EXPLOITING ILP

B649
Parallel Architectures and Pipelining

What is ILP?

Pipeline CPI = Ideal pipeline CPI + Structural stalls +
Data hazard stalls + Control stalls

- Use hardware techniques to minimize stalls
  - ⋆ branch prediction
  - ⋆ dynamic scheduling
  - ⋆ speculation
- Pick instructions across branches (i.e., across basic blocks) to overlap
  - ⋆ on MIPS average **dynamic** branch frequency is 15% to 25%
  - ⋆ Pick instructions across loop iterations

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

Data Dependences

- Conditions for data dependence from instruction *i* to instruction *j*
  - ⋆ *i* and *j* access a common memory location / register
  - ⋆ at least one of those accesses is a write
  - ⋆ there is a valid control-flow path from *i* to *j*
- Data dependence is transitive
  - ⋆ *j* depends on *i*, *k* depends on *j* ⇒ *k* depends on *i*
- Three types of data dependences
  - ⋆ true dependence (RAW)
  - ⋆ anti-dependence (WAR)
  - ⋆ output dependence (WAW)

Control Dependences

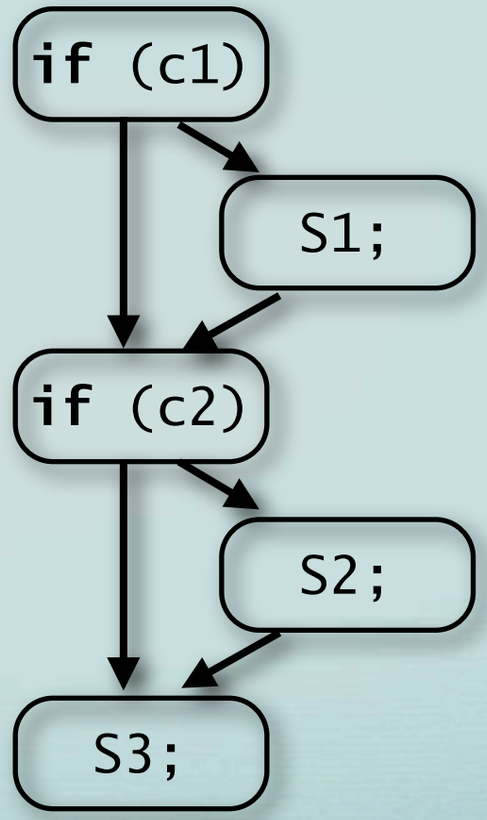
- Dependences arising out of program control-flow
- Prevent reordering to maintain program correctness (just like data dependences)

```
if (c1) {
    S1;
}
if (c2) {
    S2;
}
S3;
```

Control Dependences

- Dependences arising out of program control-flow
- Prevent reordering to maintain program correctness (just like data dependences)

```
if (c1) {
    S1;
}
if (c2) {
    S2;
}
S3;
```



Another (Equivalent) View

- Data dependences
  - ⋆ true dependences (RAW)
- Name dependences
  - ⋆ anti-dependences (WAR) and output dependences (WAW)
  - ⋆ not “true” dependences
- Control dependences

Another (Equivalent) View

- Data dependences
  - ⋆ true dependences (RAW)
- Name dependences
  - ⋆ anti-dependences (WAR) and output dependences (WAW)
  - ⋆ not "true" dependences
- Control dependences

Both software and hardware will reorder to improve performance as long as the "observed behavior" of the program does not change.

(*Principle of observational equivalence*)

BASIC COMPILER TECHNIQUES: LOOP UNROLLING AND SCHEDULING

Simple Example

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

Simple Example

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

```
Loop:   L.D     F0,0(R1)      ;F0=array element
        ADD.D   F4,F0,F2      ;add scalar in F2
        S.D     F4,0(R1)      ;store result
        DADDUI  R1,R1,#-8     ;decrement pointer
                              ;8 bytes (per DW)
        BNE     R1,R2,Loop    ;branch R1!=R2
```

Pipelined Machine

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

Scheduled 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

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

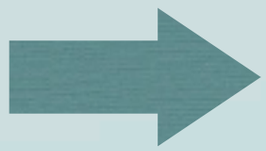
```
                                          Clock cycle issued
Loop:   L.D      F0,0(R1)                 1
        stall                             2
        ADD.D    F4,F0,F2                 3
        stall                             4
        stall                             5
        S.D      F4,0(R1)                 6
        DADDUI   R1,R1,#-8                7
        stall                             8
        BNE      R1,R2,Loop               9
```

Scheduled 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

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

```
                                    Clock cycle issued
Loop:   L.D      F0,0(R1)           1
        stall                       2
        ADD.D    F4,F0,F2           3
        stall                       4
        stall                       5
        S.D      F4,0(R1)           6
        DADDUI   R1,R1,#-8          7
        stall                       8
        BNE      R1,R2,Loop         9
```



```
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
```

Loop Unrolling

```
Loop:   L.D      F0,0(R1)     ;F0=array element
        ADD.D    F4,F0,F2     ;add scalar in F2
        S.D      F4,0(R1)     ;store result
        DADDUI   R1,R1,#-8    ;decrement pointer
                              ;8 bytes (per DW)
        BNE      R1,R2,Loop   ;branch R1!=R2
```

```
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
```

Loop Unrolling

```
Loop:   L.D      F0,0(R1)     ;F0=array element
        ADD.D    F4,F0,F2     ;add scalar in F2
        S.D      F4,0(R1)     ;store result
        DADDUI   R1,R1,#-8    ;decrement pointer
                              ;8 bytes (per DW)
        BNE      R1,R2,Loop   ;branch R1!=R2
```

Takes 27 cycles

```
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
```

Schedule 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
```

Schedule 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
```

Takes 14 cycles

Loop Unrolling

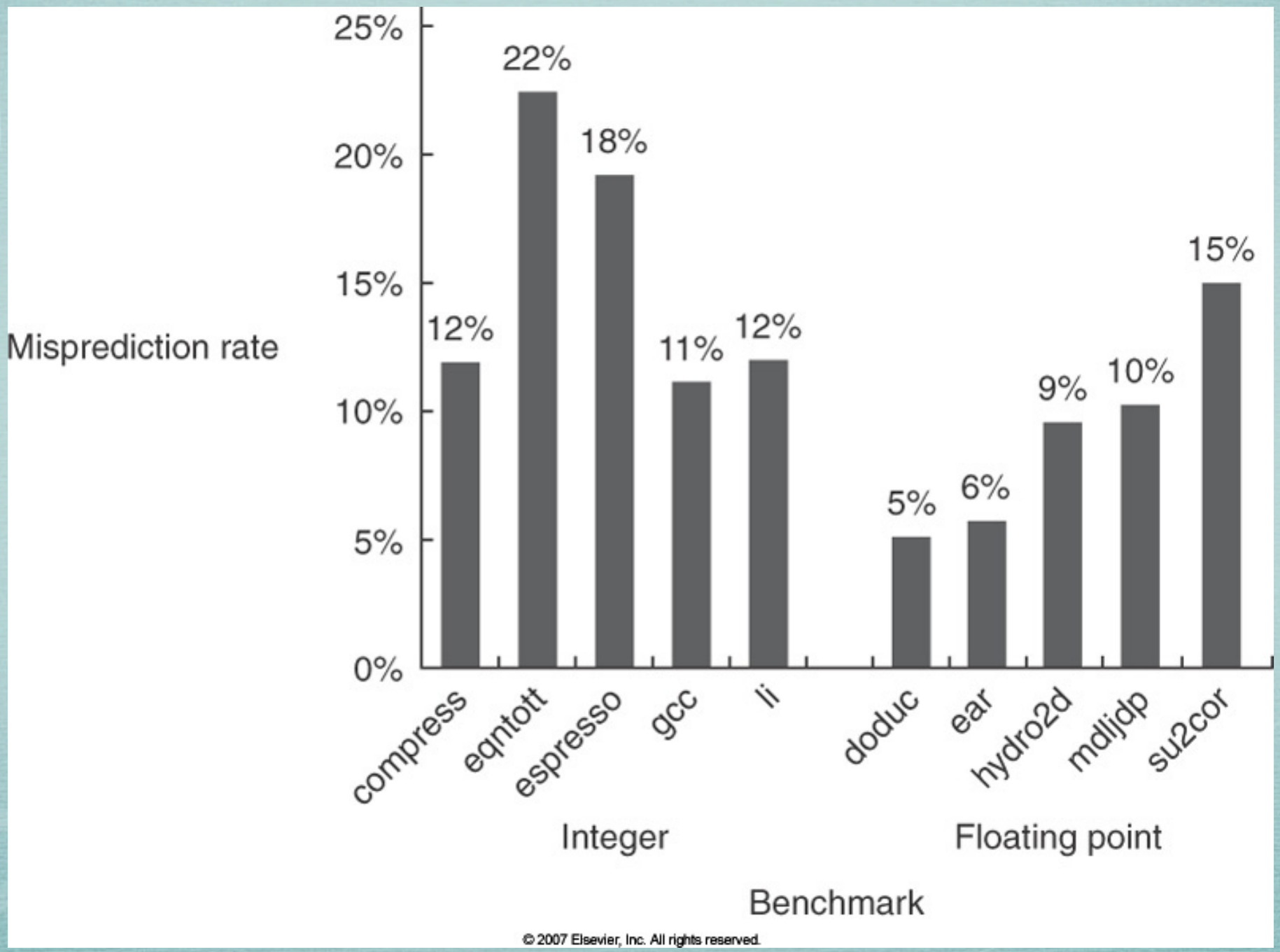
- Unroll a small number of times (called unroll factor)
  - ⋆ reduces branches
  - ⋆ bigger body enables better instruction scheduling
- Rename registers to avoid name dependences
  - ⋆ too much unrolling can cause **register pressure**
- Reorder instructions to reduce stalls
- Generate startup and / or cleanup loops for iterations that are not multiple of unroll factor

BRANCH PREDICTION

Static Branch Prediction

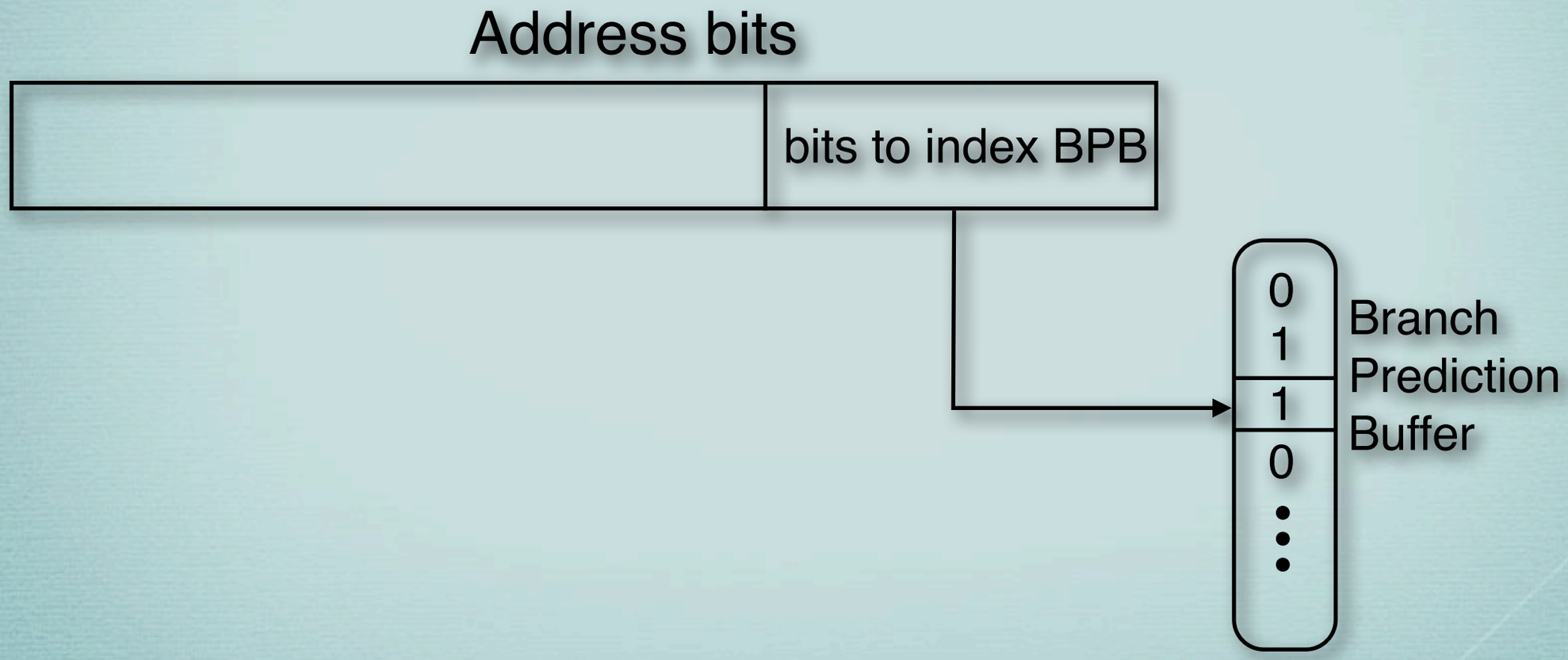
- Delay slots help
  - ⋆ can schedule instructions in the delay slots from the branch direction taken more often
- Static prediction approaches
  - ⋆ predict taken (average misprediction for SPEC is 34%, ranging from 9% to 59%)
  - ⋆ use profile information

Misprediction Rate Based on Profile Data Spec92 Benchmarks



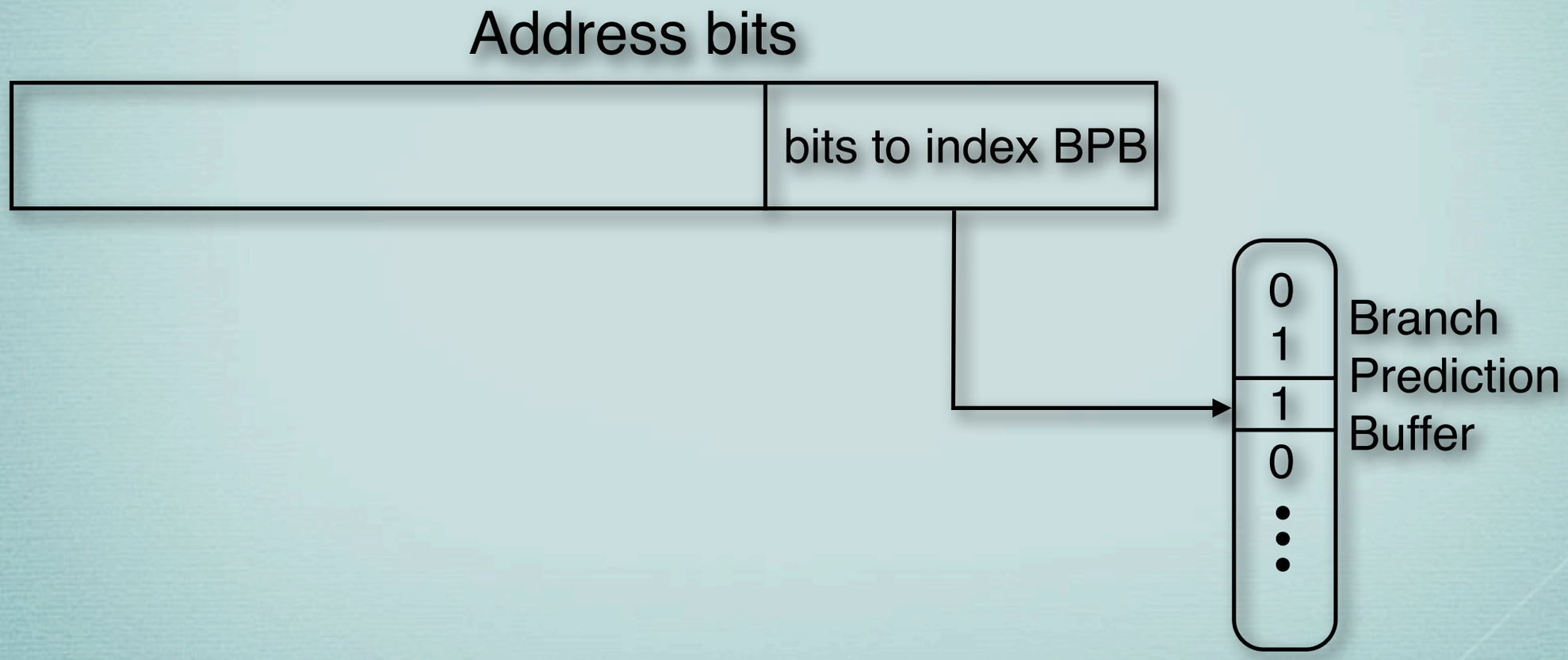
Dynamic Branch Prediction

- "Branch prediction buffer" or "branch history table"
- Small 1-bit memory (cache) indexed by lower bits of address



