OoO Processor: Dynamically Scheduled Pipelines

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

October 2020 PMPH Lecture Slides

Course Organization

W	HARDWARE		SOFTWARE	LAB/CUDA
1	Trends Vector Machine	⟵	List HOM (Map-Reduce)	Intro & Simple Map Programming
2	In Order Processor	⟶ ⟵	VLIW Instr Scheduling	Scan & Reduce
3	Cache Coherence		Reasoning About Parallelism	Sparse Vect Matrix Mult
4	Interconnection Networks		Case Studies & Optimizations	Transpose & Matrix Matrix Mult
5	Memory Consistency		Optimising Locality	Sorting & Profiling & Mem Optimizations
6	OoO, Spec Processor		Thread-Level Speculation	Project Work

Three narative threads: the path to complex & good design:

- Design Space tradeoffs, constraints, common case, trends.
- Reasoning: from simple to complex, Applying Concepts.

Acknowledgments

This lecture presents selected Topics from Chapter 3 of the
"Parallel Computer Organization and Design" book,
by Michel Dubois, Murali Annavaram and Per Stenstrom.

1. Tomasulo Algorithm
2. Dynamic Branch Prediction
3. Speculative Execution
4. Multiple Instructions Per Clock
5. Pentium III and Pentium IV
6. OoO Processors: Memory Consistency Models (MCM)

Dynamic Instruction Scheduling

- Static pipelines: exploit parallelism exposed by the compiler

Dynamic Instruction Scheduling

- Static pipelines: exploit parallelism exposed by the compiler
  - instrs are stalled in ID until they are hazard free
  - compiler is limited, e.g., indirect arrays, unbiased branches, etc.
- Potential for exploiting a lot of ILP across 100s of instrs:

Dynamic Instruction Scheduling

- Static pipelines: exploit parallelism exposed by the compiler
  - instrs are stalled in ID until they are hazard free
  - compiler is limited, e.g., indirect arrays, unbiased branches, etc.
- Potential for exploiting a lot of ILP across 100s of instrs:
  - must cross basic block boundaries,
  - data-flow execution order instead of thread order.
- Dynamic instr scheduling separates decoding from scheduling

Dynamic Instruction Scheduling

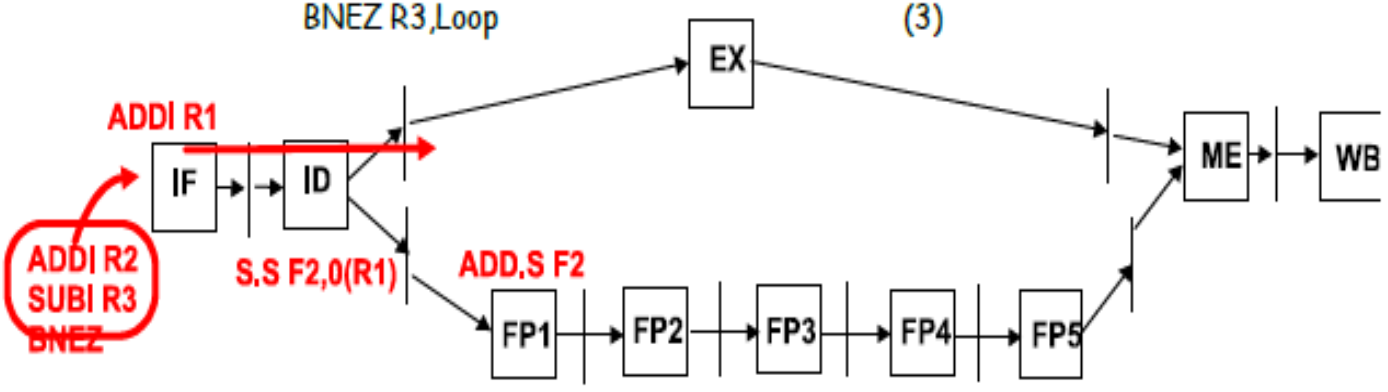
- Static pipelines: exploit parallelism exposed by the compiler
  - instrs are stalled in ID until they are hazard free
  - compiler is limited, e.g., indirect arrays, unbiased branches, etc.
- Potential for exploiting a lot of ILP across 100s of instrs:
  - must cross basic block boundaries,
  - data-flow execution order instead of thread order.
- Dynamic instr scheduling separates decoding from scheduling
  - decode instrs and dispatch them to queues, where
  - they wait for input operands to be available, then are executed.
  - Hence no stalls in ID due to data hazards (only structural).
- Several challanges

Dynamic Instruction Scheduling

- Static pipelines: exploit parallelism exposed by the compiler
  - instrs are stalled in ID until they are hazard free
  - compiler is limited, e.g., indirect arrays, unbiased branches, etc.
- Potential for exploiting a lot of ILP across 100s of instrs:
  - must cross basic block boundaries,
  - data-flow execution order instead of thread order.
- Dynamic instr scheduling separates decoding from scheduling
  - decode instrs and dispatch them to queues, where
  - they wait for input operands to be available, then are executed.
  - Hence no stalls in ID due to data hazards (only structural).
- Several challanges
  - all data hazards possible (RAW, WAR, WAW), on both memory and registers,
  - speculative execution: execute beyond conditional branches,
  - enforce precise exception model.

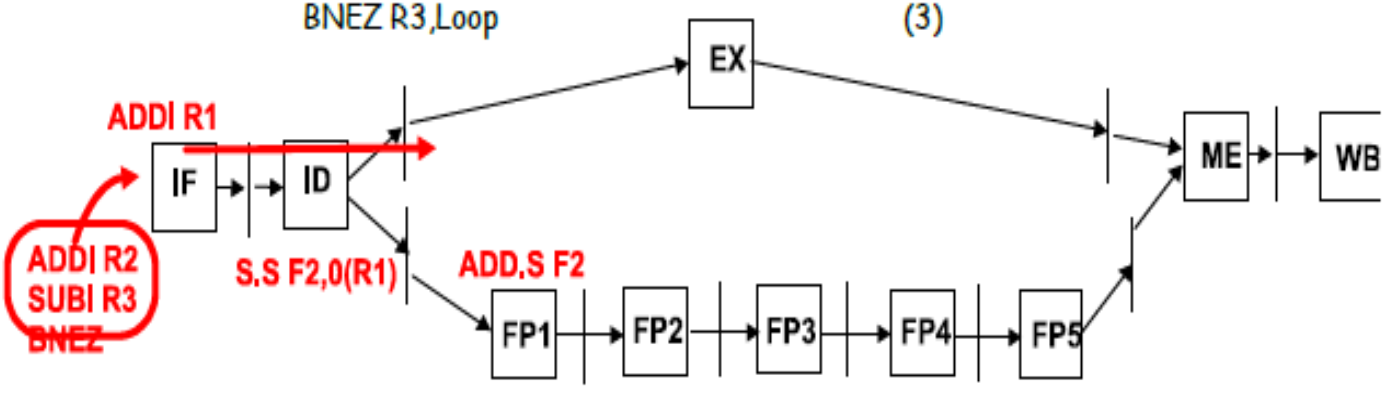
Motivation for Dynamic Instruction Scheduling

```
Loop    L.S F0,0(R1)        (1)
        L.S F1,0(R2)        (1)
        ADD.S F2,F1,F0      (2)
        S.S F2,0(R1)        (5)
        ADDI R1,R1,#4       (1)
        ADDI R2,R2,#4       (1)
        SUBI R3,R3,#1       (1)
        BNEZ R3,Loop        (3)
```



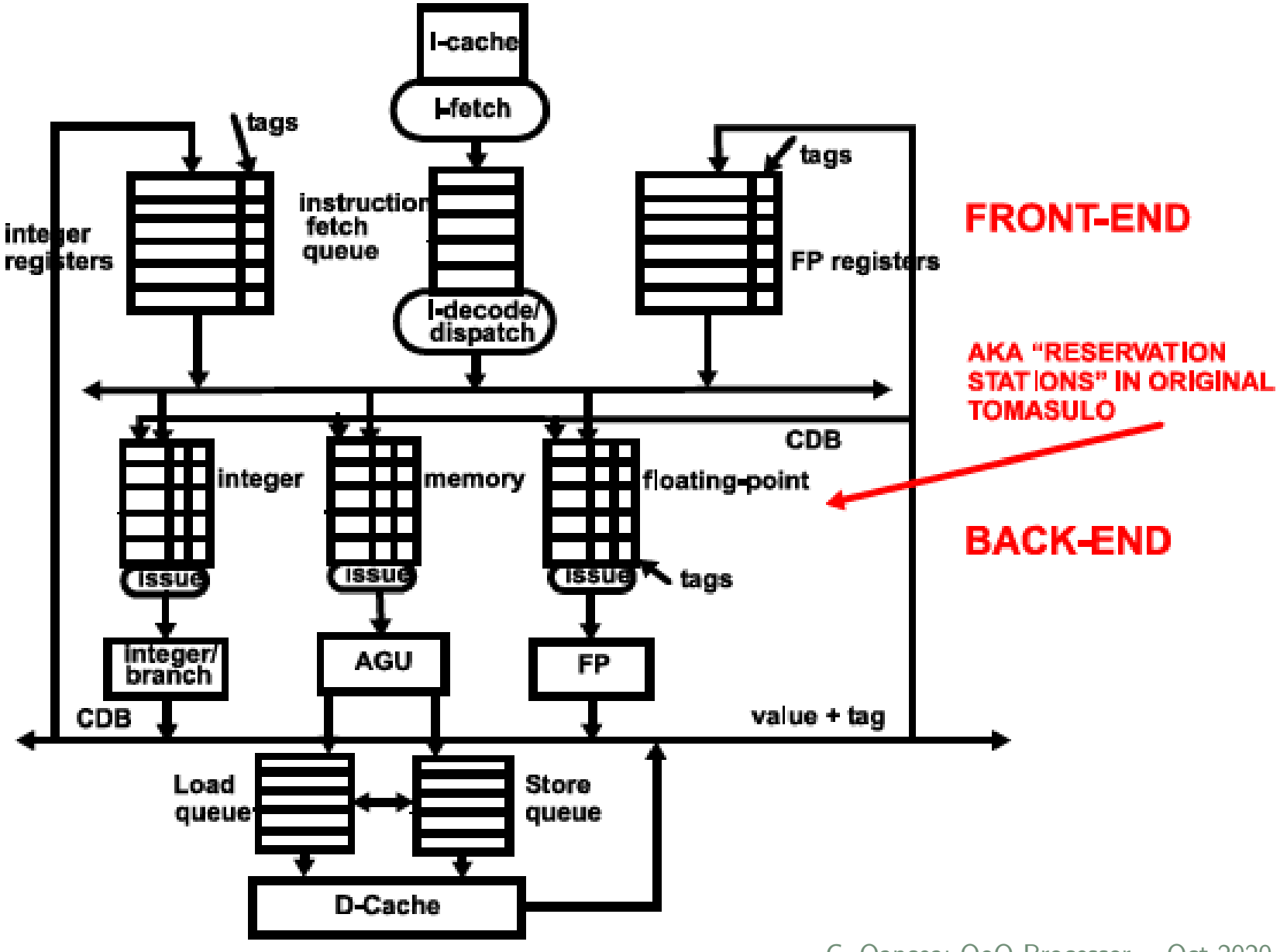
Motivation for Dynamic Instruction Scheduling

```
Loop    L.S F0,0(R1)        (1)
        L.S F1,0(R2)        (1)
        ADD.S F2,F1,F0      (2)
        S.S F2,0(R1)        (5)
        ADDI R1,R1,#4       (1)
        ADDI R2,R2,#4       (1)
        SUBI R3,R3,#1       (1)
        BNEZ R3,Loop        (3)
```



- `ADDI R1` passes through ID while the store waits (watch out for for WAR on R1)
- `ADDI R2` and `SUBI R3` fetched, decoded and bypass the store.
- By that time `ADD.S` is in FP4 and `S.S F2` stalls 1 more cycle.
- Even the branch could bypass the store! Need new Architecture!

Tomasulo Algorithm



Tomasulo Algorithm (cont)

Front-End:

- Instrs fetched and stored in *instruction fetch queue (IFQ)* (FIFO).
- When an instr reaches the top of IFQ, it is decoded and
  - dispatched to an *issue queue* (integer/branch, memory, float),
  - even if some of its input operands are not ready (being computed).

Back-End:

- Instructions in issue queues wait for their input (reg) operands
- Once operands are ready, instrs can be sceduled for execution,
- modulo conflicts for *common-data bus (CDB)* or *functional units*.
- Instrs execute in FU and results are placed on CDB.
- All instrs in queues and all registers in register files *snoop the CDG and grab the value they are waiting for*.

