Dynamic Instruction Level Parallelism (ILP)

- Introduction to ILP
- Data and name dependences and hazards
- CPU dynamic scheduling
- Branch prediction
- Hardware speculation
- Multiple issue
- Theoretical limits of dynamic ILP

Instruction Level Parallelism (ILP): Introduction

- Pipeling partially overlaps instructions execution, thus leveraging on the potential *parallelism* embedded in the instructions.
- This type of parallelism is named "**Instruction Level Parallelism**" (usually **ILP**).
- **Static ILP** places on the *compiler* the task of finding instructions that can be "overlapped" (a POE that gives a good ROE)
- **Dynamic ILP** places on the hardware all of the effort to extract parallelism from any POE

Instruction Level Parallelism (ILP)

- The actual amount of parallelism embedded in the instructions of a program that can be exploited depends on the pipelining issues examined before. Three are three types of conflicts:
- **structural**: limited number of available functional units
- **data**: an instruction has to wait for the outcome of another one, and the two instructions cannot proceed in parallel
- **control**: as long as the outcome of a branch is unknown, so is the next instruction to be executed

Instruction Level Parallelism (ILP)

- In literature, these conflicts are called "**hazard**" and they are responsible for pipeline stalls.
- Let us ignore these conflicts, and let us focus on the two basic ways to increase the level of parallelism embedded in instructions that can be exploited:

1. **Increasing the number of phases in the pipeline** (the so called pipeline **depth**)

- This increases the number of instruction whose execution can be potentially overlapped, and so the level of parallelism (*potentially*, since hazards are still there…)

Instruction Level Parallelism (ILP)

- For a fixed amount of work necessary to carry out an instruction, splitting the work in more phases (each performed by a pipeline stage) makes each phase shorter, so requiring a shorter clock cycle. Otherwise stated: **the CPU clock frequency can be raised**.
- Equivalently: for a given multicycle pipelined architecture, to increase instruction execution speed, it is possible to raise clock frequency.

Instruction Level Parallelism (ILP)

- Higher frequencies imply shorter clock cycles, which allow for less work; thus the architecture must be re-designed by breaking down work into a **larger number of phases**, each performing less activity …
- This technique has been heavily exploited recently, most notably in Pentium IV, that sported a peak frequency of 4 GHz with an almost 30-stage pipeline.

Instruction Level Parallelism (ILP)

- Intel CPUs evolution (Tanenbaum, Fig. 1.11):

Chip	Date	MHz	Transistors	Memory	Notes
4004	4/1971	0.108	2300	640	First microprocessor on a chip
8008	4/1972	0.108	3500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5–10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5–8	29,000	1 MB	Used in IBM PC
80286	2/1982	8–12	134,000	16 MB	Memory protection present
80386	10/1985	16–33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25–100	1.2M	4 GB	Built-in 8-KB cache memory
Pentium	3/1993	60–233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150–200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233–450	7.5M	4 GB	Pentium Pro plus MMX instructions
Pentium III	2/1999	650–1400	9.5M	4 GB	SSE Instructions for 3D graphics
Pentium 4	11/2000	1300–3800	42M	4 GB	Hyperthreading; more SSE instructions

Instruction Level Parallelism (ILP)

- Of course, this trend cannot continue indefinitely:
  - because of architectural problems: the more complex the pipeline, the more complex its control unit;
  - because of technological limitations: width of paths that interconnect transistors, paths interference, energy consumption and heat dissipation.
- The deeper the pipeline depth, the longer the time to complete a single instruction, the larger the number of instructions carried out in parallel (potentially…)

Instruction Level Parallelism (ILP)

- Instead of increasing pipeline depth (actually, when this option is no longer feasible), it is possible to replicate some of the processor functional units:

2. **issuing the execution of multiple instructions in parallel**, a technique commonly referred to as **"multiple issue"**.

- Multiple issue requires a "larger" datapath, capable of transferring from one pipeline stage to the following all informations associated to the instructions issued in parallel.

- And also more sophisticated functional units:

Instruction Level Parallelism (ILP)

a. There must be a number of funtional units for the parallel execution of instructions. As an example, ALU, integer/floating point multiplication, and so on.

b. It must be possible to fetch within each clock cycle more instructions from Instruction Memory, and multiple operands from Data Memory (cache memories, that store instructions and data, usually have a “bandwidth” taylored to this goal)

c. It must be possible to address in parallel multiple registers, and reading/writing the registers used by the different instructions in execution, in the same clock cycle.

Instruction Level Parallelism (ILP)

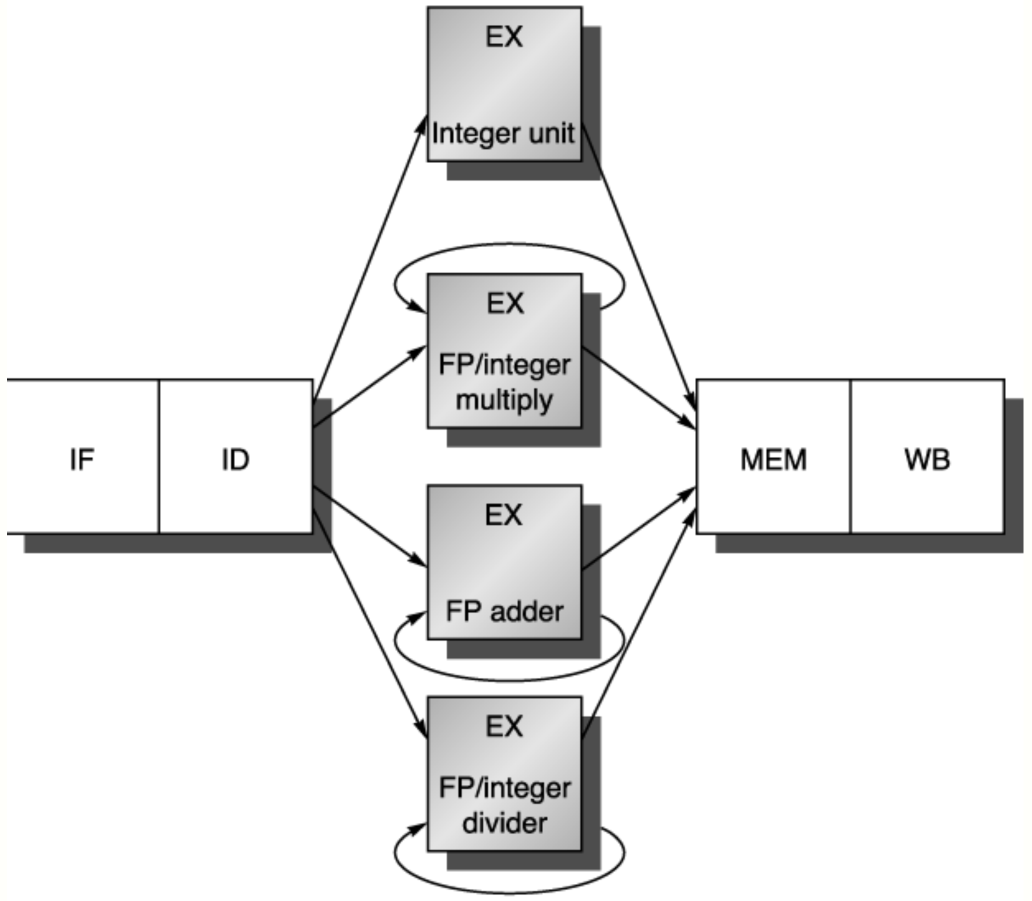
- In a simple pipelined architecture, with no dependences, the execution of a new instruction is completed at each new clock cycle:
  Clock Per Instruction (CPI) = 1

- In multiple issue pipelined architecture, it is potentially possible to complete the execution of more instructions per clock cycle, thus CPI < 1 (hazards make things a bit difficult, in reality…)

- Multiple issue architectures are usually refereed to as "**superscalar**", even though this name should be reserved to "dynamic" multiple issue architectures (to be discussed shortly…)

Instruction Level Parallelism (ILP)

- The basic scheme of a superscalar architecture. Actually, pipeline stages are more than 5, but instruction execution can be split into 5 main phases (Hennessy-Patterson, fig. A.29)



Instruction Level Parallelism (ILP)

- Modern high level processors combine pipelining with multiple issue, and can issue 3 to 8 instructions per clock cycle.
- As an instance, a 3 GHz processor capable of issuing up to 4 instructions per clock cycle would sport a peak execution speed of 12 billion instructions per second, and a CPI equal to 0,25.
- Actually, pipeline stalls limit heavily these performances, and even guaranteeing CPI = 1 is often very difficult…

Instruction Level Parallelism (ILP)

- To implement multiple issues, two basic problems must be tackled:

a. Estabilishing which and how many instructions can be sent to execution in a given clock cycle. Usually, the selected instructions are assembled in an **issue packet**, and issued in the same **issue slot**.

b. Solving possible structural hazards, both on data and on control.

- Multiple issue processors can be split into two large categories, according to how (and when) these two problems are solved.

Instruction Level Parallelism (ILP)

- In **static multiple issue** processors (to be discussed in the next chaporter), it is the **compiler** (software…) that chooses the instructions to be issued in parallel (which informations are required by the compiler?)
- When the processor fetches from IM an "instruction packet", it knows already that they can be executed in parallel without conflicts from hazards among the instructions.
- The number of instructions in a packet is set a-priori during processor design, so these are **static issue** architectures.
- When most activity to extract parallelism among instructions is carried out by the compiler, one speaks of **Static Instruction Level Parallelism**

Instruction Level Parallelism (ILP)

- In **dynamic multiple issue** processors (discussed in this chapter) it is the processor that estabilished "at run time" which instructions to issue, and how to solve conflicts from hazards.
- The processor decides on the spot even the number of instructions to issue (with a maximum depending on the specific architecture): these are **dynamic issue** architectures and the scheme is called **dynamic Instruction Level Parallelism**
- The two approaches are not completely distinct, in real cases, and often features from each of them are actually exploited toghether.

Instruction Level Parallelism (ILP)

- Almost all "general purpose" processors (both old ones, such as Pentium, PowerPC, SPARC, MIPS, and more recent multi-core ones) basically adopt dynamic ILP.
- In dynamic ILP, instructions are fetched in the order produced by the compiler, but many architectures implement some form of **pipeline dynamic scheduling**: the issue order (that is, the order by which instructions are forwarded to the EX phase in the pipeline) can be changed, to minimize the effects of dependences.
- Warning: to fully understand this mechanism, we shall cover first the *single issue* case (one instruction per clock cycle), and we will revert to multiple issue later.

