Instruction-Level Parallelism

and

Its Dynamic Exploitation

Chapter 3 - Computer Architecture : *A Quantitative Approach* - Hennessy & Patterson

Instruction-level Parallelism

- Parallelism among instructions.
- Dynamic technique used by hardware to locate parallelism: Pentium III & 4, MIPS R10000, Sun UltraSPARC, Alpha
- Static technique used by compiler.

Data Dependences

The instruction J is data dependent on instruction I if

- The instruction J uses the result produce by instruction I or
- There is an instruction K data dependent on I and J is data dependent on K.
- Data dependence through register or through memory.

Dependences

- Data dependences are properties of a program.
- Hazard and stall caused by such dependence is characteristic of a pipelined implementation.
- A dependence determines the order of value computation, indicates the possibility of hazards and gives an upper bound of IPL.

Overcoming Dependences

- **Detection of dependence - more difficult to detect if it is through memory.**
- **Transform the code to eliminate the dependence.**
- **Avoid the hazards by hardware - forwarding, dynamic scheduling.**

Other Dependences

- Name dependence - anti-dependence and output dependence.
- Control dependence - program order.

Name Dependence

- **The instruction I is anti-dependent on a following instruction J if I reads from a location written by J. The instruction I should read before the instruction J writes.**
- **The instruction I is output dependent on the instruction J if both of them writes the same register or memory location. The order of there write should be the program order.**

Different Data Hazards

- **RAW - read after write in a pipeline.**
- **WAW - write in more than one pipe stages, multiple functional units etc.**
- **WAR - out of order issue, writes early or reads late.**

Control Dependence

- Program order is to be preserved.
- An instruction dependent on branch cannot be moved out of the scope of the branch.
- An instruction not dependent on branch cannot be moved within the scope of the branch.

Control Dependence

- Control dependence may be violated provided the program correctness is guaranteed.
- The critical properties for program correctness are - data flow and behavior on exception.
- Data flow to an instruction is not static due to the presence of branch. Static data dependence analysis is not sufficient.

Data Flow

```
        DADD    R1,R2,R3
        BEQZ    R1, L1
        LD      R4, 0(R1)
L1:
```

The load is not data dependent on any of the previous instructions. But it is control dependent on the branch.

What is the problem to take it above the branch?

Data Flow

```
        DADDU   R1,R2,R3
        BEQZ    R4, L1
        DSUBU   R9,R5,R6
        DSUBU   R1,R8,R9
L1:
        OR      R7,R1,R8
```

The value of R1 received by OR R7,R1,R8 depends on the branch.

If it is known that R9 will not be used after L1 (dead), it can be shifted above the branch.

Dynamic Scheduling

- **Instructions are rearranged by the hardware to reduce stalls.**
- **The data flow and exception behavior is maintained.**
- **Data dependences unknown at the compilation time can be handled better (data dependence through memory).**
- **Code compiled for different pipeline model may run efficiently.**
- **More hardware complexity.**

Dynamic Scheduling

- **The main problem of a pipeline is in-order instruction issue and execution.**

```
DIV.D   F0,F2,F4
ADD.D   F10,F0,F8
SUB.D   F12,F8,F14
```

The ADD.D instruction will be stalled due to the DIV.D instruction resulting a stall for SUB.D which is not dependent.

Dynamic Scheduling

- In the previous example if there is no structural hazard i.e. a functional unit is available,
- an instruction can be issued if there is no data hazard.
- The issue and start of execution are separated out.
- In-order issue, but out of order start and completion of execution.

Out-of-Order Execution and Exception

- Out-of-Order execution and completion may give rise to WAW and WAR hazards.
- Complication in exception behavior.
- No instruction should raise exception until it is guaranteed to be executed in program order.
- The exception may be imprecise but the program order exception behavior must be preserved.

Dynamic Scheduling

- The ID stage is splitted into
  - Issue: decode the instruction and check for structural hazard - stall if there is any.
  - Read Operand: wait if there is a data hazard, then read operands and start execution.
- The execution has a begin and an end.

Robert Tomasulo's Algorithm

- **Designed for IBM 360 machines.**
- **Keeps track of the availability of instructions.**
- **Register renaming is done through the unnamed registers in the reservation stations. This avoids WAW and WAR hazards.**

Register Renaming

```
DIV.D   F0,F2,F4
ADD.D   F6,F0,F8
S.D     F6, 0(R1)
SUB.D   F8,F10,F14
MUL.D   F6,F10,F8
```

There is a anti-dependence between ADD.D (F8) and SUB.D (F8). There is output dependence between ADD.D (F6) and MUL.D (F6).

Register Renaming

```
DIV.D   F0,F2,F4
ADD.D   V1,F0,F8
S.D     V1, 0(R1)
SUB.D   V2,F10,F14
MUL.D   F6,F10,V2
```

Every use of F8 defined by SUB.D is substituted by V2.

Reservation Station

- Buffers an instruction, operands and the result.
- When empty, it gets an instruction from an instruction queue.
- It fetches and buffers the operands as soon as they are available (reduces register access).
- If the current value of the operand is not from a register but as an output of some previous instruction, it keeps track of the corresponding reservation station.

Reservation Station

- The **register specifier** for pending operands are **renamed** to the name of the reservation station (virtual registers).
- There are more reservation stations (in IBM 360) than ISA registers and the algorithm can eliminate more name dependences than a compiler!