Tomasulo: Data Hazards on Registers

The **TAG** implements **dynamic register renaming**: register values are renamed to queue-entry index. (Multiple values for same register may be pending in back end at any time.)

At **dispatch**, **the output register operand** is assigned a **TAG**, which is the issue queue entry number where the instr is dispatched!

- TAG is stored in the register file and is reclaimed when the instr retires (writes output on CDB and releases its Q entry)
- TAG & result put on CDB and "snooped" by queues & reg. file.
- When a tag match occurs:
  - its value is stored in queue entry and in register (file) and
  - register TAG turns invalid, i.e., value NOT pending in back end.

At **dispatch**, **input register operand & TAG** fetched from reg file:

- **If tag is NOT valid** ⇒ value NOT pending in back end, hence
- register value valid and sent to the queue entry (operand ready).
- **If tag is valid** ⇒ the value is pending in back end, hence
- stale reg value ⇒ **TAG** sent to the queue entry (op NOT ready).

Tomasulo: Data Hazards on Memory

All data hazards possible on memory (RAW, WAW, WAR).
Load/Store Queues (L/S Q): staging buffers to solve mem hazards.
Load/store addresses computed by **Address-Generation Unit AGU**.

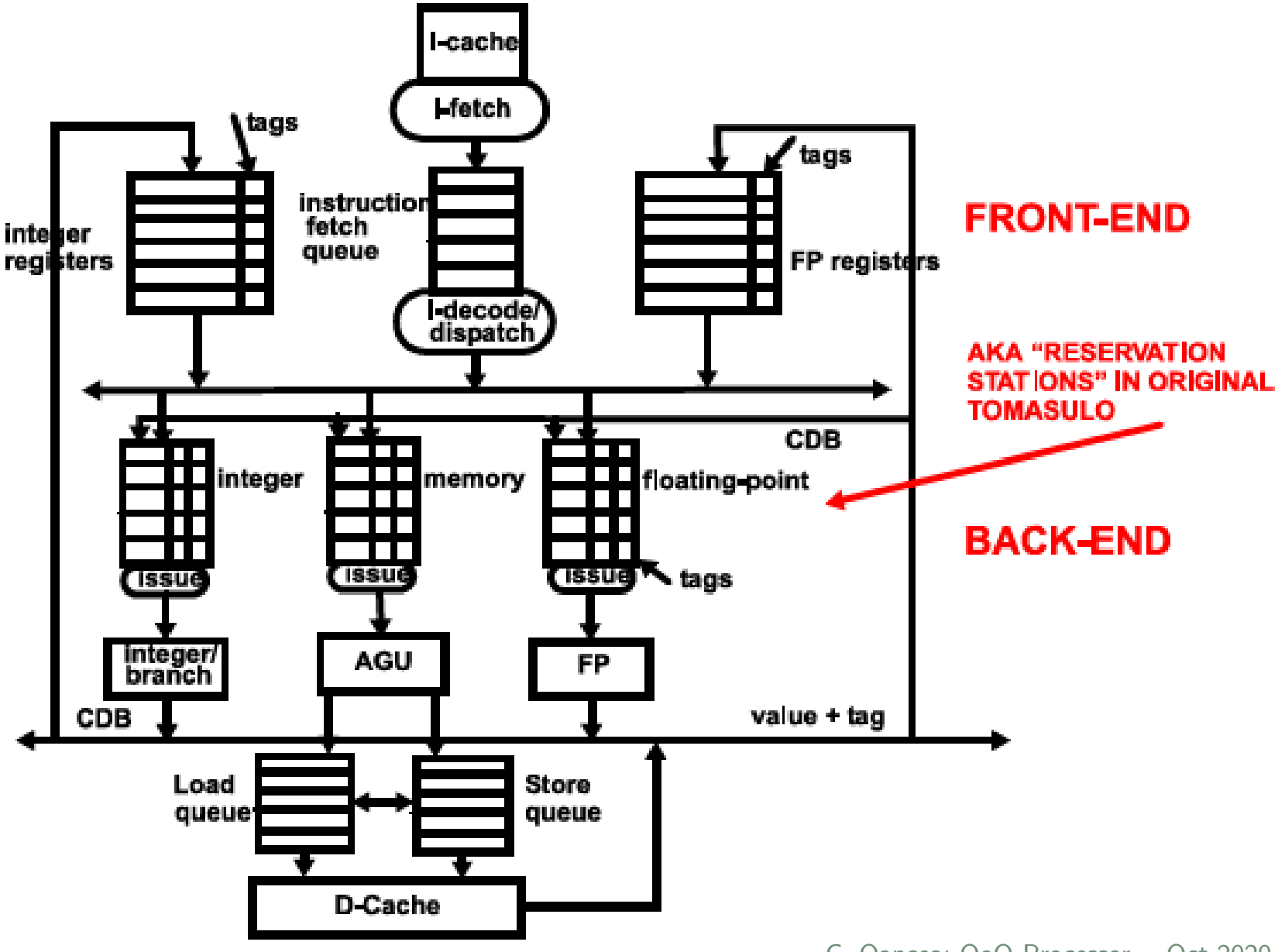
Stores are split in two sub instructions:

- one computes the address, the other waits for the data,
- both are dispatched to memory, results are latched in L/S Q,
- L/S performs memory disambiguation (solves memory hazards).

Issue to cache from memory Q:

- loads/stores can issue to AGU and L/S Q when address input operands are ready; both store sub-instrs issue to AGU & L/S Q.
- Memory hazards (RAW, WAR, WAW) are resolved in L/S Q:
  - L/S Q keeps track of the dispatch order of loads and stores,
  - L/S Q entries are reserved at dispatch.
  - Load can issue to cache if no same-address store is before it.
  - Store can issue to cache if no same-address load/store before it.
- If an address is unknown, conservatively assumes it is the same.

Tomasulo Algorithm



Tomasulo: Structural and Control Hazards

Structural Hazards:

- IF must stall if IFQ is full,
- dispatch must stall if all entries in the issue Q or L/S queue are occupied,
- instructions cannot issue in case of CDB or FU conflicts.

Control Hazards (Conditional Branches):

- dispatch stalls when it reaches a branch instruction,
- branches dispatched to integer issue Q, and treated as integer instructions
- they wait for their register operands, outcome placed on CDB:
  - **untaken** $\Rightarrow$ dispatch is resumed from the IFQ,
  - **taken** $\Rightarrow$ dispatch clears the IFQ and directs I-Fetch to fetch the target I-Stream.
- **Optimization**: front end could pre-fetch-and-decode some instrs in the target I-Stream while the branch is in the back end.

Precise Exceptions: not supported!

Discussion

WAW and WAR are also called "false" OR "name" dependencies:

- RAW dependencies are called "true dependencies",
- false or name deps are due to limited memory resources,
- Tomasulo algorithm solves false dependencies on memory and register operands by dispatching store instructions in order and by renaming registers to issue Q entry numbers.

```
I1   L.S   F0,  0(R1)
I2   ADD.S F1, F1, F0
I3   L.S   F0,  0(R2)
```

- **I3 may finish before I1, if I1 misses and I3 hits in cache**,

Discussion

WAW and WAR are also called "false" OR "name" dependencies:

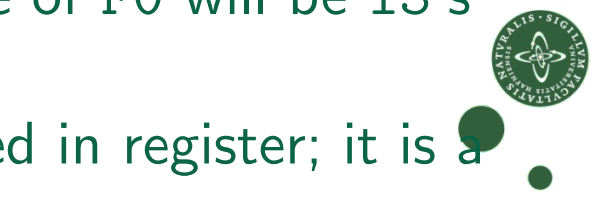
- RAW dependencies are called "true dependencies",
- false or name deps are due to limited memory resources,
- Tomasulo algorithm solves false dependencies on memory and register operands by dispatching store instructions in order and by renaming registers to issue Q entry numbers.

```
I1   L.S   F0,  0(R1)
I2   ADD.S F1, F1, F0
I3   L.S   F0,  0(R2)
```

- **I3 may finish before I1, if I1 misses and I3 hits in cache**,
- BUT I2 waits on the tag of I1, i.e., not on F0 or tag of I3,
- Meaning I2 waits for the value of F0 from I1 $\Rightarrow$ WAR hazard on F0 is solved.

The TAG of F0 in register file is set to I1's TAG upon I1's dispatch, THEN to I3's TAG upon I3 dispatch,

- even if I3 completes before I1, the final value of F0 will be I3's value $\Rightarrow$ WAW hazard on F0 is solved.
- The value of F0 produced by I1 is never stored in register; it is a fleeting value consumed by I2!



Tomasulo Execution Example

		Dispatch	Issue	Exec/ start	Exec/ complete	Cache	CDB	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	
I5	S.S-D F2,0(R1)	5	14	(15)	15	(16)	--	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	wait for R3
I10	L.S F0,0(R1)	15	16	(17)	17	(18)	(19)	wait for I9 (in dispatch)
I11	L.S F1,0(R2)	16	17	(18)	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	(22)	26	--	(27)	wait for F1

- Exec of integer & branch instr: 1 cycle, FP pipelined 5 cycles,
- AGU and cache hit: 1 cycle each, **Dispatch: 1 cycle complex**,
- Issuing (or scheduling) also takes 1 cycle, CDB result: 1 cycle.
- **Large overhead to manage instrs ⇒ latency effectively increased**

Tomasulo Execution Example

		Dispatch	Issue	Exec/ start	Exec/ complete	Cache	CDB	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	
I5	S.S-D F2,0(R1)	5	14	(15)	15	(16)	--	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	wait for R3
I10	L.S F0,0(R1)	15	16	(17)	17	(18)	(19)	wait for I9 (in dispatch)
I11	L.S F1,0(R2)	16	17	(18)	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	(22)	26	--	(27)	wait for F1

- Exec of integer & branch instr: 1 cycle, FP pipelined 5 cycles,
- AGU and cache hit: 1 cycle each, **Dispatch: 1 cycle complex**,
- Issuing (or scheduling) also takes 1 cycle, CDB result: 1 cycle.
- **Large overhead to manage instrs ⇒ latency effectively increased**, e.g., `BNEZ` waits three cycles for `SUBI` to complete!

Tomasulo Execution Example

		Dispatch	Issue	Exec/ start	Exec/ complete	Cache	CDB	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	
I5	S.S-D F2,0(R1)	5	14	(15)	15	(16)	--	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	wait for R3
I10	L.S F0,0(R1)	15	16	(17)	17	(18)	(19)	wait for I9 (in dispatch)
I11	L.S F1,0(R2)	16	17	(18)	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	(22)	26	--	(27)	wait for F1

- **Instructions issued and start execution OUT OF ORDER!**

  Critical path

Tomasulo Execution Example

		Dispatch	Issue	Exec/ start	Exec/ complete	Cache	CDB	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	
I5	S.S-D F2,0(R1)	5	14	(15)	15	(16)	--	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	wait for R3
I10	L.S F0,0(R1)	15	16	(17)	17	(18)	(19)	wait for I9 (in dispatch)
I11	L.S F1,0(R2)	16	17	(18)	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	(22)	26	--	(27)	wait for F1

- **Instructions issued and start execution OUT OF ORDER!**
  Critical path comprises the loads followed by `ADD.S` followed by store; the other instrs executed in parallel with critical path!
- `SUBI`→`BNEZ` **could take advantage of static scheduling**

Tomasulo Execution Example

		Dispatch	Issue	Exec/ start	Exec/ complete	Cache	CDB	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	
I5	S.S-D F2,0(R1)	5	14	(15)	15	(16)	--	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	wait for R3
I10	L.S F0,0(R1)	15	16	(17)	17	(18)	(19)	wait for I9 (in dispatch)
I11	L.S F1,0(R2)	16	17	(18)	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	(22)	26	--	(27)	wait for F1

- **Instructions issued and start execution OUT OF ORDER!**
  Critical path comprises the loads followed by `ADD.S` followed by store; the other instrs executed in parallel with critical path!
- `SUBI`→`BNEZ` **could take advantage of static scheduling**, e.g., if `SUBI` moved up then branch result known earlier!
- **Branches still act as a barrier to parallelism!**