Instruction Level Parallelism (ILP)

- Let us consider this code fragment, supposing that the value addressed by 100(R2) is not in the cache memory:

```
LD      F0, 100(R2)
FADD F10, F0, F8
SUB    R12, R8, R1    // what if it were: "FSUB F12, F8, F1" ?
```

- The execution of FADD depends on LD, but in a standard pipeline (usually referred to as **statically scheduled pipeline**) instructions flow one after another in the pipeline: until FADD can be forwared to the EX phase, it blocks the execution of DSUB, which could be executed, since it has no dependences from preceding instructions.

Instruction Level Parallelism (ILP)

- Instead, processors with a **dynamically scheduled pipeline** are capable of detecting that DSUB does not depend on preceding instructions, and can forward it to the esecution to an integer unit, while FADD is still waiting for the value in F0.
- Instructions can overrun each other within the pipeline and this implies not only an **out-of-order execution**, but also their **out-of-order completion**.
- Otherwise stated, instructions can execute the MEM and WB phases in an order different from that by which they are ordered in the program.

Instruction Level Parallelism (ILP)

- In a system working with static ILP, the compiler could detect such a potential stall, and could generate code in which the DSUB is moved before the LD.
- With static ILP, the compiler carries out this and more complex code movements, to produce a more efficient execution.
- Obvioulsy, all these transformations must preserve the global behavior of the program (*program semantics*)

Instruction Level Parallelism (ILP)

- In dynamic ILP, some form of **dynamic branch prediction** is customary: for each executed branch instruction, the CPU hardware records the outcomes of preceding executions (that is, whether that branch was taken or not).
- When the same branch is executed, past history of that branch is used to predict if the brach will be taken, and so choosing the next instruction to execute.
- Obviously, this techniques works only for branches that do *have an history*, namely, those that are executed more than once

Instruction Level Parallelism (ILP)

- A natural extension of branch prediction is **hardware speculation**. Let us consider this situation:

```
        FMUL   F4, F0, F2               // may need 10 clock cycles
        BLE    F4, #0.66666, next      // branch if less or equal
        FADD   F1, F1, #0.5
        ADD    R1, R1, #2
        FADD   F2, F2, #0.25
        ADD    R2, R2, #1
next:
```

Instruction Level Parallelism (ILP)

- The instructions controlled by the branch can be (possibly) executed only when the outcome of BLE is known, which in turn depends on the result of FMUL.
- The processor can **speculate** on the outcome of the branch, on the basis of some form of branch prediction, and if the prediction is for a not-taken branch, it could issue the instructions within the highlighted block.
- Of course, the prediction can be wrong, and the processor must be capable of undoing the effects of the executed instructions, which should not have been issued.
- Speculation requires a very sophisticated hardware, and it is seldom applied in processors supporting dynamic ILP.

Dependences and hazards

- Let us start discussing **dynamic ILP** by better formalizing the notion of **dependence among instructions**, which is the primary effect in limiting the exploitation of the parallelism embedded in the instructions of a program.

- This discussion will be useful even when we will cover **static ILP**, where it is the compiler that has the task of understanding which instructions can be executed in parallel, being independent of one another, and which cannot.

- In dynamic ILP instead, it is the processor that must discover possible dependences among instructions, right at the issue moment, and must be capable of handling them.

Dependences and hazards

- **Dependences** are a feature of *programs* (algorithms) and have no relationship whatsoever with hardware.
- **Hazards** are a feature of *hardware*, specifically of any *microarchitecture* for a given ISA.
- A certain set of dependences can give rise to hazards on a given microarchitecture and to NO hazards on a different microarchitecture.

Dependences and hazards

- Establishing how an instruction depends on another is instrumental to assessing the amount of **parallelism** in a program, and how it can be exploited.
- **If two instructions are independent**, they can be executed in parallel and/or in any order in the pipeline, if there are enough resources (that is: enough functional units)
- **If two instructions are dependent**, they must be executed in order, and can be executed in overlapped way only to a certain degree.
- So, to exploit ILP, it is mandatory to establish instructions dependences.

Dependences

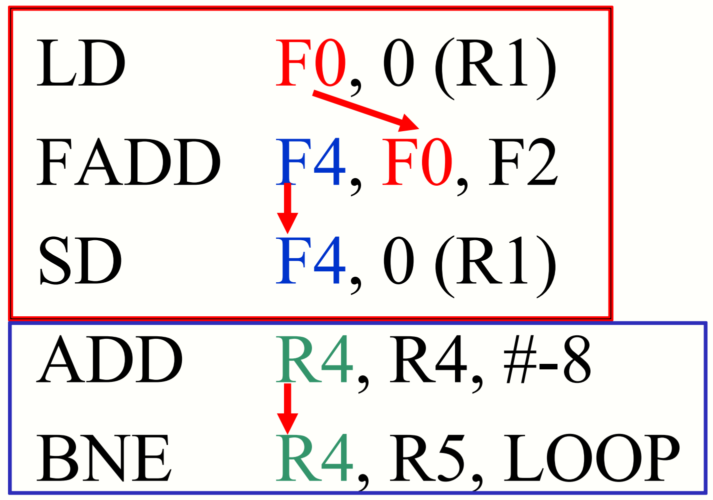
- Let us examine dependences that can exist in a group of consecutive instructions that *do not contain branches,* in the following *"linear segment"*.
- Alternatively, we might assume that the CPU always knows if the branch will be taken (using some form of prediction) that is to say, we disregard control dependences, so far.
- There are two types of dependences:

1. data dependences
2. name dependences

Data dependences

- Instruction *j* is (truly) **data dependent** on instruction *i* if at least one of the following conditions holds:
  1. *i* produces a value that can be used by *j*
  2. *j* is data dependent on *k*, and *k* is data dependent on *i*
- Condition 2 simply states that there can be an arbitrary long chain of dependences.
- *i* is the *producer*, *j* is the *consumer*

Data dependences



```
LOOP:     LD     F0, 0 (R1)
          FADD   F4, F0, F2
          SD     F4, 0 (R1)
          ADD    R4, R4, #-8
          BNE    R4, R5, LOOP
```

- The second instruction is data dependent on the first, so is the third on the second, and the fifth on the fourth.
- The arrows show the order of execution of instructions, which can cause stalls in the pipeline.

Name dependences

- A **name dependence** exists when two instructions use the same register or memory location, but there is no transmission of data between instructions using the same name (instructions are not *data dependent*)

- If instruction *I precedes* instruction *j,* there is:

1. an **antidependence** between *i* and *j* if *j* writes in a register (or memory location) read by *i*. The initial order must be maintained for *i* to read the correct value;

2. a **output dependence** if *i* and *j* write in the same register (or memory location). The order must be preserved for the final value to be that written by instruction *j*.

Name dependences

- Here are some examples for antidependence and output dependence:

```
FDIV  F0, F2, F4
FADD  F6, F0, F8
FSUB  F8, F10, F14
FMUL  F6, F10, F12
```

- **Antidependence** between FADD (waiting for FDIV) that reads F8, and FSUB that wants to write F8. What happens if FSUB modifies F8 before FADD has used it for the sum?
- **Output dependence** between FADD and FMUL: what happens if FADD (because of FDIV) completes after FMUL?

Eliminating name dependences

- A name dependence is not a true data dependence, since there are no values transmitted through instructions (this is why we say *true* data dependence).
- Instructions involved in a name dependence can be executed concurrently if the register name (or the memory address) is changed, to avoid conflicts (this is much easier with registers, an advantage of RISC instructions).
- Renaming can be done statically by the compiler, or dynamically by the hardware.

Eliminating name dependences

- Here is an example on the previous code fragment, to remove antidependences and output dependences with proper register renaming:

```
FDIV   F0, F2, F4                FDIV   F0, F2, F4
FADD F6, F0, F8                  FADD F6, F0, F8
FSUB  F8, F10, F14               FSUB  F9, F10, F14
FMUL F6, F10, F12                FMUL F11, F10, F12
```

- Obviously, for renaming to be possible, both F11 and F9 must be available and free. These changes must be accounted for in instructions following those just modified.

- What about resorting to some invisible (at the ISA level) CPU register ?

Dependences and hazards

- The different problems caused by data and name dependences can be categorized as follows. Let *i* and *j* be two instructions, where *i* occurs *before j*. There can be three types of **data hazards**:

1. **RAW (Read After Write)**: *j* tries to read a register (but even a memory location) before *i* writes it. *j* reads in errors the old, stale value.

– It is due to a *true data dependence*, and it is by large the most common. Instructions order has to be preserved so that j receives the correct value from *i*.

– Example: **LD R2, #100(R3)** followed by **ADD R5,R4,R2**. The pipeline structure can produce a RAW hazard.

Dependences and hazards

- Basically, with a RAW instruction *j* cannot procede its execution because the data it requires are not yet available.
- The processor must detect the dependence of *j* from a previous instruction and withhold execution until the missing data is available.
- *Forwarding* can solve some RAW hazards:

```
ADD    R1, R2, R3
SUB    R4, R1, R5
AND    R6, R1, R7
```

- but some remain:

Dependences and hazards

- especially when "long lasting" instructions are involved":

LD	R1, 0(R2)	*The data item addressed by LD will be present in MEM/WB register only at the endo of clock cycle 4, but SUB requires it at the beginnign of that cycle.*
SUB	R4, R1, R5	
AND	R6, R1, R7	

- In a processor with a **statically scheduled pipeline**, it is necessary to suspend SUB (and so AND too) for a clock cycle, waiting for the data from LD to be available (assuming the data are in the cache, otherwise…)

- This effect can be obtained by repeating the ID phase of SUB (it is decoded again, and the registers are read again) and the IF phase of AND (fetched again from IM)

Dependences and hazards

2. **WAW (Write After Write)**: *j* tries to write a register before it is written by *i*. The double write terminates in the wrong order, leaving in the register the value produced by *i* instead of that produced by *j*.

– It is due to and *output dependence*, and only happens in pipelines where an instruction can proceed even if a previous one is stalled.

– Example: in a pipeline with a multi-stage floating point unit, a simple case of WAW hazard:

FDIV	F0, F2, F4	
FADD	**F6**, F0, F8	**stalled** by FDIV because of RAW on F0
FSUB	F8, F10, F14	**proceeds** (is this possible ?)
FMUL	**F6**, F10, F12	

How can this case be solved?

Dependences and hazards

2. **WAW (Write After Write)**.

– Another example: in a pipeline with a multi-stage floating point unit, a simple case of WAW hazard:

```
FMUL   F4, F5, F6
LD     F5, #0 (R7)
FADD   F4, F7, F10
SD     F5, #0 (R8)
```

– What is the meaning of this piece of code ?

