

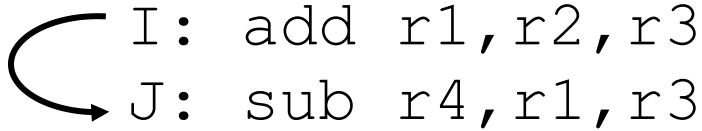
# Chapter 3: Instruction Level Parallelism and Its Exploitation

-Abdullah Muzahid

# Instruction Level Parallelism (ILP)

- Would like to exploit the independence of instructions in order to allow overlap of these instructions in the pipeline
- Amount of parallelism among instructions may be small - need ways to exploit the parallelism within the code
- Can substantially reduce the amount of time that is needed to run the code
- Obstacles to ILP -> **Dependences**

# Data Dependence and Hazards

- Instr<sub>j</sub> is **data dependent** (aka **true dependence**) on Instr<sub>i</sub>:
  1. Instr<sub>j</sub> tries to read operand before Instr<sub>i</sub> writes it  


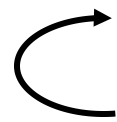
```
I:  add  r1, r2, r3
J:  sub  r4, r1, r3
```
  2. or Instr<sub>j</sub> is data dependent on Instr<sub>k</sub> which is dependent on Instr<sub>i</sub>
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence  
⇒ 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**

# ILP and Data Dependences, Hazards

- HW/SW must preserve **program order**:  
Dependences are a property of **programs**
- Presence of dependence indicates **potential** for a hazard, but actual hazard and length of any stall is property of the **pipeline**
- Importance of the data dependencies
  - 1) indicates the possibility of a hazard
  - 2) determines order in which results must be calculated
  - 3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order **only where it affects the outcome of the program**

# Name Dependence #1: Anti-dependence

- **Name dependence:** when 2 instructions use same register or memory location, called a **name**, but no flow of data between the instructions associated with that name; **2 versions of name dependence**
- Instr<sub>j</sub> writes operand **before** Instr<sub>i</sub> reads it



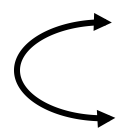
```
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

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

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

# Name Dependence #2: Output dependence

- Instr<sub>j</sub> writes operand before Instr<sub>i</sub> writes it.



```
I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

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

# Control Dependencies

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

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

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

# Control Dependence Ignored

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



# Exception Behavior

- Preserving exception behavior  
⇒ any changes in instruction execution order must not change how exceptions are raised in program  
(⇒ no new exceptions)
- Example:  
    DADDU       R2, R3, R4  
    BEQZ       R2, L1  
    LW         R1, 0(R2)  
L1:  
    – (Assume branches not delayed)
- Problem with moving LW before BEQZ?

# Data Flow

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

- **Example:**

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

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

# Eliminating Dependences

- Name Dependences - Register renaming - can be static or dynamic - simply use different registers

```
1 L5: LD    F4, 0(R1)
2      ADDD  F6, F4, F2
3      SD    F6, 0(R1)
4      SUBI  R1, R1, #8
5      BNEZ  R1, L5
```



Original loop

```
1 L5: LD    F4, 0(R1)
2      ADDD  F6, F4, F2
3      SD    F6, 0(R1) ; end of body 1
4      LD    F4, -8(R1)
5      ADDD  F6, F4, F2
6      SD    F6, -8(R1) ; end of body 2
7      SUBI  R1, R1, #16
8      BNEZ  R1, L5
```



2 iterations of the loop

# Eliminating Dependences

- Name Dependences - Register renaming - can be static or dynamic - simply use different registers

```
1 L5: LD      F4, 0(R1)
2      ADDD   F6, F4, F2
3      SD     F6, 0(R1)
4      SUBI   R1, R1, #8
5      BNEZ   R1, L5
```

Original loop



```
1 L5: LD      F4, 0(R1)
2      ADDD   F6, F4, F2
3      SD     F6, 0(R1) ; end of body 1
4      LD     F9, -8(R1)
5      ADDD   F11, F9, F7
6      SD     F11, -8(R1) ; end of body 2
7      SUBI   R1, R1, #16
8      BNEZ   R1, L5
```

2 iterations of the loop



# Eliminating Dependences

- Control Dependences – eliminate intermediate branches

```
1 L5: LD      F4, 0(R1)
2      ADDD   F6, F4, F2
3      SD     F6, 0(R1)
4      SUBI   R1, R1, #8
5      BNEZ   R1, L5
```

Original loop



```
1 L5: LD      F4, 0(R1)
2      ADDD   F6, F4, F2
3      SD     F6, 0(R1)    ; end of body 1
4      SUBI   R1, R1, #8
5      BNEZ   R1, L5
6      LD     F9, -0(R1)
7      ADDD   F11, F9, F7
8      SD     F11, -0(R1)  ; end of body 2
9      SUBI   R1, R1, #8
10     BNEZ   R1, L5
```

2 iterations of the loop



# Eliminating Dependences

- Control Dependences – eliminate intermediate branches

```
1 L5: LD      F4, 0(R1)
2      ADDD   F6, F4, F2
3      SD     F6, 0(R1)
4      SUBI   R1, R1, #8
5      BNEZ   R1, L5
```

Original loop



```
1 L5: LD      F4, 0(R1)
2      ADDD   F6, F4, F2
3      SD     F6, 0(R1) ; end of body 1
4      LD     F9, -8(R1)
5      ADDD   F11, F9, F7
6      SD     F11, -8(R1) ; end of body 2
7      SUBI   R1, R1, #16
9      BNEZ   R1, L5
```