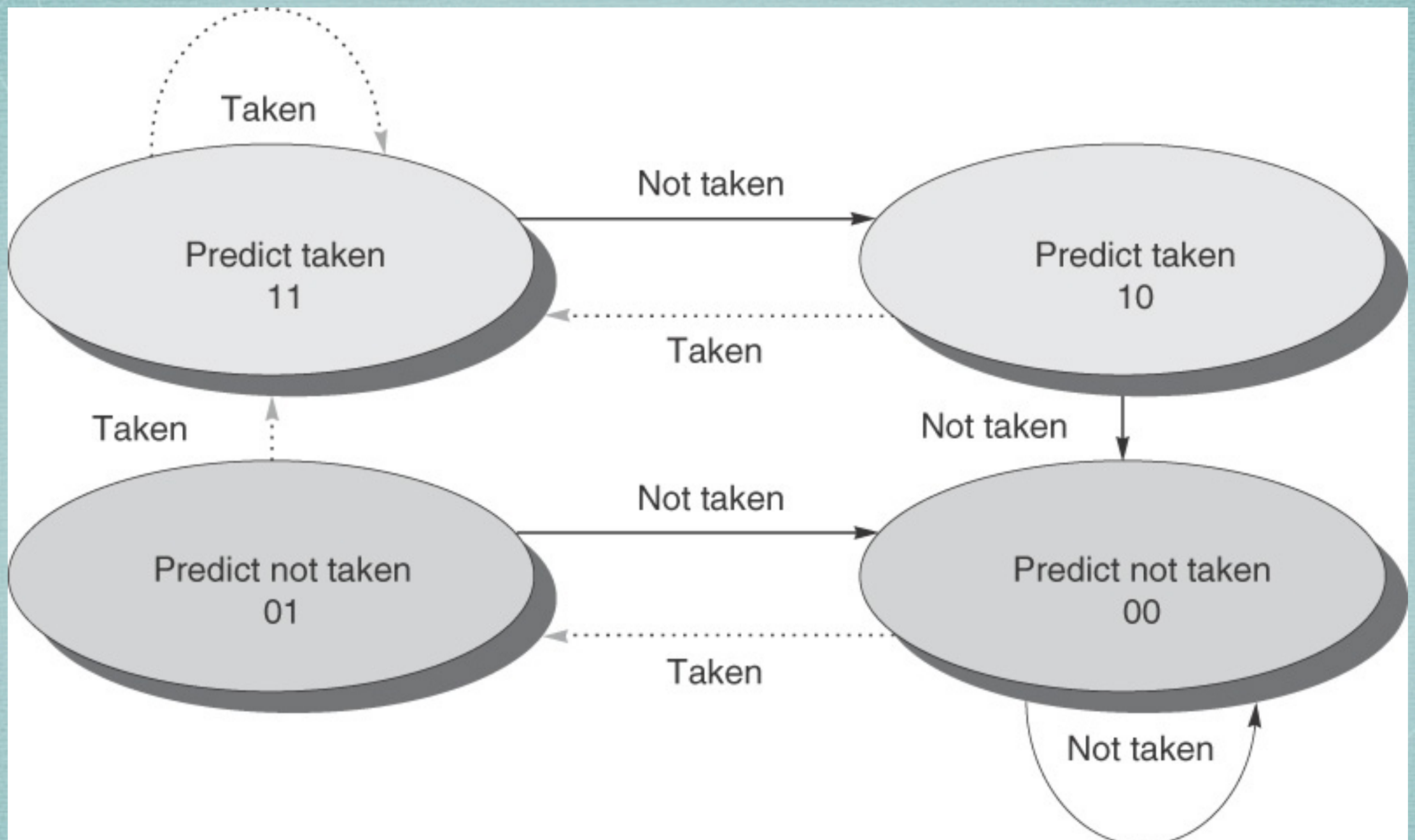
Dynamic Branch Prediction

- "Branch prediction buffer" or "branch history table"
- Small 1-bit memory (cache) indexed by lower bits of address

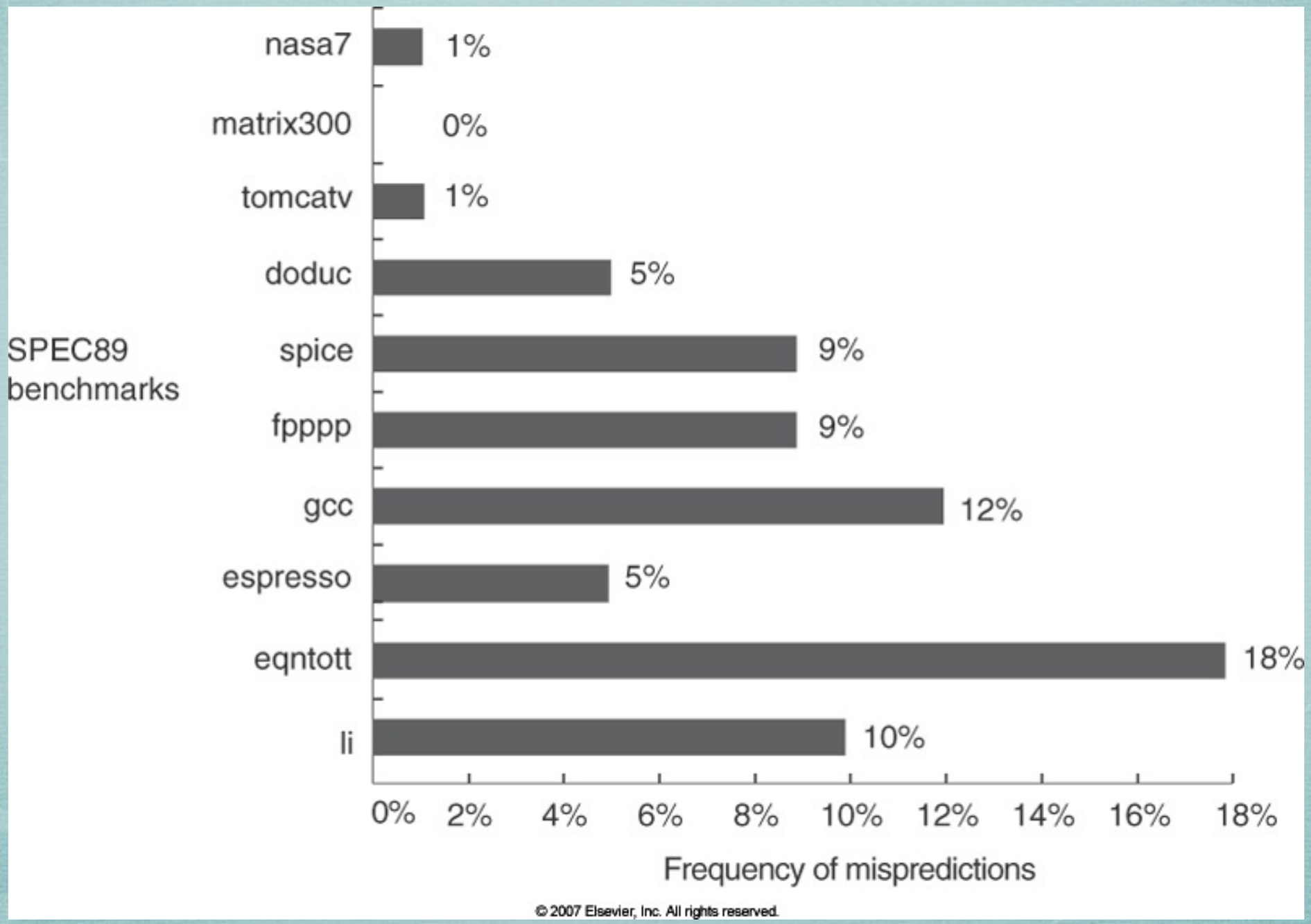


Problem: Predicts incorrectly twice

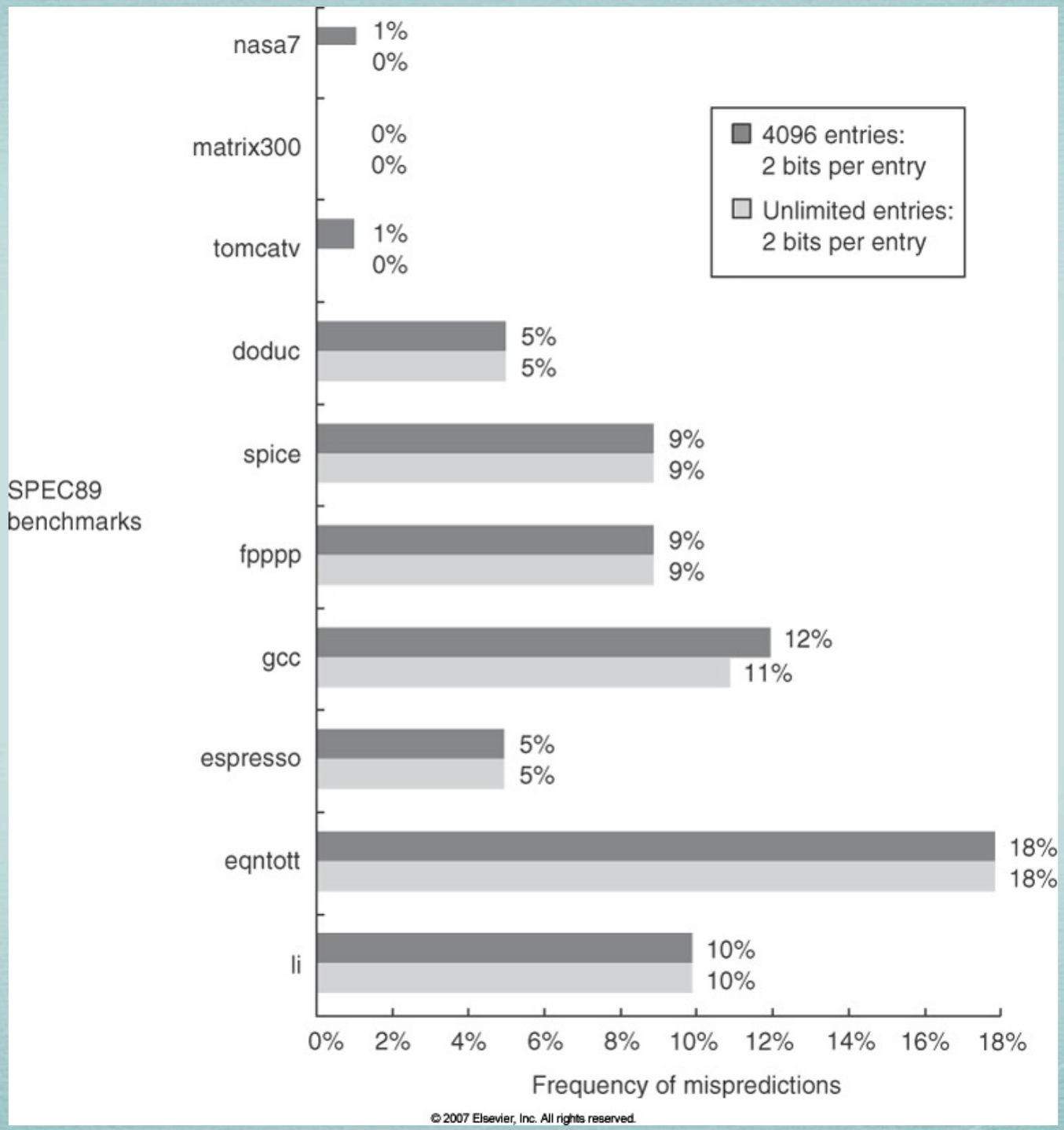
Two-bit Predictors



Prediction Accuracy: 4K 2-bit Entries (Spec89 Benchmarks)



Prediction Accuracy: 4K vs Infinite



Correlating Branch Predictors

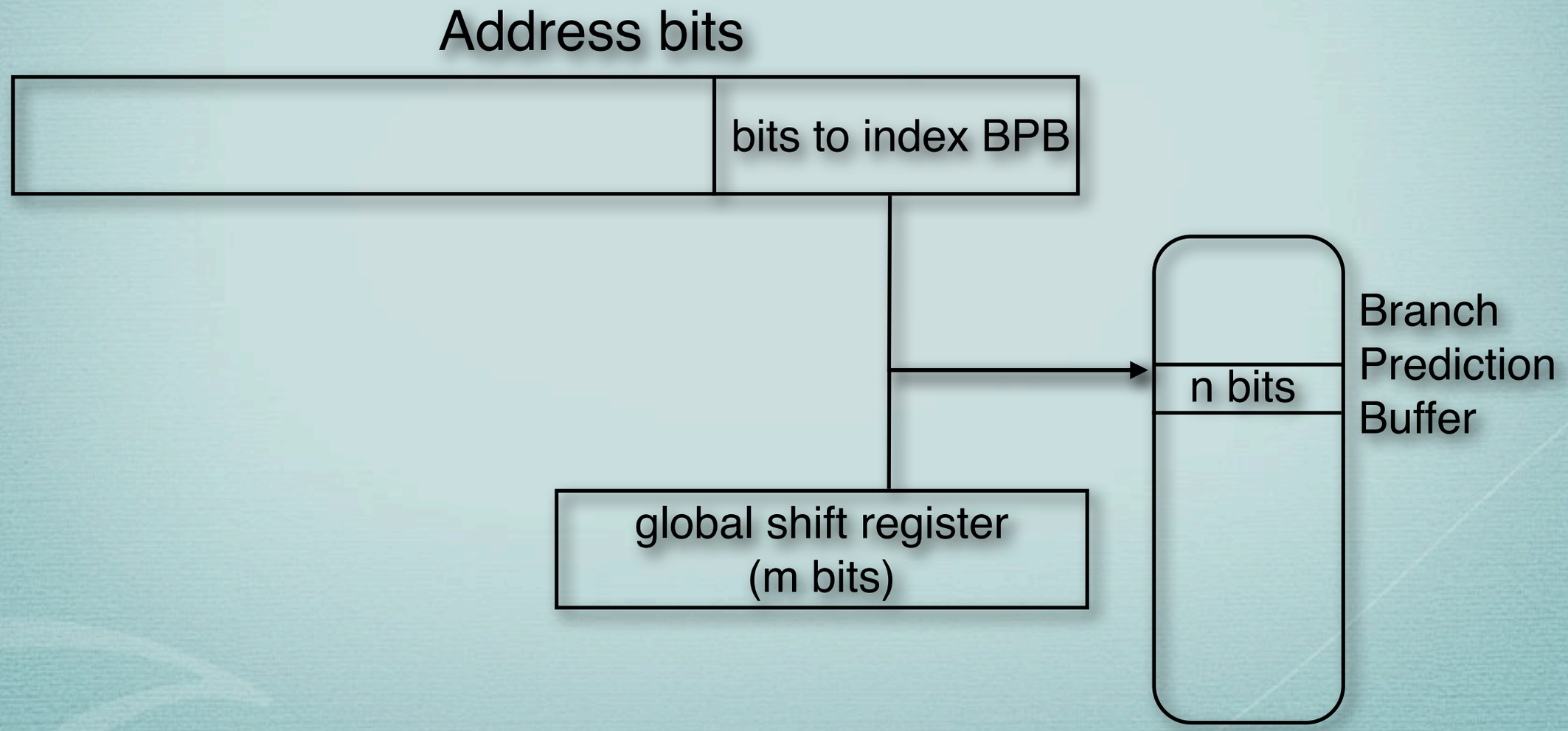
- Motivating example

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

- Idea
  - ★ "correlate" last m branches

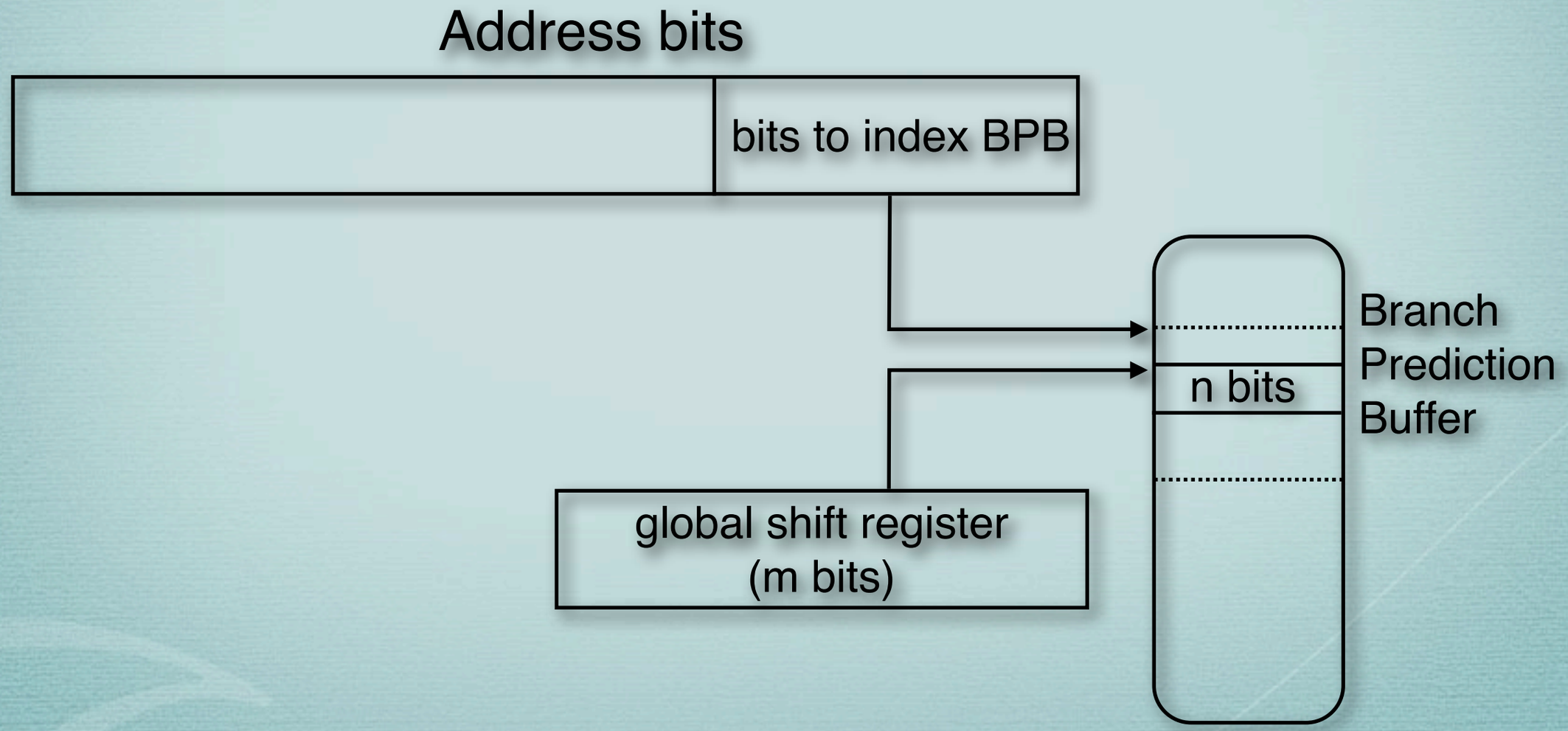
General Correlating Predictors

- (m,n) predictor
  - ⋆ use last m branches to predict using n bit saturating counters