Dependences and hazards

2. **WAW (Write After Write)**:

Example: in a pipeline with a multi-stage floating point unit, a simple case of WAW (is this a hazard ?):

```
LD      F4, #0 (R7)        a long lasting instruction
SD      F7, #0 (R8)
FMUL    F4, F5, F6
```

Dependences and hazards

3. **WAR (Write After Read)**: *j* tries to write a register before it is read by *i*. *i* reads in error the new value.

(i) FADD F6, F0, **F8**	FADD **F6**, F0, **F8**
(j) FSUB **F8**, F10, F14	FSUB **F9**, F10, F14

– It is due to an *antidependence*, and it seldom ensues, since in most pipelines, operands are read (stage ID) "long" before being written (stage WB).

– WAR hazards can arise if instructions can write results in some early stage of the pipeline and can read operands in final stages. WAR hazards are also possible in instructions are re-ordered (try to build a case for a WAR hazard).

- What about RAR hazards?

Data and name hazards reduction through dynamic scheduling

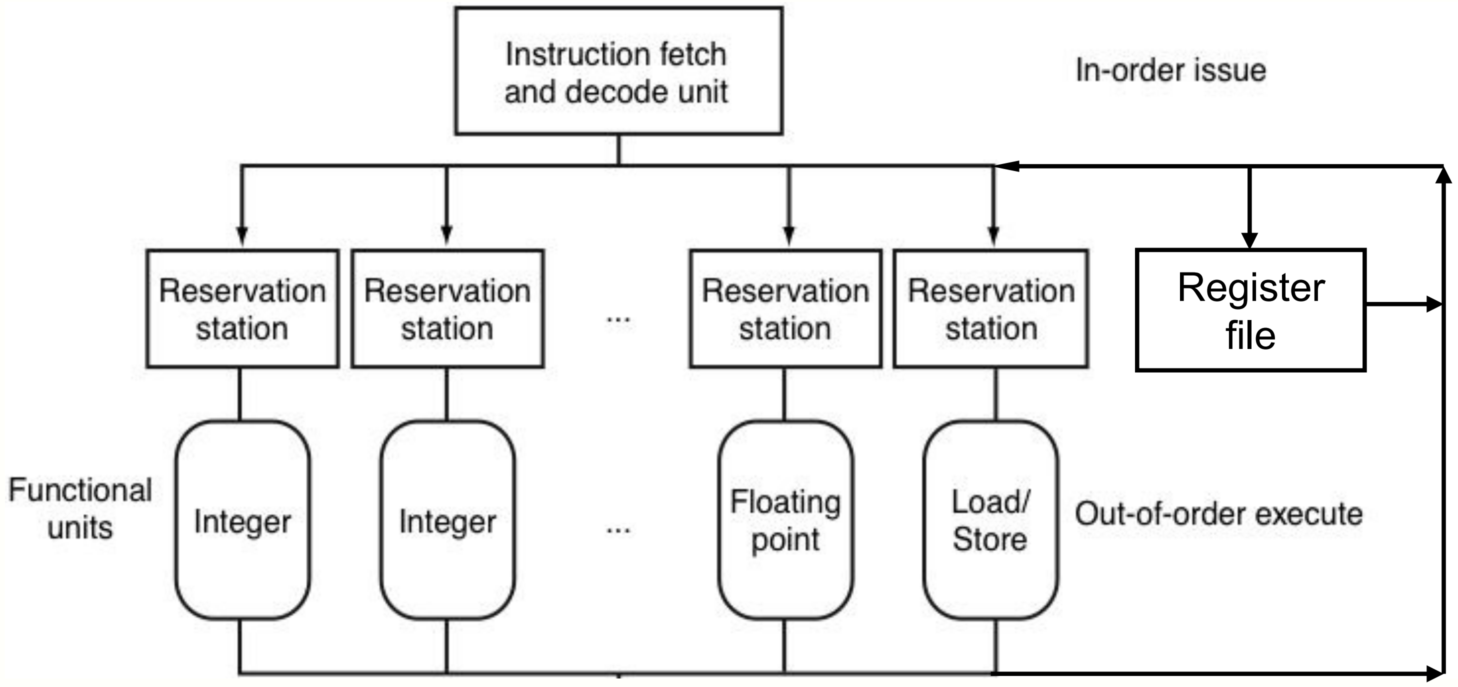
- **Pipeline dynamic scheduling** is the set of CPU run-time hardware techniques that help reduce the frequency and the length of pipeline stalls, **including possibly a change in the order of instructions**, so as to minimize RAW hazards effects.
- Pipeline dynamic scheduling includes **register renaming** (at instructions execution time) to reduce effects by WAW and WAR hazards.

Dynamic scheduling: Tomasulo’s approach

- The basic technique for dynamic scheduling was developed already in 1967 for IBM360/91 by Robert Tomasulo, a researcher with IBM
  - To minimize RAW hazards, it tracks availability of operands, independently of the execution order of instructions.
  - To minimize WAW and WAR hazards, it uses a set of internal registers (invisible to the ISA level) to carry out register renaming.
- Variations on the original techniques are currently used in all CPUs that implement dynamic ILP.

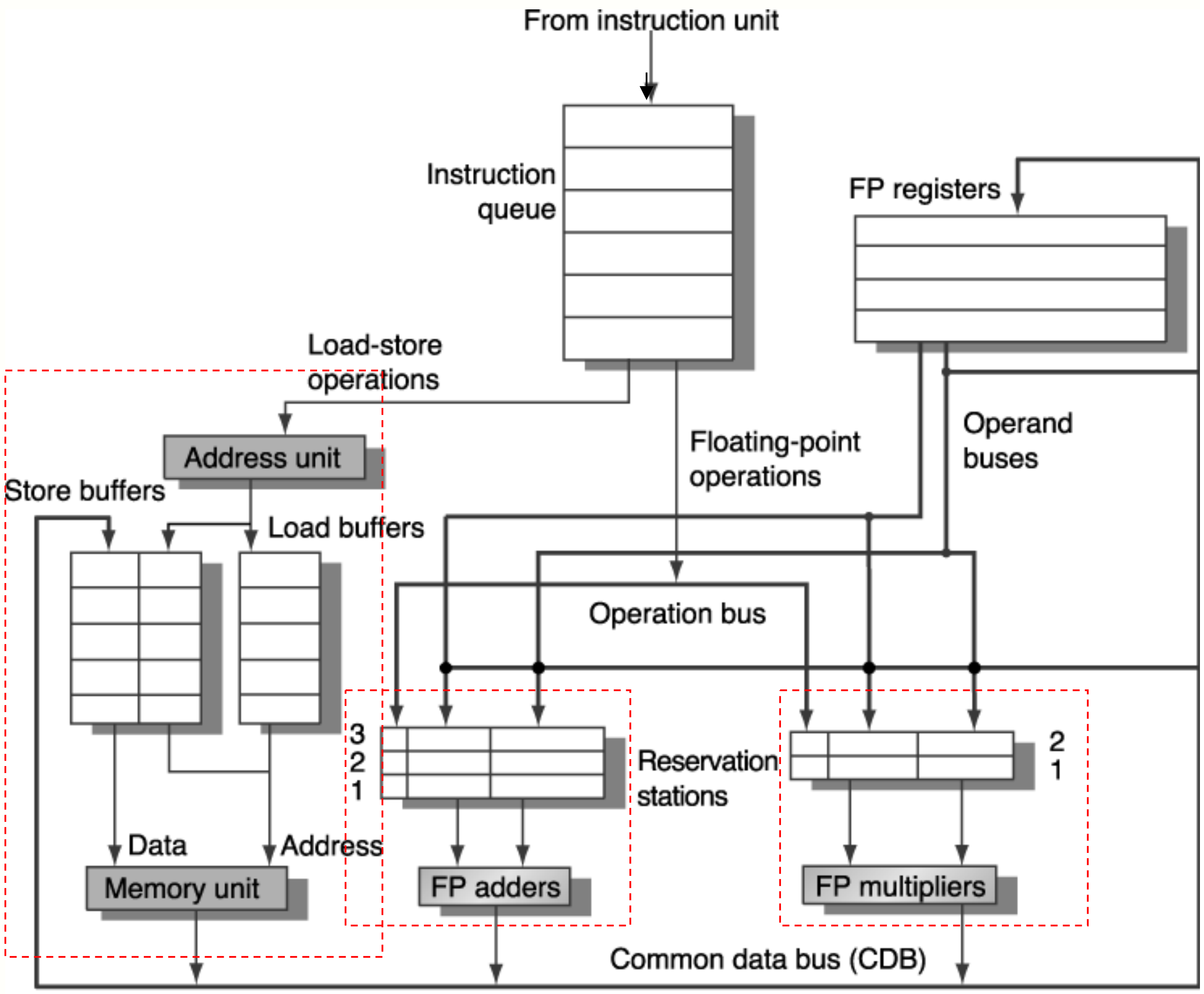
Pipeline dynamic scheduling

- The basic feature of Tomasulo's scheme are **reservation stations** associated to functional units, where instructions, fetched from IM and decoded, wait until ready to be executed (Patterson-Hennessy, fig. 6.49 modified).



Pipeline dynamic scheduling

The basic structure of the MIPS F.P. unit that implements dynamic ILP according to Tomasulo’s scheme. Disregard the “Instruction queue” and think as it were the Instruction Memory (Hennessy-Patterson, Fig. 3.2)



Pipeline dynamic scheduling

- Each reservation station can hold one or more instructions waiting to use the corresponding funtional unit and, for each instruction:
  - the operands that will be used by that instruction *or*
  - the names of the reservation stations that will produce those operands.
- a **Common Data Bus** (CDB) connects all functional units, registers and reservation stations, and allows to transmit the result produced by a functional unit to all units that need it, in parallel
- In Tomasulo’s scheme, the execution of an instruction can be split into three “macro-steps” (each can correspond to more pipeline stages):

Pipeline dynamic scheduling

- **ISSUE**. An instruction is fetched from instruction memory and is decoded (phases IF and ID in the simple pipeline). If a proper reservation station is available, the instruction is forwarded to it.
- If there are no reservation station, the pipeline stalls.
- If the instruction operands are available in the register file, or in some reservation station, they are fetched and forwarded to the reservation station where the instruction has been moved: the instruction is ready for execution in the corresponding functional unit.
- For each missing operand, the reservation station receives the name of the F.U. that will produce that operand.