2 iterations of the loop



# Eliminating Dependences

- What about data dependences?
  - We do not eliminate them
  - We try to avoid hazards caused by them with **scheduling**

# Scheduling

- Static Scheduling :
  - if there is a hazard, stop the issue of the instruction and the ones that follow
- Dynamic Scheduling :
  - hardware rearranges the exec of instructions to reduce stalls
  - + handles cases when dependences are unknown at compile time ( e.g. involve a mem. ref.)
  - + simplify compiler
  - + allows code compiled for 1 pipeline to run on another
  - significant hardware complexity



# Dynamic Scheduling

DIVD F0,F2,F4

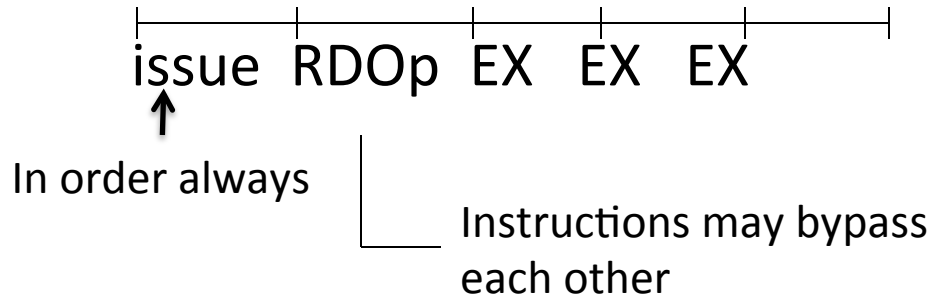
ADDD F10,F0,F8

SUBD F12,F8,F14 ← stuck, even though not dependent

- After the instruction fetch:
  - Check structural hazards
  - Wait for the absence of data hazard

┌───────────┐ decode , check for structural hazard  
Issue RDOp → wait until no data hazards , then read  
                    operands

┌──────────┐  
ID



May have WAR hazards:

DIVD F0

ADDD F10, F0, F8

SUBD F8

May have WAW hazards:

DIVD F0

ADDD F8, F0

SUBD F8

# Techniques

- This type of machine is called dynamically scheduled or out of order execution machine
- 2 techniques:
  - Scoreboarding (**out of syllabus**)
  - Tomasulo's algorithm

# Tomasulo's Algorithm

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

# Tomasulo

- Basic idea
  - Reservation stations
    - Fetch and buffer operands as soon as they are available
    - No need to operate from registers
  - As instructions are issued
    - Reg specifiers for pending operands are renamed to names of reservation stations
    - This avoids WAW and WAR

# How to resolve it

	DIV	F0,F2,F4	
————	ADD	F6,F0,F8	————
	SD	F6,0(R1)	
	SUB	F8,F10,F14	————
————	MUL	F6,F10,F8	

# Renaming

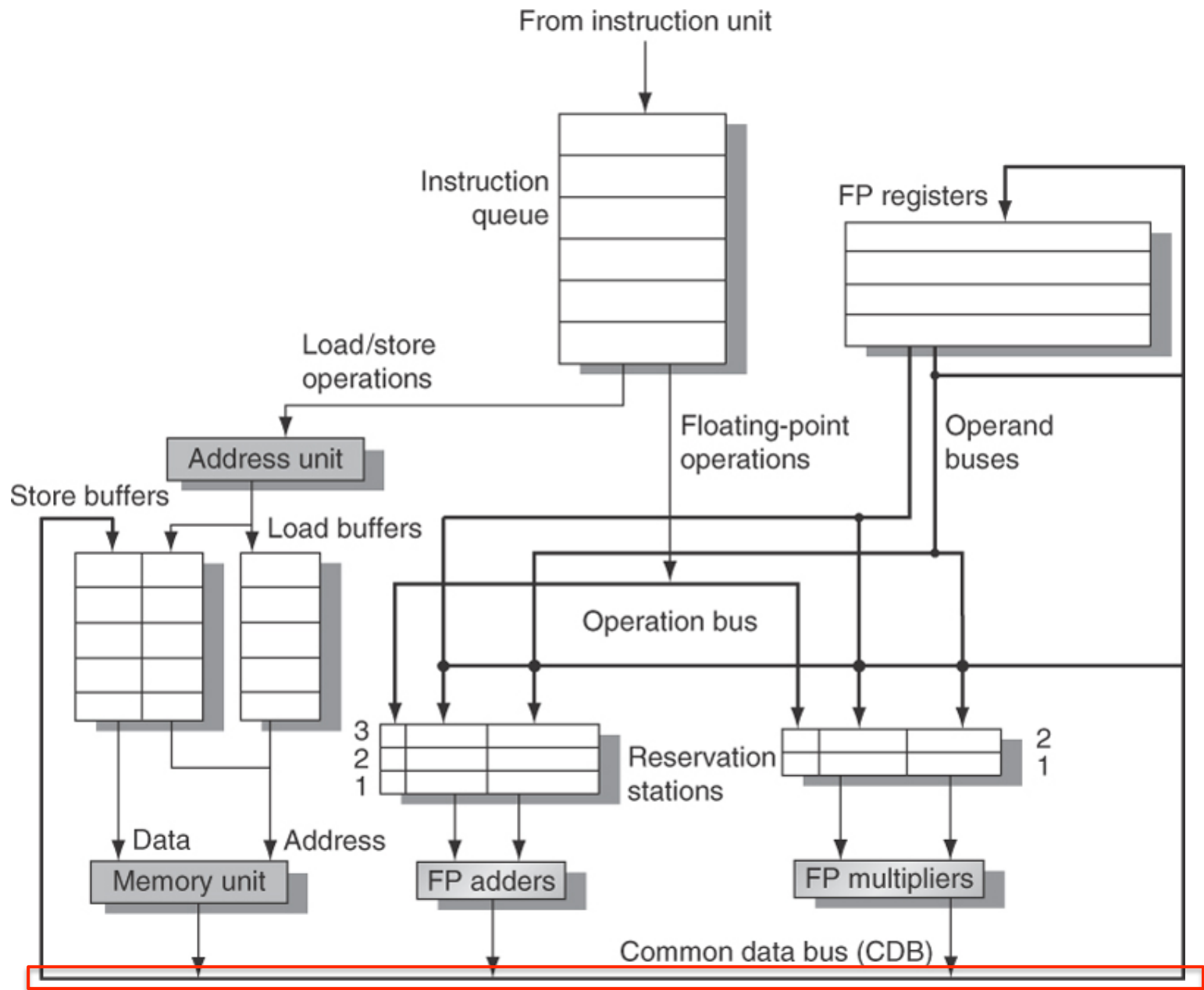
	DIV	F0,F2,F4	
————	ADD	<b>F6</b> ,F0, <b>F8</b>	     
	SD	F6,0(R1)	
	SUB	<b>F8</b> ,F10,F14	
————	MUL	<b>F6</b> ,F10,F8	