Reservation Station

- Hazard detection and execution control is distributed as the information available in a reservation station decides the start of execution.
- The results are passed to different functional units through **common data bus(s) (CDB)**.

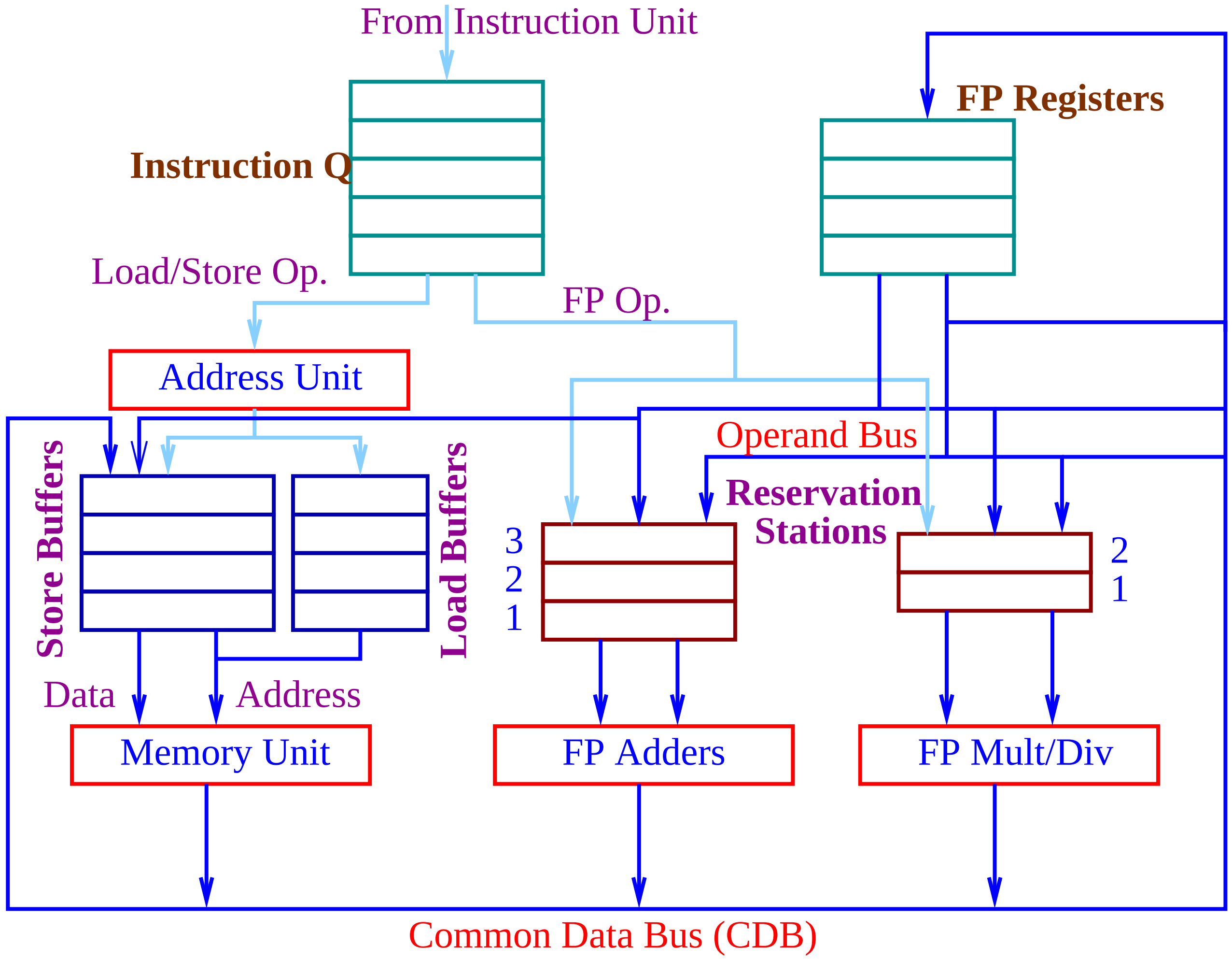


Figure 1: **MIPS Tomasulo's Structure**

MIPS Tomasulo Structure

- The structure has floating-point units and load-store units.
- Execution control tables are not shown.
- A reservation station holds an issued instruction that waits for execution in a functional unit.
- It also holds the operand values, if already available, or the name(s) of the reservation station(s) that will provide it.

Load/Store Buffer

- **Holds address and data coming from and going to memory.**
- **Very similar to reservation stations.**

Bus Connections

- There is a pair of data buses from the floating point registers to the reservation stations and a single bus to the store buffer.
- Data from all functional units and memory are sent over a common data bus (CDB) to every place other than the load buffer.

Three Steps of Execution

- **Instruction Issue.**
- **Instruction Execute.**
- **Write the result.**

Instruction Issue

- The next instruction comes from the FIFO queue (for correct data flow).
- If a reservation station (or LS buffer) corresponding to the required functional unit is available, the instruction is issued and operand values are fetched (if available in registers).
- If no reservation station is available (structural hazard), the instruction is stalled.