Pipeline dynamic scheduling

- Note that knowing that the instruction operands:
  - can be in another reservation station, or
  - still have to be produced by an instruction currently in another reservation station
- implies that the CPU control logic, during the ISSUE phase, has checked previously issued instructions, already forwarded to some reservation station, for dependences with current one.

Pipeline dynamic scheduling

- **EXECUTE**: (EX in the basic scheme) the instruction is in a reservation station. If one or more operands are not yet available, the CDB is checked for them.

- When an operand is available (it is the outcome of another instruction, and it is transmitted to the register file through the CDB) it is "captured" and it is forwarded to the reservation station station waiting for it.

- When all operands are available, the instruction can be sent for execution in the corresponding F.U.

- If more instructions are ready for execution in the same F.U., a FIFO policy is used to choose the one to execute.

Pipeline dynamic scheduling

- In LOAD and STORE instructions, execution is plit in two steps:
- in the first step, the effective memory address is computed, when the base register is available, and the computed address is stored in a **load** or **store buffer**
- LOAD instructions can complete (second step) as soon as the (data) memory access unit is available.
- STORE instructions can possibly wait for the data to be stored in RAM.
- To prevent hazards while accessing RAM locations, the execution order of LOAD and STORE instructions obeys some additional constraints, that will be discussed later through an example (what makes these hazards so difficult to manage?)

Pipeline dynamic scheduling

- **WRITE RESULT**: (MEM and WB in the basic scheme) when the instruction executed in the F.U. has produced its result, it is written on the CDB, and it is thus forwarded to the register file, to the reservation stations and to the load/store buffers waiting for it.

- STORE instructions write data to the data memory in this phase, when both the address and the data for that address are ready.

- Analogously, LOAD instructions fetch data from RAM and write it into the destination register.

Pipeline dynamic scheduling

- The reservation stations, the register file, load and store buffers have internal registers (unavailable to the ISA level) used for storing the infos required to handle the whole procedure
- When an instruction I is in a station, it refers to an operand it is waiting for by writing in an internal register of that station the number of the station S holding the instruction that will produce the result required by I.
- When S produces a result to be written (for example) in R5, the value is transmitted through the CDB to R5 and to all reservations stations where I is waiting that result.

Pipeline dynamic scheduling

- At this point, if another instruction wants to overwrite R5, it can do so with no harm for I, that has already a local copy of R5 saved in its reservation station.
- Otherwise stated, the reservation station internal registers act as temporary registers, and are used to implement **register renaming**.
- Moreover, an instruction can be executed as soon as its operands are available, possibly before other instructions that preceded it in the IM: **the pipeline is scheduled dynamically**.
- Obviously, if two independent instructions require the same F.U., they cannot be executed in parallel (if the F.U. is itself a pipeline, they can proceed in pipeline fashion)

Pipeline dynamic scheduling

- Let us know examine some cases with Tomasulo's scheme, assuming that each reservation station can hold a single instruction. Each **station** has seven fields:
  - **Op**: the operation to be executed on certain operands
  - **Qj, Qk**: the stations that will produce the result required by Op. A value zero means that the operand is already in Vj or Vk
  - **Vj, Vk**: operands values (for a LOAD or a STORE, Vj contains the offset)
  - **A**: contains initially the immediate value for the LOAD or STORE, and the effective address, once computed
  - **Busy**: The station and its corresponding F.U. are occupied

Pipeline dynamic scheduling

- Each **register** in the register file has a field:
  - **Qi**: the number of the station containing the instruction whose result is bound for the register.
  - If Qi is empty or has a value 0, there is no instruction that is computing a value for that register.

- LOAD and STORE **buffers** each have a field, **A**, containing the effective address on which the operation will work.

Dynamic scheduling : example 1

```
LD      F6, 34 (R2)
LD      F2, 45 (R3)
MUL     F0, F2, F4
SUB     F8, F2, F6
DIV     F10, F0, F6
ADD     F6, F18, F12
```

WAR hazard on ADD: it writes F6, that is just used by DIV. If ADD completes before DIV reads F6, a wrong value is used. (Hennessy-Patterson, Fig. 3.3)

Dynamic scheduling : example 1

Reservation stations / Register Status (Hennessy-Patterson, Fig. 3.3)

tag	busy	Oi.	Vj	Vk	Qj	Qk	A

Field	F0	F2	F4	F6	F8	F10	F12	...	F31
Qi									

Dynamic scheduling : example 1

LD	F6, 34 (R2)
LD	F2, 45 (R3)
MUL	F0, F2, F4
SUB	F8, F2, F6
DIV	F10, F0, F6
ADD	F6, F18, F12

WAR hazard on ADD: it writes F6, that is just used by DIV. If ADD completes before DIV reads F6, a wrong value is used. (Hennessy-Patterson, Fig. 3.3)

Instruction	Issue	Execute	Write Res.
LD F6, 34 (R2)	√	√	√
LD F2, 45 (R3)	√	√	
MUL F0, F2, F4	√		
SUB F8, F2, F6	√		
DIV F10, F0, F6	√		
ADD F6, F8, F2	√		

Dynamic scheduling : example 1

Reservation stations / Register Status (Hennessy-Patterson, Fig. 3.3)

tag	busy	Oi.	Vj	Vk	Qj	Qk	A
load1	yes	load					34+[R2]
load2	yes	load					45+[R3]
float1	yes	MUL		[F4]	load2		
fadd1	yes	SUB		mem[34+[R2]]	load2		
fdiv1	yes	FDIV		mem[34+[R2]]	float1		
fadd2	yes	add	[F18]	[F12]			

Field	F0	F2	F4	F6	F8	F10	F12	...	F31
Qi	float1	load2		fadd2	fadd1				

Dynamic scheduling : example 1

Reservation stations / Register Status (Hennessy-Patterson, Fig. 3.3)

tag	busy	Oi.	Vj	Vk	Qj	Qk	A
load1	yes	load					34+[R2]
load2	yes	load					45+[R3]
float1	yes	MUL		[F4]	load2		
fadd1	yes	SUB		mem[34+[R2]]	load2		
fdiv1	yes	FDIV		mem[34+[R2]]	float1		
fadd2	yes	add	[F18]	[F12]			

Field	F0	F2	F4	F6	F8	F10	F12	...	F31
Qi	float1	load2		fadd2	fadd1				

Dynamic scheduling : example 1

Reservation stations / Register Status (Hennessy-Patterson, Fig. 3.3)

name	busy	Oi.	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		

Field	F0	F2	F4	F6	F8	F10	F12	...	F31
Qi	Mult1	Load2		Add2	Add1	Mult2			

Dynamic scheduling : example 1

- please note:
  - Regs[ ] shows register file access
  - Mem[ ] data memory
  - the second Load has completed the computation of the effective address and it is waiting to access the data memory unit
  - the SUB has an operand already available (Vk) and waits for the second from the outcome of the second LOAD (Qj)
  - **ADD waits for its opernads from SUB (Qj) and from the second LOAD (Qk): as soon as available, the ADD can start execution**
  - **the WAR hazard is solved with register renaming: DIV does not get its second operand from F6, but from Vk. Completion of ADD before DIV causes no harm.**

Pipeline dynamic scheduling

- Tomasulo's scheme has two basic characteristics (that, by the way, make it superior with respect to other simpler schemes, such as **scoreboarding**, an obsolete dynamic scheduling):

1. **operand access is distributed**

   – This is obtained by using multiple reservation stations and the CDB. When more instructions are waiting for the same operand A (the other one being already available), as soon as A is ready all instructions can be launched (provided they use different F.U., or these are pipelined).

   – If the operands should be fetched from the register file, each F.U. should access sequentially the register holding the operand.

Pipeline dynamic scheduling

2. WAW and WAR are eliminated

- This is obtained by register renaming on reservation station internal registers, and by forwarding operands just produced to all stations awaiting for them, as soon as possible and in parallel.
- Let us consider the WAR example just discussed. If the first LOAD had not yet completed, Qk for DIV would contain Load1, and DIV would be independent of ADD.
- ADD can execute without waiting for DIV to read its operands (specifically the second one)

Dynamic scheduling: example 1a

- With the same code as the previous example, let us consider the situation when MUL instruction is ready to produce its result (Hennessy-Patterson, Fig. 3.4):

Instruction	Issue	Execute	Write Res.
LD F6, 34 (R2)	√	√	√
LD F2, 45, (R3)	√	√	√
MUL F0, F2, F4	√	√	
SUB F8, F2, F6	√	√	√
DIV F10, F0, F6	√		
ADD F6, F8, F2	√	√	√

Dynamic scheduling: example 1a

Reservation stations / Register Status (Hennessy-Patterson, Fig. 3.4)

nome	busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MU L	Mem[45+Regs[R3]]	Regs[F4]			
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

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

Pipeline dynamic scheduling

- The real effectiveness of Tomasulo's scheme in handling name and data hazards shows best in loop management.
- In the following code fragment, let us assume that the CPU assumes that the branch will be taken (with static branch prediction: *backwards branches are taken*, or another dynamic branch prediction technique to be covered later).

```
Loop:   LD    F0, 0 (R1)        (Hennessy-Patterson, Fig. 3.6)
        MUL   F4, F0, F2        // multiplies by a scalar in F2
        SD    F4, 0 (R1)        // the elements of an array
        ADD   R1, R1, -8
        BNE   R1, R2, Loop;     //branches if R1 <> R2
```

Pipeline dynamic scheduling

- If the CPU is able to fetch a new instruction from IM at every clock cycle, reservation stations allow to sustain the concurrent execution of instructions from different (consecutive) iterations, and the stations themselves act as additional registers.
- In the code fragment just considered, let us assume that all instructions from two consecutive iterations have already gone through the ISSUE phase, but none has completed.
- Also, let us not consider the ADD operation, that only handles loop management, and let us assume that the branch is taken

Dynamic scheduling: example 2

```
Loop:   LD    F0, 0 (R1)      (Hennessy-Patterson, Fig. 3.6)
        MUL   F4, F0, F2      // / multiplies by a scalar in F2
        SD    F4, 0 (R1)      // the elemts of an array
        ADD   R1, R1, -8
        BNE   R1, R2, Loop; //  branch if R1 <> R2
```

Instruction	Iteration	Issue	Execute	Write Res.
LD F0, 0 (R1)	1	√	√	
MUL F4, F0, F2	1	√		
SD F4, 0 (R1)	1	√		
LD F0, 0 (R1)	2	√	√	
MUL F4, F0, F2	2	√		
SD F4, 0 (R1)	2	√		

Dynamic scheduling: example 2

- After set up, there are two consecutive iterations of the loop in execution, a feature known as **dynamic loop unrolling**, with clear gains in execution speed.
- The critical point is the execution of LOAD and STORE from different iterations. They can be executed in any order, if they access different addresses, but **if they use the same address**, and hazards ensues as follows:
  - WAR, if the LOAD precedes the STORE and they are executed out of order
  - RAW, if the STORE precedes the LOAD and they are executed out of order
  - (n.b.: exchanges STOREs causes a WAW)

Dynamic scheduling: example 2

- To establish if a LOAD can be executed, the CPU must check if any STORE not yet completed and preceding the LOAD uses the same memory location (RAW hazard)
- Similarly, a STORE must wait if other STORE or LOAD not yet completed and preceding the STORE use the same memory location (WAW and WAR hazard)
- To detect these hazards, the CPU must have computed and stored all RAM addresses associated to any operation involving a memory access preceding the one being scheduled.