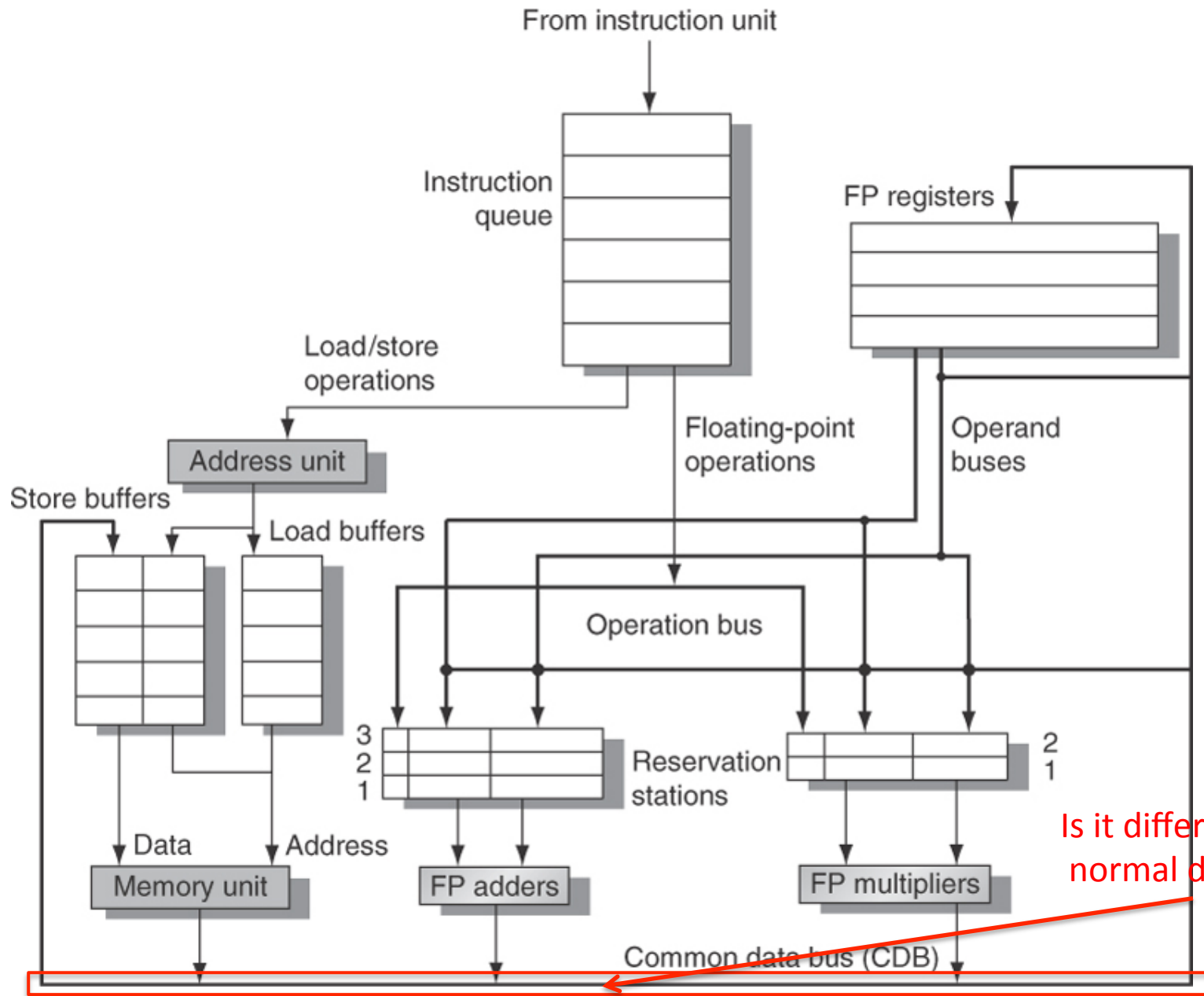
DIV F0,F2,F4  
ADD **S**,F0,F8  
SD **S**,0(R1)  
SUB **T**,F10,F14  
MUL F6,F10,**T**