General Correlating Predictors

- (m,n) predictor
  - ⋆ use last m branches to predict using n bit saturating counters



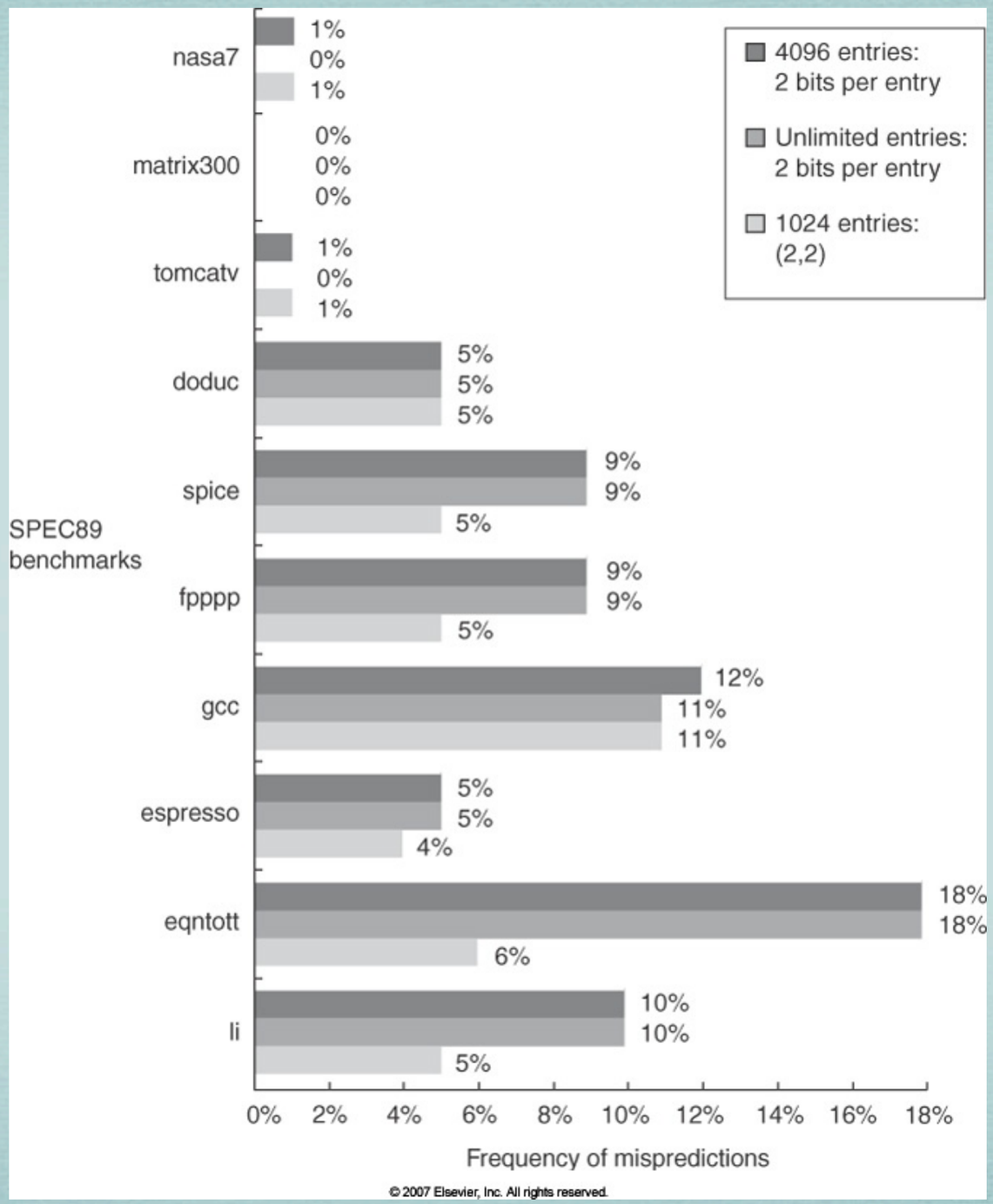
General Correlating Predictors

- (m,n) predictor
  - ⋆ use last m branches to predict using n bit saturating counters

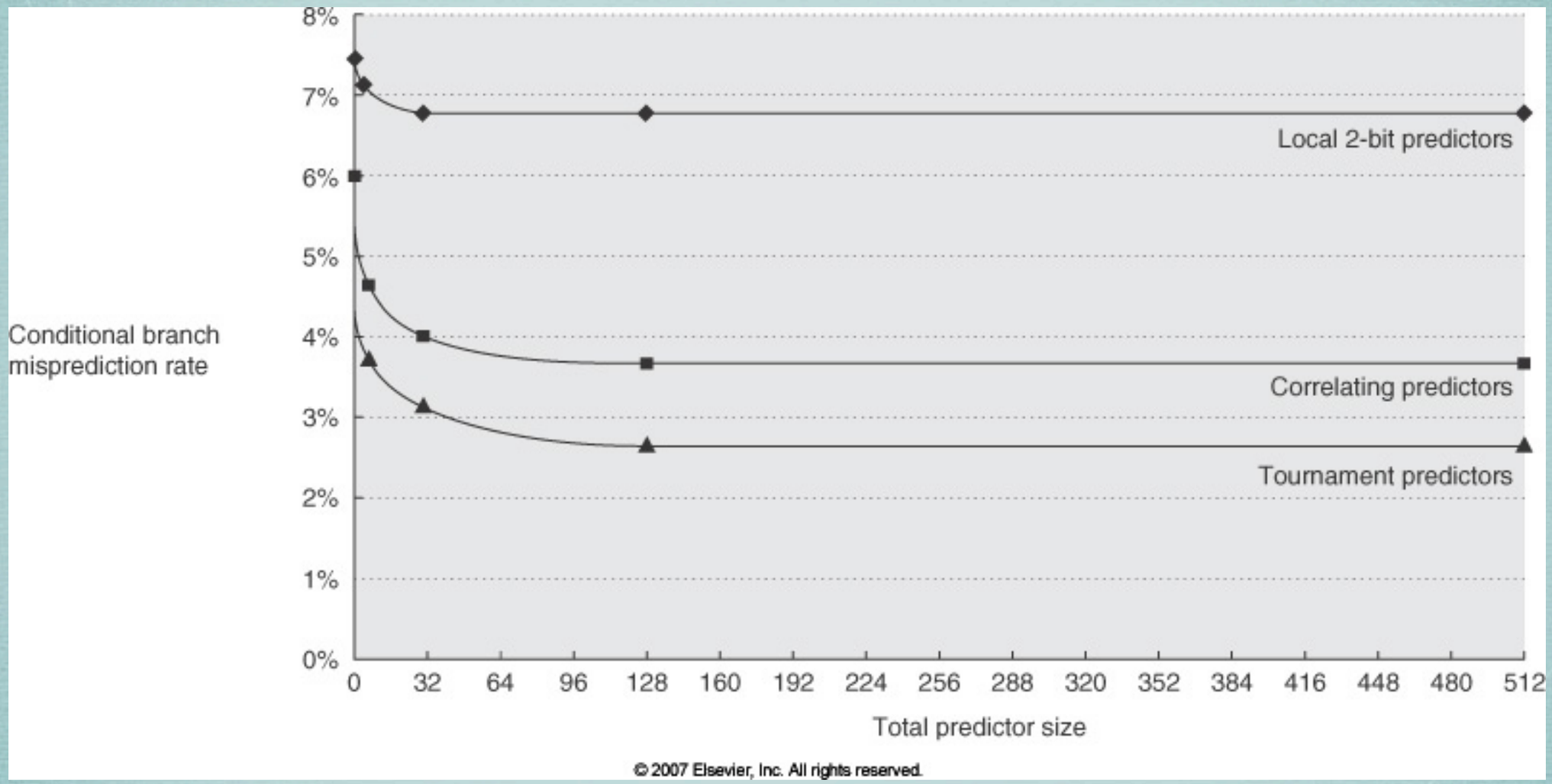
if b = number of entries in BTB,

number of bits in BTB = 2^m × 2^n × b

Comparison of 2-bit Predictors



Tournament Predictors (Spec89 Benchmarks)



DYNAMIC SCHEDULING

Comments on Assignment 1

- Start NOW!
- Use course blog for discussions
- Needs work, but rewards await you
- You may mix languages
  - ⋆ e.g., parsing might be easier with a scripting language
- Matrix-matrix computation == matrix multiply
- Note differences in Tomasulo's approach for assignment from the textbook Figure 2.9
- Extra credit possibilities (case-by-case)
  - ⋆ forwarding in pipelined architecture
  - ⋆ speculation in Tomasulo's approach
  - ⋆ other applications
  - ⋆ transformations such as, loop unrolling

Why Dynamic Scheduling?

- Reduces stalls
  - ⋆ tolerates data hazards
  - ⋆ tolerates cache misses
- Code optimized for one pipeline can run on another
- Can work with statically scheduled code
- BUT, needs significantly more hardware

Dynamic Scheduling: Idea

- Out-of-order execution
- Out-of-order completion
- Additional issues:
  - ⋆ WAR and WAW hazards
  - ⋆ precise exceptions
- Implementation:
  - ⋆ split ID into *Issue* and *Read Operands*
  - ⋆ multiple instructions *in execution* need multiple functional units
- In order issue, out-of-order execution

Dynamic Scheduling: Idea

- Out-of-order execution
- Out-of-order completion
- Additional issues:
  - ⋆ WAR and WAW hazards
  - ⋆ precise exceptions
- Implementation:
  - ⋆ split ID into *Issue* and *Read Operands*
  - ⋆ multiple instructions *in execution* need multiple functional units
- In order issue, out-of-order execution

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

Dynamic Scheduling: Idea

- Out-of-order execution
- Out-of-order completion
- Additional issues:
  - ⋆ WAR and WAW hazards
  - ⋆ precise exceptions
- Implementation:
  - ⋆ split ID into *Issue* and *Read Operands*
  - ⋆ multiple instructions *in execution* need multiple functional units
- In order issue, out-of-order execution

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

```
DIV.D   F0, F2, F4
ADD.D   F6,F0,F8
SUB.D   F8,F10,F14
MUL.D   F6,F10,F8
```

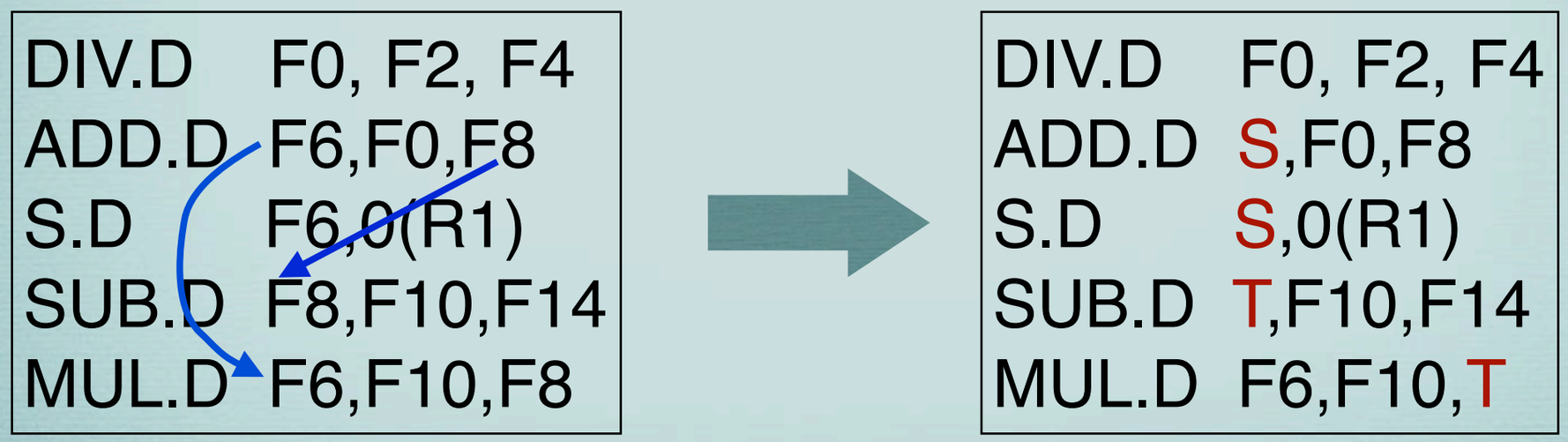
Tomasulo's Approach

- Invented by Robert Tomasulo for IBM 360
  - ⋆ 360 had only 4 FP regs and long floating point delays
- Avoids RAW and WAW hazards by *register renaming*
  - ⋆ use of *reservation stations*

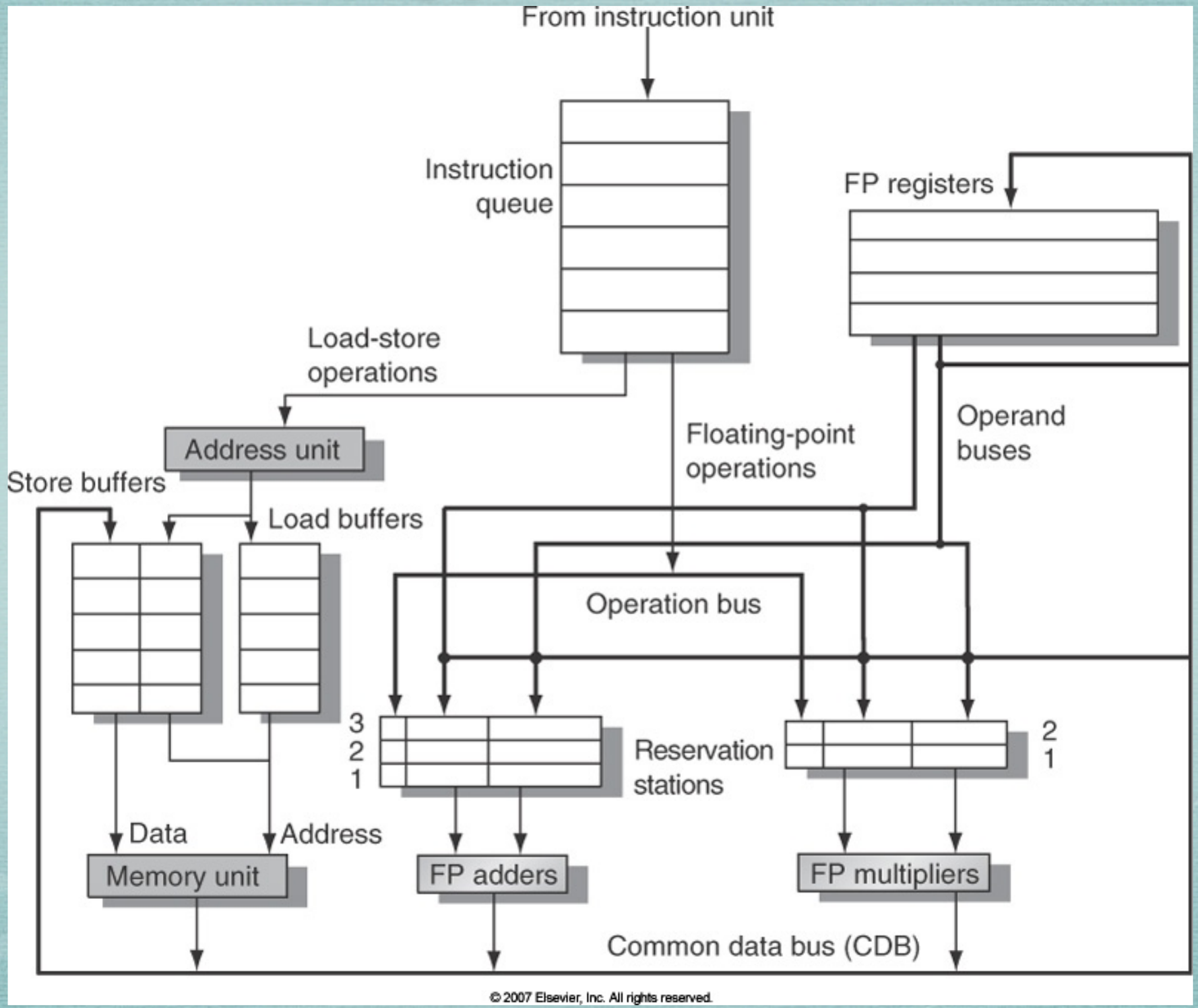
```
DIV.D    F0, F2, F4
ADD.D    F6,F0,F8
S.D      F6,0(R1)
SUB.D    F8,F10,F14
MUL.D    F6,F10,F8
```