Dynamic scheduling: example 2

- A simple way to do so is to compute effective memory addresses (immediate + base register, in our ISA) according to the sequel of instructions in the program.

- Relative order must be preserved only for STORE vs other STOREs and LOADs, while LOADs can be interchanged freely among themselves (why ? ..)

Dynamic scheduling: esempio 2

- **LOAD**: let us assume that effective addresses are computed in program order: when the computation for a given LOAD is completed, the CPU examines field A of all active STORE buffers, to detect address conflicts.

  If the LOAD address matches any active entry in a STORE buffer, the LOAD is not forwarded to the LOAD buffer until the the conflict is solved (that is, the relative STORE has completed).

- **STORE**: as with the LOAD, but the CPU controls both STORE and LOAD active buffers, because STOREs to a given address cannot be reorder with respect to LOADs from the same address.

Dynamic scheduling: esempio 2

```
Loop:   LD    F0, 0 (R1)      (Hennessy-Patterson, Fig. 3.6)
        MUL   F4, F0, F2      // multiplies by a scalar in F2
        SD    F4, 0 (R1)      // the elements of a vector
        ADD   R1, R1, -8
        BNE   R1, R2, Loop; //  branch if R1 <> R2
```

Instruction	Iteration	Issue	Execute	Write Res.
LD F0, 0 (R1)	1	√	√	
MUL F4, F0, F2	1	√		
SD F4, 0 (R1)	1	√		
LD F0, 0 (R1)	2	√	√	
MUL F4, F0, F2	2	√		
SD F4, 0 (R1)	2	√		

Hennessy-Patterson, Fig. 3.6:

nome	busy	Op	Vj	Vk	Qj	Qk	A
Load1	yes	Load					Regs[R1]
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	

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

Dynamic scheduling: example 2

– Try to describe the status of reservation stations and to analyse completion (the *write phase*) of loads and stores in the following situation:

Instruction	**Iteration**	**Issue**	**Execute**	**Write Res.**
LD F0, 0 (R1)	1	√	√	√
MUL F4, F0, F2	1	√	√	
SD F4, 0 (R1)	1	√	√	
LD F0, 0 (R1)	2	√	√	√
MUL F4, F0, F2	2	√	√	
SD F4, 0 (R1)	2	√	√	

Dynamic scheduling: notes

- As already highlighted, all modern processors supporting dynamic scheduling use some variation of Tomasulo's scheme.
- Dynamic ILP requires a complex and costly hardware (each reservation station must be realized as a high speed, associative buffer, to minimize delays) and a very sophisticated control logic.
- The CDB itself requires a complex and high speed circuitry.
- A dynamically scheduled pipeline can have very good performances (close to one instruction completed per clock cycle – we do not consider so far multiple-issue architectures), **provided branches are predicted in a most accurate manner**.

Branch Prediction (BP)

- We have already introduced the issues of control. Let us consider the instructions:

```
           ADD   R1, R4, R5
           BNE   R1, R2, branchtag;
           SUB   R5, R6, R7
branchtag: ADD   R5, R6,R7
```

- In theory, until the outcome of the comparison between R1 and R2 is known, the CPU does not know which is the next instruction to fetch (SUB or ADD), and should stall the pipeline.

Branch Prediction (BP)

- In RISC architecture, by analyzing the various types of programs, it turns out that there is a branch every 4-:-7 instructions.
- In a 5-stage pipeline with “aggressive” branch execution in only 2 clock cycles, the pipeline must be stalled for 1 clock cycle, waiting for the comparison outcome, thus roughly 1/5 of all clock cycles are wasted.
- In real architectures, pipelines are much deeper in stages, and since comparison result is known only after quite a number of clock cycles, the waste is much larger.

Branch Prediction (BP)

- A simple yet effective alternative consists of fetching anyway the instructions that "follow" the branch (SUB in the above example) and of starting their execution in the pipeline.
- It is a very simple form of static prediction: the branch is supposed not-taken.
- If the prediction is correct, no clock cycle is wasted
- Otherwise, the pipeline must be flushed, with proper control signals from the CU, by eliminating all instructions following the branch, that should not have been executed.

Branch Prediction (BP)

- In a misprediction, the waste in clock cycles is the larger, the deeper the CPU, that is the more apart the stage in the pipeline that executes the comparison.
- Unfortunately, any static assumption on branch behaviour cannot guarantee a sufficiently low error rate
- All modern processors apply some form of dynamic branch prediction, with the outcome of the branch speculated on the basis of the "history" of that branch, namely previous executions of the same instruction.

Branch Prediction (BP)

- Branch prediction (the dynamic version, from this point onward) is effective because many branch instructions in the programs are executed many times, and they tend to behave as in the preceding cases.
- The most obvious instance is a branch that controls a loop: if a program uses a loop, it is assumed that the instructions within the body of the loop are executed many times, and consequently the branch will have the same behaviour almost all of the times.
- No branch can be reliably predicted if it is executed only once …

Branch-Prediction Buffer

- The most simple branch prediction technique is using a **branch prediction buffer**: an associative memory addressed with the n least significant bits of a branch instruction address.
- Every entry in the buffer contains two fields:
  - **the n least significant bits of the memory address** where the branch instruction is located (it is the key into the associative memory)
  - a **prediction bit** that stores the outcome of the last decision taken for that branch

Actually, the buffer is a cache for branch instructions.

Branch-Prediction Buffer

- At the first execution of the branch, the relevant information (address and prediction bit) are used to update the buffer.
- If the branch is executed again, and its prediction bit says that the branch must not be taken, the CPU continues to execute the instructions that follow the branch.
- Meanwhile, the branch instruction completes execution, and if the prediction is wrong (the branch must be taken), the pipeline is flushed and execution is restarted with the destination instruction. The prediction bit is complemented.

Branch-Prediction Buffer

- Question (1): what happens if the prediction bit says that the branch must be taken ?
- Question (2): and what if this prediction is wrong ?
- Question (3): how can one know if the data in a specific entry are actually specific of the branch that is being executed ?

Branch-Prediction Buffer

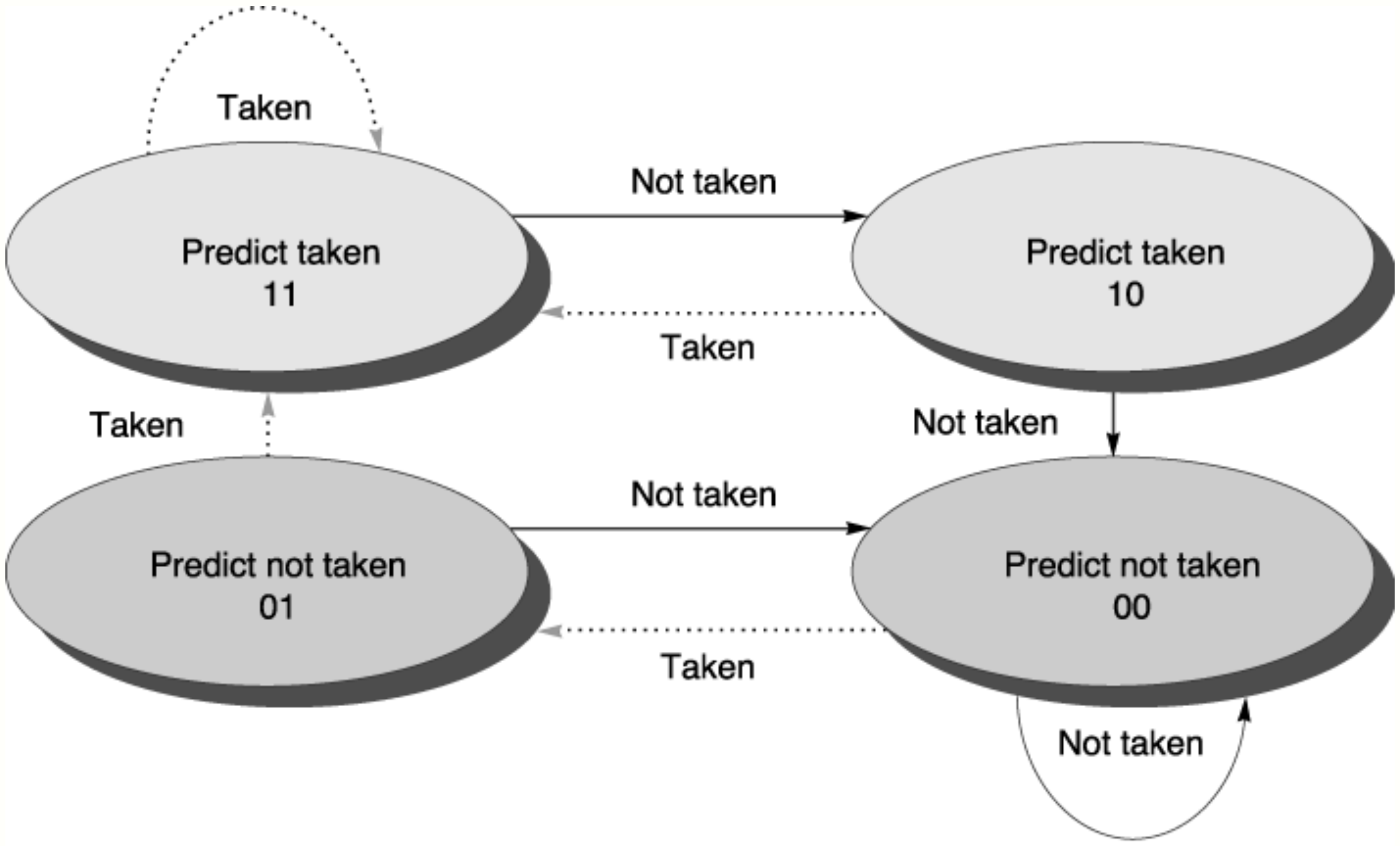
- Actually, this form of prediction can produce more errors than one can expect.
- Let us consider a loop executed 9 times, after which the loop exits, to re-enter in a later phase. What is the accuracy of the prediction?
- After 9 iterations, a prediction error cannot be avoided: the prediction error calls for another iteration, but the loop is finished. The bit is complemented.
- When the program re-enters the loop, the first prediction is wrong once again ...