# Tomasulo

- Register renaming provided by the reservation stations:
  - buffer the operands of instructions waiting to execute
  - Pending instructions designate the reservation station that will provide their input → effectively a register
  - When successive writes to a register overlap in execution, only the last one is actually used to update the register
- Other characteristics of Tomasulo:
  - Hazard detection and execution control are distributed
  - Bypassing everywhere (use the common data bus CDB – all units waiting for a result can load it simultaneously)







# Steps of an Instruction

- Issue :
  - get next instr from instruction queue
  - issue it to empty reservation station
  - send operands to the rs; if ops not ready, write the rs that will produce them
  - if no reserv stations / buffers: structural Hz , stall
  - This step renames registers, eliminating WAR and WAW
- EX:
  - monitor bus for available operand
  - when available, put it in rs
  - when all ops ready, execute
  - By delaying until all ops are available, handle RAW
  - Independent functional units can begin executing in the same cycle
  - If two rs in the same FU become ready in the same cycle, one is chosen to execute

# Steps of an Instruction (Cont)

- EX (cont):
  - Ld/st have a two-step execution
  - First step: compute effective address when base register is available and then eff. addr. is placed in the load or store buffer
  - Second step: actual mem access
  - For now: do not allow EX of any instruction following a branch until the branch is resolved (later: allow EX, not allow WB)
- WR:
  - write result on bus. From there, it goes to regs & res. Stations/buffers

# Workout Table

Instruction	Reservation Station	Exec FU	Issue	Exec begin-end	Mem Access	CDB write
LD F6, 32(R2)						
LD F2, 44(R3)						
MULD F0, F2, F4						
SUBD F8, F2, F6						
DIVD F10, F0, F6						
ADDD F6, F8, F2						

### Typical assumptions:

- IS,WR take one cycle each
- One instruction IS per cycle
- Functional Units (FUs) not pipelined
- Results are communicated via the CDB
- Assume you have as many load/store buffers as needed
- Loads/stores take 1 cycle to execute and 1 cycle to access memory
- Loads/stores share a memory access unit
- Branches and stores do not have WR/CDB write stage
- If an instruction is in its WR stage in cycle  $x$ , then an instruction that is waiting on the same FU (due to a structural hazard) can start executing on cycle  $X$ , unless it needs to read the CDB, in which case it can only start executing on cycle  $X+1$
- Only one instruction can write to the CDB in a clock cycle
- Whenever there is a conflict for the FU, assume that the first (in program order) of the conflicting instructions gets access, while the others are stalled.
- Latency: Int FU 1 cycle, F. Add 2 cycles, Mul 6 cycles, Div 12 cycles

# More on elimination of WAW, WAR

```
Loop : LD      F0,0(R1)
        MULTD  F4,F0,F2
        SD      F4,0(R1)
        DADDUI R1,R1,-8
        BNE     R1,R2,Loop
```

- predict that branches will be taken
  - loop is unrolled dynamically by the hardware ( no need many regs)

Instruction	Reservation Station	Exec FU	Issue	Exec begin-end	Mem Access	CDB write
LD F0, 0(R1)						
MULD F4, F0, F2						
SD F4, 0(R1)						
DADDIU R1, R1, -8						
BNE R1, R2, Loop						
LD F0, 0(R1)						
MULD F4, F0, F2						
SD F4, 0(R1)						
DADDIU R1, R1, -8						
BNE R1, R2, Loop						



...however , need dynamic disambiguation of address  
stall if  $\text{addr Ld} = \text{addr pending stores}$  (or forward)  
or if  $\text{addr St} = \text{addr pending loads or stores}$

else , could execute iterations out of order

Problems:

- hardware intensive

- CDB bottleneck (if replicate, replicate logic too)

key features: dynamic scheduling  
register renaming  
dynamic memory disambiguation

# Dynamic Hardware Branch Prediction

- Control hazards are sources of losses , especially for processors that want to issue  $> 1$  instr / cycle
  - this approach : use H/W to dynamically predict the outcome of a branch (may change with time)
- 1 Branch prediction buffer ( branch history table)
    - Small memory indexed by lower bits of addr of branch instruction
    - Contains 1 bit that says if branch was recently taken

Note : the prediction may refer to another branch that has some low-order address bits

- if hint is wrong : prediction bit is inverted
- Scenario: A loop iterates 9 times

