3	DADDIU R1,R1,#8					
3	BNE R2,R3,LOOP					

Two-Issue Dynamically Scheduled Processor without Speculation (5)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5			Wait for LD
1	SD R2,0(R1)	2	3			Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3				Wait for DADDIU
2	LD R2,0(R1)	4				Wait for BNE
2	DADDIU R2,R2,#1	4				Wait for LD
2	SD R2,0(R1)	5				
2	DADDIU R1,R1,#8	5				
2	BNE R2,R3,LOOP					
3	LD R2,0(R1)					
3	DADDIU R2,R2,#1					
3	SD R2,0(R1)					
3	DADDIU R1,R1,#8					
3	BNE R2,R3,LOOP					

Two-Issue Dynamically Scheduled Processor without Speculation (6)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3			Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3				Wait for DADDIU
2	LD R2,0(R1)	4				Wait for BNE
2	DADDIU R2,R2,#1	4				Wait for LD
2	SD R2,0(R1)	5				Wait for DADDIU
2	DADDIU R1,R1,#8	5				Wait for BNE
2	BNE R2,R3,LOOP	6				
3	LD R2,0(R1)					
3	DADDIU R2,R2,#1					
3	SD R2,0(R1)					
3	DADDIU R1,R1,#8					
3	BNE R2,R3,LOOP					

Two-Issue Dynamically Scheduled Processor without Speculation (7)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4				Wait for BNE
2	DADDIU R2,R2,#1	4				Wait for LD
2	SD R2,0(R1)	5				Wait for DADDIU
2	DADDIU R1,R1,#8	5				Wait for BNE
2	BNE R2,R3,LOOP	6				Wait for DADDIU
3	LD R2,0(R1)	7				
3	DADDIU R2,R2,#1	7				
3	SD R2,0(R1)					
3	DADDIU R1,R1,#8					
3	BNE R2,R3,LOOP					

Two-Issue Dynamically Scheduled Processor without Speculation (8)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8			Wait for BNE
2	DADDIU R2,R2,#1	4				Wait for LD
2	SD R2,0(R1)	5				Wait for DADDIU
2	DADDIU R1,R1,#8	5	8			Wait for BNE
2	BNE R2,R3,LOOP	6				Wait for DADDIU
3	LD R2,0(R1)	7				Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8				
3	DADDIU R1,R1,#8	8				
3	BNE R2,R3,LOOP					

Two-Issue Dynamically Scheduled Processor without Speculation (9)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9		Wait for BNE
2	DADDIU R2,R2,#1	4				Wait for LD
2	SD R2,0(R1)	5	9			Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6				Wait for DADDIU
3	LD R2,0(R1)	7				Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8				Wait for DADDIU
3	DADDIU R1,R1,#8	8				Wait for BNE
3	BNE R2,R3,LOOP	9				

Two-Issue Dynamically Scheduled Processor without Speculation (10)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4				Wait for LD
2	SD R2,0(R1)	5	9			Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6				Wait for DADDIU
3	LD R2,0(R1)	7				Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8				Wait for DADDIU
3	DADDIU R1,R1,#8	8				Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (11)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11			Wait for LD
2	SD R2,0(R1)	5	9			Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6				Wait for DADDIU
3	LD R2,0(R1)	7				Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8				Wait for DADDIU
3	DADDIU R1,R1,#8	8				Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (12)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9			Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6				Wait for DADDIU
3	LD R2,0(R1)	7				Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8				Wait for DADDIU
3	DADDIU R1,R1,#8	8				Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (13)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7				Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8				Wait for DADDIU
3	DADDIU R1,R1,#8	8				Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (14)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14			Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8				Wait for DADDIU
3	DADDIU R1,R1,#8	8	14			Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (15)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15		Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8	15			Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (16)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7				Wait for LD
3	SD R2,0(R1)	8	15			Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (17)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17			Wait for LD
3	SD R2,0(R1)	8	15			Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

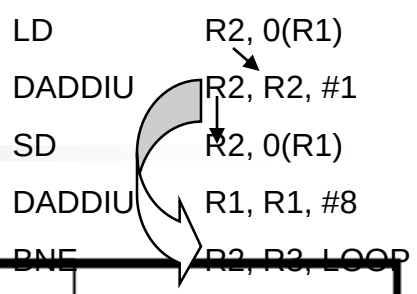
Two-Issue Dynamically Scheduled Processor without Speculation (18)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LD
3	SD R2,0(R1)	8	15			Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor without Speculation (19)