Branch-Prediction Buffer

- To prevent this problem, usually a new scheme is used, a (**local**) **2-bit predictor**: a prediction must be wrong twice before it is changed
- The two bits are used as a counter:
  - When a branch is taken, the counter is incremented by one (with a saturation to 11)
  - When it is not taken, it is decremented (with saturation to 00)
- If count = 11 or 10 → prediction taken
- If count = 00 or 01 → prediction not taken

Branch-Prediction Buffer

- Actually, this is a 4-state automaton (Hennessy-Patterson, Fig. 3.7, Patterson-Hennessy fig. 6.39) :



Branch-Prediction Buffer

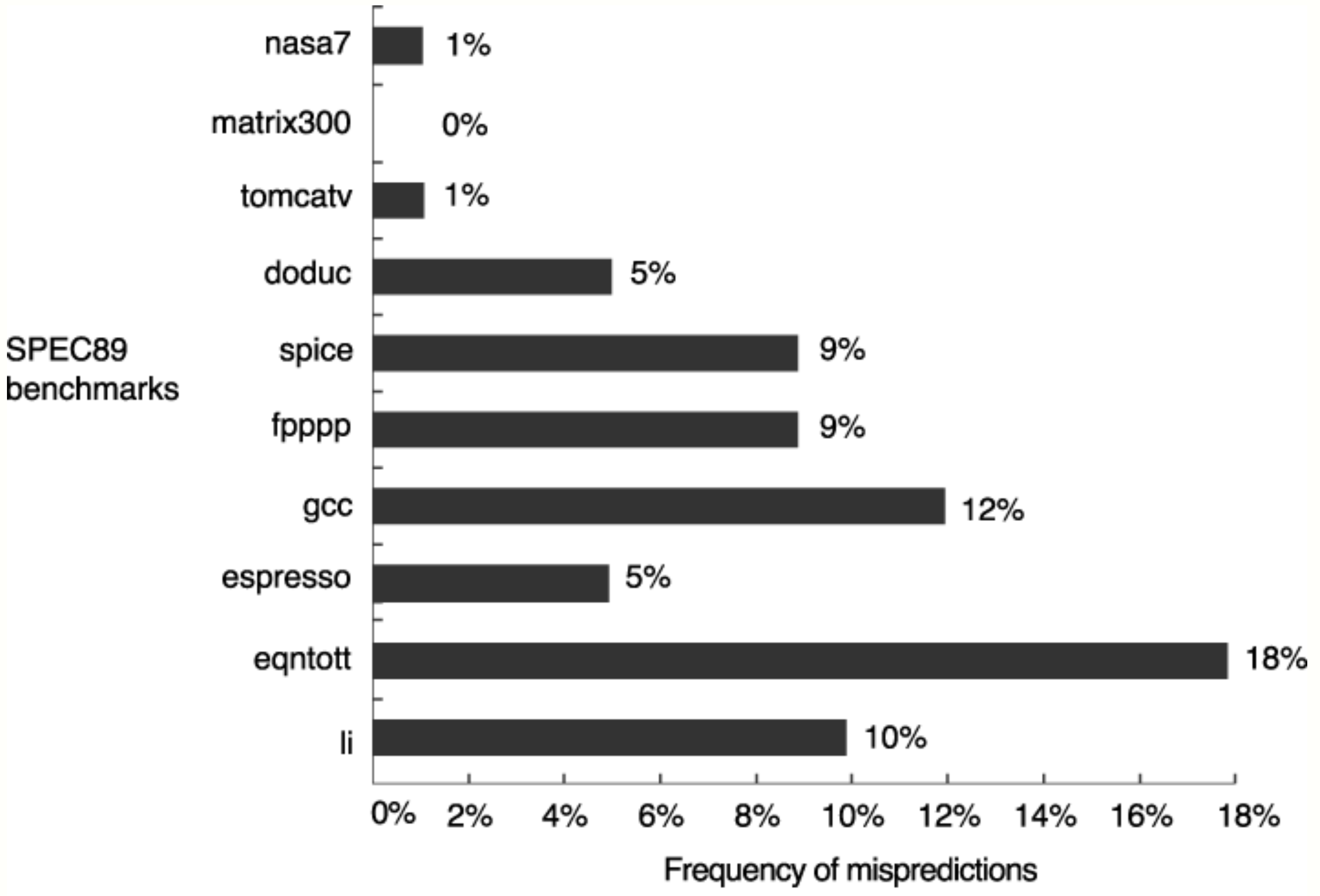
- The two-bit scheme, though more complex, works well if, for each branch executed, the ratio between taken and not taken instances is really unbalanced.
- In this case, the 2-bit scheme is considerably more efficient than the 1-bit scheme.
- Since 2 bits yield a better prediction than 1, it could be natural to further increase prediction accuracy by moving to a 3-bit scheme
- Oddily enough, prediction accuracy does not increase effectively using more than 2 bits.

Branch-Prediction Buffer

- Moreover, the efficacy of the scheme depends also on the number of entries in the associative memory holding the prediction bits for branch instructions.
- Typically, these buffers are caches with 4096 entries, a number considered sufficient in most situations (though this does not guarantee that the bits of the correct branch are indeed used…).
- Simulations have shown that larger buffers do not offer effectively better preformances

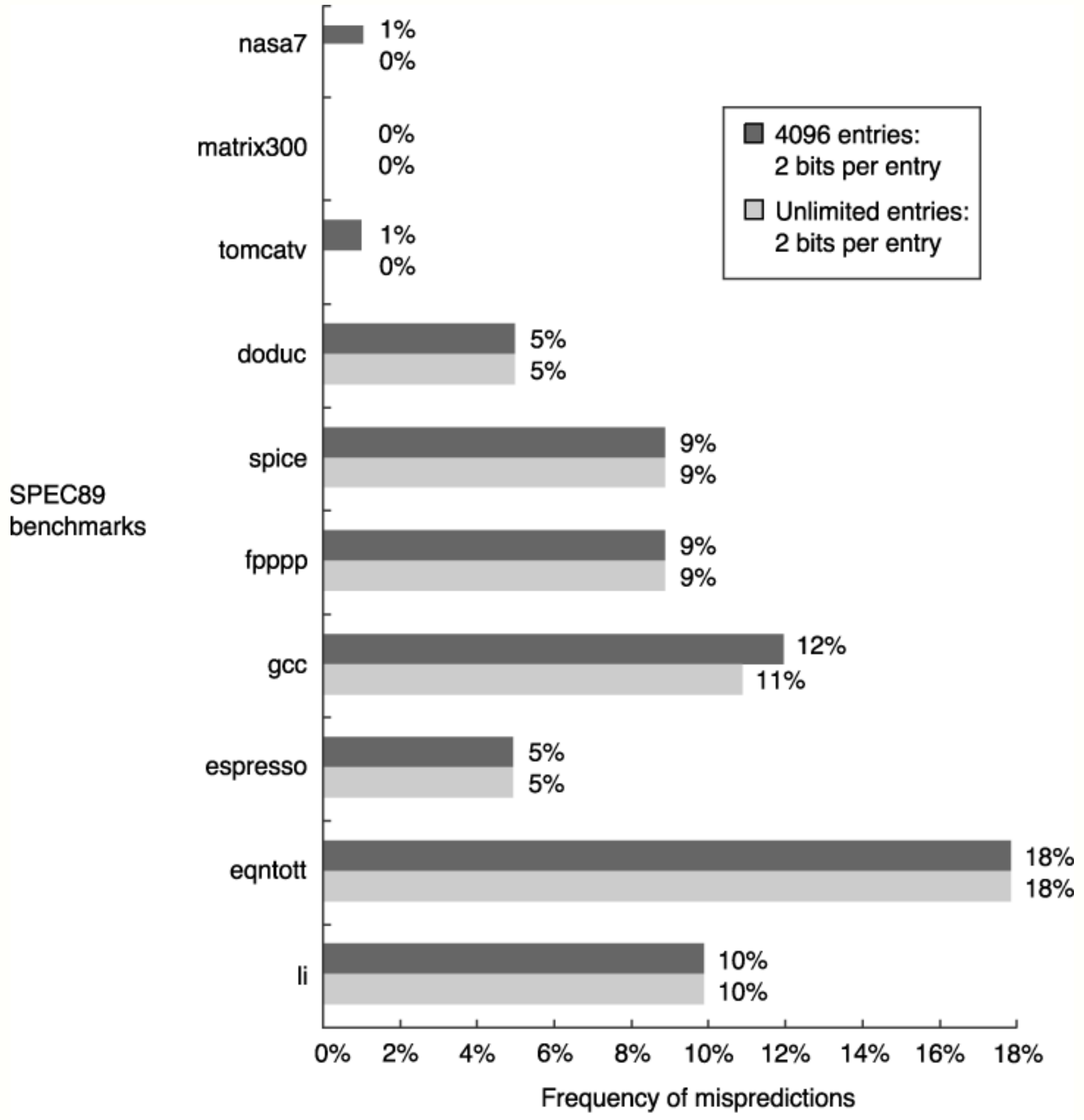
Branch-Prediction Buffer

- prediction accuracy for 2-bit, 4096-entry BPB (Hennessy-Patterson, Fig. 3.8):



Branch-Prediction Buffer

increase in accuracy with a 2-bit BPB and an infinite number of entries with respect to 4096 entries: almost no difference !! (Hennessy-Patterson, Fig. 3.9).