1 Tomasulo Algorithm

2 Dynamic Branch Prediction

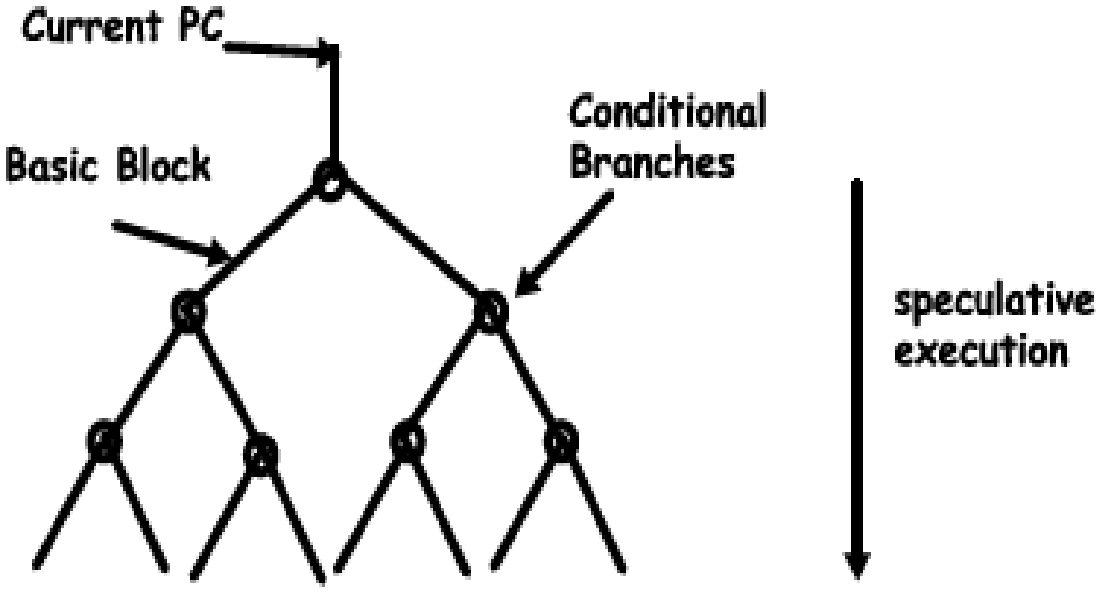
3 Speculative Execution

4 Multiple Instructions Per Clock

5 Pentium III and Pentium IV

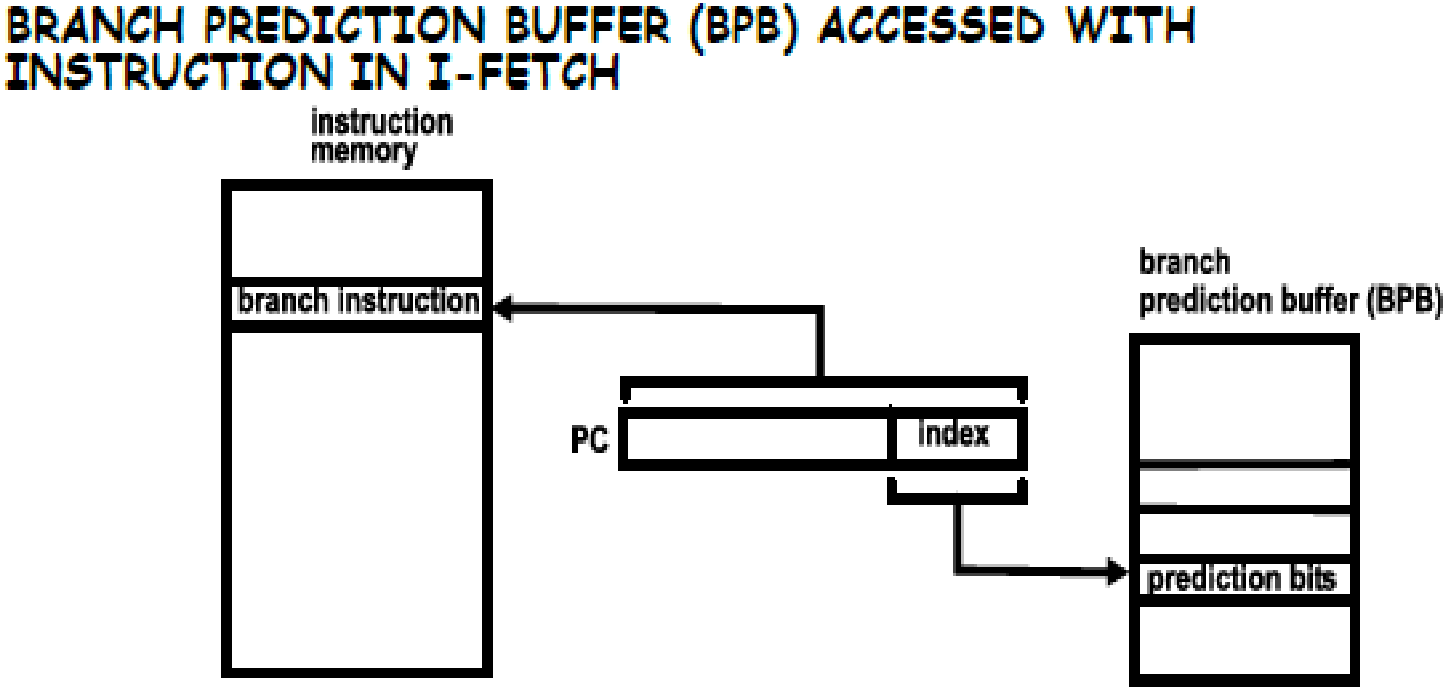
6 OoO Processors: Memory Consistency Models (MCM)

Execution Beyond Unresolved Branches: A Tree of Possibilities

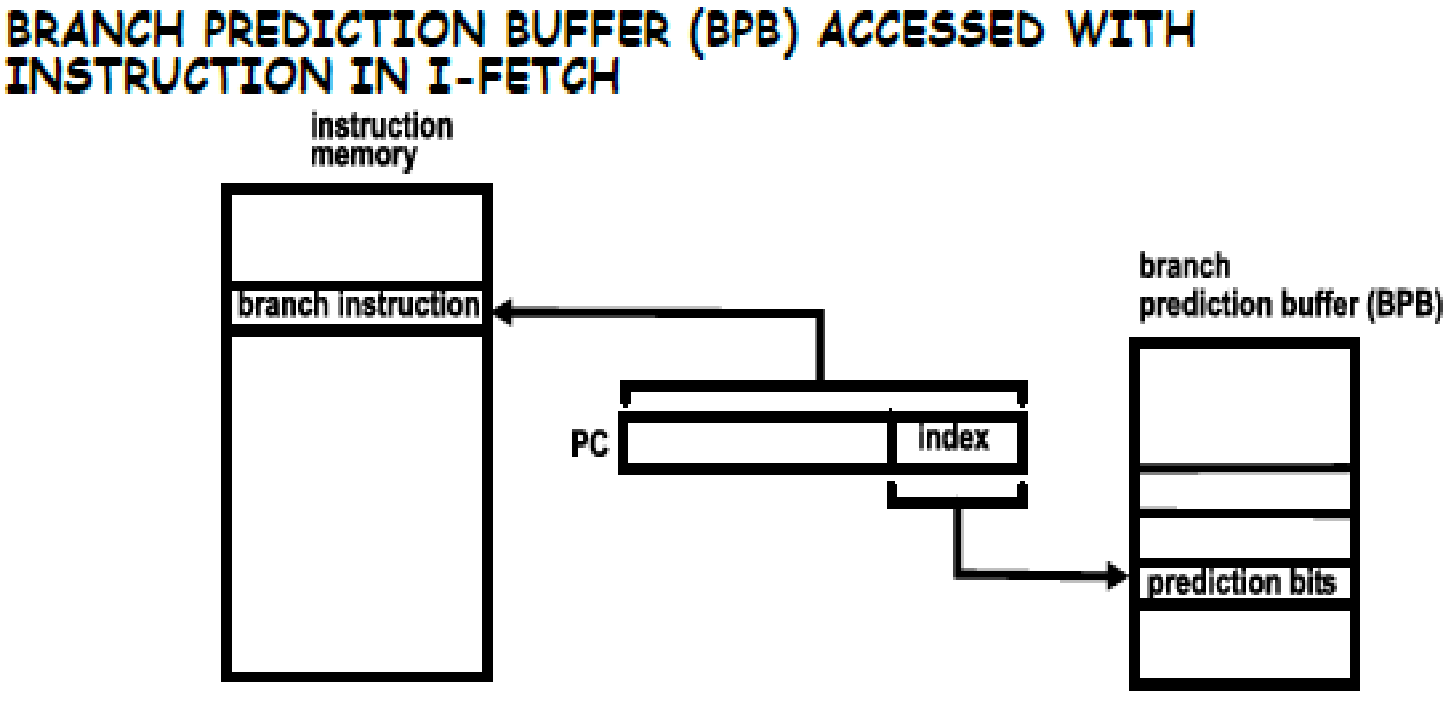


- **All-Path Execution** then cancel all but one path!
  - very hardware intensive, unwanted exceptions
  - hard to keen track of order of instructions in a tree.
- **Predict branches and execute the most likely path** requires a rollback mechanism in case prediction is wrong.
- **Multi-Path Execution**: follow most likely path, and follow both paths if branch is not predictable.

Dynamic Branch Prediction



Dynamic Branch Prediction



- Small memory indexed with LSBs of PC in I-Fetch
- Prediction is dropped if not a branch
- Otherwise, prediction bits are decoded into T/F prediction
- When branch condition known, if incorrect prediction $\Rightarrow$ rollback.
- Update prediction bits.
- Aliasing in BPB: different branches affect each other predictions.

1-Bit Predictor

- **Each BPB entry is 1 bit:**
  - bit records the last outcome of a branch
  - next outcome predicted as the last one

- In the context of loops:

```
Loop1: ...
       ...
       Loop2: ...
              ...
              BEZ R2, Loop2
       ...
       BNEZ R3, Loop1
```

- BEZ **mispredicted twice for every inner loop execution:**
  - once on entry and once on exit
  - the mispredict on exit is unavoidable, but
  - the mispredict on entry could be avoided.

1-Bit Predictor

- **Each BPB entry is 1 bit:**
  - bit records the last outcome of a branch
  - next outcome predicted as the last one

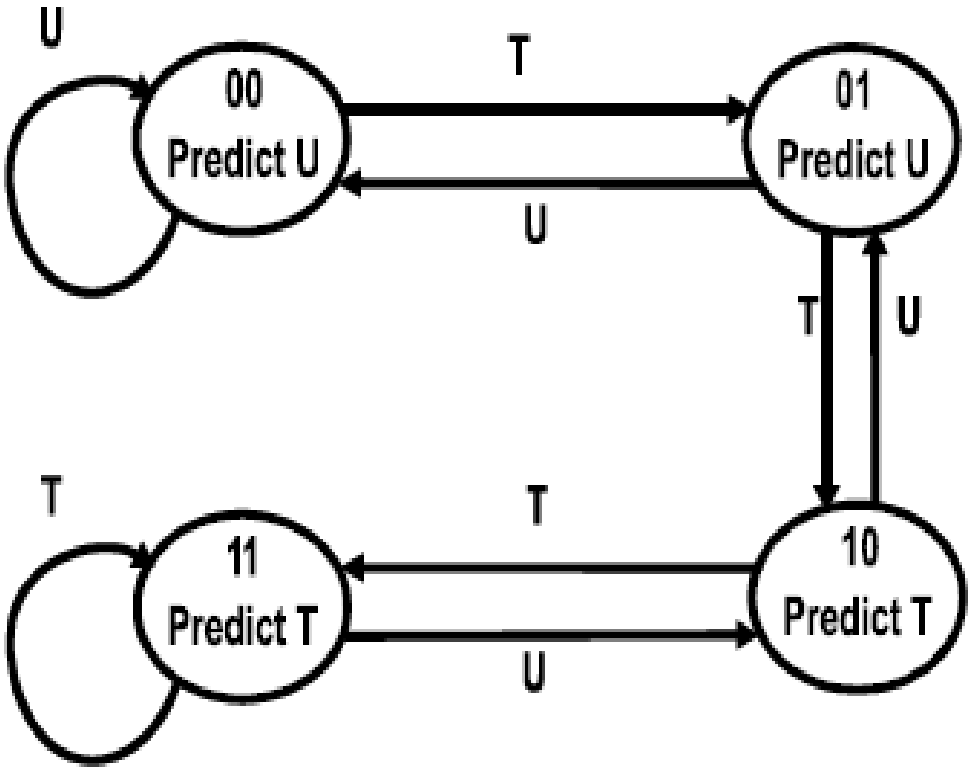
- In the context of loops:

```
Loop1: ...
       ...
       Loop2: ...
              ...
              BEZ R2, Loop2
       ...
       BNEZ R3, Loop1
```