Instruction Issue

- **If the operands are not in the registers, the reservation unit keeps track of the functional unit that will produce it.**
- **This step renames the registers and eliminates the WAR and WAW hazards.**

Instruction Execute

- If one or more operands are not available, it monitors the CDB and place the operand in the proper reservation station when available. By this RAW hazard is avoided.
- More than one instruction may be ready for a functional unit. There may be choice or parallel execution depending on the number of available units.

Execution of Load/Store

- **Load/store has two steps - effective address computation (when the base register is available).**
- **The effective address, after computation, is stored in the load/store buffer**
- **Load in the load buffer can be executed as soon as the memory unit is available.**
- **Store may have to wait for the availability of the data to be stored.**

Execution of Load/Store

- **The program order of load/store is maintained through the effective address computation.**
- **This will prevent hazard through memory.**

Exception Behavior

- No instruction following a branch in program order is executed before the branch is complete.
- This restriction guarantees that an instruction causes an exception would have been actually executed in program order.

Exception Behavior

- If there is a branch prediction, the processor must know that the prediction is correct before allowing the instruction following the branch to execute.
- Exceptions may be recorded but not raised.
- It allows the instruction execution to continue before it enters the write result.

Write Result

- The result when available will be written on the CDB and from there to registers.
- The data is also written in reservation stations, store buffers and address computation unit waiting for this data.
- If the memory unit is free, the address of the location and the data to store are available, the store can be completed.

Data Structures

- Data Structures used to detect and eliminate hazards are attached to the reservation stations, load/store units and registers files.
- Data structures are tags (4-bit in this example) for names of the actual and virtual registers (reservation stations).
- In this example there are five (5) reservation stations and six (6) load buffers.

Seven (7) Fields of a Reservation Station

- **Op** - operation to perform on source operands **S1** and **S2**.
- **Qj** and **Qk** - the reservation stations that will produce the source operand. If the value is **zero (0)**, it indicates that the value is available in **Vj** or **Vk**; or it may be unnecessary.
- **Vj** and **Vk** - values of the source operands. Both **V** and **Q** fields cannot be valid.

Seven (7) Fields of a Reservation Station

- **A** - stores the immediate data. Stores the offset and finally the effective address for load/store buffer.
- **Busy** - the reservation station and the corresponding functional unit is busy.

Register Status

- **Qi - The number of the reservation station computing the new value for this register. If it is zero (0), no currently running instruction is computing its value.**

Tomasulo's Algorithm: An Example

```
L.D    F6,34(R2)
L.D    F2,45(R3)
MUL.D  F0,F2,F4
SUB.D  F8,F2,F6
DIV.D  F10,F0,F6
ADD.D  F6,F8,F2
```

We consider the state when the first load is complete and the second load has not yet written the register F2.

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
L_1	no						
L_2	yes	load					45 + R3
A_1	yes	sub		M[34+R2]	L_2		
A_2	yes	add			A_1	L_2	
A_3	no						
M_1	yes	mul		F4	L_2		
M_2	yes	div		M[34+R2]	M_1		

Register Status (Qi)

F0	F2	F4	F6	F8	F10	F12	$\cdots$	F30
M_1	L_2		A_2	A_1	M_2			

Instruction Issue: FP Operations

- **Wait until a station r is available.**
- **Action:**

```
if (RegisterStatus[rs].Qi != 0)
  ReservationStation[r].Qj = RegisterStatus[rs].Qi ;
else {
  ReservationStation[r].Vj = Reg[rs] ;
  ReservationStation[r].Qj = 0 ;
}
```

Instruction Issue: FP Operations

```
if (RegisterStatus[rt].Qi != 0)
  ReservationStation[r].Qk = RegisterStatus[rt].Qi ;
else {
  ReservationStation[r].Vk = Reg[rt] ;
  ReservationStation[r].Qk = 0 ;
}
ReservationStation[r].Busy = yes ;
RegisterStatus[rd].Qi = r ;
```

Instruction Issue: Load/Store

- **Wait until a buffer r is available.**
- **Action:**

```
if (RegisterStatus[rs].Qi != 0)
  ReservationStation[r].Qj = RegisterStatus[rs].Qi ;
else {
  ReservationStation[r].Vj = Reg[rs] ;
  ReservationStation[r].Qj = 0 ;
}
ReservationStation[r].A = imm;
ReservationStation[r].Busy = yes ;
```

Instruction Issue: Load

```
RegisterStatus[rt].Qi = r ;
```

Instruction Issue: Store

```
if (RegisterStatus[rt].Qi != 0)
  ReservationStation[r].Qk = RegisterStatus[rt].Qi ;
else {
  ReservationStation[r].Vk = Reg[rt] ;
  ReservationStation[r].Qk = 0 ;
}
```

Execute: Floating-point Op.

- **Wait until** `ReservationStation[r].Qi = 0` **and** `ReservationStation[r].Qk = 0`.
- **Action: Compute with operands from Vj and Vk.**

Execute: Load/Store Op.