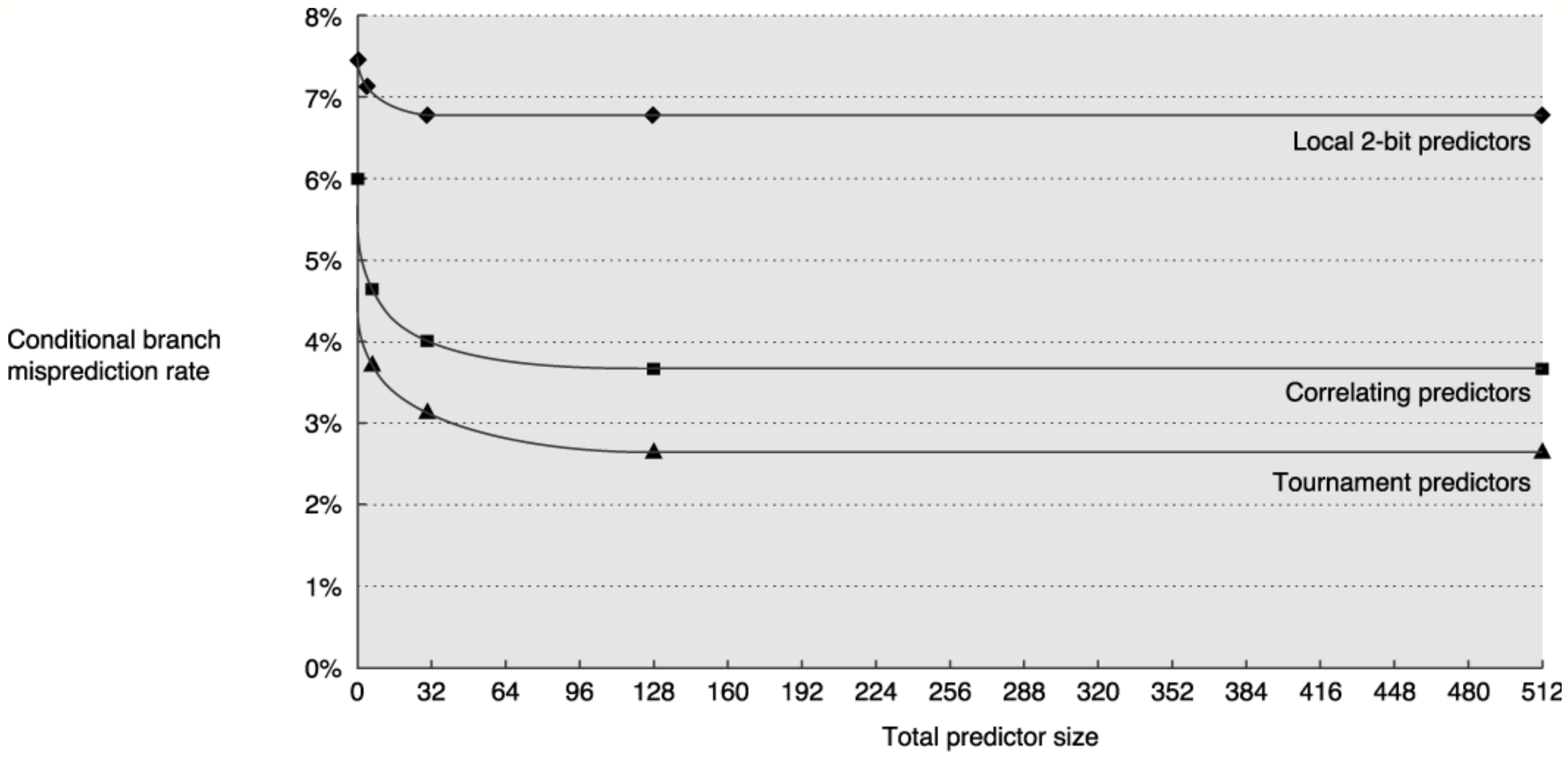


Branch-Prediction Buffer

- With any prediction scheme, there are limits to the precision that can be reached (as an instance, the example just considered does not increase performance in consistent way by moving beyond 2 bits and 4096 entries). Furthermore, prediction capability varies on the basis of the actual application in execution.
- Modern processors use advanced variations on the BP techniques examined so far, that guarantee actual increments to precision accuracy.
- **Correlating predictors**: the 2-bit predictors of two consecutive branches are correlated, thus combining the history of a branch with the behaviour of another “nearby” branch.
- **Tournament predictors**: each branch has two predictors, a 1-bit and a 2-bit one, and every time the prediction is based on the predictor that behaved best in the preceding case.

Branch-Prediction Buffer

Average mis-prediction rate in different benchmarks and in 3 different prediction techniques (Hennessy-Patterson, Fig. 3.18):

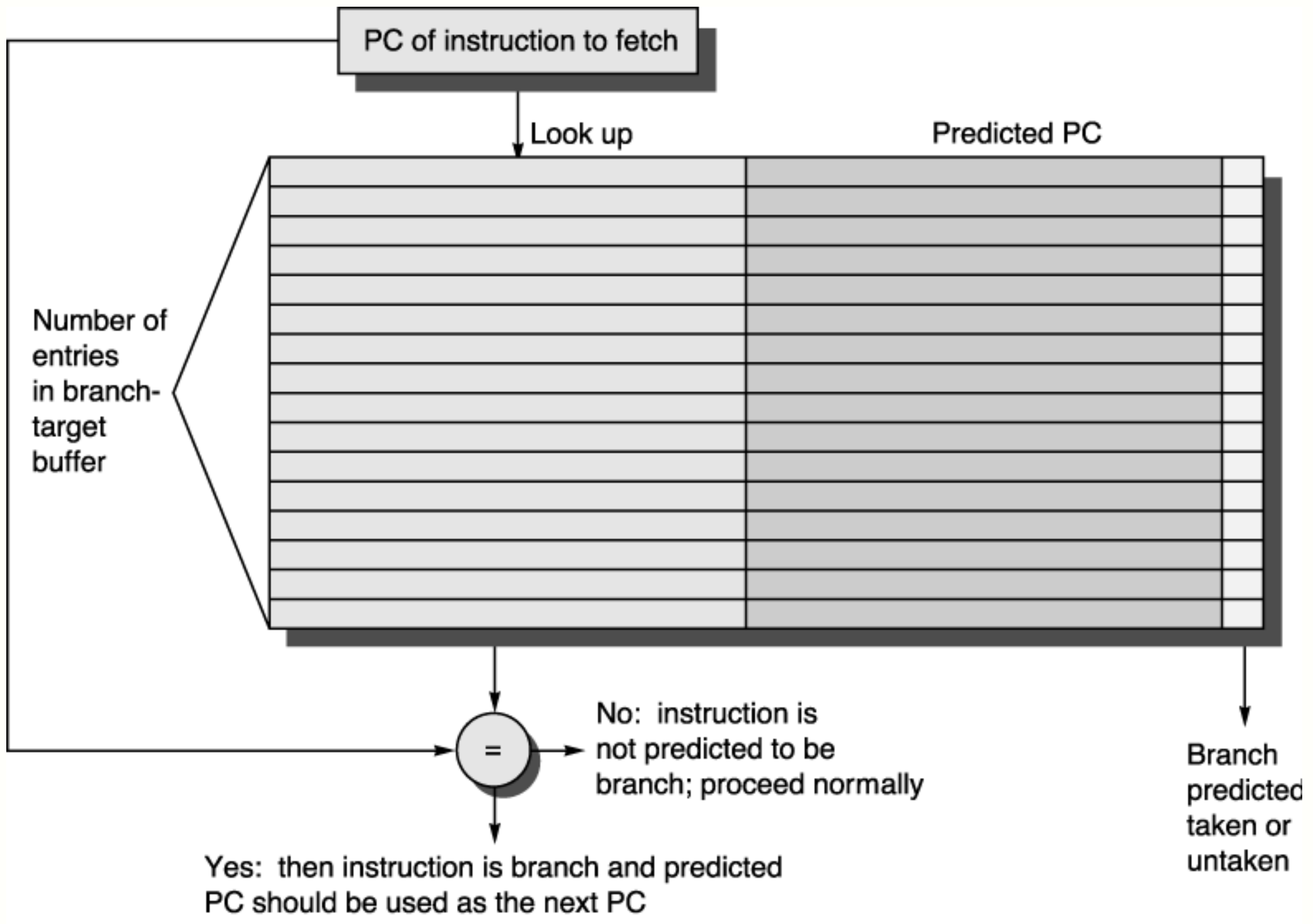


Branch Target Buffer

- So far, an important feature of BP has been neglected: il the predictor votes for a taken branch, the CPU cannot start fetching the destination instruction until its address is known (PC + offset in the branch instruction).
- To overcome this problem , many CPUs use a **branch target buffer**, (also **branch prediction cache**), that, for each branch, stores the destination address to be used if the prediction is "taken"
- The value "PC + offset" is computed and stored in the BTB the first time the branch is executed.
- In subsequent executions, no computation is necessary: in case of "taken" prediction, the PC is loaded with the destination address stored in the buffer.

Branch Target Buffer

Each entry in the buffer stores a branch instruction from the program in execution (Hennessy-Patterson, Fig. 3.19):


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Hardware speculation

- In BP, the CPU starts executing instructions before knowing if they should actually be run.
- If the prediction turns out to be wrong, instructions in pipeline in stages before that of the branch are nullified, and the correct instruction is stared.
- Let us consider the situation depicted in the next chart, where a true data dependence involving the branch can stall the CPU much longer than the time required to execute the branch.

Hardware speculation

```
1.              LD      F4, 100(R4)             // value not in cache…
2.              BLE     F4, #0.66666, jump      // branch if less or equal
3.              FADD    F1, F1, #0.5
4.              DADD    R1, R1, #2
5.      jump:   FADD    F1, F1, #0.25
6.              DADD    R1, R1, #1
```

- If the data loaded from the LOAD is not in the cache, it might be necessary using tenths of clock cycles to fetch it from RAM.
- One can use BP on BLE, and **start and complete** the execution of the sums controlled by the branch, if this requires a number of clock cycles much shorter than that required to fetch the date for the BLE.
- But what is the prediction is wrong?