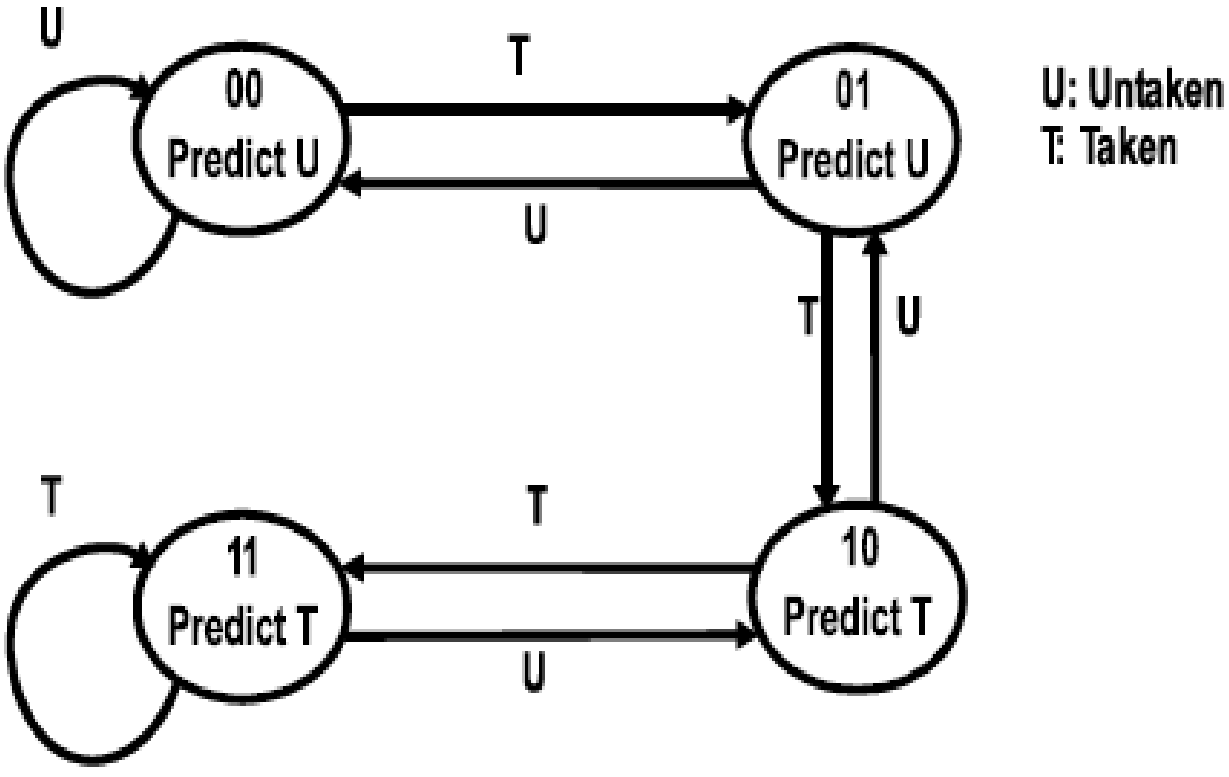
- `BEZ` **mispredicted twice for every inner loop execution:**
  - once on entry and once on exit
  - the mispredict on exit is unavoidable, but
  - the mispredict on entry could be avoided.

- **Use a 2-bit predictor**

2-Bit Predictor



2-Bit Predictor



- **Each BPB entry: 2-bit up-down saturating counter**
- **TAKEN ⇒ ADD 1; UNTAKEN ⇒ SUBTRACT 1**
  - changing prediction requires 2 mispredictions in a row,
  - for the nested loop, the misprediction at entry is avoided.
- **Two bits cover most patterns** (could have more than 2).

Correlating Branch Predictors

- **Branches other than loop branches motivate improving branch predictors:**

```
if (a==2) then a := 0;
if (b==2) then b := 0;
if (a!=b) then ...
```

  If the first two conditions succeed then the third will fail
  - meaning the 3rd branch is correlated with the first two
  - previous predictors cannot get this because they keep track of one branch history only.
- **In general a branch may behave differently if it is reached via different code sequences**
  - e.g., the code sequence can be characterized by the outcome of the latest branch to execute.

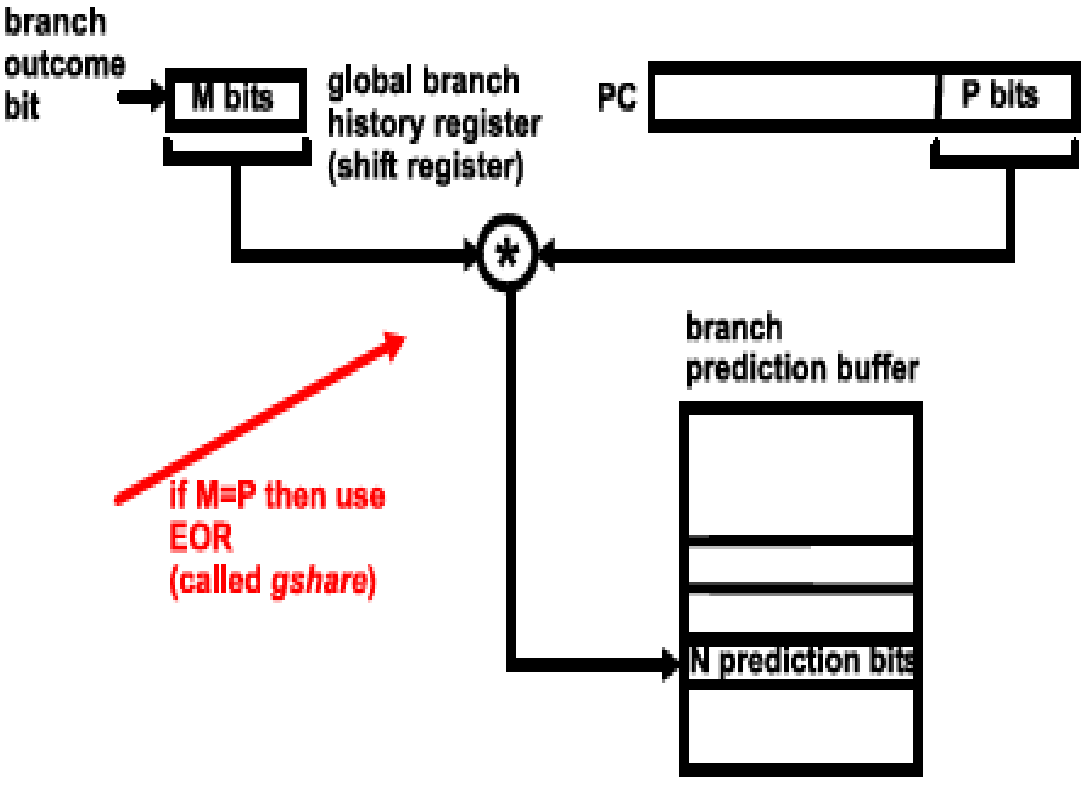
Correlating Branch Predictors

- **Branches other than loop branches motivate improving branch predictors:**

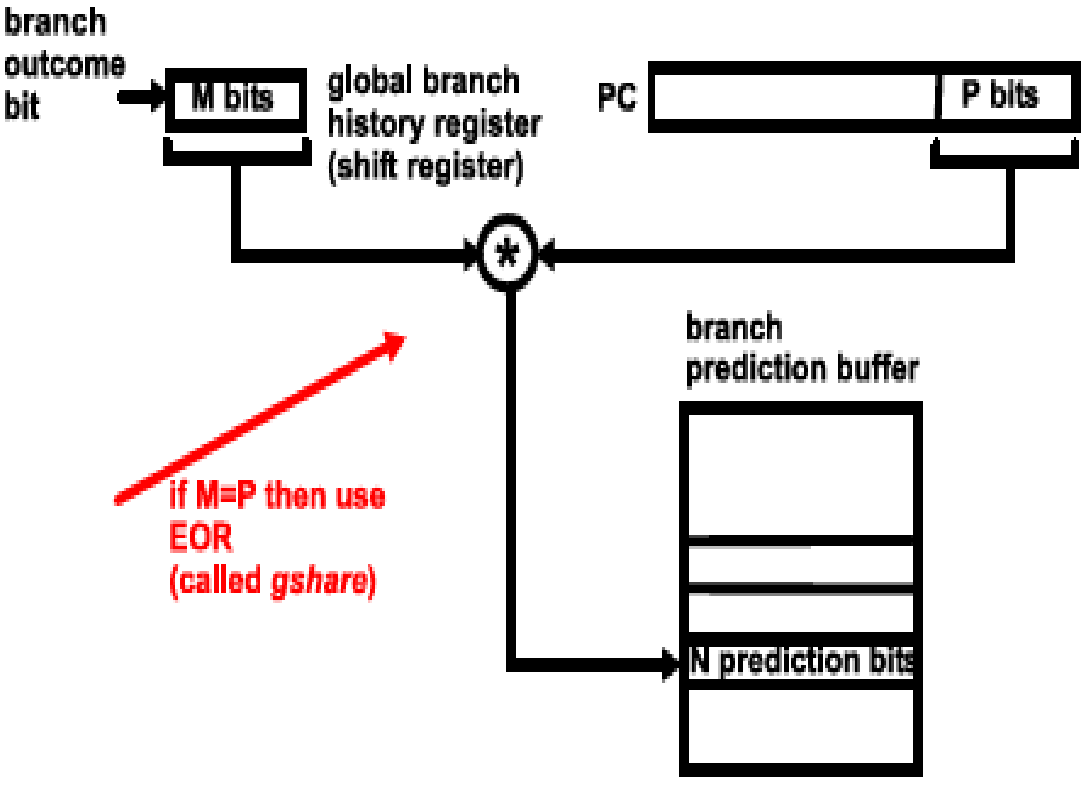
```
if (a==2) then a := 0;
if (b==2) then b := 0;
if (a!=b) then ...
```

  If the first two conditions succeed then the third will fail
  - meaning the 3rd branch is correlated with the first two
  - previous predictors cannot get this because they keep track of one branch history only.
- **In general a branch may behave differently if it is reached via different code sequences**
  - e.g., the code sequence can be characterized by the outcome of the latest branch to execute.
- **Can use `N` prediction bits, and the outcome of the last `M` branches to execute**:
  - global *vs* local history
  - the branch itself may be part of global history

(M,N) BPB

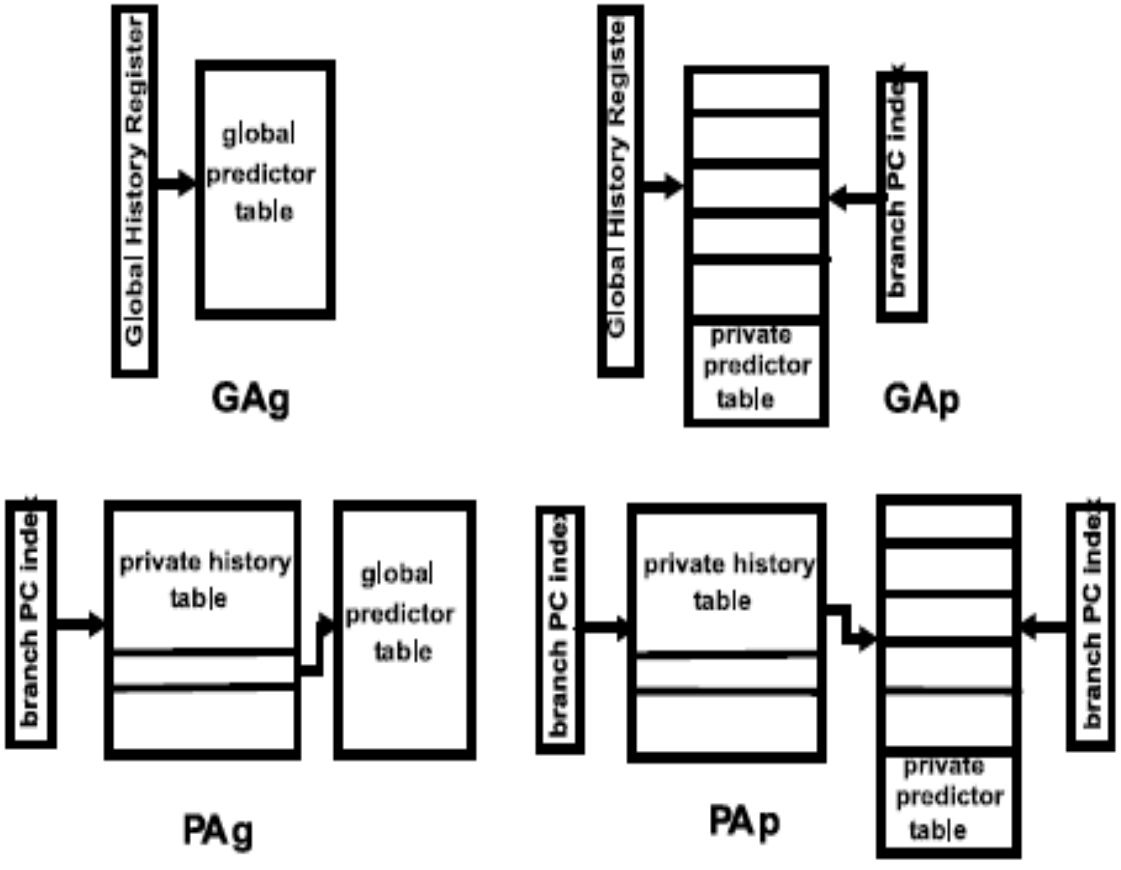

branch
outcome
bit
M bits
global branch
history register
(shift register)
PC
P bits
*
branch
prediction buffer
if M=P then use
EOR
(called gshare)
N prediction bits