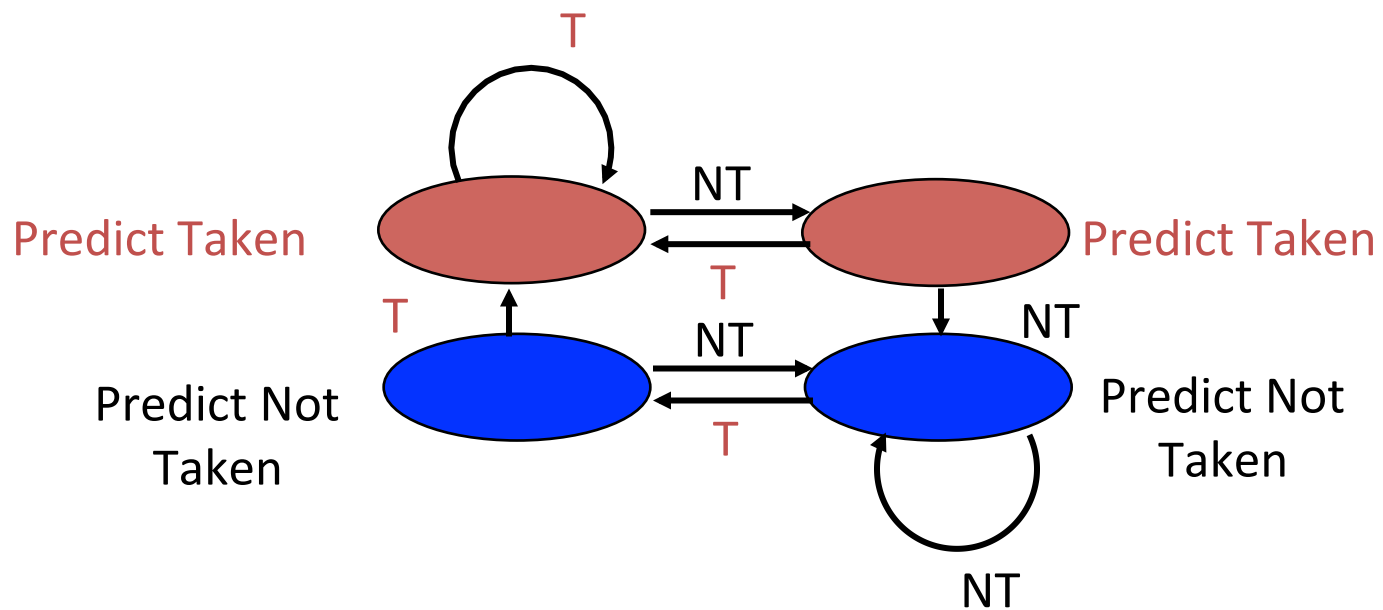
```
loop_begin: ...
```

```
...
```

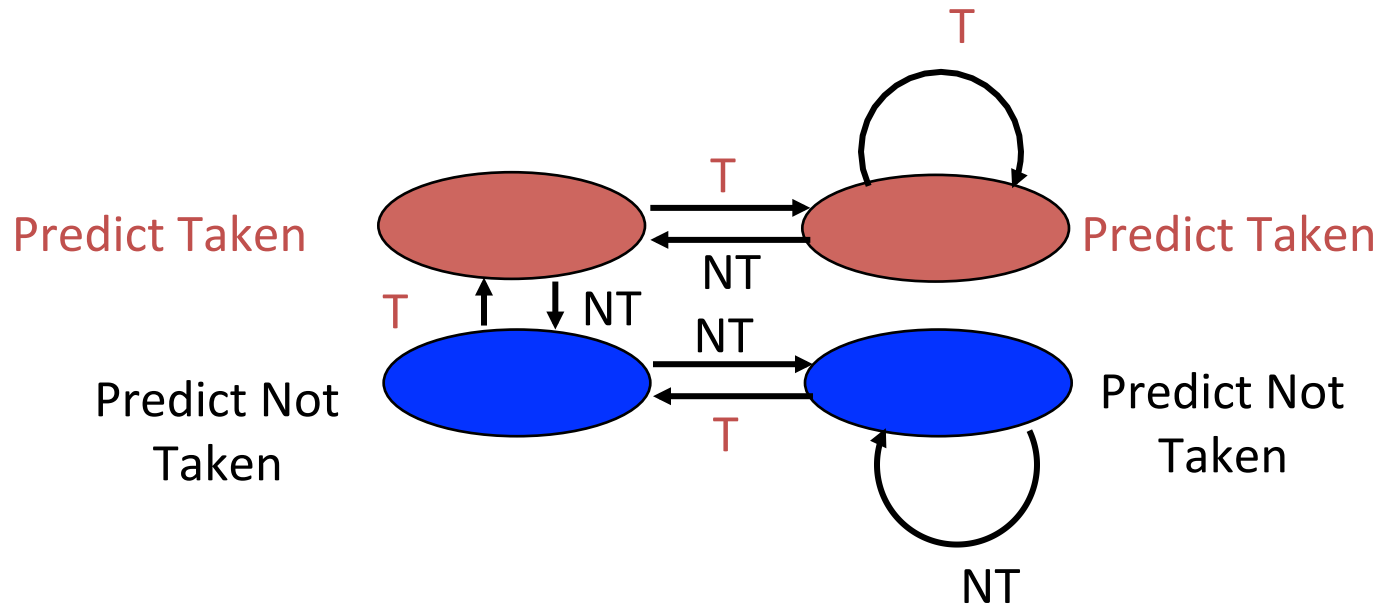
```
BEQZ loop_begin
```

How many mispredictions?