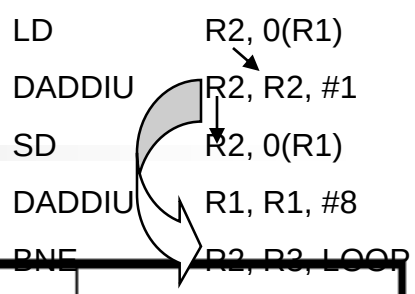
Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LD
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

Two-Issue Dynamically Scheduled Processor with Speculation (1)



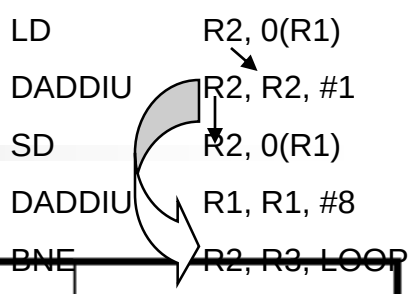
	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1					First issue
1	DADDIU R2,R2,#1	1					
1	SD R2,0(R1)						
1	DADDIU R1,R1,#8						
1	BNE R2,R3,LOOP						
2	LD R2,0(R1)						
2	DADDIU R2,R2,#1						
2	SD R2,0(R1)						
2	DADDIU R1,R1,#8						
2	BNE R2,R3,LOOP						
3	LD R2,0(R1)						
3	DADDIU R2,R2,#1						
3	SD R2,0(R1)						
3	DADDIU R1,R1,#8						
3	BNE R2,R3,LOOP						

Two-Issue Dynamically Scheduled Processor with Speculation (2)



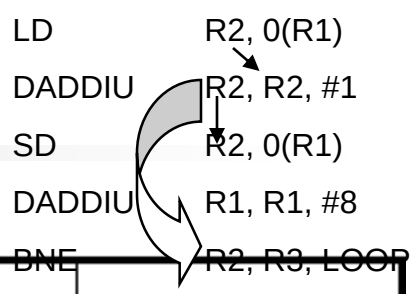
	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2				First issue
1	DADDIU R2,R2,#1	1					Wait for LD
1	SD R2,0(R1)	2					
1	DADDIU R1,R1,#8	2					
1	BNE R2,R3,LOOP						
2	LD R2,0(R1)						
2	DADDIU R2,R2,#1						
2	SD R2,0(R1)						
2	DADDIU R1,R1,#8						
2	BNE R2,R3,LOOP						
3	LD R2,0(R1)						
3	DADDIU R2,R2,#1						
3	SD R2,0(R1)						
3	DADDIU R1,R1,#8						
3	BNE R2,R3,LOOP						

Two-Issue Dynamically Scheduled Processor with Speculation (3)



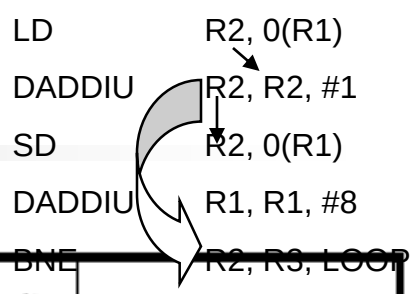
	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3			First issue
1	DADDIU R2,R2,#1	1					Wait for LD
1	SD R2,0(R1)	2	3				
1	DADDIU R1,R1,#8	2	3				
1	BNE R2,R3,LOOP	3					
2	LD R2,0(R1)						
2	DADDIU R2,R2,#1						
2	SD R2,0(R1)						
2	DADDIU R1,R1,#8						
2	BNE R2,R3,LOOP						
3	LD R2,0(R1)						
3	DADDIU R2,R2,#1						
3	SD R2,0(R1)						
3	DADDIU R1,R1,#8						
3	BNE R2,R3,LOOP						

Two-Issue Dynamically Scheduled Processor with Speculation (4)



	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4		First issue
1	DADDIU R2,R2,#1	1					Wait for LD
1	SD R2,0(R1)	2	3				Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4		
1	BNE R2,R3,LOOP	3					Wait for DADDIU
2	LD R2,0(R1)	4					
2	DADDIU R2,R2,#1	4					
2	SD R2,0(R1)						
2	DADDIU R1,R1,#8						
2	BNE R2,R3,LOOP						
3	LD R2,0(R1)						
3	DADDIU R2,R2,#1						
3	SD R2,0(R1)						
3	DADDIU R1,R1,#8						
3	BNE R2,R3,LOOP						