Tomasulo's Approach

- Invented by Robert Tomasulo for IBM 360
  - ⋆ 360 had only 4 FP regs and long floating point delays
- Avoids RAW and WAW hazards by *register renaming*
  - ⋆ use of *reservation stations*



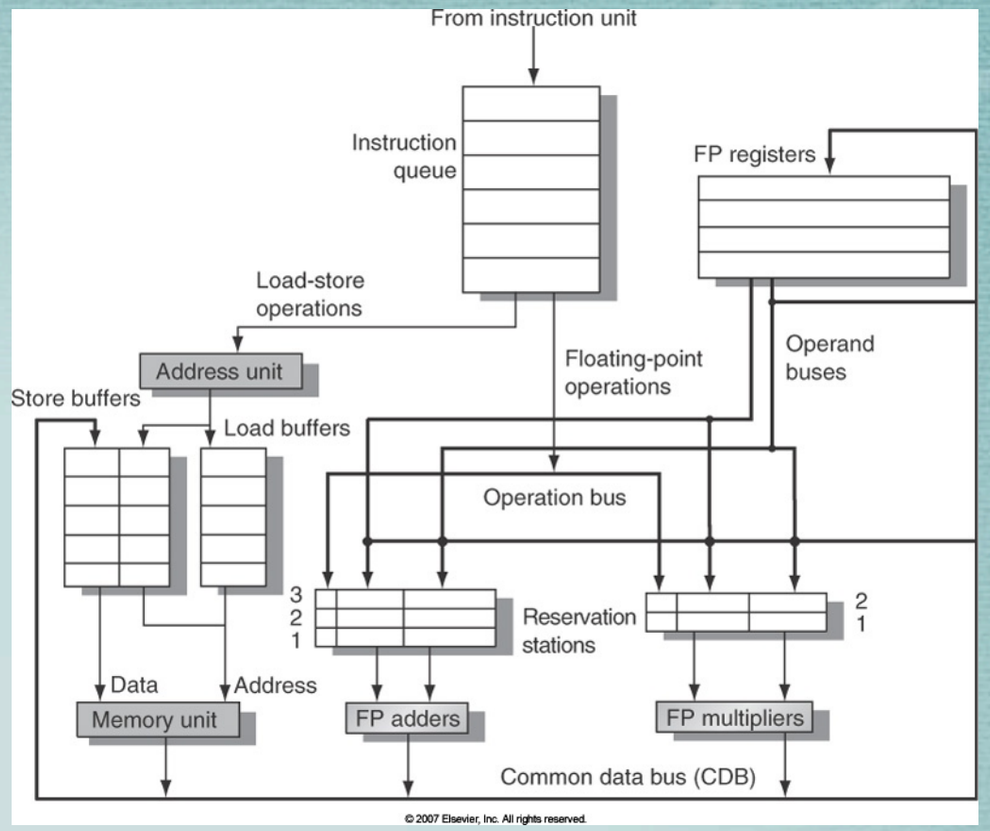
Tomasulo's Approach: Basic Structure



Tomasulo's Approach: Steps

- Issue
  - ⋆ get next instruction from queue
  - ⋆ move to a matching reservation station (or stall if none available)
  - ⋆ get operand values if in registers, else keep track of units that will produce them
- Execute
  - ⋆ if an operand not available, monitor the CDB
  - ⋆ if all operands are ready, execute the instruction when the functional unit becomes available
  - ⋆ loads and stores take two steps: read register and wait for memory
- Write results
  - ⋆ functional units write into CDB (and from there into registers)
  - ⋆ stores write to memory when both value and store register are available

Tomasulo's Approach: Fields



- Reservation station:
  - ⋆ Op: operation
  - ⋆ Qj, Qk: operands, to come from reservation stations
  - ⋆ Vj, Vk: operands, available in registers
  - ⋆ A: effective address (initialized with immediate value)
  - ⋆ Busy: bit to indicate the reservation station is busy
- Register file:
  - ⋆ Qi: the number of the reservation station whose result will go into this register; blank (or zero) indicates the register already has the value

Example

```
1.  L.D     F6,32(R2)
2.  L.D     F2,44(R3)
3.  MUL.D   F0,F2,F4
4.  SUB.D   F8,F2,F6
5.  DIV.D   F10,F0,F6
6.  ADD.D   F6,F8,F2
```

Example

Instruction		Instruction status		
		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					45 + Regs[R3]
Add1	yes	SUB		Mem[34 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

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

Example (contd.)

		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	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL	Mem[45 + Regs[R3]]	Regs[F4]			
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

Register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1					Mult2			

Observations

- RAW hazards handled by waiting for operands
- WAR and WAW hazards handled by register renaming
  - ⋆ only WAR and WAW hazards between instructions currently in the pipeline are handled; is this a problem?
  - ⋆ larger number of hidden names reduces name dependences
- CDB implements forwarding

Loop Example

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,-8
        BNE      R1,R2,loop
```

Loop Example

		Instruction status			
Instruction		From iteration	Issue	Execute	Write Result
L.D	F0,0(R1)	1	√	√	
MUL.D	F4,F0,F2	1	√		
S.D	F4,0(R1)	1	√		
L.D	F0,0(R1)	2	√	√	
MUL.D	F4,F0,F2	2	√		
S.D	F4,0(R1)	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	yes	Load					Regs[R1] + 0
Load2	yes	Load					Regs[R1] – 8
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL		Regs[F2]	Load1		
Mult2	yes	MUL		Regs[F2]	Load2		
Store1	yes	Store	Regs[R1]			Mult1	
Store2	yes	Store	Regs[R1] – 8			Mult2	

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

Summary of Tomasulo's Approach

- Need to check WAR and WAW hazards
  - ⋆ through registers
  - ⋆ through loads and stores
- Works very well if branches predicted accurately
  - ⋆ instruction not allowed to execute unless all preceding branches finished
- Out-of-order completion results in imprecise exceptions
- Widely popular
  - ⋆ high performance without compiler assistance
  - ⋆ can hide cache latencies
  - ⋆ reasonable performance for code difficult to schedule statically
  - ⋆ key component of speculation

HARDWARE-BASED SPECULATION

Speculation: Handling Control Dependences

- Fetch, issue, and **execute** instructions as if branch predictions always right
- Mechanism to handle situation where prediction was incorrect
- Combines three key ideas:
  - ⋆ dynamic branch prediction
  - ⋆ speculation to execute without waiting for control dependences to resolve
  - ⋆ dynamic scheduling

Speculation: Handling Control Dependences

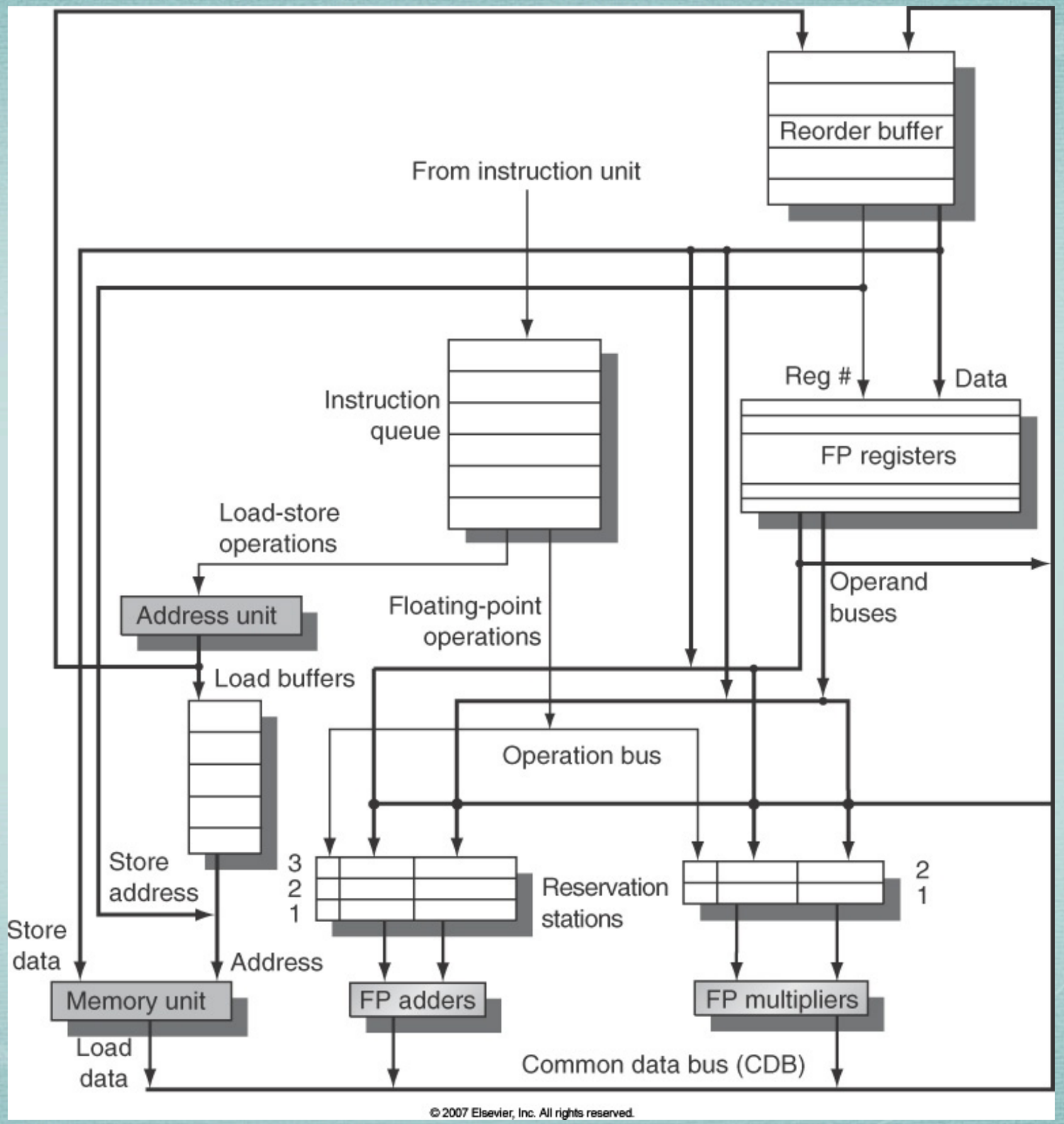
- Fetch, issue, and **execute** instructions as if branch predictions always right
- Mechanism to handle situation where prediction was incorrect
- Combines three key ideas:
  - ⋆ dynamic branch prediction
  - ⋆ speculation to execute without waiting for control dependences to resolve
  - ⋆ dynamic scheduling

Data flow execution

Extending Tomasulo's Algorithm

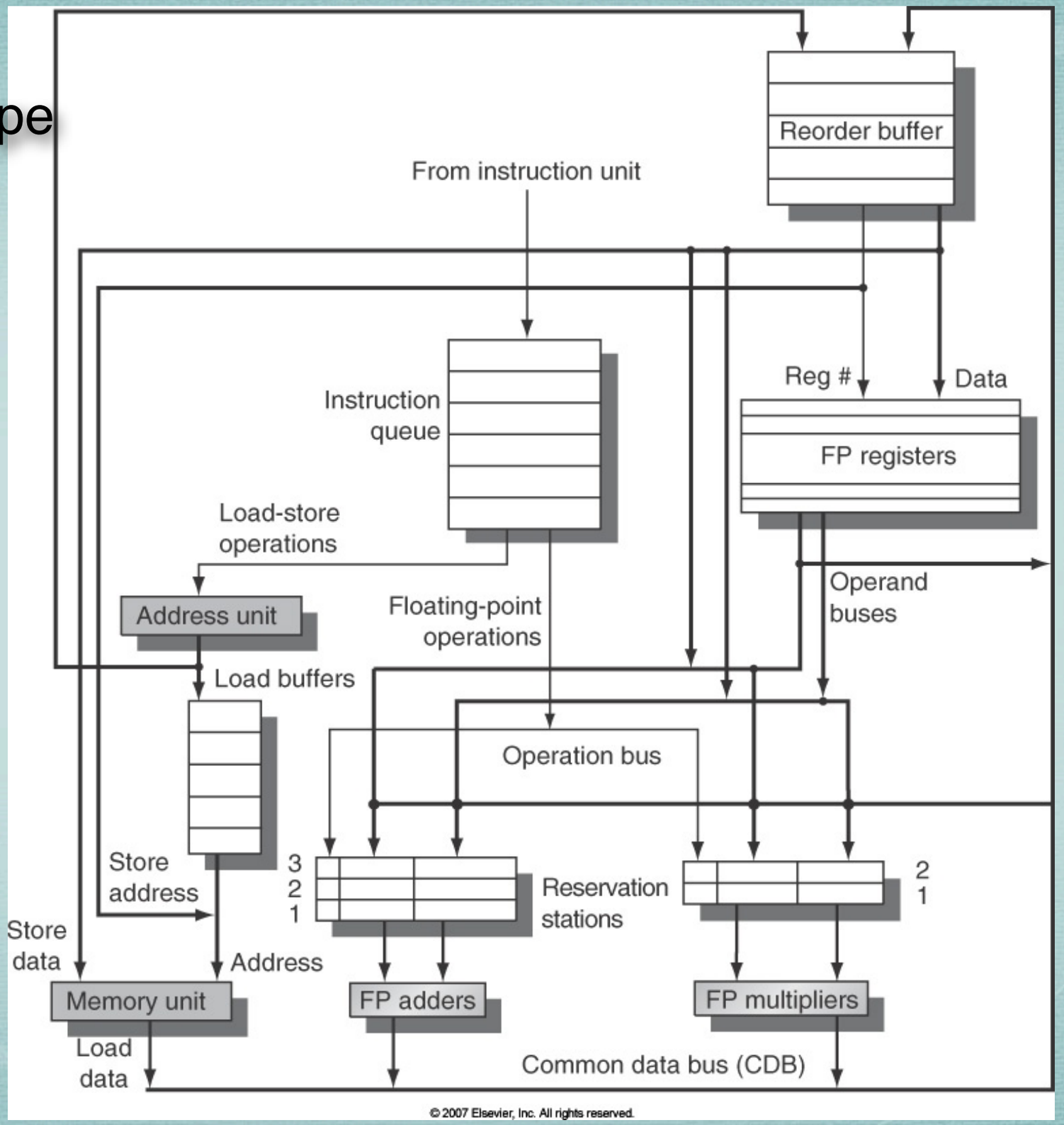
- Separate execution from completion
  - ⋆ **execute**: when data dependences are resolved
  - ⋆ **commit**: when control dependences are resolved
- Out-of-order execution, but in-order commit
- Store uncommitted instructions in a **reorder buffer** (ROB)
- Written results found in ROB, until committed
  - ⋆ similar to store buffer
- Register file / memory written upon commit

Tomasulo's Approach + Speculation



Tomasulo's Approach + Speculation

Fields in ROB
1. Instruction type
2. Destination
3. Value
4. Ready



Speculation Steps

Issue → *Execute* → *Write* → *Commit*

- Issue
  - ⋆ get instruction from queue
  - ⋆ issue if empty reservation station and empty ROB slot
    - ✳ otherwise, stall
  - ⋆ update control fields to indicate buffers are in use
  - ⋆ send the reserved ROB entry number to the reservation station for tagging

Speculation Steps

Issue → ***Execute*** → *Write* → *Commit*

- Execute
  - ⋆ if an operand not ready, monitor the CDB
    - ✳ checks RAW hazards
  - ⋆ execute when both operands available
    - ✳ loads require two steps (why?)
    - ✳ stores only need base register (why?)

Speculation Steps

Issue → *Execute* → ***Write*** → *Commit*

- Write
  - ⋆ upon completion, write result+tag on CDB
  - ⋆ CDB is read by ROB and any waiting reservation stations
  - ⋆ mark reservation station available
  - ⋆ for store:
    - ✳ write in ROB's Value field if value available
    - ✳ otherwise, keep monitoring CDB for the value

Speculation Steps

Issue → *Execute* → *Write* → ***Commit***

- Commit (also called "completion" or "graduation")
  - ⋆ normal commit
    - ✳ update the register with the result, remove entry from ROB
  - ⋆ store
    - ✳ update memory with the result, remove entry from ROB
  - ⋆ correctly predicted branch
    - ✳ finish the branch
  - ⋆ incorrectly predicted branch
    - ✳ flush the ROB
    - ✳ restart at the correct successor

Example