- 2 Two-bit prediction schemes
  - need to mispredict twice before changing the predict



Alternative Design



### 3. N-bit saturating counter

- n bit counter can take from 0 to  $2^n - 1$
- Half of them are used for predict taken and other half for predict untaken

- Design a 3 bit saturating counter

Accuracy : 4096 entries in table , 2 bits (large)

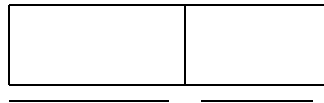
- misprediction rate  $\approx 1 - 18\%$  in spec89
- usually better at FP programs (more loops)

4 Look at recent behavior of other branches too  
“correlating predictors or two-level predictors”

if ( d == 0)

    d = 1 ;

if (d == 1)



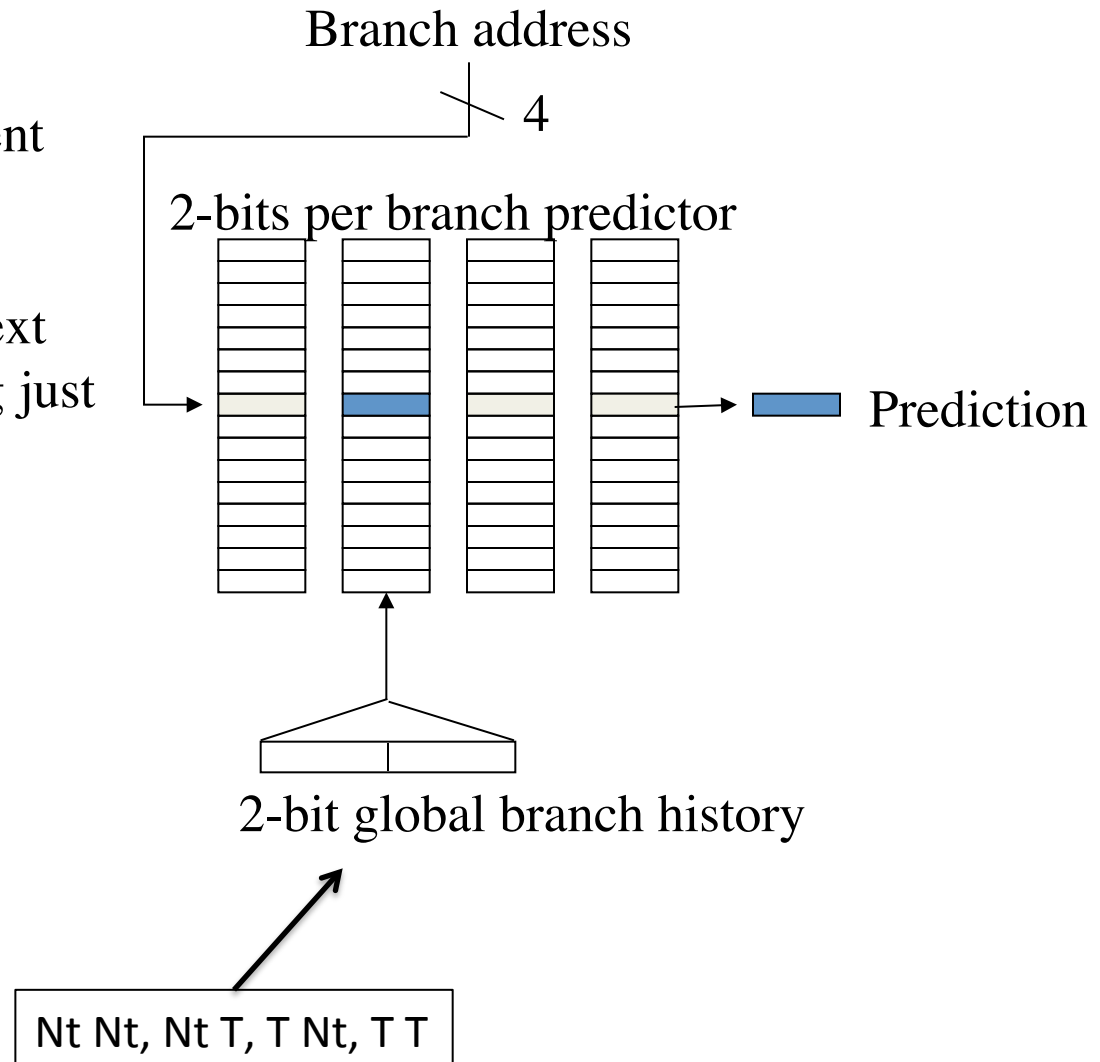
prediction used  
if the last branch  
not taken

prediction used if  
the last branch taken

# Correlating Branches

## (2, 2) predictor

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





# An Example

ADDUI R1, R0, #2

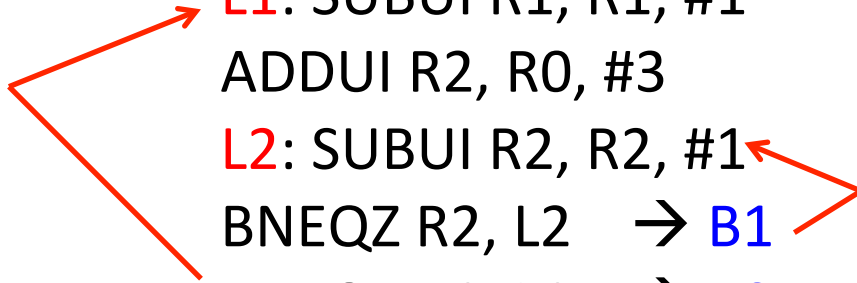
**L1:** SUBUI R1, R1, #1

ADDUI R2, R0, #3

**L2:** SUBUI R2, R2, #1

BNEQZ R2, L2 → **B1**

BNEQZ R1, L1 → **B2**



What is sequence of branches and their outcomes?