(M,N) BPB

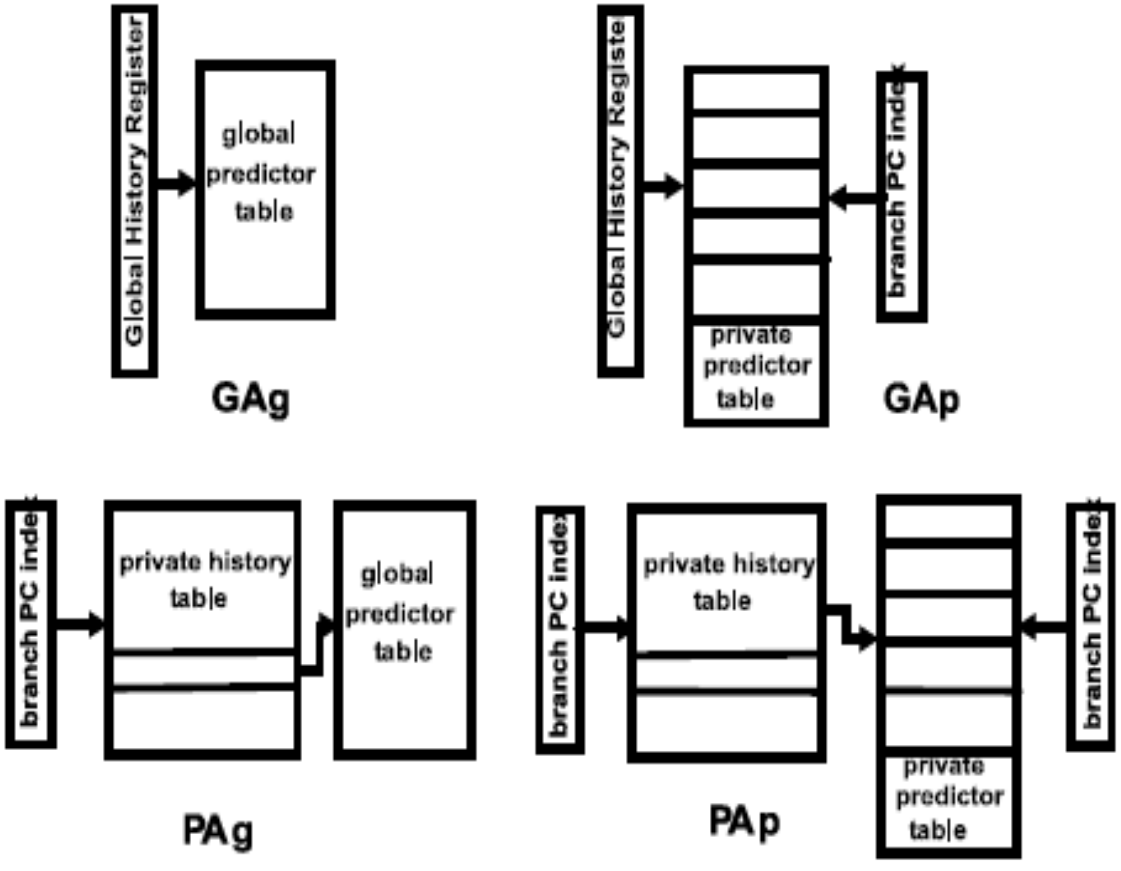


- **Global History**: the outcome of the last M (executed) branches,
- i.e., use 2^M N-bit predictors for each of P-bit branch entry.
- BPB size: $N \times 2^M \times 2^P$, EOR = bitwise exclusive OR.
- **This predictor uses global history to differentiate between various behaviors of a particular branch**
- **Can be generalized: two level predictors (next)!**

Two-Level Predictor



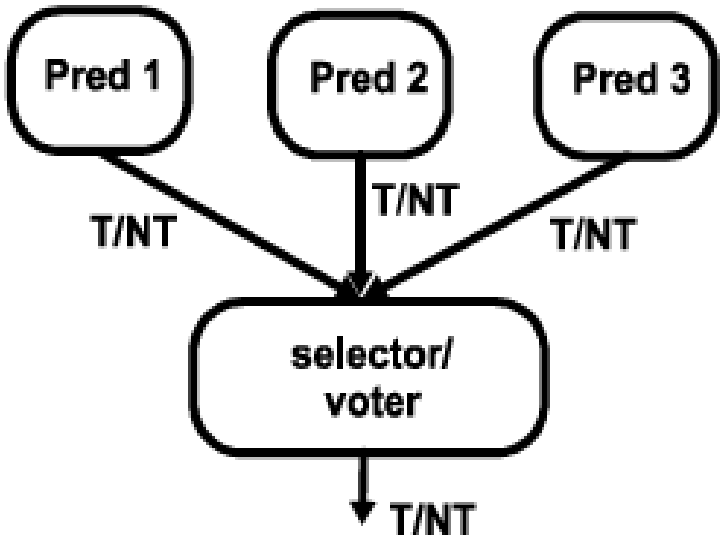
Two-Level Predictor



- history: **global** (all branches), or **private** (for each branch)
- First letter → type of history; last letter → whether each predictor is private in the table.
- `G` or `g` means global; `P` or `p` means private.

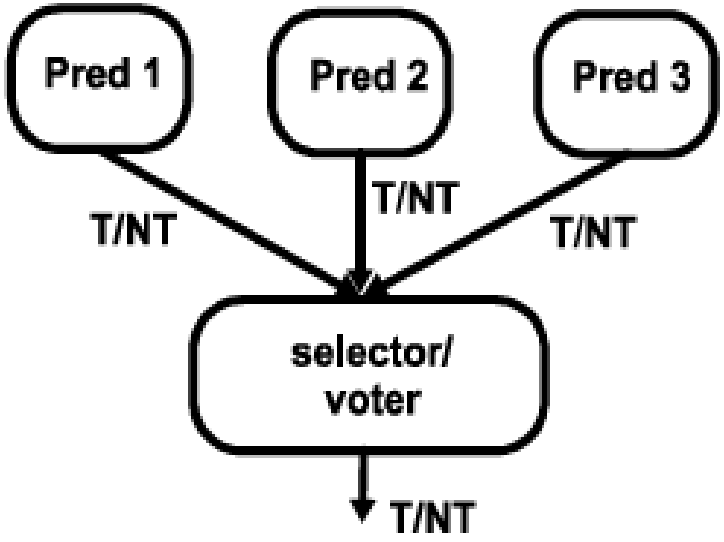
Combining Predictors

- The branches in different phases of the program may be predicted better with different predictors.
- Or if two predictors agree ⇒ then they are probably right.



Combining Predictors

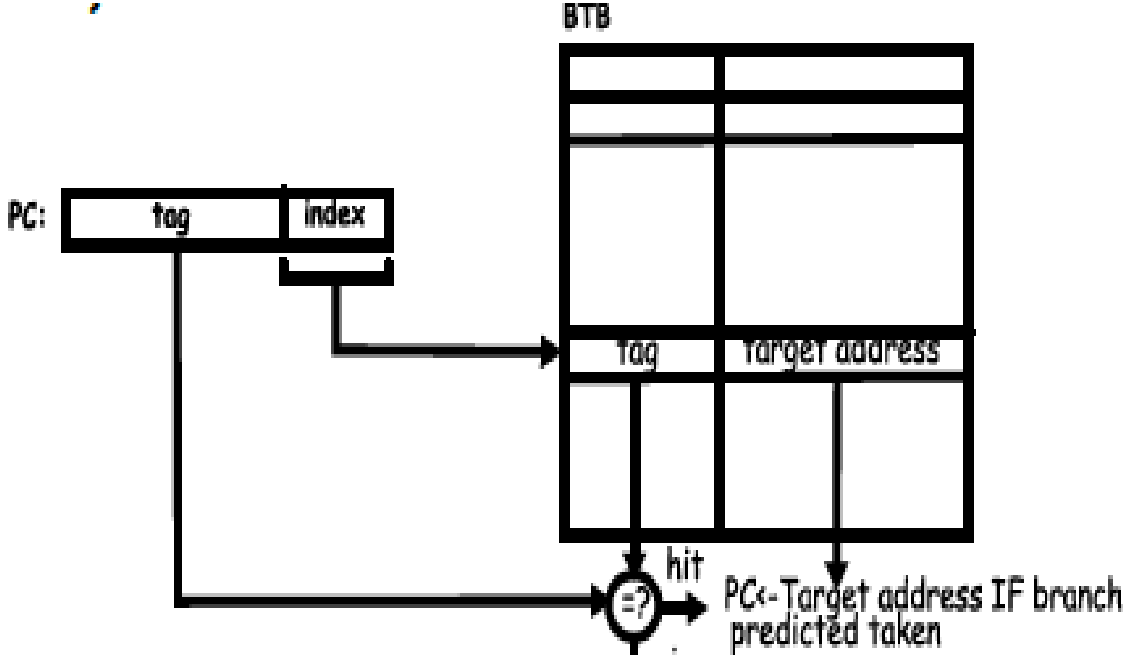
- The branches in different phases of the program may be predicted better with different predictors.
- Or if two predictors agree ⇒ then they are probably right.



- A selector keeps the track record of each predictor (globally or for each branch). Can be done with 1 or 2 bits.
- A voter takes a majority vote of the three predictors.

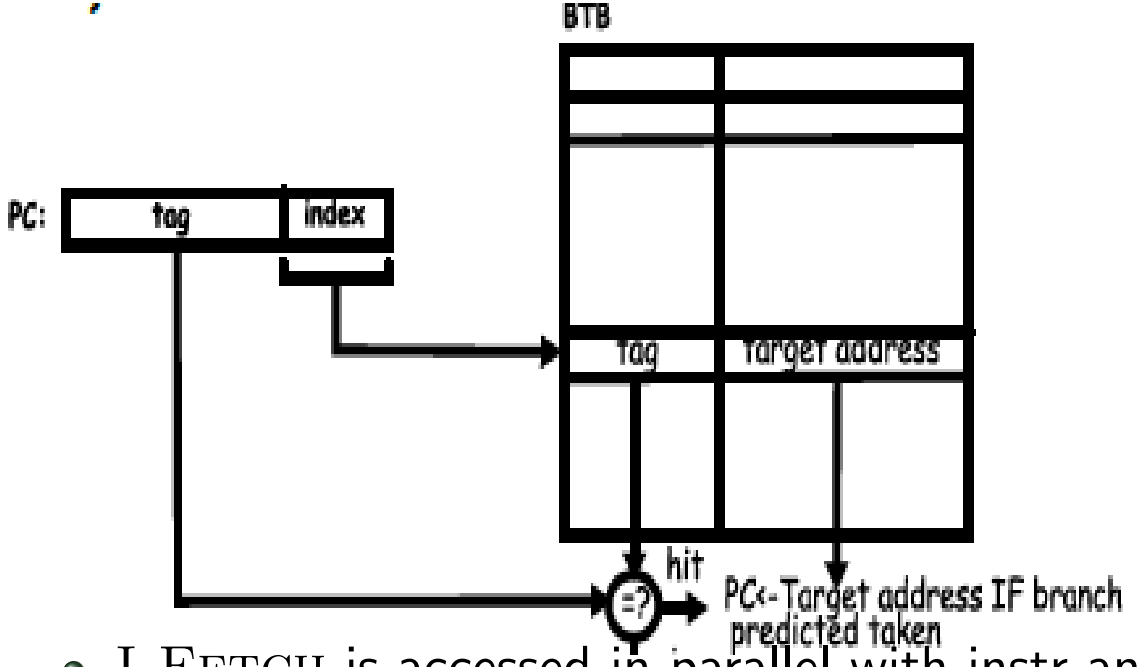
Branch Target Buffer (BTB)

- Eliminating the branch penalty: need to know the target address by the end of I-Fetch!
- BTB: cache for all branch target addresses (no aliasing)



Branch Target Buffer (BTB)

- Eliminating the branch penalty: need to know the target address by the end of I-Fetch!
- BTB: cache for all branch target addresses (no aliasing)



  - I-Fetch is accessed in parallel with instr and BPB entry
  - relies on target address of a branch never changing.