- **Wait until** `ReservationStation[r].Qi = 0` **and r is head of load/store queue.**
- **Action 1:** `ReservationStation[r].A = ReservationStation[r].Vj + ReservationStation[r].A`
- **Action 2 (load only): read from** `Mem[ReservationStation[r].A]`

Write Result: Floating-point Op. or Load

- **Wait until execution complete at r and CDB is available.**
- **Action:**
  - $\forall$x (if (RegisterStatus[x].Qi = r) {
    Reg[x] = result;
    RegisterStatus[x].Qi = 0})
  - $\forall$x (if (ReservationStation[x].Qj = r) {
    ReservationStation[x].Vj = result;
    ReservationStation[x].Qj = 0 })

Write Result: Floating-point Op. or Load

- $\forall$x (if (ReservationStation[x].Qk = r) {
  ReservationStation[x].Vk = result;
  ReservationStation[x].Qk = 0 })
- ReservationStation[r].Busy = no

Write Result: Store

- **Wait until execution complete at r and** `ReservationStation[r].Qk = 0`.
- **Action:**

```
Mem[ReservationStation[r].A] =
                  ReservationStation[r].Vk ;
ReservationStation[r].Busy = no
```

Order of load/store

- **If the memory addresses are different, load and store can be done in any order.**
- **But if the address is same then reordering load-store and store-load and store-store give rise to WAR, RAW and WAW hazards respectively.**

Avoidance of Load/Store Hazard

- To determine whether a **load can be executed**, the processor should check whether there is any **incomplete store** in the same location.
- To determine whether a **store can be executed**, the processor should check whether there is any **incomplete load or store** in the same location.
- To check this it is necessary to **compute the effective address** of the data memory in **program order**.

Tomasulo's Scheme

- Renaming of architectural registers by the reservation stations.
- Buffering of source operands from the register file.
- Broadcast of result over the CDB.

Dynamically Scheduled Pipeline: Performance

- A **dynamically scheduled pipeline can give high performance** provided the **branches are predicted correctly**.
- The **hardware complexity of Tomasulo's scheme** is **very high**.

Dynamically Scheduled Pipeline: Performance

- Each reservation station must contain a high speed associative buffer with a complex control logic.
- There should be more than one CDB for high performance, but every additional CDB will add extra tag-matching hardware for each reservation station.

Reducing Branch Costs: Hardware Prediction

- **The control dependences become very important for processors with high ILP.**
- **A processor that can issue n instructions per clock cycle, encounters branches n times faster.**
- **The impact of control stalls is larger for such a low CPI machine.**

Dynamic Branch Prediction

- The hardware can **dynamically predict** the outcome of a branch.
- The prediction mechanism should be **accurate and fast**.
- Branch penalty depends on the structure of the pipeline, type of predictor and the recovery cost from mis-prediction.

Branch-Prediction Buffer

- A simplest **Branch prediction buffer** is a **branch history table** indexed by the **lower portion** of the **address** of the **branch instruction**.
- The **table (memory)** in its simplest form may take a **single bit** to indicate whether the branch was recently **taken or not**. The current branch is **predicted** on the basis of the bit.
- The buffer is a **hash table** and the entry may correspond to **another branch instruction** with the **same low order bit**.

Branch-Prediction Buffer

- The fetching of next instructions starts depending on the hint of the prediction buffer.
- If the prediction is wrong, the prediction bit is inverted.
- The target address should be known along with the prediction.
- Little impact on 5-stages MIPS pipeline.

Problem with 1-bit Prediction Scheme

```
.L6: cmpl    $9, -8(%ebp)
     jle     .L9
     jmp     .L4
.L9: movl    -12(%ebp), %eax
     sall    $1, %eax
     movl    %eax, -12(%ebp)
     leal    -8(%ebp), %eax
     incl    (%eax)
     jmp     .L6
.L4:
```

Problem with 1-bit Prediction Scheme

- **If the initial prediction is not-taken (inner loop), then there are two mis-predictions - the initial and the final.**
- **The branch will be taken ten (10) out of eleven (11) times.**
- **If it is an inner loop there will be large number of mis-predictions.**

2-bit Prediction Scheme

- A **2-bit prediction scheme** uses a **2-bit saturation counter.**
- The counter is **incremented** on a **taken branch** and is **decremented** on an **untaken branch.**
- The prediction is changed only after **two misses.**