Two-Issue Dynamically Scheduled Processor with Speculation (5)



	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5				Wait for LD
1	SD R2,0(R1)	2	3				Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4		Commit in order
1	BNE R2,R3,LOOP	3					Wait for DADDIU
2	LD R2,0(R1)	4	5				No execute delay
2	DADDIU R2,R2,#1	4					Wait for LD
2	SD R2,0(R1)	5					
2	DADDIU R1,R1,#8	5					
2	BNE R2,R3,LOOP						
3	LD R2,0(R1)						
3	DADDIU R2,R2,#1						
3	SD R2,0(R1)						
3	DADDIU R1,R1,#8						
3	BNE R2,R3,LOOP						

Two-Issue Dynamically Scheduled Processor with Speculation (6)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6		Wait for LD
1	SD R2,0(R1)	2	3				Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4		Commit in order
1	BNE R2,R3,LOOP	3					Wait for DADDIU
2	LD R2,0(R1)	4	5	6			No execute delay
2	DADDIU R2,R2,#1	4					Wait for LD
2	SD R2,0(R1)	5	6				
2	DADDIU R1,R1,#8	5	6				
2	BNE R2,R3,LOOP	6					
3	LD R2,0(R1)						
3	DADDIU R2,R2,#1						
3	SD R2,0(R1)						
3	DADDIU R1,R1,#8						
3	BNE R2,R3,LOOP						

Two-Issue Dynamically Scheduled Processor with Speculation (7)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4		Commit in order
1	BNE R2,R3,LOOP	3	7				Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7		No execute delay
2	DADDIU R2,R2,#1	4					Wait for LD
2	SD R2,0(R1)	5	6				Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7		
2	BNE R2,R3,LOOP	6					Wait for DADDIU
3	LD R2,0(R1)	7					
3	DADDIU R2,R2,#1	7					
3	SD R2,0(R1)						
3	DADDIU R1,R1,#8						
3	BNE R2,R3,LOOP						

Two-Issue Dynamically Scheduled Processor with Speculation (8)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7		No execute delay
2	DADDIU R2,R2,#1	4	8				Wait for LD
2	SD R2,0(R1)	5	6				Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7		Commit in order
2	BNE R2,R3,LOOP	6					Wait for DADDIU
3	LD R2,0(R1)	7	8				
3	DADDIU R2,R2,#1	7					
3	SD R2,0(R1)	8					
3	DADDIU R1,R1,#8	8					
3	BNE R2,R3,LOOP						

Two-Issue Dynamically Scheduled Processor with Speculation (9)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9		Wait for LD
2	SD R2,0(R1)	5	6				Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7		Commit in order
2	BNE R2,R3,LOOP	6					Wait for DADDIU
3	LD R2,0(R1)	7	8	9			
3	DADDIU R2,R2,#1	7					Wait for LD
3	SD R2,0(R1)	8	9				
3	DADDIU R1,R1,#8	8	9				Execute earlier
3	BNE R2,R3,LOOP	9					

Two-Issue Dynamically Scheduled Processor with Speculation (10)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LD
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7		Commit in order
2	BNE R2,R3,LOOP	6	10				Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10		
3	DADDIU R2,R2,#1	7					Wait for LD
3	SD R2,0(R1)	8	9				Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10		Execute earlier
3	BNE R2,R3,LOOP	9					Wait for DADDIU

Two-Issue Dynamically Scheduled Processor with Speculation (11)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LD
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10		
3	DADDIU R2,R2,#1	7	11				Wait for LD
3	SD R2,0(R1)	8	9				Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10		Execute earlier
3	BNE R2,R3,LOOP	9					Wait for DADDIU

Two-Issue Dynamically Scheduled Processor with Speculation (12)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LD
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12		Wait for LD
3	SD R2,0(R1)	8	9				Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10		Execute earlier
3	BNE R2,R3,LOOP	9					Wait for DADDIU

Two-Issue Dynamically Scheduled Processor with Speculation (13)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LD
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LD
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10		Execute earlier
3	BNE R2,R3,LOOP	9	13				Wait for DADDIU

Two-Issue Dynamically Scheduled Processor with Speculation (14)