- **Indirect Jump**, e.g., caused by procedure returns.
  Use a stack to track the return addresses of procedure calls.

1 Tomasulo Algorithm

2 Dynamic Branch Prediction

3 Speculative Execution

4 Multiple Instructions Per Clock

5 Pentium III and Pentium IV

6 OoO Processors: Memory Consistency Models (MCM)

Hardware-Supported Speculation

Combines Three Main Ideas:

- Dynamic OoO instruction scheduling (Tomasulo algorithm),
- Dynamic branch prediction allows inst sched across branches,
- Speculative execution: execute instrs before all control dependencies are resolved.

Hardware-based speculation uses a data-flow approach:

- instrs execute when ops are available, across predicted branches:
- requires to separate the completion of instr execution and the commit of its result
  - results are speculative between completion and commit,
  - commit results to registers and storage in process order.

Required extensions to Tomasulo Algorithm:

- branch prediction,
- temporary storage of speculative results,
- mechanism to rollback execution when speculation fails.

Also Need to Support Precise Exceptions!

Tomasulo With Speculative Execution

New Structures:

- Reorder Buffer (ROB),
- Branch Prediction Buffer (BPB),
- Branch Target Buffer (BTB).

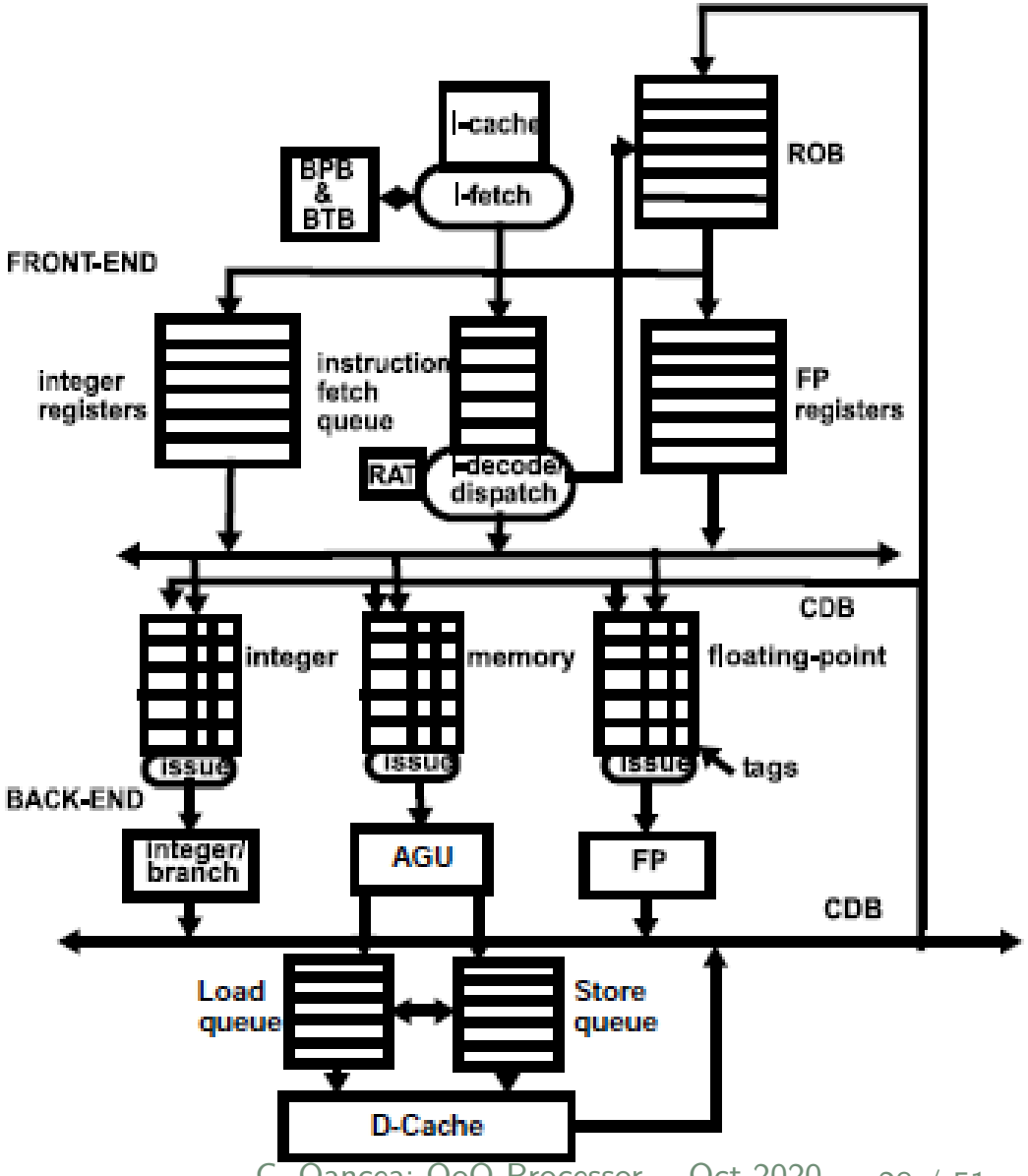
Reorder Buffer (ROB):

- keeps track of process order (FIFO),
- holds speculative results,
- no more register-file snooping.

Register Values Can Be:

- pending in back end, or
- speculative in ROB, or
- committed in register file.

ROB entry numbers are as TAGs for register renaming!



Speculative Tomasulo Semantics (Steps)

Normal operation, i.e., no mispredictions, no exceptions:

1. **I-Fetch:**
   - fetch instruction, predict branches and their targets,
   - fill IFQ with instrs following the branch prediction.
2. **I-Decode/Dispatch:**
   - decode opcode, allocate 1 issue-queue entry & 1 ROB entry & 1 L/S entry for load/store instructions.
   - rename destination registers (TAG), with ROB entry #, mark "pending" in register alias table (RAT).
   - split store in two instructions (address + cache).
   - fill input register operand fields:
     - if value marked "pending" in RAT $\Rightarrow$ fill with TAG (not ready)
     - if marked "completed" in RAT $\Rightarrow$ fetch value from ROB (ready)
     - if marked "committed" in RAT $\Rightarrow$ fetch value from register file.
   - stall while ROB or issue Q or L/S Q (if load/store) are full!
3. **Issue:**
   - wait in issue queue until all inputs are ready (snoop the CDB)
   - issue in a cycle when conflicts for FU and CDB can be avoided.

Speculative Tomasulo Semantics (Steps)

4 **Complete Execution and Write Result**
- result written to the ROB via the CDB,
- destination register is marked "completed" in the RAT.

5 **Commit (Graduate or Retire):**
- wait to reach the head of ROB, then
- write result to the register or memory (in store case).

- **In Case Of Branch Misprediction:**
  - all instrs following the branch in the ROB must be flushed:
  - wait until the branch reaches top of ROB and flush the ROB together with all instrs in the back end.
  - Finally, start fetching instrs from the correct target!
- **Exceptions:**
  - are flagged in ROB, but remain "silent" until the instr is ready to retire at the top of ROB,
  - the entire ROB must then be flushed,
  - share the hardware dedicated to mispredicted branch recovery.
  - Finally, the exception handler is fetched and executed.

Solving Memory Hazards

- **Load/Stores Use the Same Approach as in Tomasulo**
  - issue to L/S Q through AGU,
  - stores are split into two instructions: one computing the address, one propagating the value,
  - each subinstruction is allocated 1 issue Q entry,
  - only one ROB entry associated with the data part of the store.
- **All WAW and WAR Hazards Automatically Solved:**
  - because stores update the cache in process order
  - when they reach the top of the reorder buffer!
- **Check RAW Hazards in L/S Q before Load Issues to Cache**
  - a load can issue to cache as soon as it reaches the L/S Q, however
  - if a store with the same address is in front of the load in L/S Q:
    - wait until the store reaches the cache (at top of ROB), or
    - return the value of the store when it is ready (may affect memory consistency model).
- **Big Problem is what to do when there are stores in front of the load with unknown address (address not ready)?**

Dynamic Memory Disambiguation

- **Figuring out if two addresses are equal to avoid hazards**
- **Conservative Approach:**
  - a ready load must wait in L/S Q until addresses of all stores preceding it in the L/S Q are known. Unfortunate, because the case when such a dependence exists is RARE!
- **Optimistic Approach: Speculative Disambiguation**
  - Use the mechanism for speculative execution (ROB+rollback).
  - If a load is ready & unknown-address stores are in front in L/S Q:
    - Speculatively assume that the store/load addresses are different,
    - issue load to cache, and later, when the store address is ready, check all following loads in L/S Q.
    - If a same-address load is found & already completed then the load and all following instructions must be replayed;
    - this is accomplished by rolling back the execution from ROB and I-Cache, as if the load was a mispredicted branch!
- **Intermediate Approach:**
  - keep track the loads that have violated in the past, and
  - treat those loads conservatively, and all others optimistically!

Speculative Tomasulo Execution Example

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	6	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	7	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	14	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	--	
I5	S.S-D F2,0(R1)	5	14	(15)	(15)	(16)	--	17	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	18	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	19	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	20	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	21	wait for R3
I10	L.S F0,0(R1)	10	12	(13)	13	(14)	(15)	22	CDB conflict with I9
I11	L.S F1,0(R2)	11	13	(14)	14	(15)	(16)	23	issue conflict with I10
I12	ADD.S F2,F1,F0	12	17	(18)	22	--	(23)	24	wait for F1

- **Loads can be dispatched right after the branch.**
- **Store to cache must wait until it reaches the top of ROB.**

1 Tomasulo Algorithm

2 Dynamic Branch Prediction

3 Speculative Execution

4 Multiple Instructions Per Clock

5 Pentium III and Pentium IV

6 OoO Processors: Memory Consistency Models (MCM)

Dispatching Multiple Instructions Per Clock

- **Goal: more ILP such that** `IPC > 1`.
- **Assume a 16-way OoO processor.**
- **Problems with I-Fetch:** must fill IFQ at same rate
  - 16 instrs contain 2-3 branches,

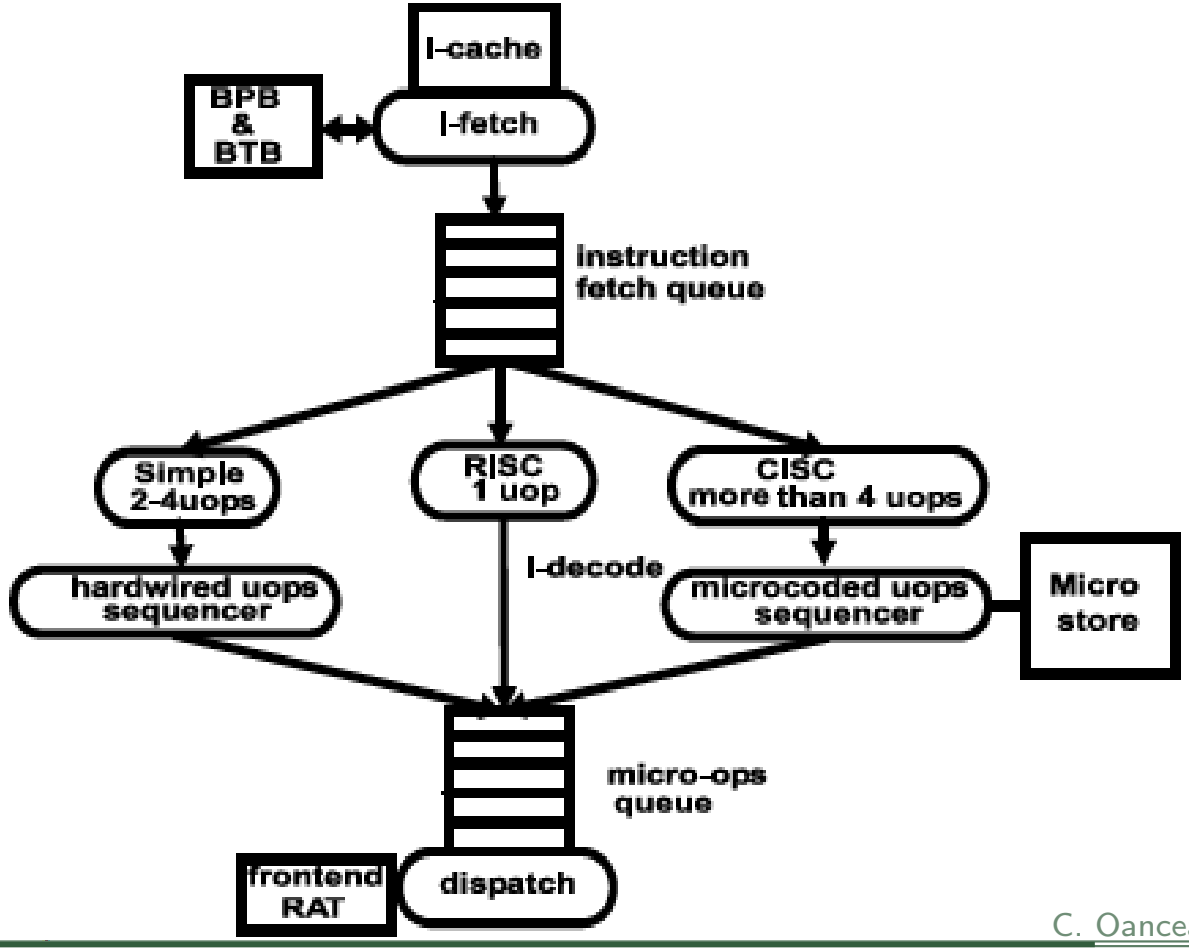
Dispatching Multiple Instructions Per Clock