```
1.  L.D     F6,32(R2)
2.  L.D     F2,44(R3)
3.  MUL.D   F0,F2,F4
4.  SUB.D   F8,F2,F6
5.  DIV.D   F10,F0,F6
6.  ADD.D   F6,F8,F2
```

Example

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	`L.D`	`F6,32(R2)`	Commit	F6	Mem[34 + Regs[R2]]
2	no	`L.D`	`F2,44(R3)`	Commit	F2	Mem[45 + Regs[R3]]
3	yes	`MUL.D`	`F0,F2,F4`	Write result	F0	#2 × Regs[F4]
4	yes	`SUB.D`	`F8,F2,F6`	Write result	F8	#2 – #1
5	yes	`DIV.D`	`F10,F0,F6`	Execute	F10	
6	yes	`ADD.D`	`F6,F8,F2`	Write result	F6	#4 + #2

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	`MUL.D`	Mem[45 + Regs[R3]]	Regs[F4]			#3	
Mult2	yes	`DIV.D`		Mem[34 + Regs[R2]]	#3		#5	

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	. . .	yes	yes

Loop Example

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,-8
        BNE      R1,R2,loop
```

Loop Example

	Reorder buffer				
Entry	Busy	Instruction	State	Destination	Value
1	no	`L.D F0,0(R1)`	Commit	F0	Mem[0 + Regs[R1]]
2	no	`MUL.D F4,F0,F2`	Commit	F4	#1 × Regs[F2]
3	yes	`S.D F4,0(R1)`	Write result	0 + Regs[R1]	#2
4	yes	`DADDIU R1,R1,#-8`	Write result	R1	Regs[R1] – 8
5	yes	`BNE R1,R2,Loop`	Write result		
6	yes	`L.D F0,0(R1)`	Write result	F0	Mem[#4]
7	yes	`MUL.D F4,F0,F2`	Write result	F4	#6 × Regs[F2]
8	yes	`S.D F4,0(R1)`	Write result	0 + #4	#7
9	yes	`DADDIU R1,R1,#-8`	Write result	R1	#4 – 8
10	yes	`BNE R1,R2,Loop`	Write result		

	FP register status								
Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	...	no

Observations on Speculation

- Speculation enables precise exception handling
  - ⋆ defer exception handling until instruction ready to commit
- Branches are critical to performance
  - ⋆ prediction accuracy
  - ⋆ latency of misprediction detection
  - ⋆ misprediction recovery time
- Must avoid hazards through memory
  - ⋆ WAR and WAW already taken care of (how?)
  - ⋆ for RAW
    - ✳ don't allow load to proceed if an active ROB entry has Destination field matching with A field of load
    - ✳ maintain program order for effective address computation (why?)

SUPERSCALAR PROCESSORS

Improving ILP: Multiple Issue

- Statically scheduled superscalar processors
- VLIW (Very Long Instruction Word) processors
- Dynamically scheduled superscalar processors

Multiple Issue Processor Types

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
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	Pentium 4, MIPS R12K, IBM Power5
VLIW/LIW	static	primarily software	static	all hazards determined and indicated by compiler (often implicitly)	most examples are in the embedded space, such as the TI C6x
EPIC	primarily static	primarily software	mostly static	all hazards determined and indicated explicitly by the compiler	Itanium

Dyn. Scheduling+Multiple Issue+Speculation

- Design parameters
  - ⋆ two-way issue (two instruction issues per cycle)
  - ⋆ pipelined and separate integer and FP functional units
  - ⋆ dynamic scheduling, but not out-of-order issue
  - ⋆ speculative execution
- Task per issue: assign reservation station and update pipeline control tables (i.e., control signals)
- Two possible techniques
  - ⋆ do the task in half a clock cycle
  - ⋆ build wider logic to issue any pair of instructions together
- Modern processors use both (4 or more way superscalar)