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LD
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LD
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	Execute earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Two-Issue Dynamically Scheduled Processor with/without Speculation

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LD
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LD
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

without

	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LD
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LD
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	Execute earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

with

Outline

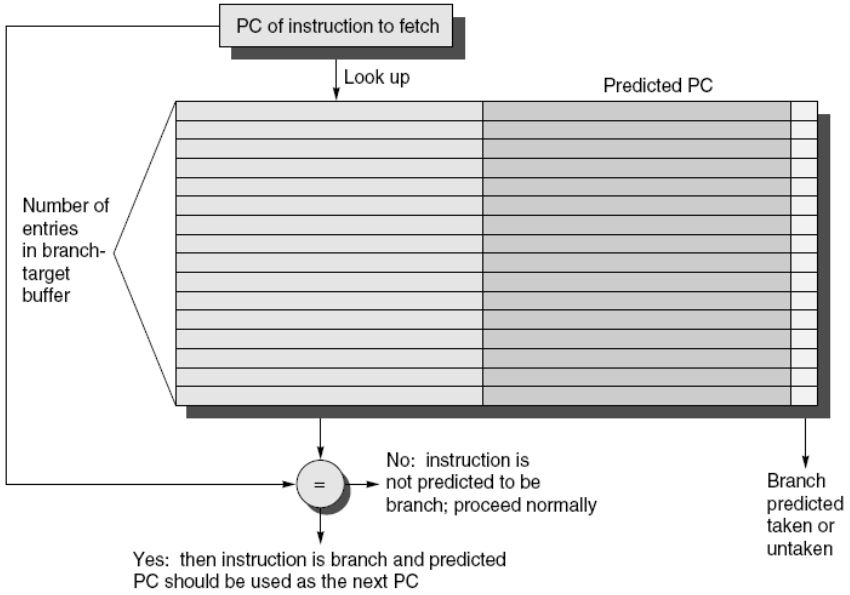
- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using dynamic scheduling, multiple issue, and speculation
- Advanced techniques for instruction delivery and speculation

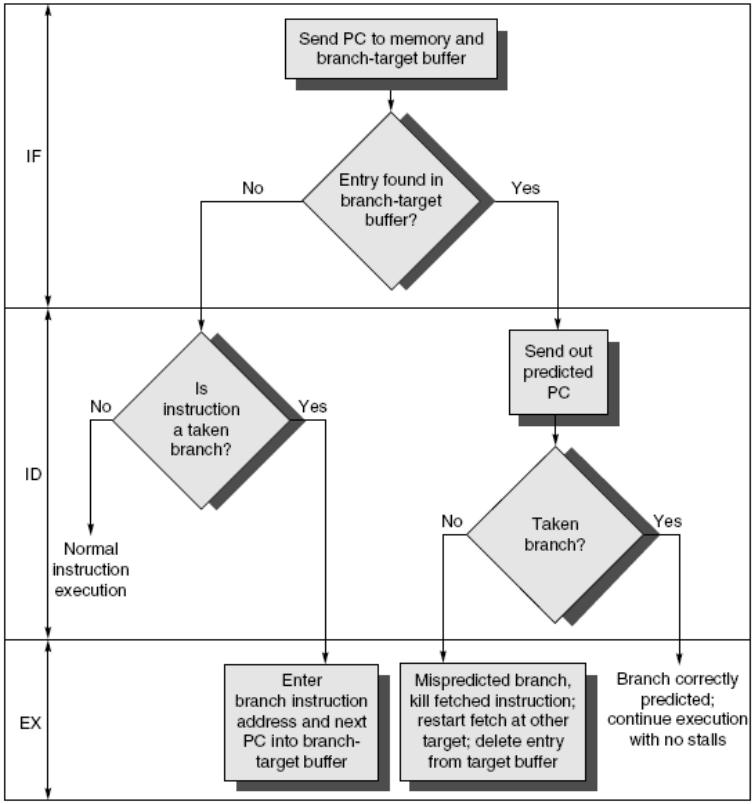
Advanced Techniques

- Increasing Instruction Fetch Bandwidth
  - Branch-Target Buffers
  - Return Address Predictors
  - Integrated Instruction Fetch Units
- Speculation: Implementation Issues and Extensions
  - Speculation Support: Register Renaming versus Reorder Buffers
  - How Much to Speculate
  - Speculating through Multiple Branches
  - Value Prediction

Branch-Target Buffer

- Need high instruction bandwidth!
  - Branch-Target buffers
    - Next PC prediction buffer, indexed by current PC





Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - “Branch folding”
    - The branch instruction is simply deleted from the instruction queue and replaced with the instruction located at the branch target.

Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack

Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative
    - Free up physical register used to hold older value
    - In other words: SWAP physical registers on commit
  - Physical register de-allocation is more difficult

Integrated Instruction Fetch Units

- Integrated branch predictor is part of instruction fetch unit and is constantly predicting branches
  - 將branch predictor整合到指令擷取元件，且讓猜測速度加外
- Instruction prefetch Instruction fetch units prefetch to deliver multiple instruct. per clock, integrating it with branch prediction (能先擷取未來可能需要執行的指令)
- Instruction memory access and buffering fetching multiple instructions per cycle: (一次擷取一個以上的指令)
  - May require accessing multiple cache blocks (prefetch to hide cost of crossing cache blocks)
  - Provides buffering, acting as on-demand unit to provide instructions to issue stage as needed and in quantity needed

Integrated Issue and Renaming

- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle (fixed number?)
    - 預先保留足夠暫存器
  - Issue logic finds dependencies within bundle, maps registers as necessary
    - Bundle中的相依性偵測
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary
    - 跨bundle

How Much?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
    - 為避免猜錯時過度影響效能，如果需要花較多執行時間的指令，將先不執行
- Speculating through multiple branches
  - Complicates speculation recovery
  - No processor can resolve multiple branches per cycle

Energy Efficiency

- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance
- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not been incorporated into modern processors
  - Similar idea--*address aliasing prediction*--is used on some processors

Speculation: Register Renaming vs. ROB

- Alternative to ROB is a larger physical set of registers combined with register renaming
  - **Extended registers replace function of both ROB and reservation stations**
- Instruction issue maps names of architectural registers to physical register numbers in extended register set
  - **On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards), i.e., Qi**
  - **Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits**
- Most Out-of-Order processors today use extended registers with renaming

How Much to Speculate

- **Speculation is not free:**
  - **It takes time and energy, and the recovery of incorrect speculation further reduces performance**
  - **The processor must have additional resources, which take silicon area and power**
  - **If speculation causes an exceptional event to occur, such as a cache or TLB miss, the potential for significant performance loss increase (if that event would not have occurred without speculation)**

How Much to Speculate (cont.)

- To maintain most of the advantage, while minimizing the disadvantages:
  - Most pipelines with speculation will allow only low cost exceptional events (such as a first-level cache miss) to be handled in speculative mode.
  - If an expensive exceptional event occurs, such as a second-level cache miss or a TLB miss, the processor will wait until the instruction causing the event is no longer speculative before handling the event.

Speculating through Multiple Branches

- Three different situations can benefit from speculating on multiple branches simultaneously:
  - A very high branch frequency
  - Significant clustering of branches
  - Long delays in functional units
- As of 2005, no processor has yet combined full speculation with resolving multiple branches per cycle.

Value Prediction

- Attempts to predict value produced by instruction
  - **E.g., Loads a value that changes infrequently**
  - **an instruction produces a value chosen from small set of potential values**
- Value prediction is useful if it significantly increases ILP
  - Focus of research has been on loads; so-so results, no processor uses value prediction
    - **The load returns a value that matches the value on the last execution of the load: 5%~80% (SPEC CPU2000)**
    - **The load to match any of the most recent 16 values returned: 80%**
- **Because of the high costs of misprediction and the likely case that misprediction rates will be significant (20% to 50%), researches have focused on accessing which loads are more predictable and only attempting to predict those.**
- **So-so results, no commercial processor has included value prediction.**

Value Prediction (cont.)

- Related topic is *address aliasing prediction*
  - RAW for load and store or
  - WAW for 2 stores
- Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict
  - Has been used by a few processors

Summary

- Instruction-level parallelism: concepts and challenges
- Basic compiler techniques for exposing ILP
- Reducing branch cots with Advanced branch prediction
- Overcoming data hazard with dynamic scheduling
- Dynamic scheduling: examples and the algorithm
- Hardware-based speculation
- Exploiting ILP using multiple issue and static scheduling
- Exploiting ILP using Dynamic Scheduling, Multiple Issue, and Speculation