- **Goal: more ILP such that `IPC > 1`.**
- **Assume a 16-way OoO processor.**
- **Problems with I-Fetch:** must fill IFQ at same rate
  - 16 instrs contain 2-3 branches,
  - branches must be predicted and code must be fetched piecewise from I-Cache in 1 clock.
  - use trace cache (maintains dynamic traces of decoded instrs).
- **Problems with dispatch:**
  - must rename many instructions in one cycle
    - inherently sequential process, must be done in thread order!
  - front-end RAT must be updated consistently with dispatching instrs one by one, and must have enough bandwidth:
    - up to 16 physical registers must be allocated in one clock,
    - up to 32 architectural input registers must be mapped.
- **Problems with back-end:** ensure no bottlenecks
  - enough FUs, enough CDB bandwidth,
  - enough register file bandwidth to store values,
  - enough retirement bandwidth.

Complex ISAs

So far RISC; for complex ISAs use microcode:

- identify a core set of RISC instructions (common case),
- make them the micro-code (don't slow them down),
- translate complex instructions to micro-ops on the fly:



1 Tomasulo Algorithm

2 Dynamic Branch Prediction

3 Speculative Execution

4 Multiple Instructions Per Clock

5 Pentium III and Pentium IV

6 OoO Processors: Memory Consistency Models (MCM)

Intel Pentium III

Several processors based on the same architecture:

- Dynamically scheduled, RISC (load/store) core,
- Complex instrs translated on the fly into RISC-core instrs,
- If instr has > 4 micro-ops then implem by a microcoded seq,
- Maximal number of micro-ops is 6 per clock,
- Micro-ops issued and executed OoO + speculation in 14 stages.

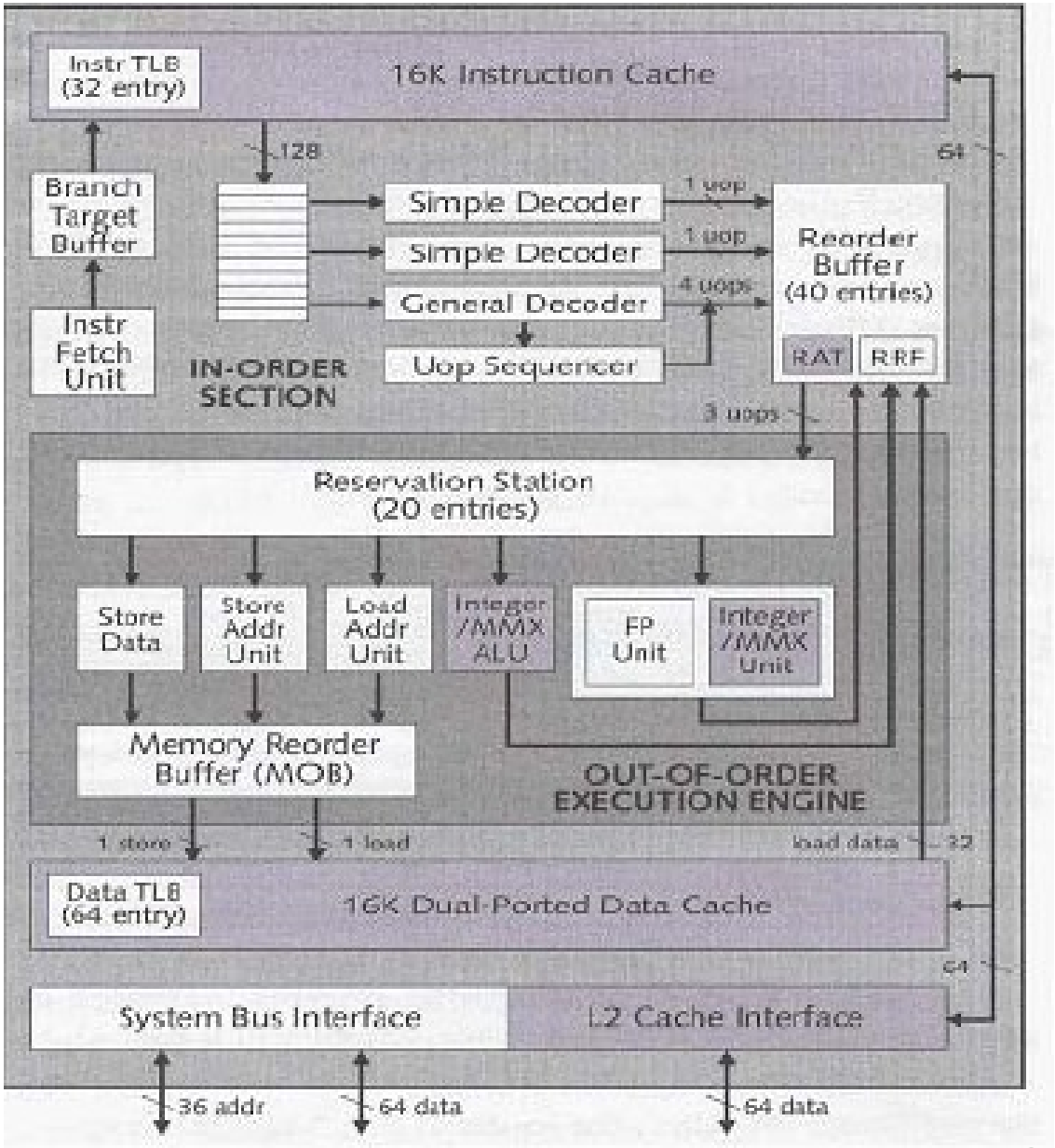
Processor	ship	clock	L1 cache	L2 cache
Pentium Pro	1995	100-200MHz	8K/8K	256-1024KB
Pentium II	1998	233-450MHz	16K/16K	256-512KB
Pentium II Xeon	1999	400-450MHz	16K/16K	512-2048KB
Celeron	1999	500-900MHz	16K/16K	128KB
Pentium III	1999	450-1100MHz	16K/16K	256-512KB
Pentium III Xeon	2000	700-900MHz	16K/16K	1024-2048KB

Intel Pentium III

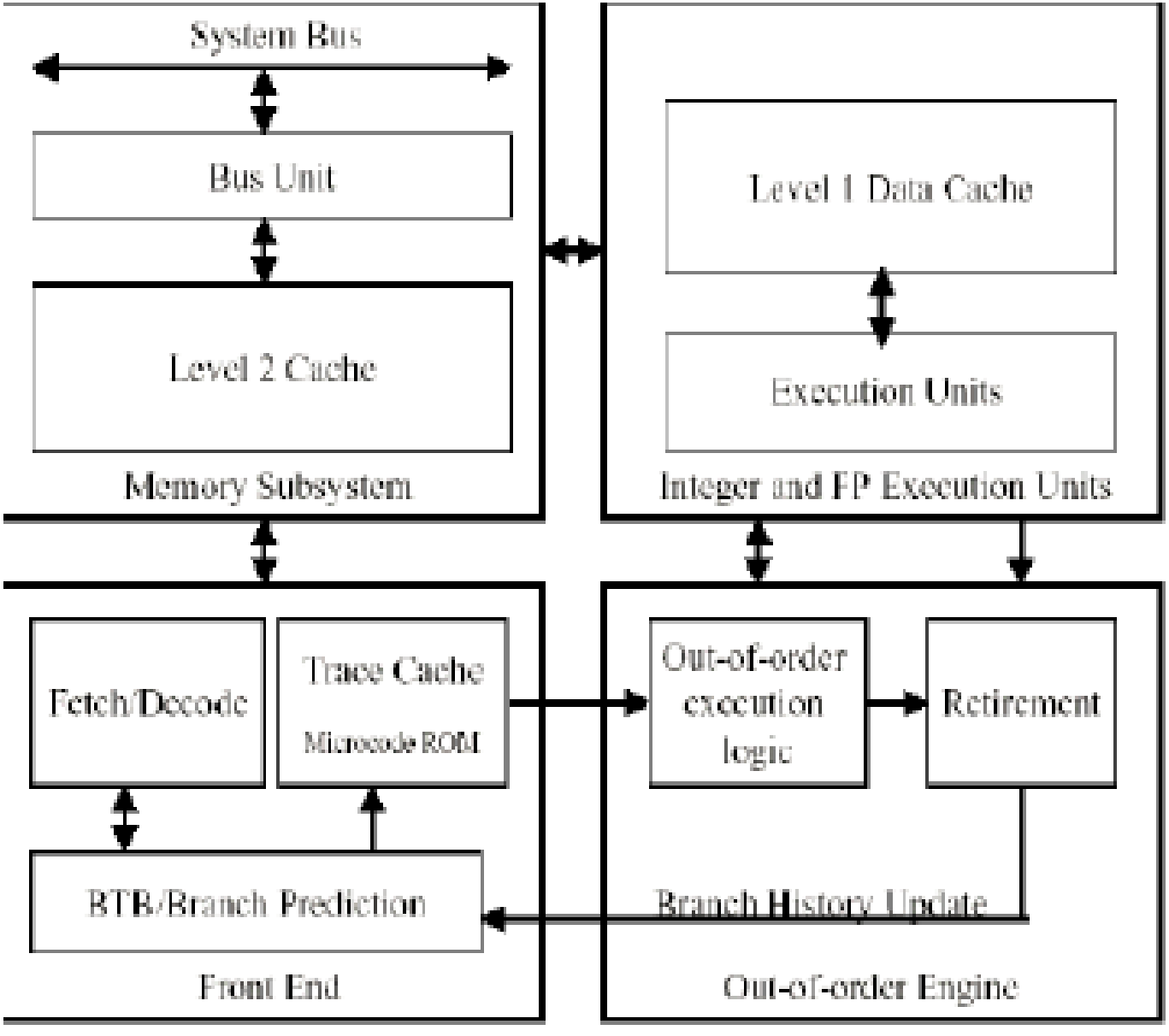
- 3-stages for execution, 8 stages for fetch, decode and dispatch, (includes 512-entry 2-level branch predictor).
- 40 virtual registers and 20 reservation stations (shared).
- Latencies and repeat (initiation) intervals:

Instruction	latency	repeat
Int. ALU	0	1
Load	2	1
Int Mult	3	1
FP add	2	1
FP Mult	4	2
FP Div	31	32

Intel Pentium III



Intel Pentium IV



Intel Pentium IV

- 42 million transistors, 217 mm^2, 55 watts at 1.5 GHz,
- **Front End:** trace cache + instruction cache,
  - 32-bit instrs decoded in up to 4 micro-ops,
  - pointer in ROM if more than 4 micro-ops,
  - micro-ops buffered in a micro-ops queue (in order),
  - trace-cache predictor (512 entries) + BTB of 4K entries.
- **OoO Logic:**
  - allocates physical registers (128 int and 128 fp), and
  - ROB entries (126-up to 48 loads and 24 stores) for 3 micro-ops in each cycle (stalls if structural hazard)
  - register renaming supported by RAT,
  - micro-ops are queued in FIFO order in two queues: memory and non-memory queue (same as reservation stations),
  - when operands available, micro-ops dispatched up to 6 at a time,
  - speculative dispatching of dependent instructions to cu latency.
- **Execution Units**: 1 LD, 1 ST, 3 INT, 2 FP/MMX
- **Memory System**: 8KB 4-way 128B L1; 256 KB 8-way 128B L2