Loop 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,R2,Loop   ;branch if not last
```

Two-Way Issue, 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 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

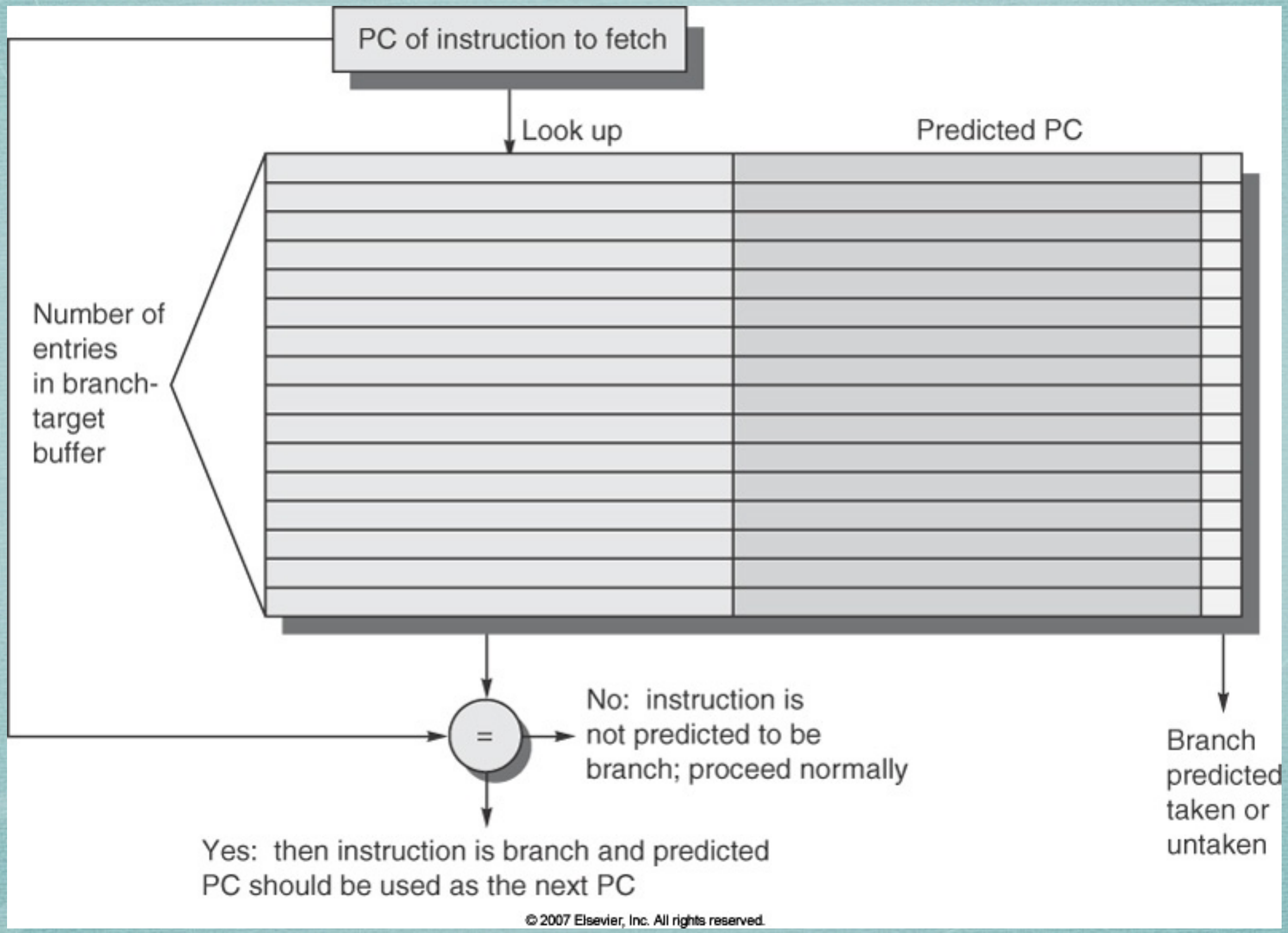
Two-Way Issue, With Speculation

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

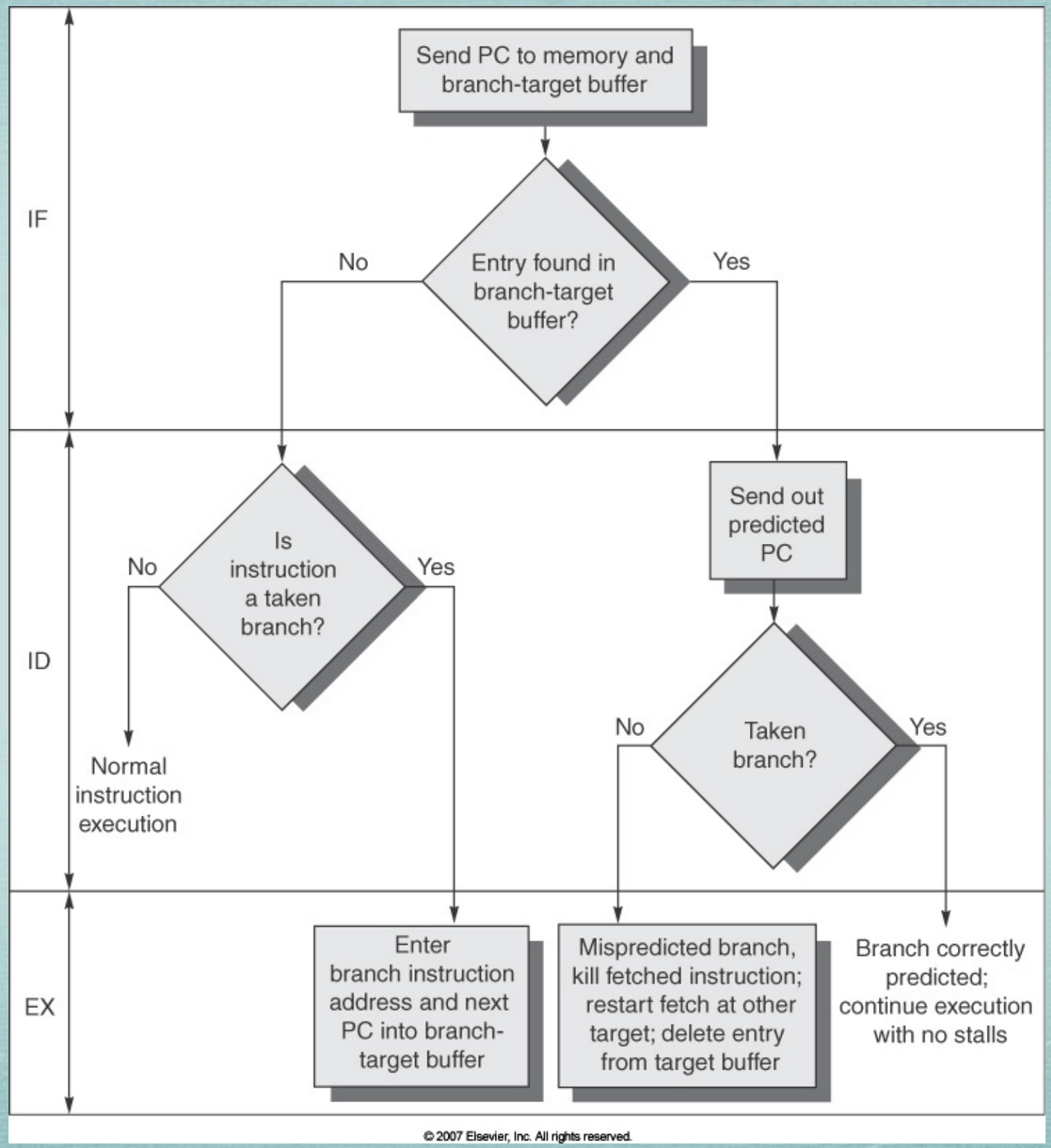
Compare with 14 and 19, without speculation

ADVANCED TECHNIQUES FOR INSTRUCTION DELIVERY AND SPECULATION

Increasing Fetch Bandwidth:
Branch Target Buffers



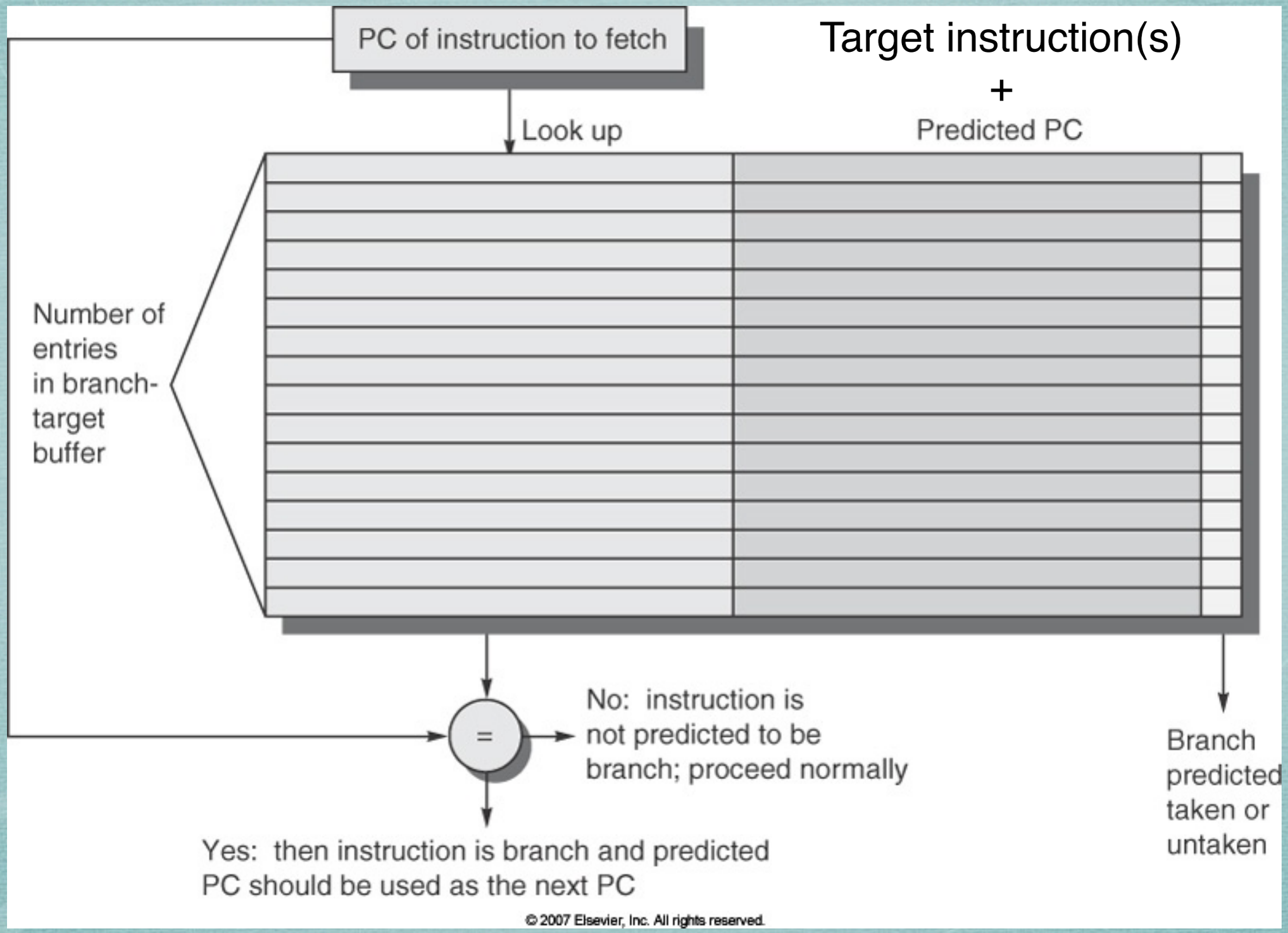
Increasing Fetch Bandwidth:
Branch Target Buffers



Increasing Fetch Bandwidth:
Branch Target Buffers

Instruction in buffer	Prediction	Actual branch	Penalty cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

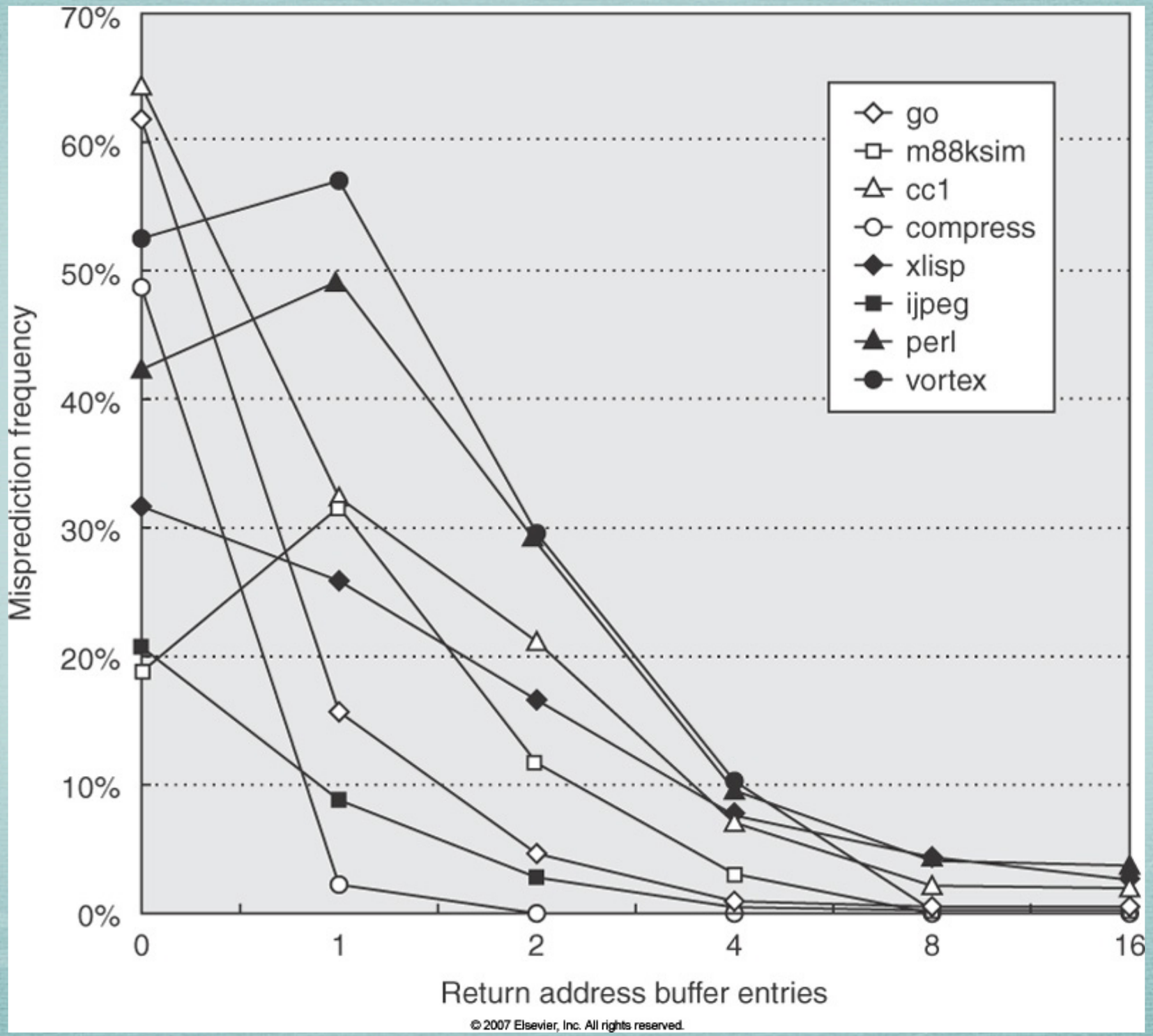
Increasing Fetch Bandwidth:
Branch Target Buffers: Variation



Increasing Fetch Bandwidth

- Return address predictors
  - ⋆ assuming a special instruction for return (not a jump)
  - ⋆ returns are indirect (why?)
  - ⋆ more important for OO and dynamic languages (esp. VMs)

Return Address Prediction Accuracy



Increasing Fetch Bandwidth

- Return address predictors
  - ⋆ assuming a special instruction for return (not a jump)
  - ⋆ returns are indirect (why?)
  - ⋆ more important for OO and dynamic languages (esp. VMs)
- Integrated (stand-alone) instruction fetch units (not just a pipeline stage)
  - ⋆ integrated branch prediction
  - ⋆ instruction prefetch (when might this be useful?)
  - ⋆ instruction memory access and buffering (e.g., trace cache on Pentium 4)

Improving Speculation: Register Renaming