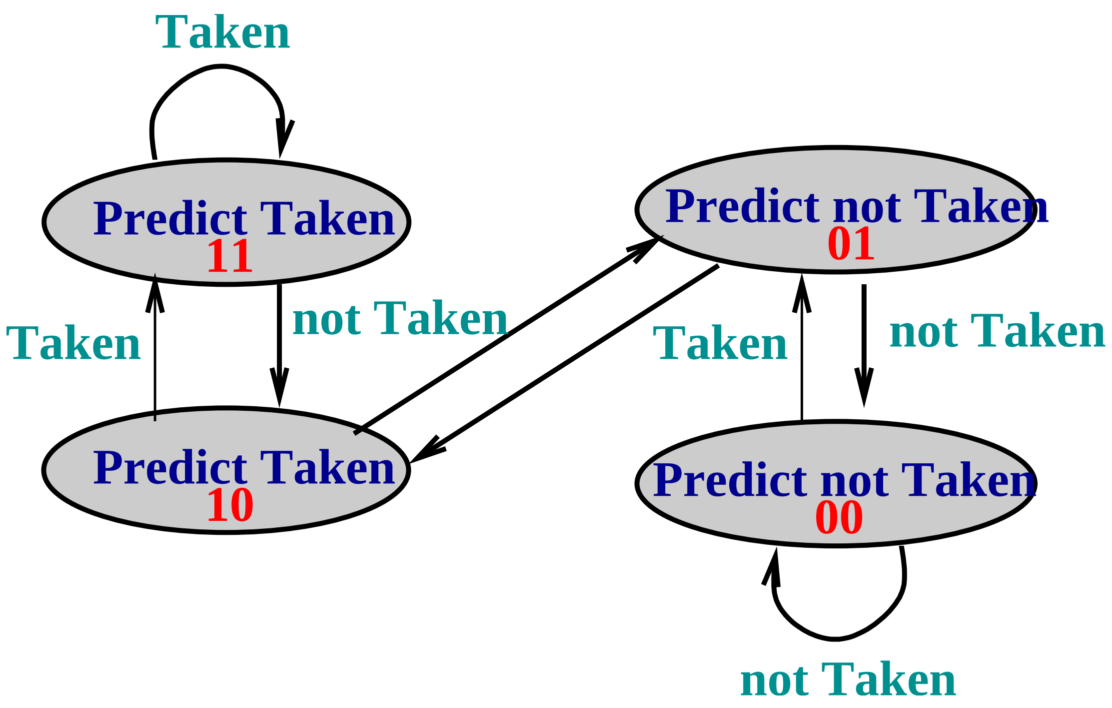


Figure 2: **2-bit Predictor**

Prediction Buffer

- Buffer may be implemented as a **special cache** addressed by the **least significant bits** of the **instruction address**.
- The **prediction bits** in another scheme may be attached to the **instruction cache**.
- The buffer is accessed in the **IF stage**.
- Once the instruction is known to be a **branch** and the **branch address is computed**, new instructions are fetched from the **target** if the **prediction is taken**.

Prediction in 5-stage Pipeline

- **In the classic 5-stage pipeline, both whether the branch is taken and the value of the branch address is known in the ID stage. The prediction scheme is of no extra help.**

Performance of 2-bit Branch Prediction

- **2-bit prediction buffer of size 4K entries and SPEC89 benchmark gives 1% to 18% mis-predictions on different programs.**
- **Calculation of performance loss also depends on branch frequency.**
- **Integer programs have higher branch frequencies than floating-point scientific computations.**

Improvement of Branch Prediction Scheme

- Increase in the **size of the prediction buffer** - 4K entry buffer is comparable to infinite buffer.
- Increase in the **accuracy of the scheme** - increase of the number of bits is not enough.

Correlation Among Branches

- So far we were looking at the **history of a branch to predict its next behavior.**
- The **prediction may be more accurate** if we look at the **history of some other branches.**

A Small Piece of Code

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

The variable `aa` and `bb` are assigned to registers `R1` and `R2` respectively.

MIPS Translation

```
         DADDI   R3, R1, #-2
         BNEZ    R3, L1     ;branch b1 (aa != 2)
         DADD    R1, R0, R0
L1:      DADDI   R3, R2, #-2
         BNEZ    R3, L2     ;branch b1 (bb != 2)
         DADD    R2, R0, R0
         DSUBU   R3, R1, R2
         BEQZ    R3, L3     ;branch b3 (aa == bb)
```

The branch b3 is correlated to b1 and b2. If both are not taken, then b3 is taken.

Correlating Predictors (2-level predictor)

- A branch predictor that uses the behavior of other branches for prediction.

1-bit of Correlation

- Each branch has two (2) separate prediction bits b1 b0.
- If the last branch was not taken, **one prediction bit** is used - b1.
- If the last branch was taken, the **other prediction bit** is used - b0.
- In general, the last branch is not the branch being predicted.

An Example

```
        BNEZ     R1, L1    ;b1 (d!=0)
        DADDIU   R1, R0, #1 ;d = 1
L1:     DADDUI   R3, R1, #-1
        BNEZ     R3, L2    ;b2 (d!=1)
     ..............
L2:
```

Assume that the code is in a loop and the values alternate between 2 and 0. As if there is no other branch!

1-bit Predictor (Initialized not taken)

d	P-b1	b1	N-b1	P-b2	b2	N-b2
2	nt	T	t	nt	T	t
0	t	NT	nt	t	NT	nt
2	nt	T	t	nt	T	t
0	t	NT	nt	t	NT	nt

P - predicted, N - new prediction.

All predictions are wrong.

1-bit Predictor + 1-bit Correlation

d	P-b1	b1	N-b1	P-b2	b2	N-b2
2	nt/nt	T	t/nt	nt/nt	T	nt/t
0	t/nt	NT	t/nt	nt/t	NT	nt/t
2	t/nt	T	t/nt	nt/t	T	nt/t
0	t/nt	NT	t/nt	nt/t	NT	nt/t

P - predicted, N - new prediction.

Though there are only two mis-predictions, the branch b1 is not correlated to b2. But b2 is correlated to b1.