Intel Pentium IV

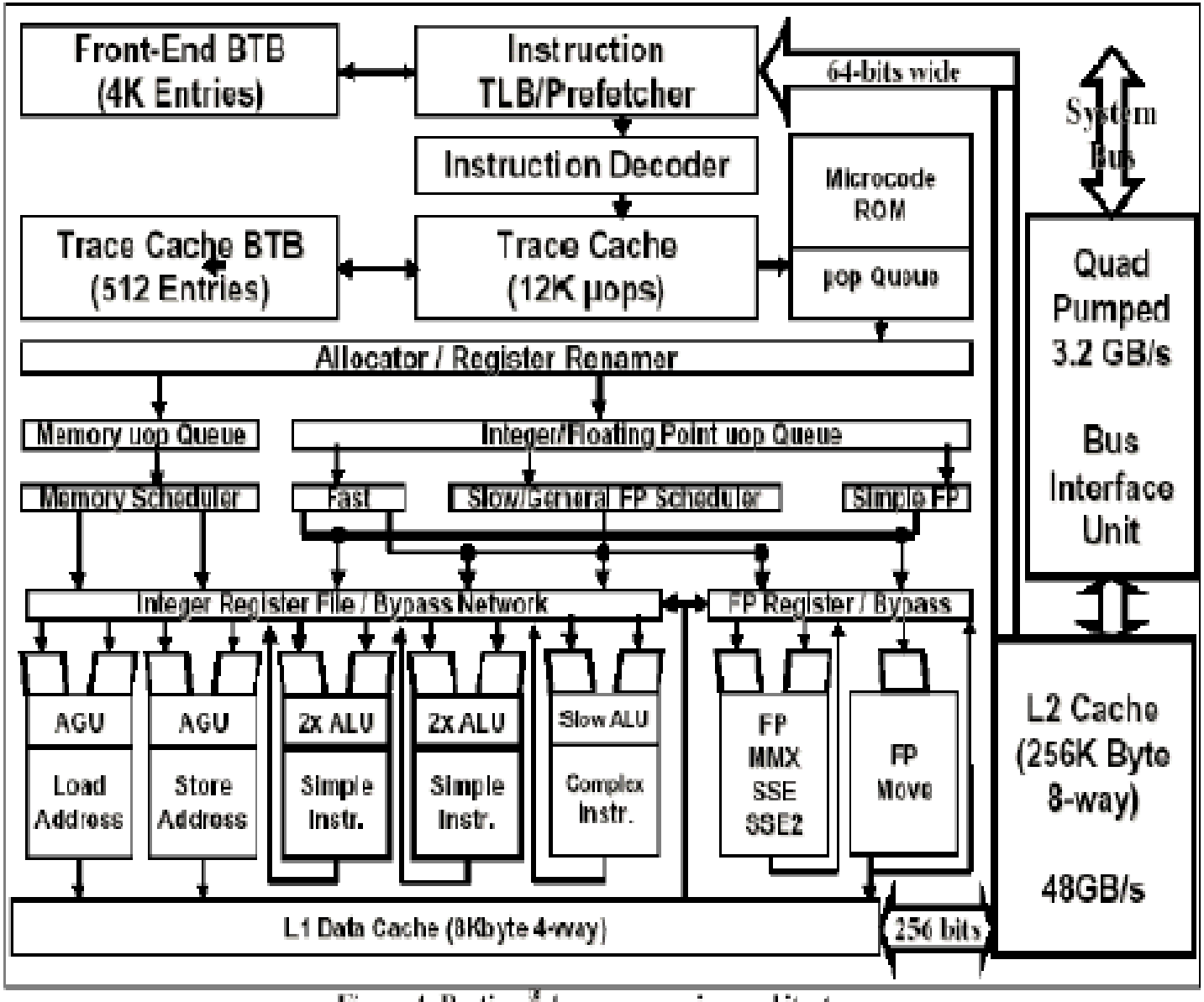


Figure 4: Pentium® 4 processor microarchitecture

1 Tomasulo Algorithm

2 Dynamic Branch Prediction

3 Speculative Execution

4 Multiple Instructions Per Clock

5 Pentium III and Pentium IV

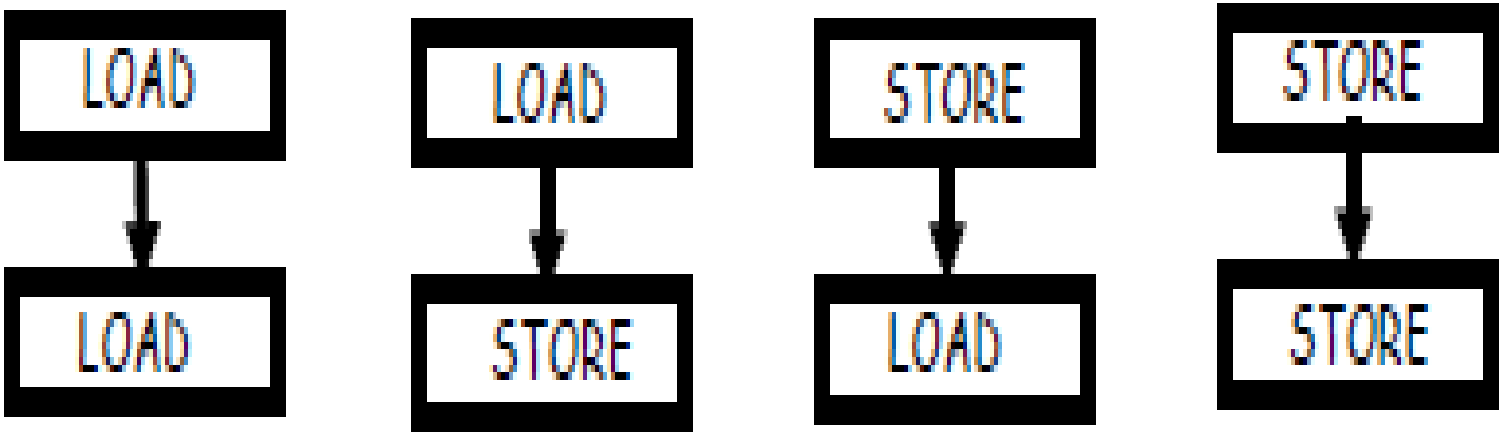
6 OoO Processors: Memory Consistency Models (MCM)

OoO Processor: Conservative MCM

- All hazards on same-memory addresses are solved in L/S Q $\Rightarrow$ helps coherence but does nothing for MCM, which refers to different addresses.
- **Conservative MCM Enforcement:**
  - a memory access is certainly performed when reaches top of ROB,
  - could wait and perform a load in cache when reaches top of ROB, downside is no memory tolerance under OoO execution
  - still, loads and stores can be prefetched (non-binding) in cache
    - if the load or store data is in the right state in cache when reaches top of ROB then performing the access takes 1 cycle,
    - however, load values still cannot be used speculatively!
- **Next Idea: exploit speculative exec to speculatively violate memory orders (works because order violations are rare!)**

Speculative Violations of Sequential Consistency

- **Orders to enforce globally:**



- **Load-Store and Store-Store orders automatically satisfied**
  - because stores are sent to store buffer in program order,
  - provided that stores are globally performed from store buffer in program order.
- **Remaining orders are load-load and store-load**
  - the SECOND access in these orders is a LOAD.
- **Exploit Speculation on Loads!**

Speculative Violations of Sequential Consistency

- **Assume a memory access (load or store) to X precedes a load of Y and that the order must be enforced:**

  ```
  LOAD X/STORE X, followed by
  LOAD Y
  ```

  - Perform `LOAD Y` speculatively before access to `X`,
  - and use the value returned by `LOAD Y` speculatively.
  - Later, when access to `X` retires at the top of ROB, check to see whether the value of `Y` has changed, and if so then rollback!
  - (Simpler, check the value of `Y` when `LOAD Y` reaches top of ROB.)

- **Two mechanisms needed: Validation and Rollback**
- **Validation of load values done at the top of ROB:**

Speculative Violations of Sequential Consistency

- **Assume a memory access (load or store) to X precedes a load of Y and that the order must be enforced:**

  ```
  LOAD X/STORE X, followed by
  LOAD Y
  ```

  - Perform `LOAD Y` speculatively before access to `X`,
  - and use the value returned by `LOAD Y` speculatively.
  - Later, when access to `X` retires at the top of ROB, check to see whether the value of `Y` has changed, and if so then rollback!
  - (Simpler, check the value of `Y` when `LOAD Y` reaches top of ROB.)

- **Two mechanisms needed: Validation and Rollback**
- **Validation of load values done at the top of ROB:**
  1. Check value of `Y` by performing the load in cache (uses extra cache bandwidth and possibly extra cycles), or
  2. Monitor events that could change the value of `Y` between the speculative load and the (load) commit.
     - Events: invalidates, updates or replacements in node's cache.
- **Load Value Recall:** ROB rollback as for mispredicted branches!

Speculative Violations of Sequential Consistency

- **Load-Load** uses load-value recall,
- **Store-Load** uses load-value recall + stalls loads at top of ROB until all previous stores have been globally performed, i.e., all prior stores from store buffer committed to cache.
- **Load-Store** safe because stores retire at top of ROB, and by that time all previous loads have retired.
- **Store-Store** safe because stores retire at top of ROB & are globally performed from store buffer one-by-one in thread order.
- **Performance issues:**
  - a load may reach the top of the ROB but cannot perform and is stalled waiting for prior stores in store buffer to commit.
  - this backs up the ROB (and other internal buffers) and eventually stalls the processor.
- **Next Idea: relax Store-Load order!**

Speculative TSO Violations

- Load-Load uses load-value recall,
- **Store-Load: no need to enforce. This value of load should not be recalled if all preceding pending accesses in L/S Q are stores. Also loads do NOT wait on store-buffer stores**.
- Load-Store safe because stores retire at top of ROB, and by that time all previous loads have retired.
- Store-Store safe because stores retire at top of ROB & are globally performed from store buffer one-by-one in thread order.
- **Performance issues:**
  - If a long latency store backs up the store buffer then stores cannot retire, which may back up the ROB and other queues and stall dispatch.
- **Next Idea: relax even further with weak-ordering and release consistency!**

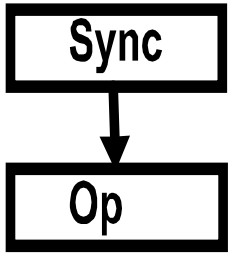
Speculative Execution of RMW Accesses

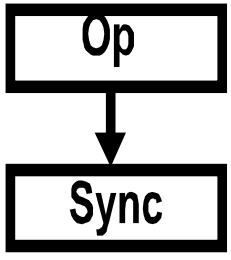
- RMW instrs are made of a load followed by a store, which execute atomically.

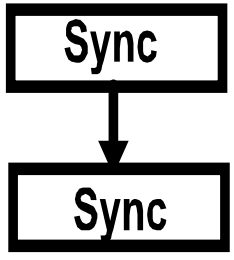
- **The value of the lock returned speculatively by the load**.
  - Based on the speculative lock value, the critical section is entered or the lock is retried speculatively. (All this is speculative in ROB.)
  - RMW's does not execute in cache until it reaches the top of ROB.
  - Loads in RMW accesses must remain subject to load-value recall until the RMW is retired; will roll back if a violation is detected.

- **Because of the store, the RMW access does not update memory until it reaches the top of ROB**
  - if the store can execute at the top of ROB, then no other thread got the lock during the time the RMW value was speculative.
  - The load and the store can be performed atomically in cache with a write-invalidate protocol.

Speculative Violations of Weak Ordering







- **OP-TO-SYNC:**
  - Any access to a SYNC variable MUST be globally performed at the top of ROB, after all previous accesses have retired and
  - after all the sores in store buffer were globally performed!
- **SYNC-TO-OP and SYNC-TO-SYNC:** No OP or SYNC access following an access to a SYNC variable can perform until the SYNC access has been globally performed:
  - automatically enforced for STORES and SYNC accesses (since they are globally performed only at the top of ROB)
  - no SYNC access may commit until the store buffer is empty.
  - a LOAD of a SYNC variable (including that of a RMW) can be speculatively performed provided it is subject to load-value recall.
- **Regular loads NOT subject to load-value recall!**