# An Example

ADDUI R1, R0, #2

**L1:** SUBUI R1, R1, #1

ADDUI R2, R0, #3

**L2:** SUBUI R2, R2, #1

BNEQZ R2, L2 → **B1**

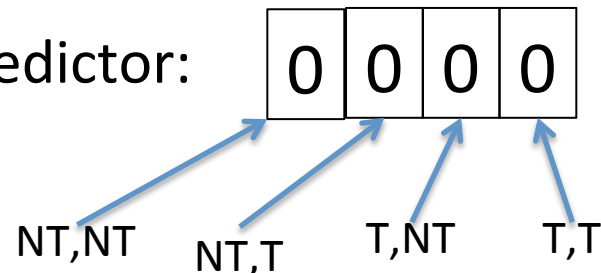
BNEQZ R1, L1 → **B2**

What is sequence of branches and their outcomes?

Outcome

**B1(T), B1(T), B1(NT), B2(T), B1(T), B1(T), B1(NT), B2(NT)**

Lets assume a (2,1) predictor:



# An Example

ADDUI R1, R0, #2

L1: SUBUI R1, R1, #1

ADDUI R2, R0, #3

L2: SUBUI R2, R2, #1

BNEQZ R2, L2 → B1

BNEQZ R1, L1 → B2

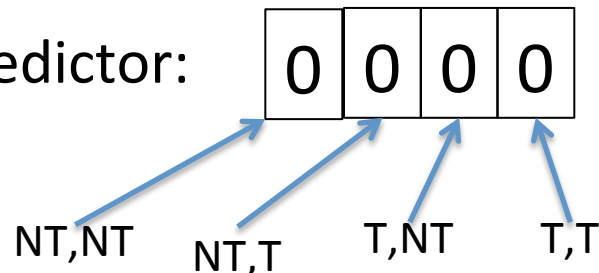
What is sequence of branches and their outcomes?

Outcome

B1(T), B1(T), B1(NT), B2(T), B1(T), B1(T), B1(NT), B2(NT)

Lets assume a (2,1) predictor:

Predictions



B1(NT), B1(NT), B1(NT), B2(NT), B1(T), B1(NT), B1(T), B2(T)

# Correlated Branch Prediction

- Idea: record  $m$  most recently executed branches as taken or not taken, and use that pattern to select the proper  $n$ -bit branch history table ( $n$ -bit counter)
- In general,  $(m,n)$  predictor means record last  $m$  branches to select between  $2^m$  entries, each with  $n$ -bit counters
  - Thus, old 2-bit BHT is a  $(0,2)$  predictor
- Global Branch History:  $m$ -bit shift register keeping T/NT status of last  $m$  branches.