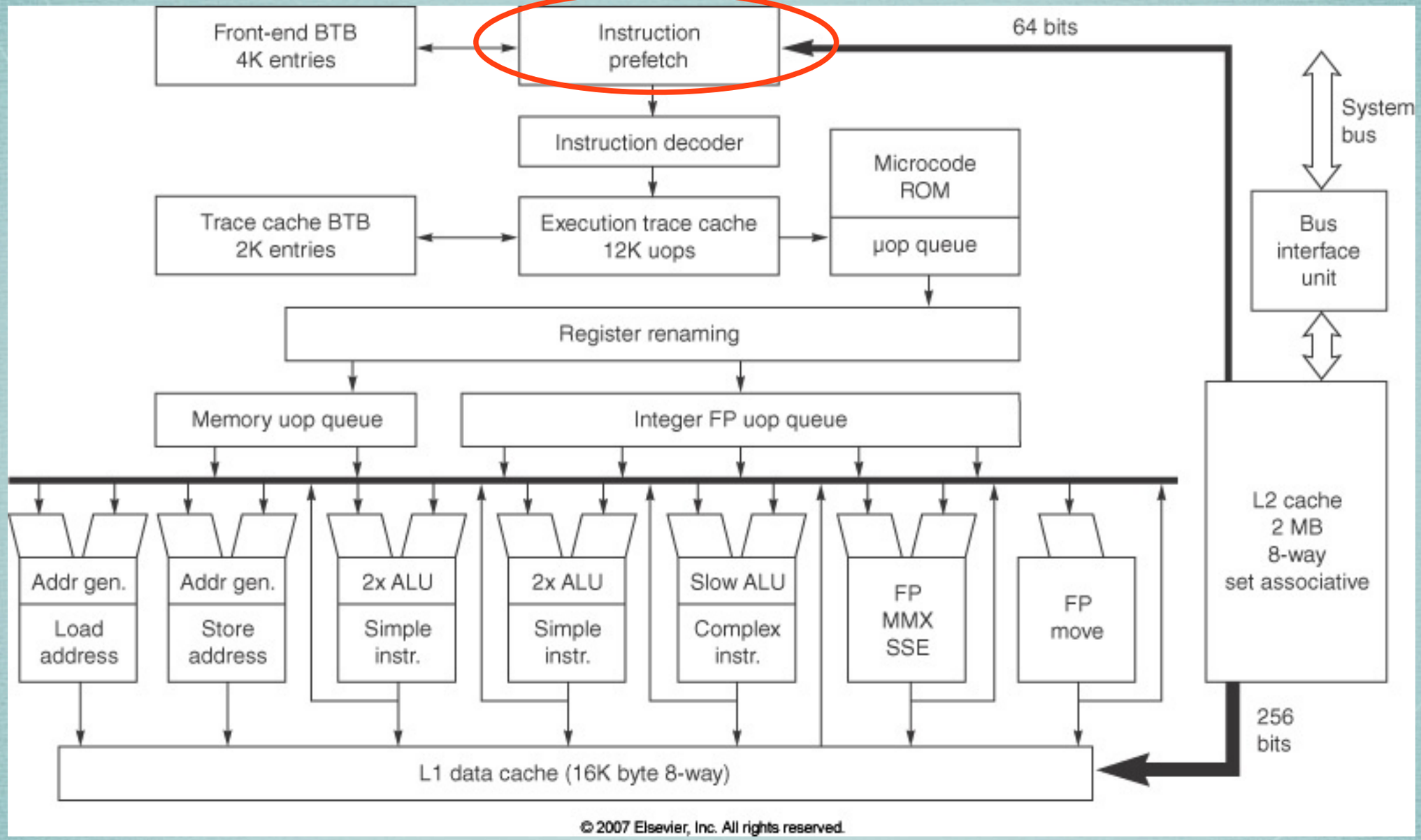
- Use an extended set of "physical" registers, instead of "architectural" registers
  - ⋆ beware of the terminology difference with memory system
- Physical registers hold values, instead of reservation stations or ROB
- Map of architectural to physical registers
- Commit steps:
  - ⋆ make the map entry for written architectural reg. "permanent"
  - ⋆ deallocate physical registers containing "older" value
- Deallocating physical registers
  - ⋆ look at the source operands to find unused physical registers
  - ⋆ wait until next instruction writing the same architectural register commits (how does this work?)

Improving Speculation

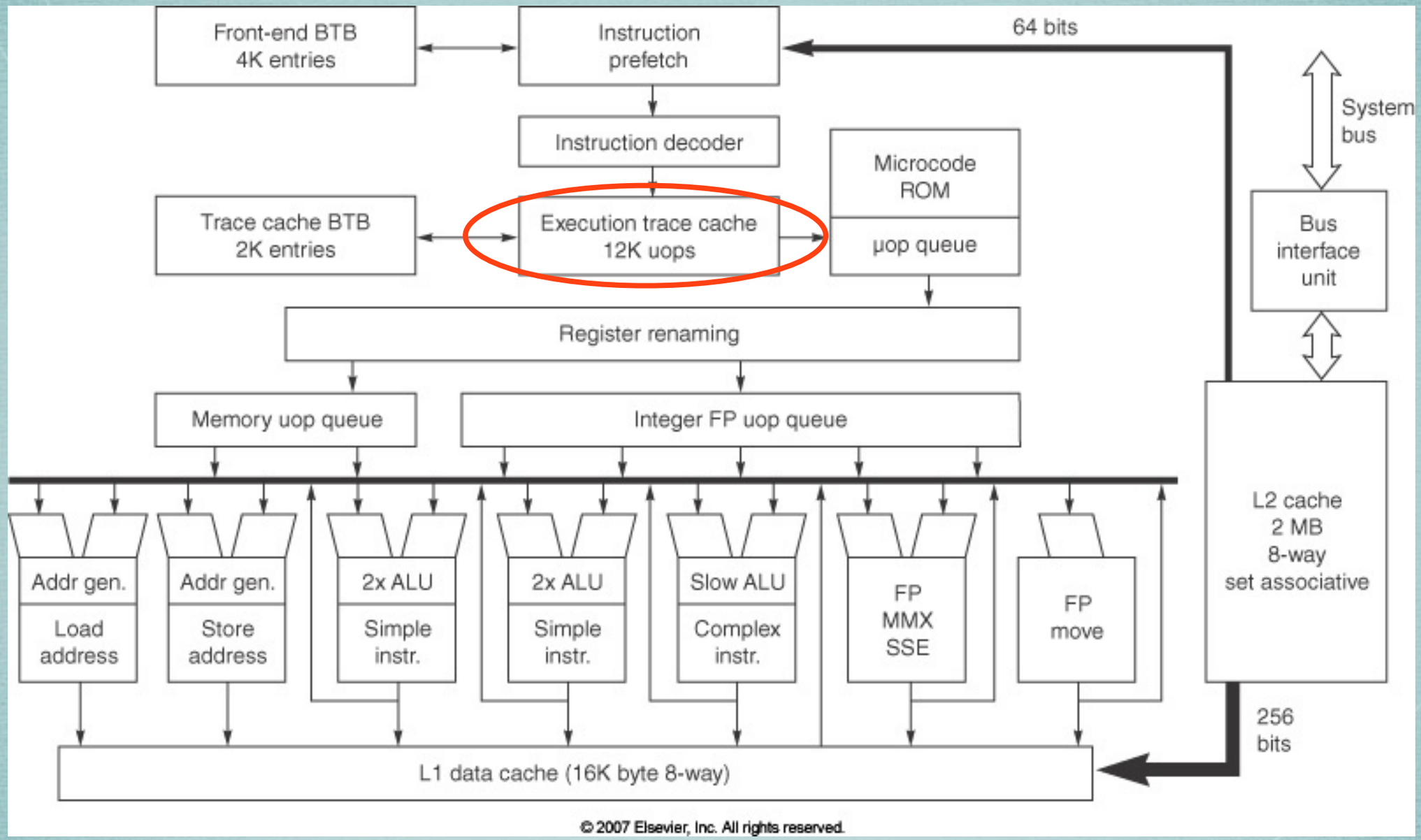
- Limiting speculation
  - ⋆ avoid speculating when it may cause an expensive exception, such as TLB miss or L2 miss
- Speculating through multiple branches
  - ⋆ speculate on a subsequent branch while the previous one still pending
    - ✳ useful when high branch frequency, clustered branches, or long functional unit delays
  - ⋆ speculating on more than branch in **one** cycle
    - ✳ no architecture combines multiple branch prediction in one cycle with full speculation
- Value prediction

INTEL PENTIUM 4

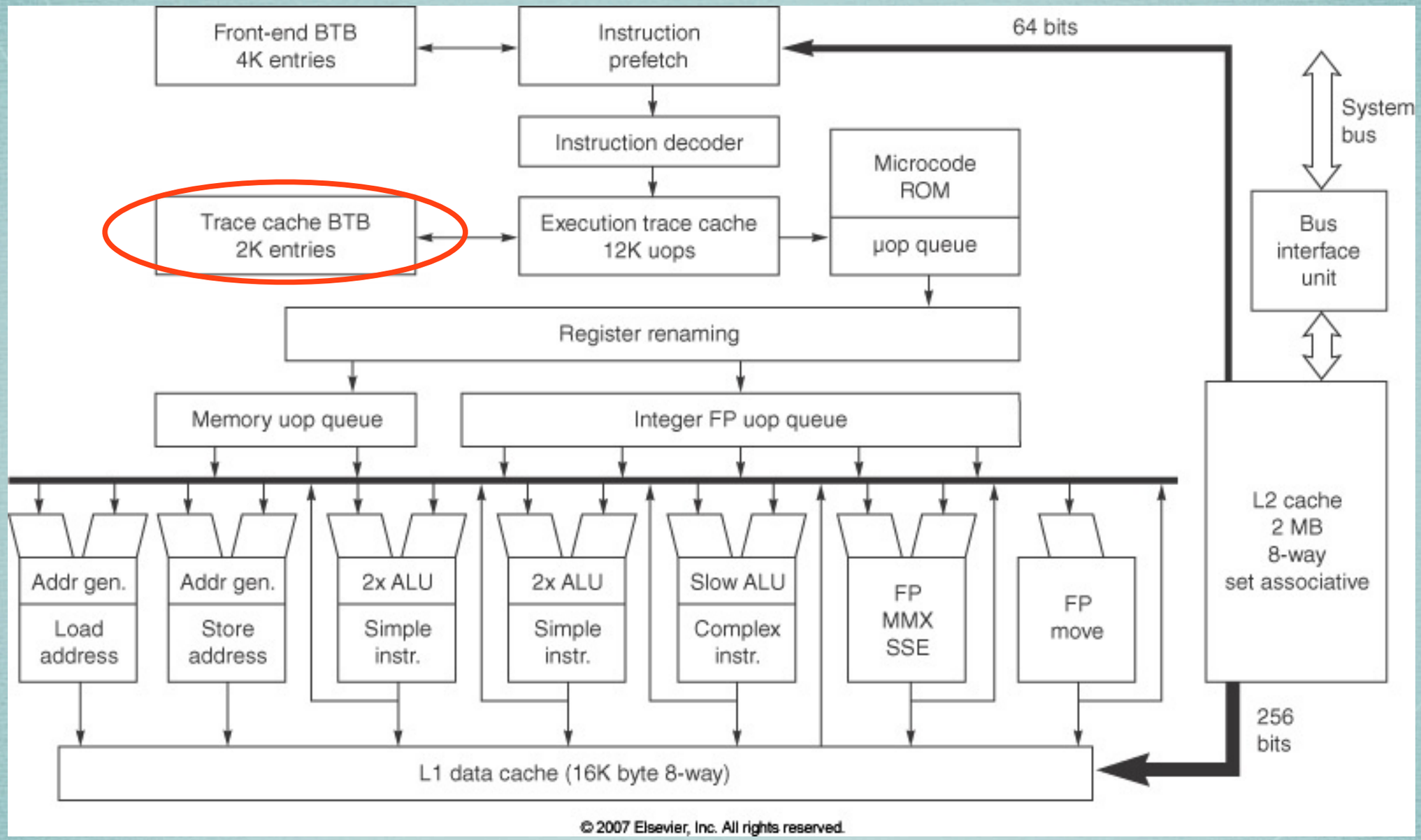
Overall Architecture



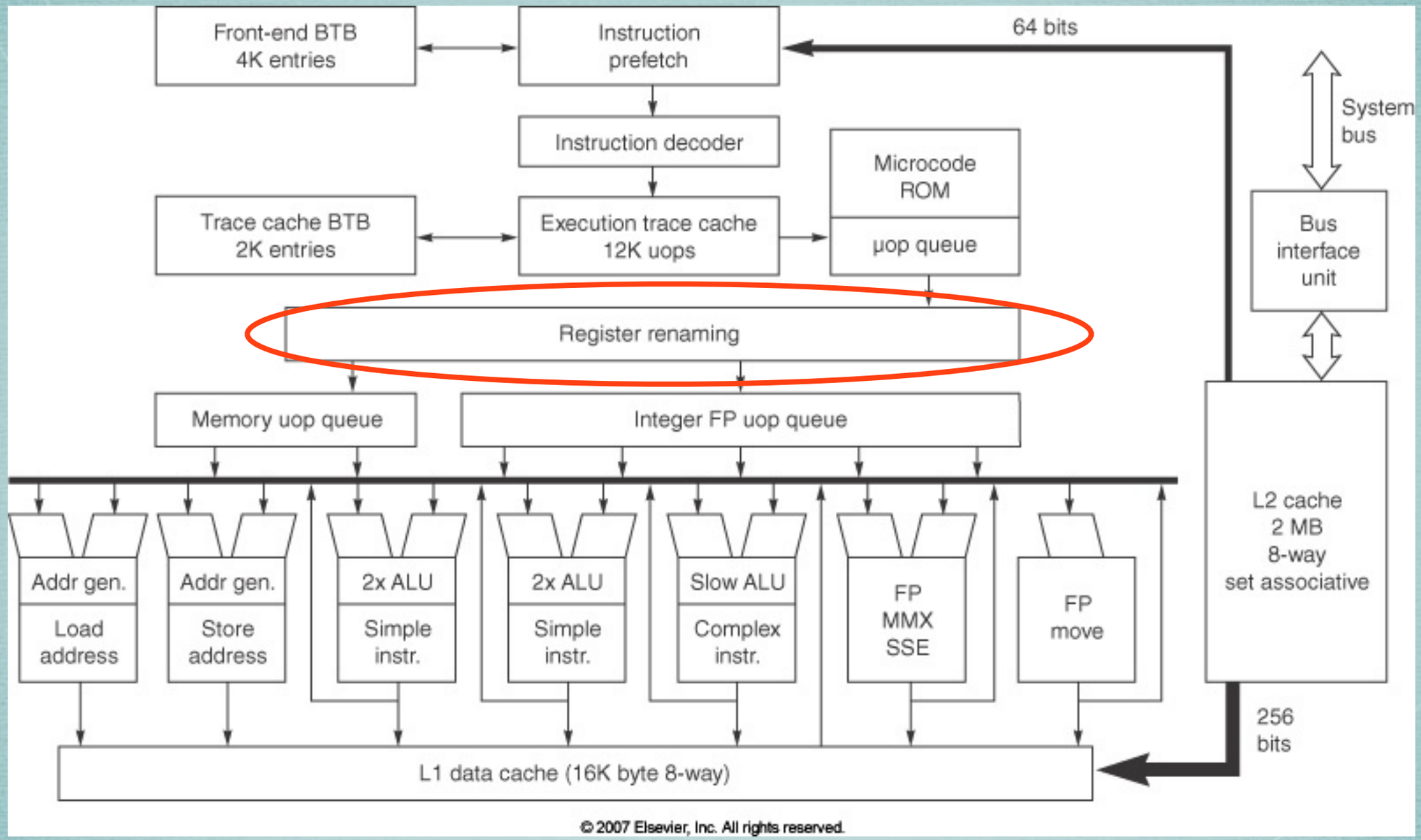
Overall Architecture



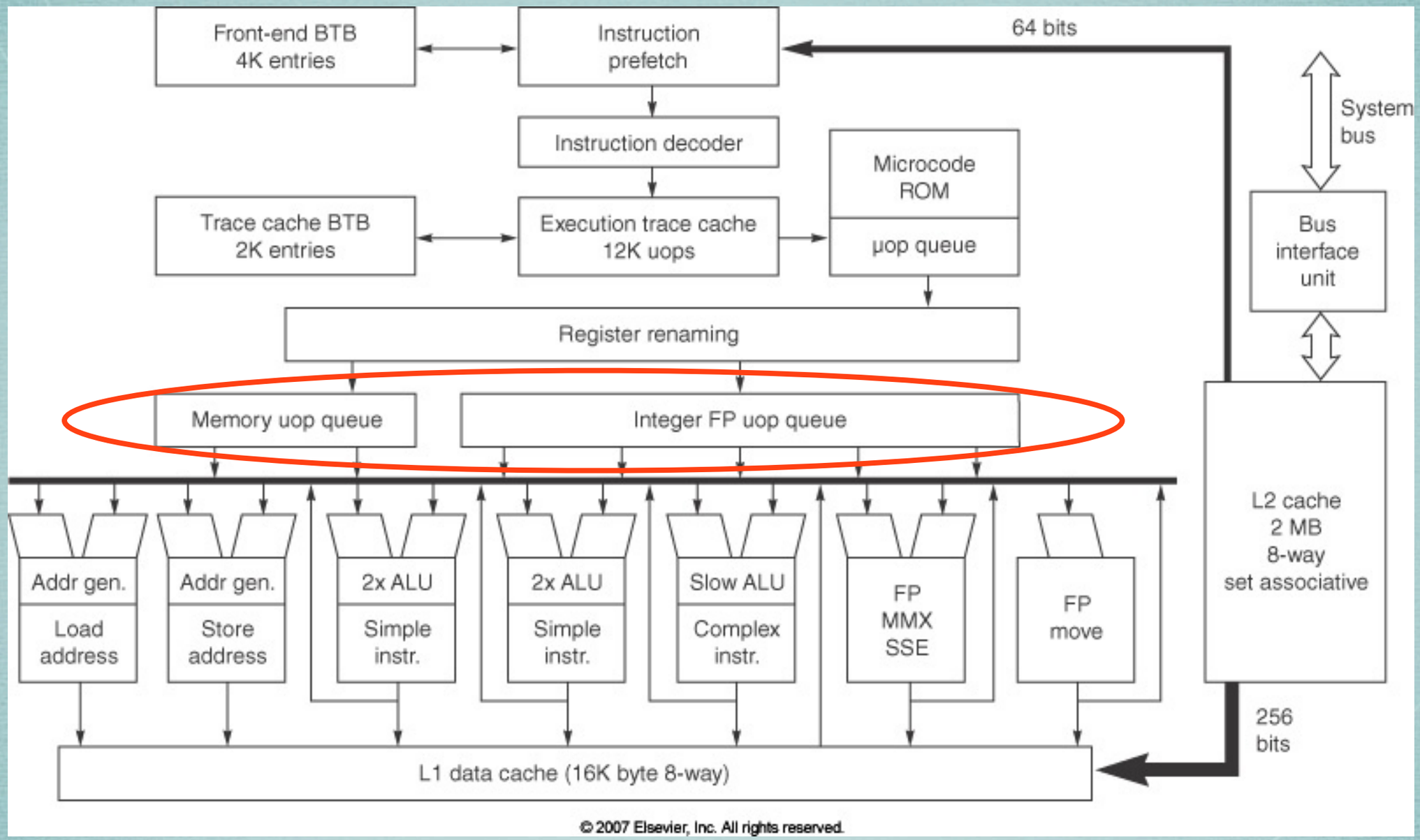
Overall Architecture



Overall Architecture



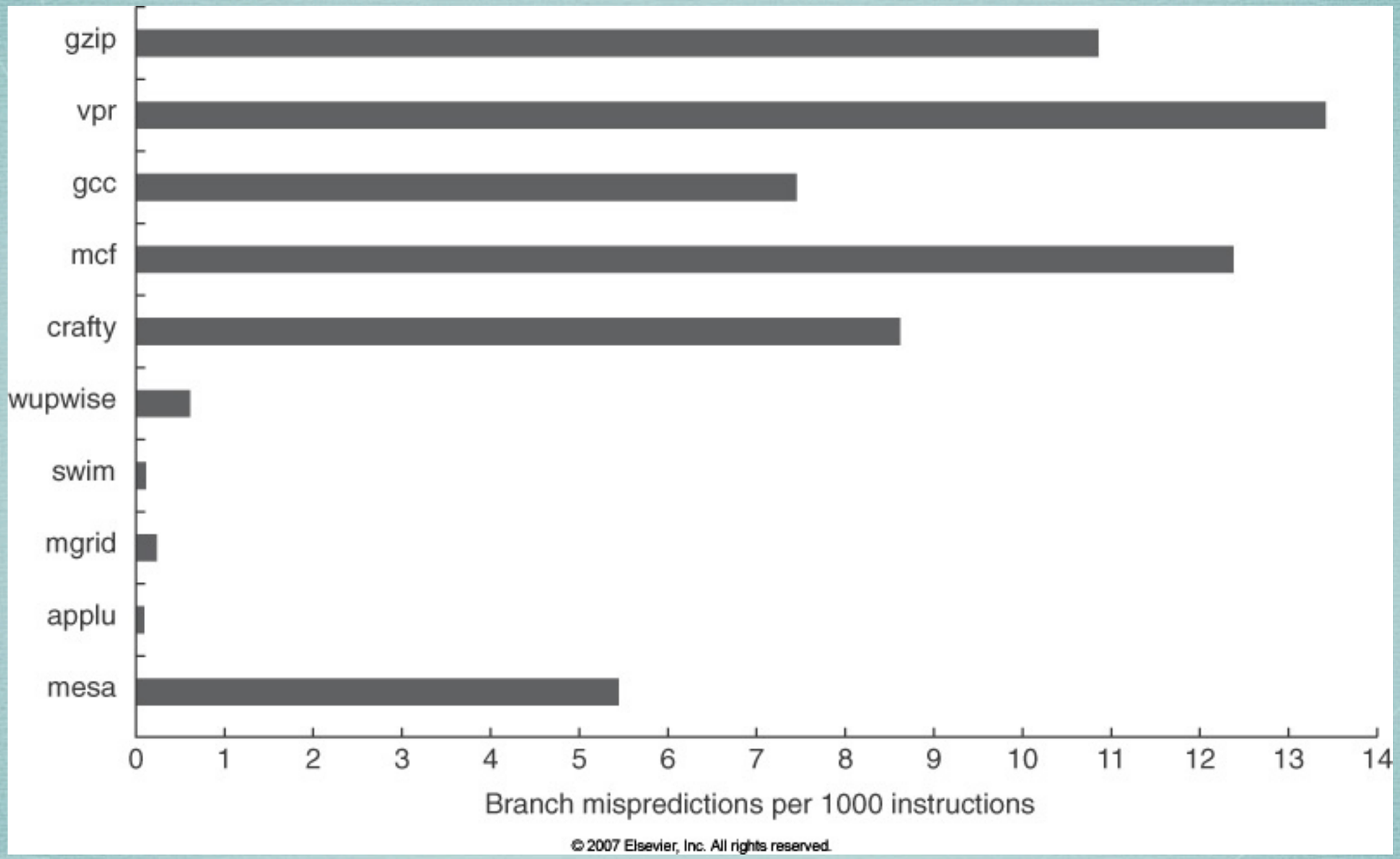
Overall Architecture



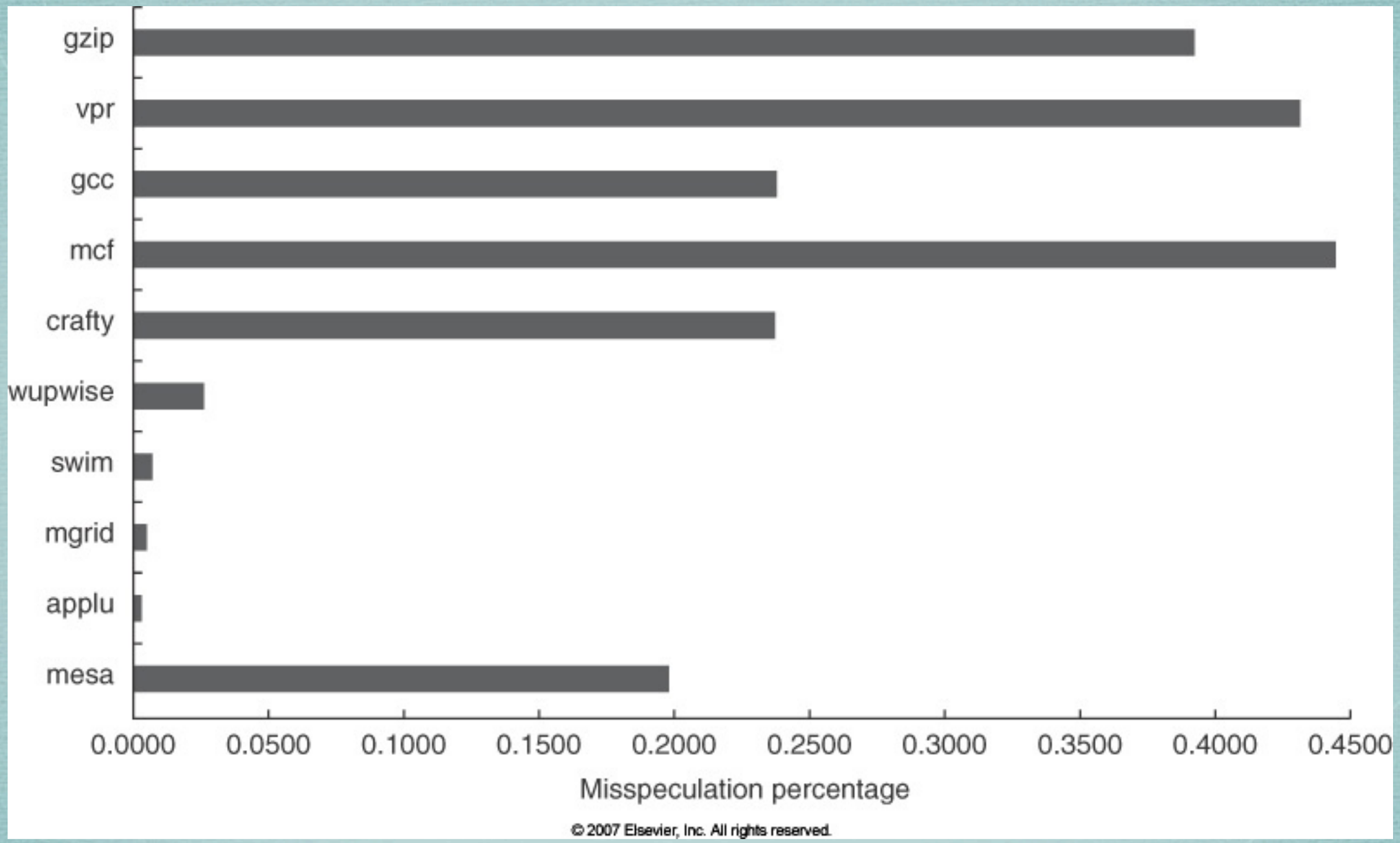
Characteristics

Feature	Size	Comments
Front-end branch-target buffer	4K entries	Predicts the next IA-32 instruction to fetch; used only when the execution trace cache misses.