(m, n) Predictor

- **The correlation predictor we have discussed is called $(1, 1)$ predictor.**
- **It uses the behavior of the last branch to choose from a pair of 1-bit branch predictors.**
- **An (m, n) predictor uses the behavior of last m branches to choose from 2^m predictors, each of which is an n-bit predictor.**

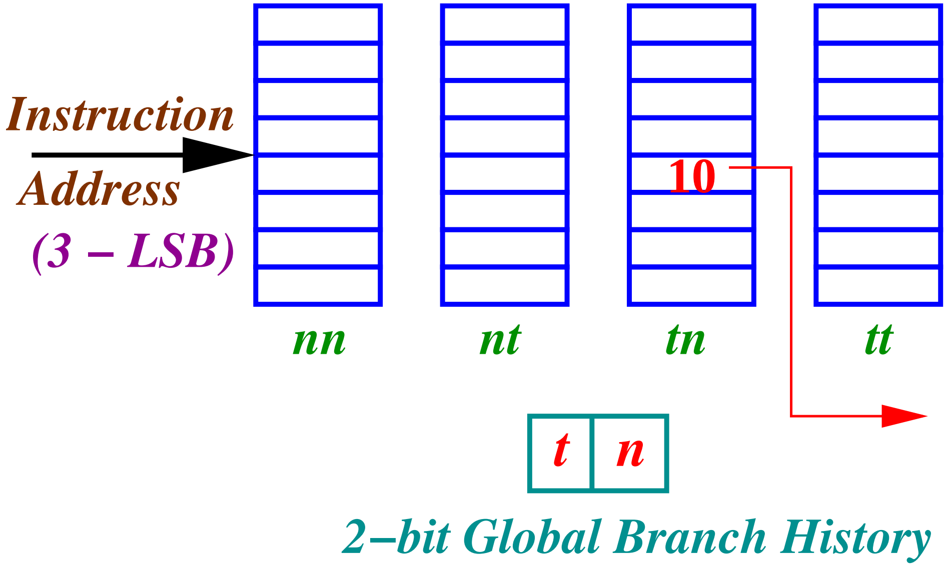


Figure 3: $(2, 2)$ **Predictor**

Performance

- Comparison of **(2,2) correlation predictor with 1K entries** shows that it not only **out performs the 2-bit simple predictor of size 4K**, but also **out performs 2-bit predictor of unlimited size.**

Tournament Predictor: Multilevel Prediction

- It uses **multiple predictors** - one base on **global information** and one based on **local information**.
- They are combined with a selector.
- The selector may be a **saturation counter per branch** to choose among the two predictors.
- The counter is incremented if the **predictor 2 is correct** and the **predictor 1 is incorrect**. It is decremented in reverse situation.

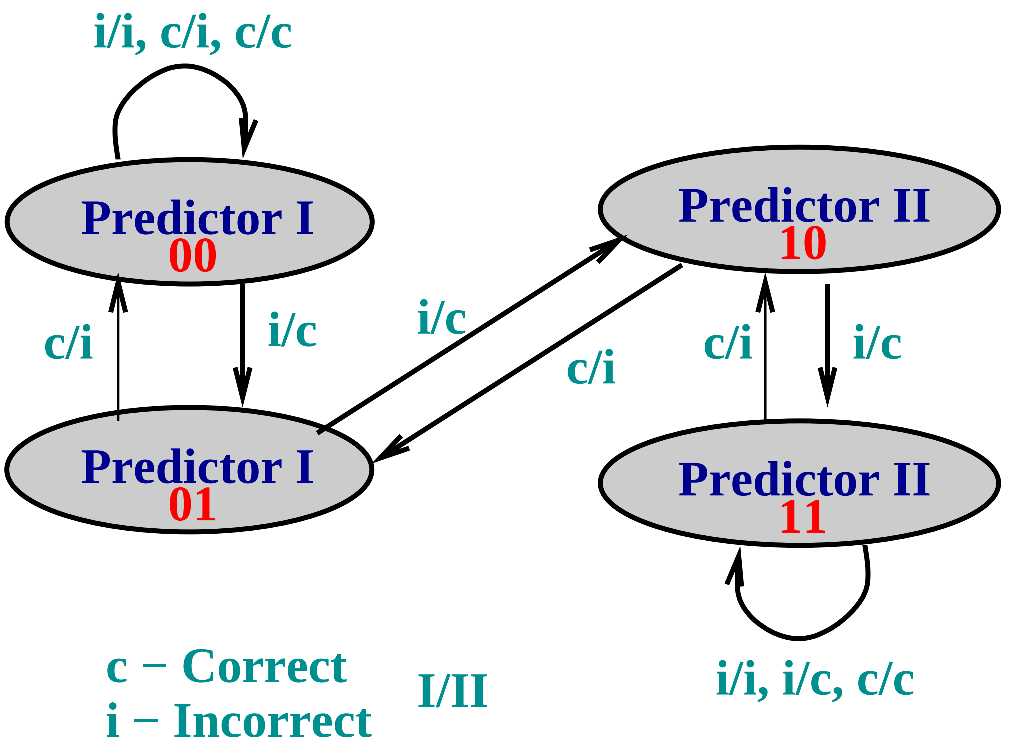


Figure 4: **Selector State Transition**

Alpha 21264: Tournament Predictor

- **4K 2-bit selector counters indexed by local branch address to choose from a global predictor and a local predictor.**
- **The global predictor has 4k entries and is indexed by the history of last 12 branches. Each one is a 2-bit simple predictor.**

Alpha 21264: Tournament Predictor

- There are **two levels** of the local predictor.
- The top level has 1024 10-bit entries. Each entry corresponds to the **most recent 10 branch outcomes for this entry**.
- The **selected entry** from the local table is used as index to a **table (1024 entries) of saturation counters (each 3-bit)**. The counter is used for local prediction.
- The total number of bits are: $2 \times 4\text{k}$ (selector) $+ 2 \times 4\text{K}$ (global) $+ (10 \times 1\text{K} + 3 \times 1\text{K}) = 29\text{K}$.