# Total Size of Predictor (m, n)

History length

Each predictor size

- Number of rows (indexed by branch address):  $x$

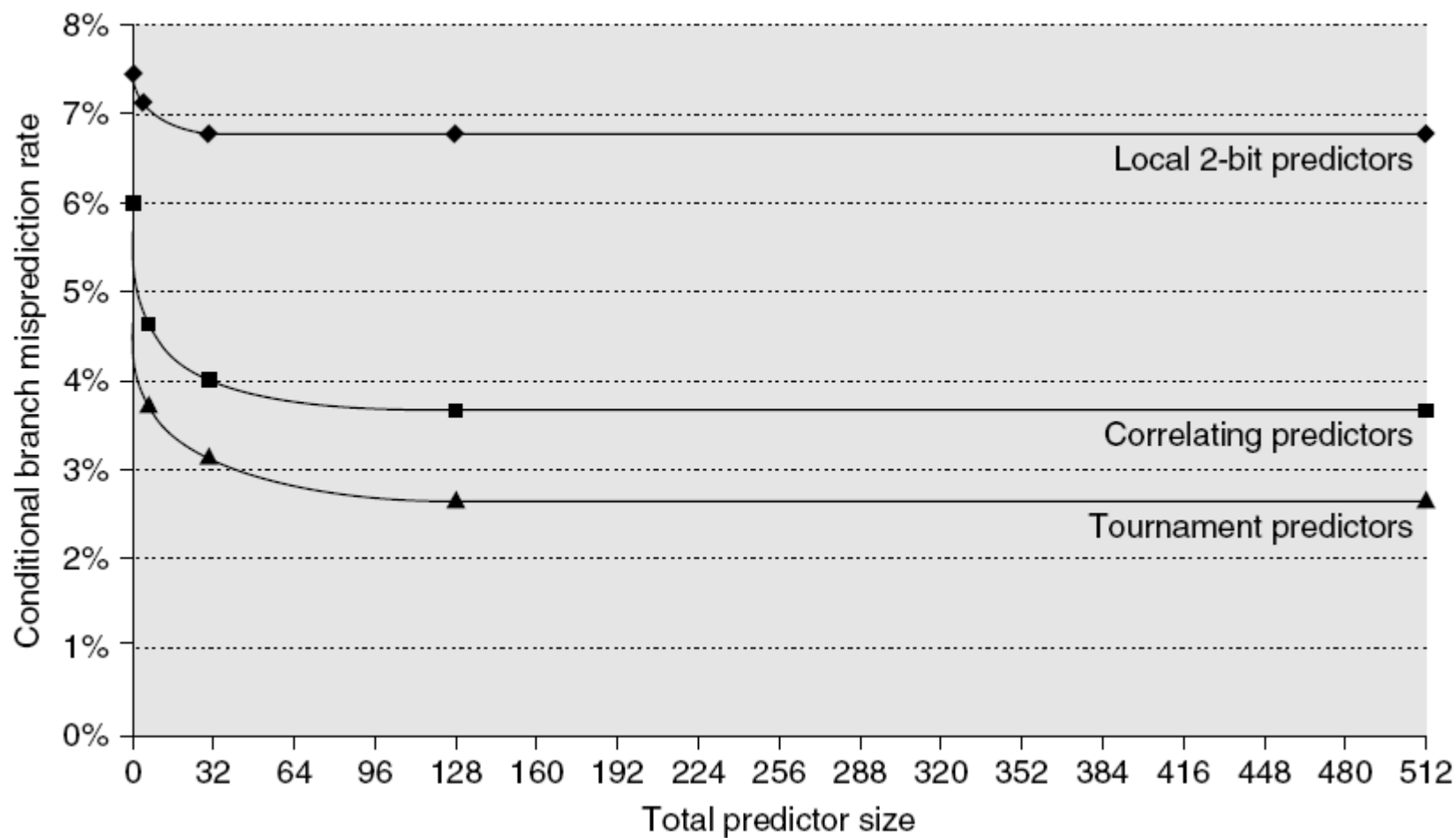
$$x * 2^m * n$$

# Tournament Predictors

- Multilevel branch predictor
- Use  $n$ -bit saturating counter to choose between [predictors](#)
- Usual choice between global and local predictors

# Comparing Predictors (Fig. 3.4)

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

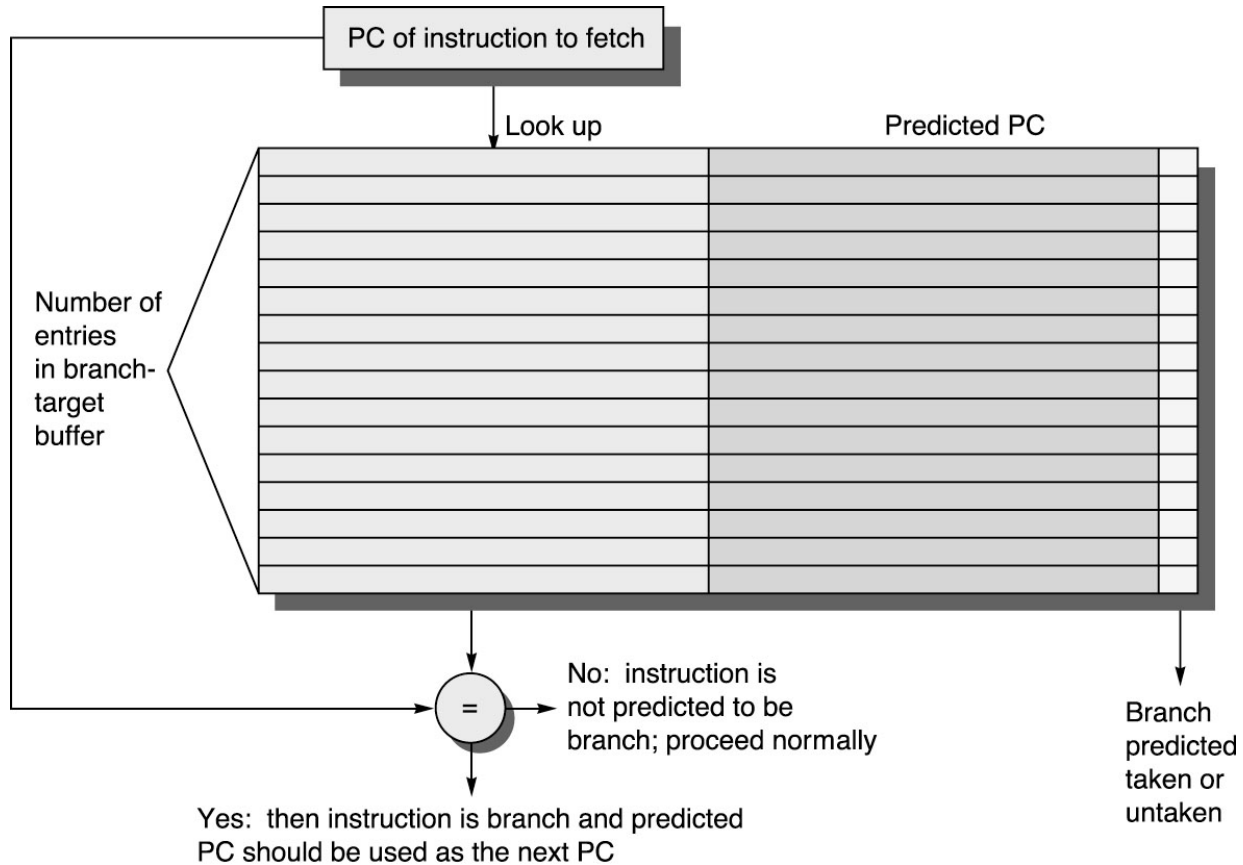




## Branch Target Buffers (BTB)

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

# Branch Target Buffers



# Return Address Predictors

- Handling indirect jumps (from procedure returns)
  - cannot use traditional BTB's (target changes)
  - use a stack : push the return addr on a call; pop it at return. For example 1 - 16 entries

## Other

- Fetch from both the predicted and unpredicted direction:
  - Some processors have used
  - costly