Execution trace cache	12K uops	Trace cache used for uops.
Trace cache branch-target buffer	2K entries	Predicts the next uop.
Registers for renaming	128 total	128 uops can be in execution with up to 48 loads and 32 stores.
Functional units	7 total: 2 simple ALU, complex ALU, load, store, FP move, FP arithmetic	The simple ALU units run at twice the clock rate, accepting up to two simple ALU uops every clock cycle. This allows execution of two dependent ALU operations in a single clock cycle.
L1 data cache	16 KB; 8-way associative; 64-byte blocks write through	Integer load to use latency is 4 cycles; FP load to use latency is 12 cycles; up to 8 outstanding load misses.
L2 cache	2 MB; 8-way associative; 128-byte blocks write back	256 bits to L1, providing 108 GB/sec; 18-cycle access time; 64 bits to memory capable of 6.4 GB/sec. A miss in L2 does not cause an automatic update of L1.

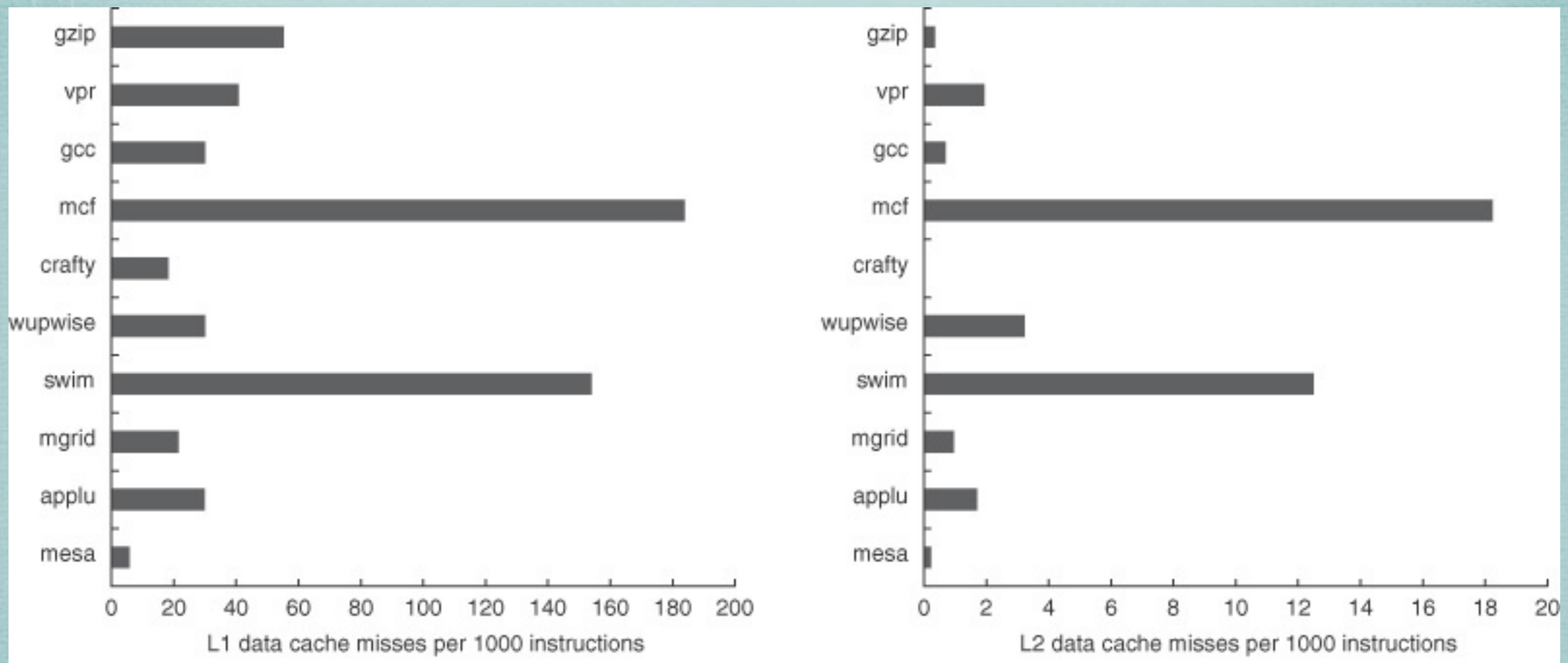
Branch Misprediction Rate



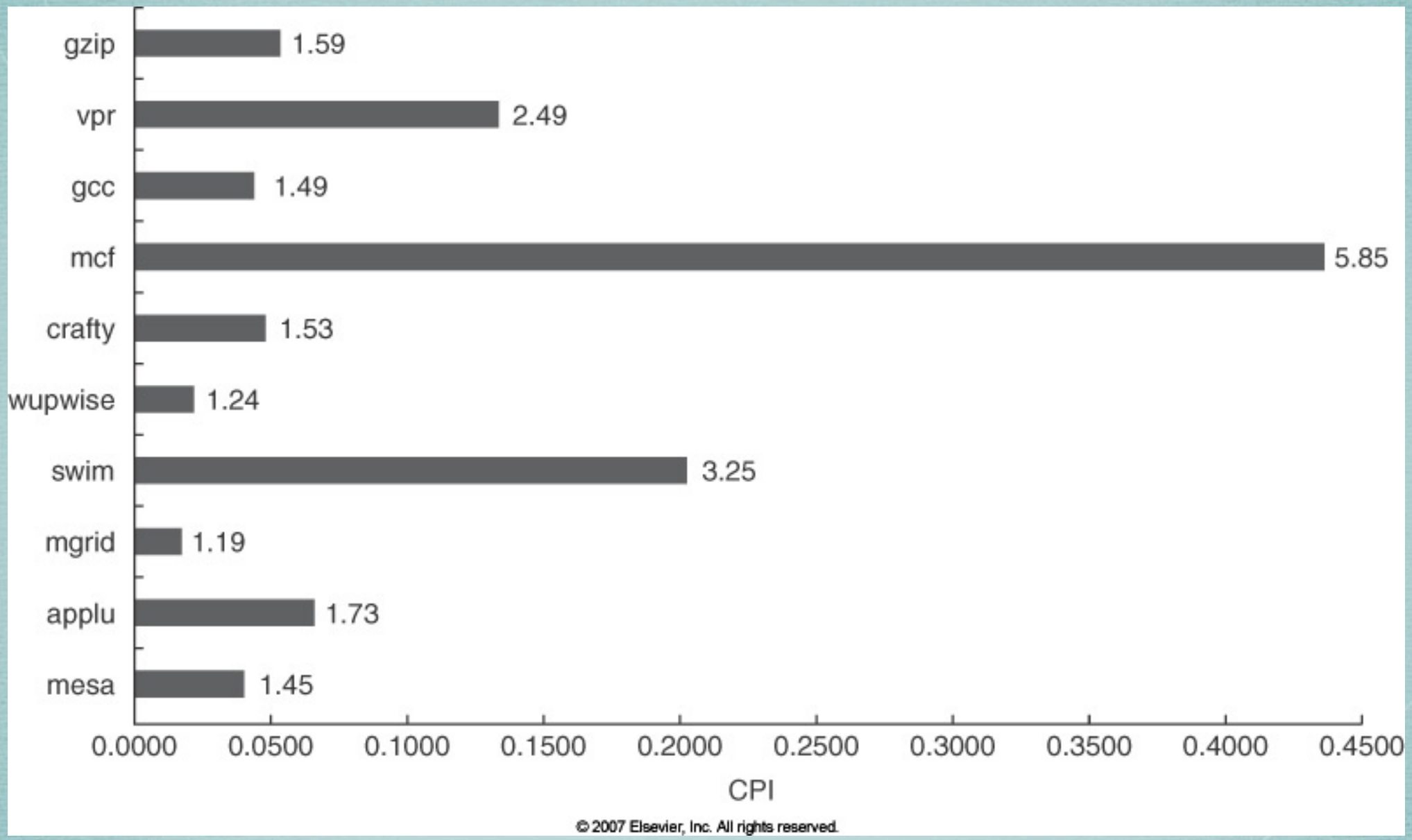
Percentage Mispredicted uops



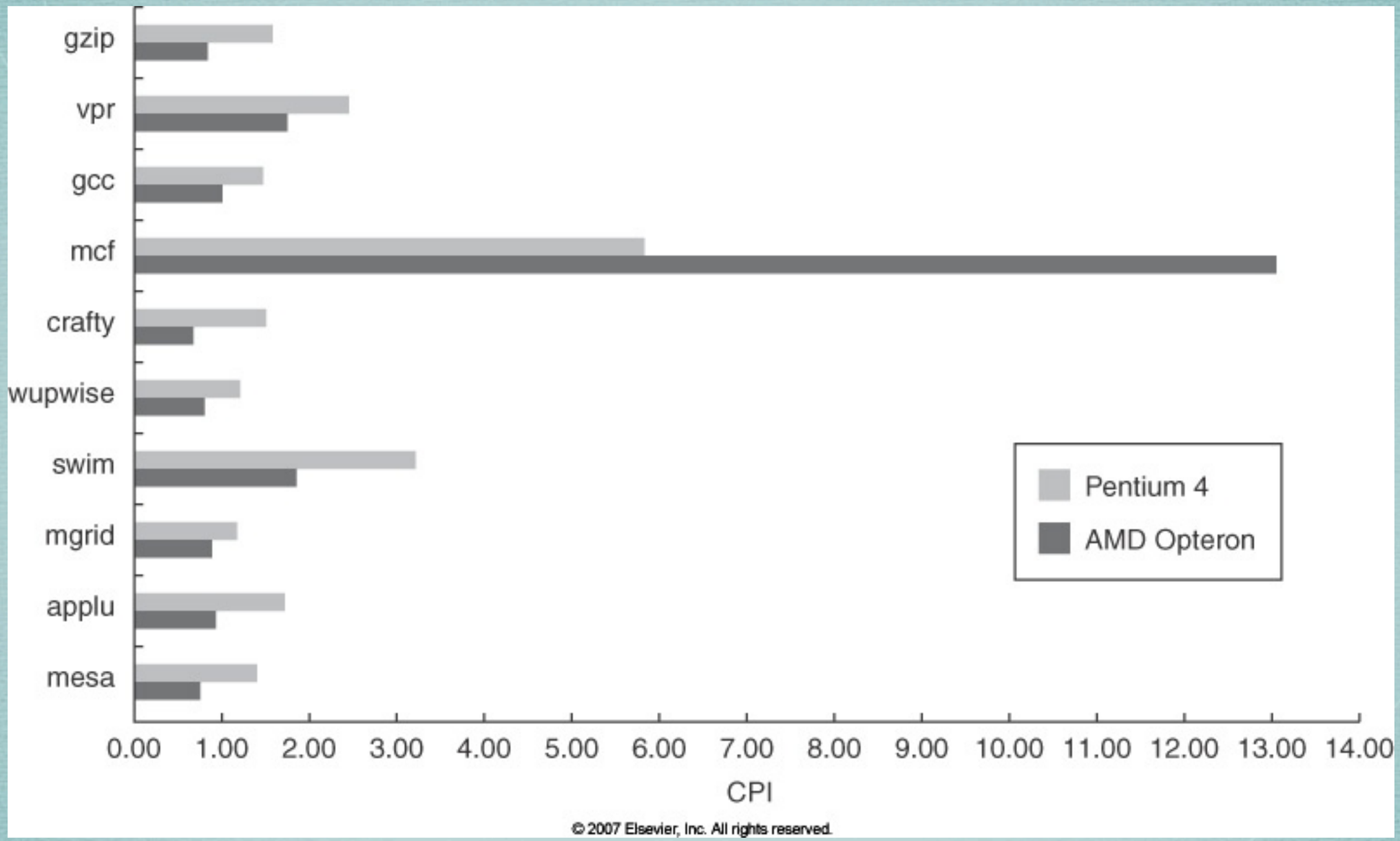
Data Cache Misses



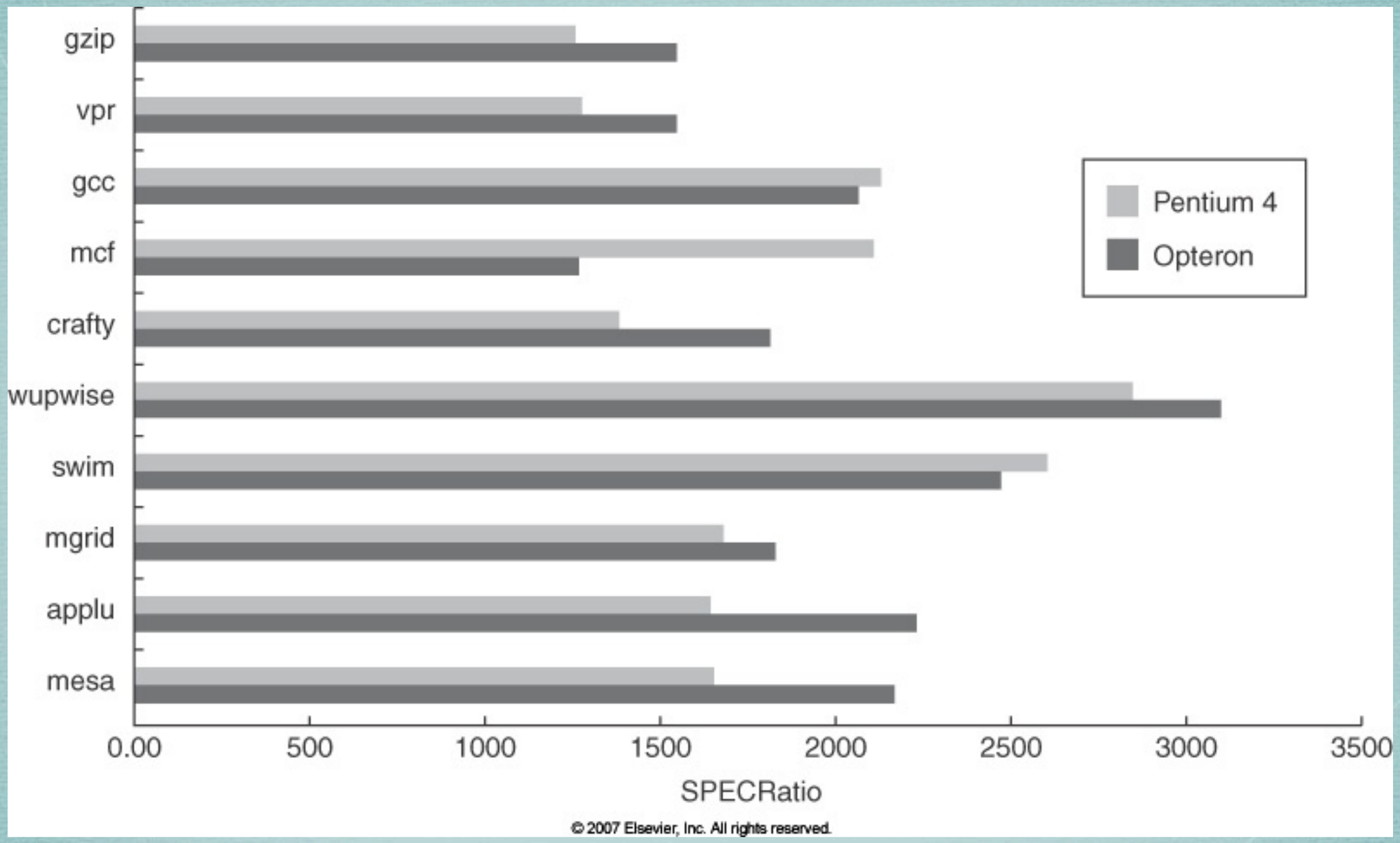
CPI



Intel Pentium 4 vs AMD Opteron: CPI

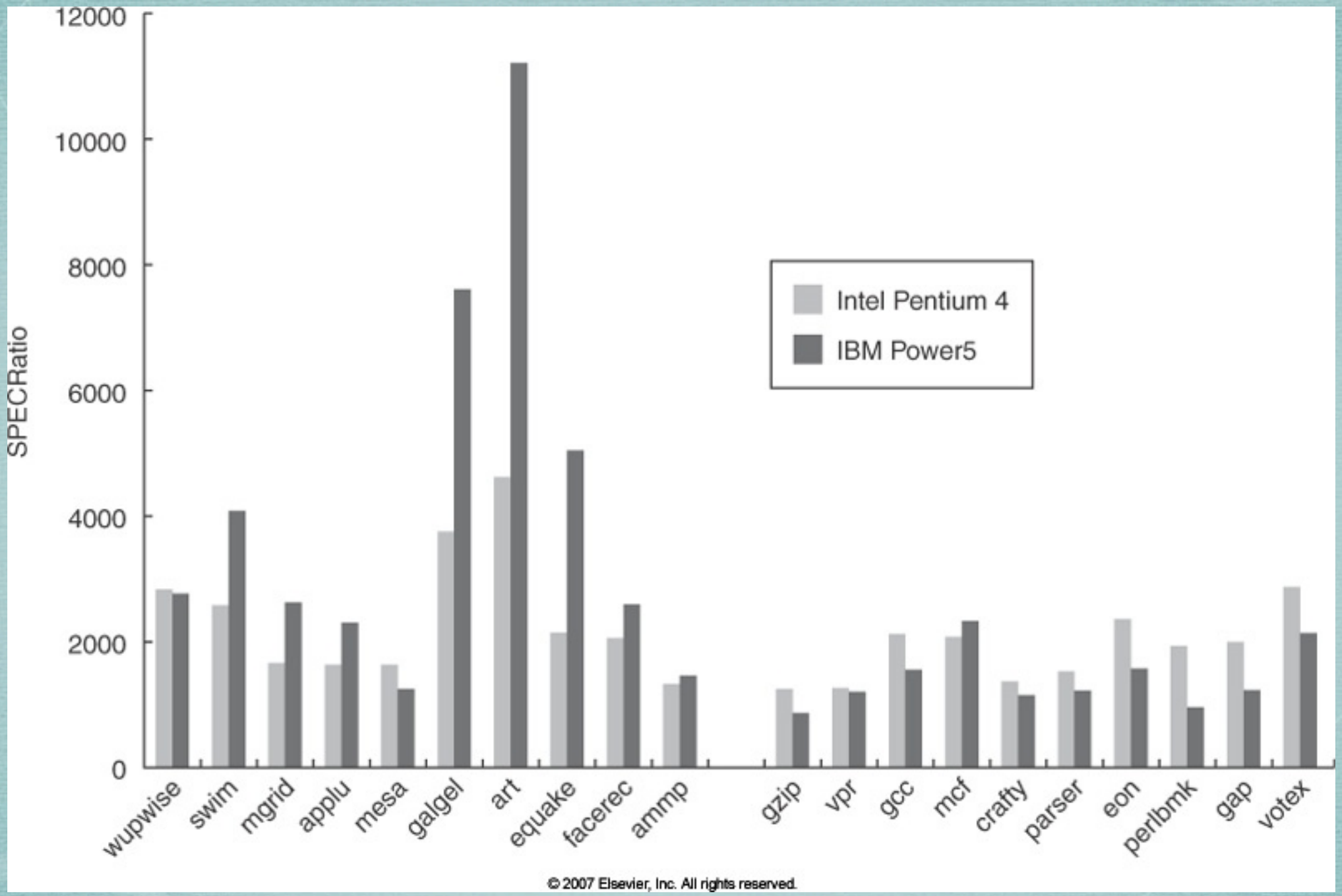


Intel Pentium 4 vs AMD Opteron: Performance



Pentium 4: 3.2 GHz
Opteron: 2.6 GHz

Intel Pentium 4 vs IBM Power5



Pentium 4: 3.2 GHz
Power5: 1.9 GHz

Recap

- Many advanced techniques on modern processors
  - ⋆ pipelining
  - ⋆ dynamic scheduling
  - ⋆ branch prediction
  - ⋆ speculation
  - ⋆ multiple issue
  - ⋆ branch target buffers
  - ⋆ register renaming
  - ⋆ ...
- Too much complexity ⇒ Multiple cores