High Bandwidth Instruction Delivery

- **Prediction of branch is not enough for a high-performance pipeline with multiple instruction issue.**
- **The necessity is to deliver 4-8 instructions per clock cycle.**

High Bandwidth Instruction Delivery

- **Branch-target buffer.**
- **Integrated instruction fetch unit.**
- **Return address prediction.**

Branch-Target Buffer

- Consider the 5-stage pipeline.
- The address of the physically-next instruction is loaded in the PC at the end of the IF stage.
- The branch-prediction and the branch-address are known only at the end of the ID stage.
- There is one cycle delay for a taken branch.

Branch-Target Buffer

- If the **branch-target buffer** and the **branch-prediction buffer** are accessed in the **IF stage** using the **address of the fetched instruction**, and it is a **hit**, the target address can be loaded in the PC without delay.

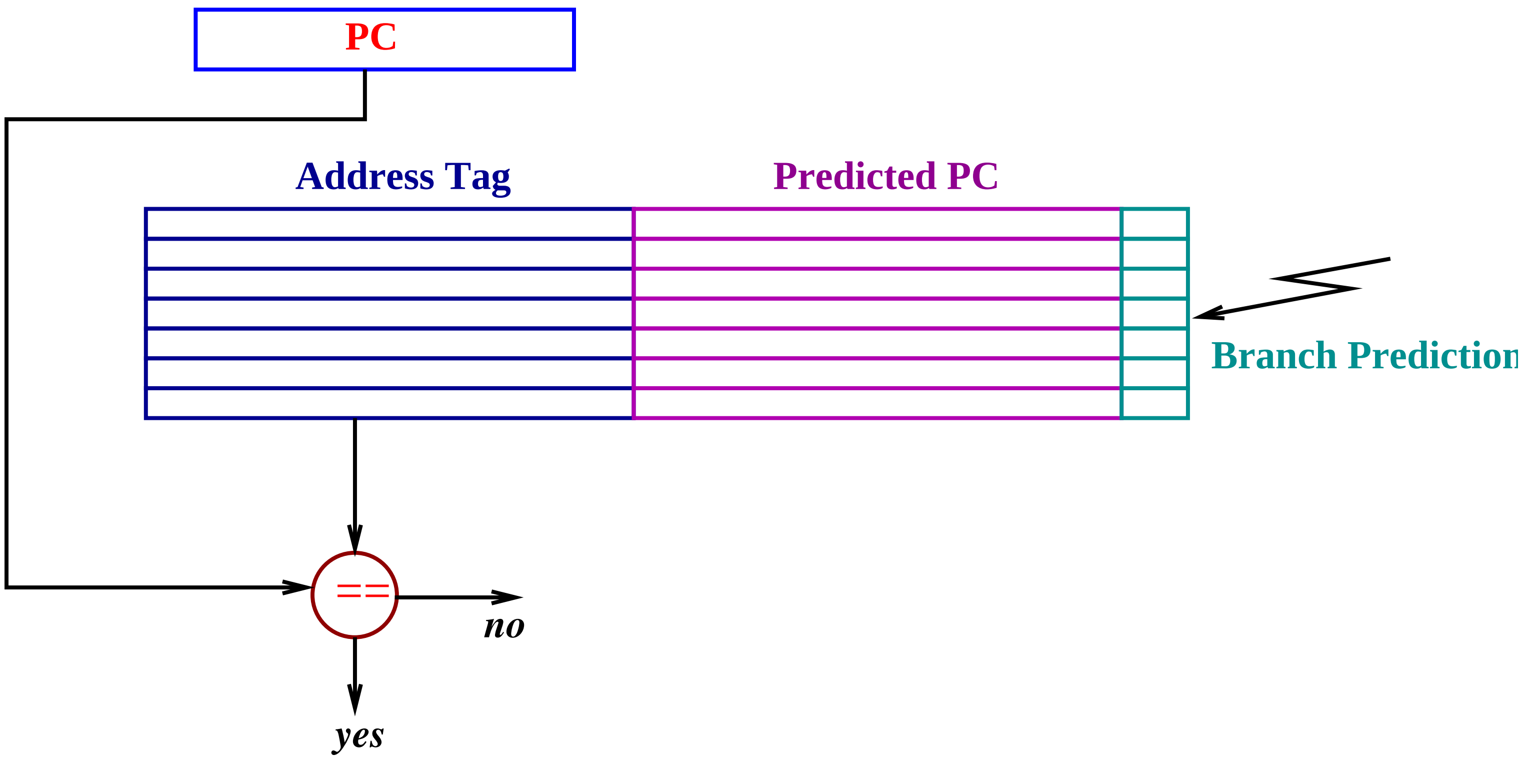
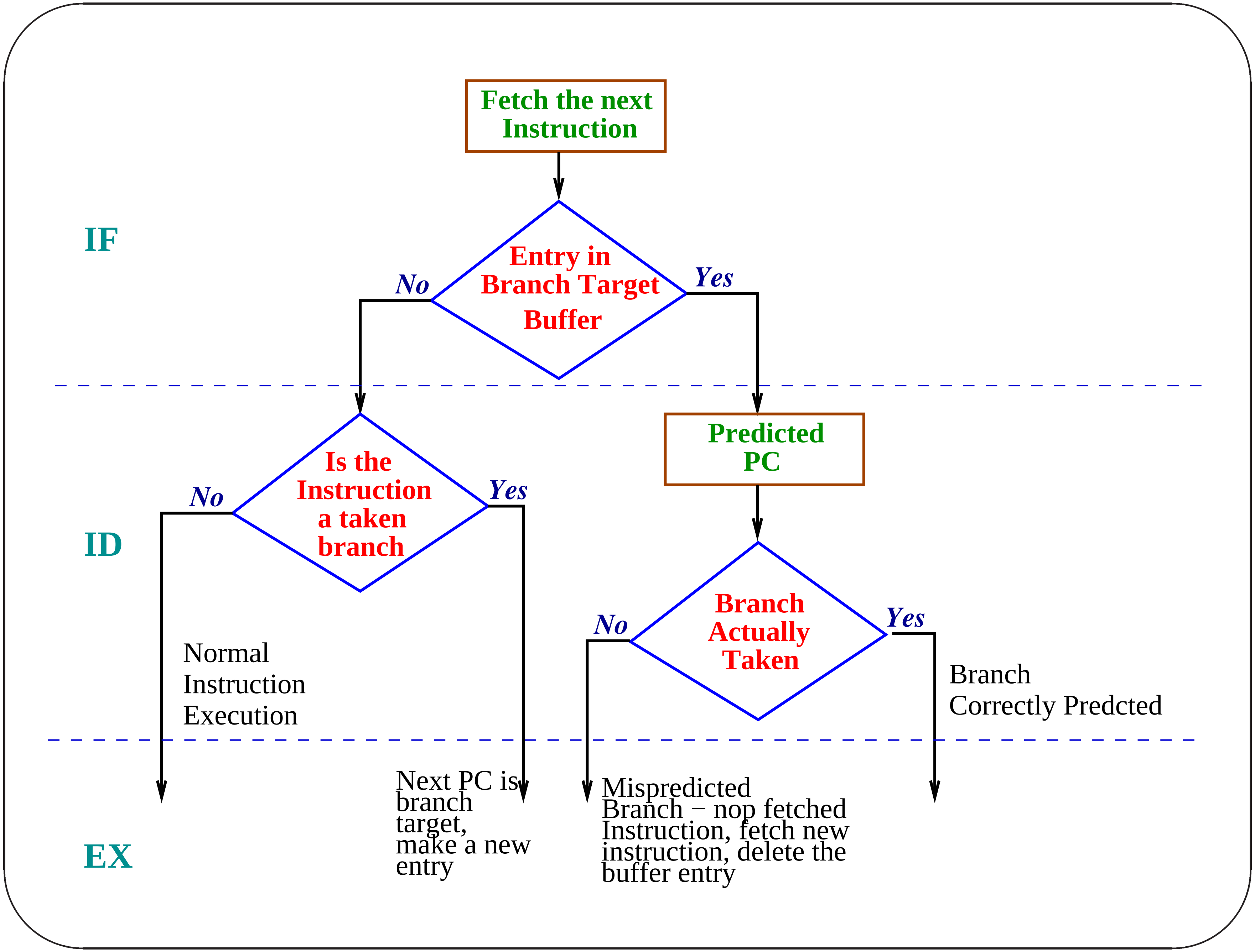


Figure 5: **Branch Target Buffer**

Branch-Target Buffer

- The **branch-target buffer** is an **associative memory** **searched** by the **address of the fetched instruction**.
- If there is a **match** and the **prediction is taken**, the PC is loaded with the predicted target PC value.
- **Entries of branch-target buffer** should correspond to **taken branches**; but for **2-bit predictor** it is necessary to **include non-taken branches** as well.



Fetch the next Instruction
IF
Entry in Branch Target Buffer
No
Yes
Is the Instruction a taken branch
No
Yes
Predicted PC
ID
Branch Actually Taken
No
Yes
Normal Instruction Execution
Branch Correctly Predcted
EX
Next PC is branch target, make a new entry
Mispredicted Branch − nop fetched Instruction, fetch new instruction, delete the buffer entry

Penalty

Inst. in Buff.	Prediction	Actual Br.	Cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

Branch-Target Buffer

- **Branch target buffer with the target instruction.**

Integrated Instruction Fetch Unit

- In a **multiple-instruction-issue processor**, **instruction fetch** is not a simple stage of the pipeline any more.
- A **separate** and **integrated instruction fetch unit** is there to **supply instructions to the pipeline**.
- Branch prediction, instruction prefetch, instruction memory access and buffering are put together in the integrated instruction fetch unit.

Indirect Jump Prediction

- In a **high-level program indirect jumps** come from **case statement**, **computed goto's of FORTRAN**, **indirect procedure calls** etc.
- But in a **benchmark large percentage of indirect jumps** comes from **procedure return.**

Procedure Return

- **Jump-target buffer** is **not very suitable** for a **procedure return** as the procedure may be called from **different points.**
- Moreover there **may not be any clustering over time** for calls from a point.
- A small **buffer of return addresses** is organized as a **stack**.

Return Address Buffer

- **The return address is pushed once the procedure is called.**
- **The predicted return address is perfect as long as the size of the buffer is larger than the call depth.**