Hardware speculation

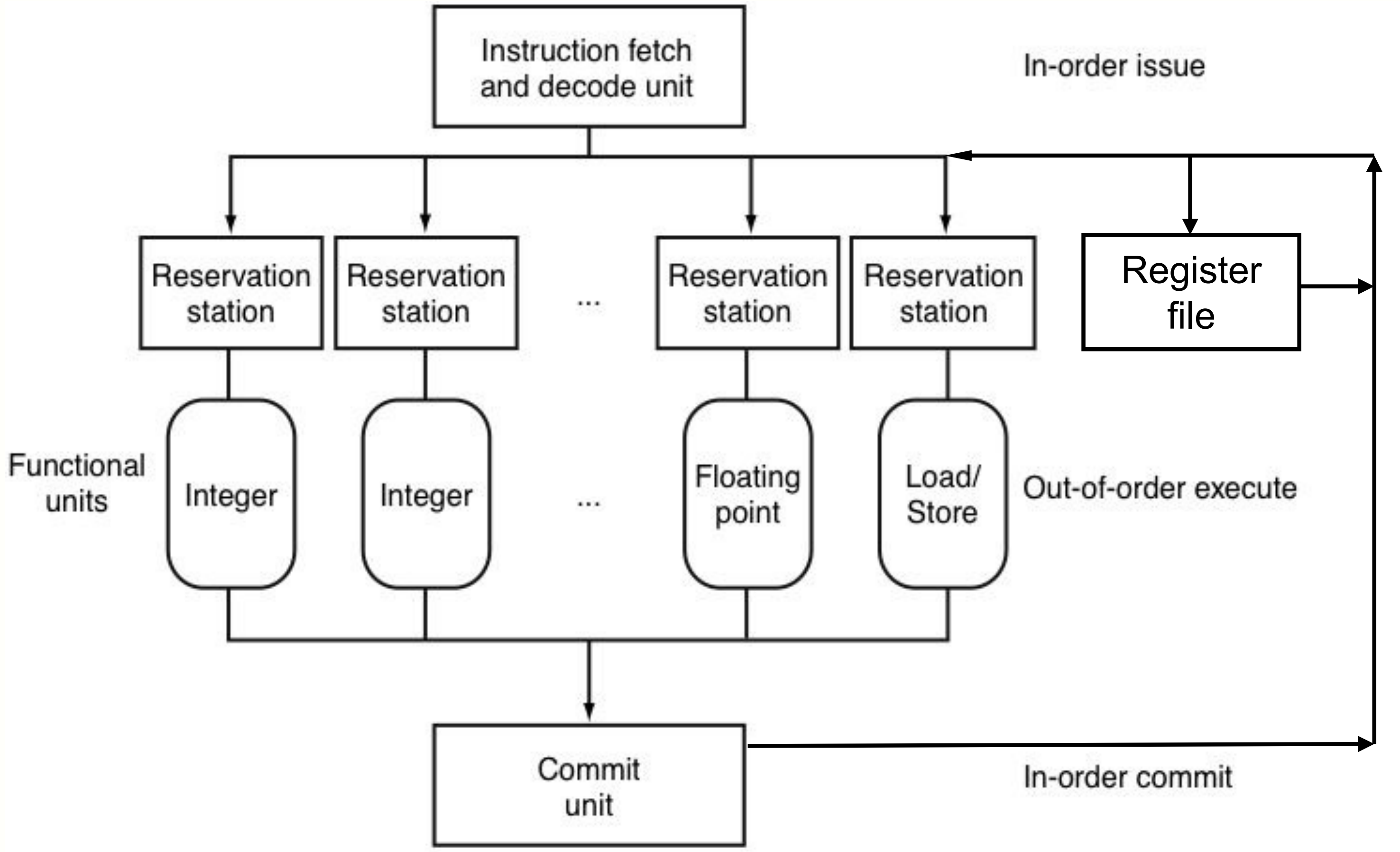
- **Hardware speculation** is the technique used in dynamic ILP to handle cases such as this.
- Branch controlled instructions are executed as if the prediction were correct (usually, one speaks of *branch speculation*, and of **speculative instructions**)
- However, it must be always possible to nullify speculative instructions, should the prediction be wrong.
- Question: in the example above, what about instructions that are not controlled by BLE (instructions 5 and 6)?

Hardware speculation

- The problem is that instructions 3 and 4, controlled by the branch, pass results on to instructions 5 and 6, that should be executed anyway.
- If the LOAD takes a lot of time, the CPU can execute instructions 5 and 6 before knowing if it should eceuted 3 and 4 also.
- However, if 3 and 4 should not be executed, the values computed by 5 and 6 are wrong, and the two instructions must be executed again from scratch.
- This is to say that 5 and 6 also must be handled as speculative instructions, until the true output from BLE is known..

Hardware speculation

- In Tomasulo's scheme, hardware speculation requires a commit unit: a bank of internal registers known as Reorder Buffer (ROB) where instructions are parked, until is is known if they should actually have been executed (Patterson-Hennessy, fig. 6.49 modified).



Hardware speculation

- In the ROB executed instructions are stored along with the result they have computed, and enties in the ROB are a further support to register renaming.

- When the CPU "knows" (the "how" to be described shortly) that an instruction must effectively be executed, it performs the **commit** on it: it is cancelled from the ROB, and the destination register (or RAM memory in STORE instructions) is updated.

- If the CPU discovers that the instruction should not have been executed (or executed with other operands), it simply removes it from the ROB.

Hardware speculation

- Even though instruction execution can be performed out-of-order, commit must be carried out in-order, namely in the order in which the instructions have entered the CPU; this justifies the name ROB, that is actually managed as a circular queue.
- This constraint makes dependences control much simpler, and lessen the burden of handling exceptions, a very complex issue with speculation.
- Indeed, what if a speculative instruction raises an exception, and then it turns out that it should not be executed altogether?

Hardware speculation

- Each ROB entry has 4 fields:

1. **instruction type**: *branch*, that produces no result; *store*, that writes to RAM; *ALU o load*, that write to a register
2. **destination**: this is the register or memory address that will be modified by the instruction
3. **value**: this field stores the result of the instruction untill commit
4. **ready**: if set, it signals that the instruction has completed execution and that the output value is available.

Hardware speculation

- In speculative execution, instructions go through four "macro-phases". The first three are equal to those in Tomasulo's basic scheme

1. **ISSUE**:

   - an instruction if fetched from Instruction Memory (simplified assumption)
   - it is forwarded to the EXECUTE phase if there are a reservation station and a ROB entry free. **Instructions are inserted into the ROB in program order**. Otherwise, stall.
   - operands required by the instructions are forwarded to the RS if they are available in the registers or in the ROB (why can they be available in the ROB?)
   - the ROB entry number that will receive the result is captured into the RS: it will be used to *tag* the result of the instruction, when it will be placed on the CDB.

Hardware speculation

2. **EXECUTE**:

   - If at least one of the operands is unavailable, the CDB is monitored to detect availability of the data
   - when all opernads are available, the instruction is forwarded to the corresponding Functional Unit

3. **WRITE RESULT**:

   - the result, once ready, is written on the CDB, and, through it, in the ROB and in any station waiting for it (note, not in the register file nor in RAM).

Hardware speculation

4. **COMMIT**: (instructions commit is in-order, the ROB is managed as a circular queue, in which instructions are inserted in the same order of fetch from IM)

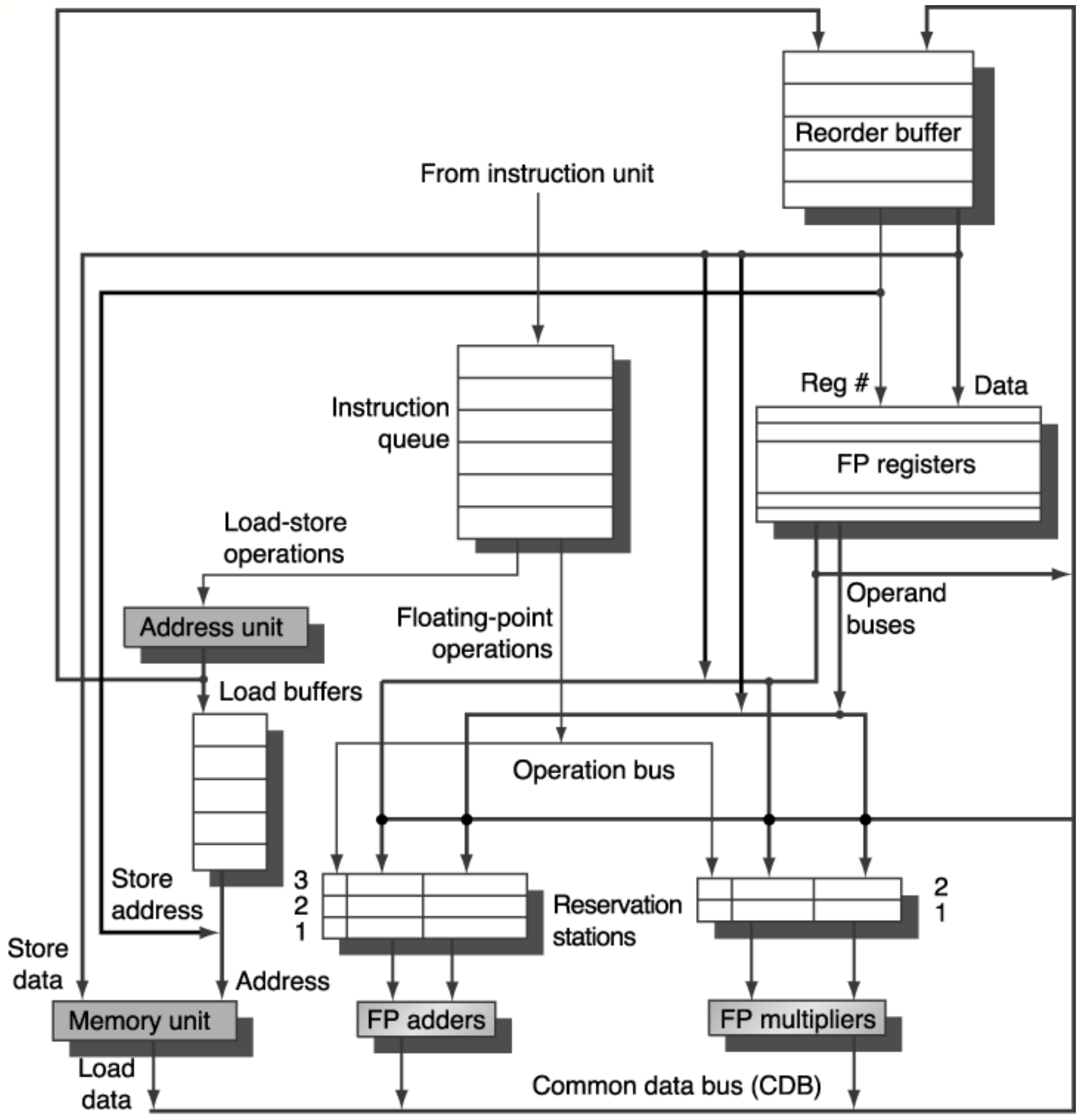
- when an instruction in Rob reaches the head of the queue (because other instructions have been inserted), commit can start
- **If the instruction is NOT a branch**, the content of the VALUE field is tranferred into the register or Ram location. The instruction is removed from the ROB
- **if the instruction is a branch with a WRONG prediction** (at this point, the execution of the branch has been completed, of course) the whole ROB is flushed and the computation restarts with the correct instruction.

Hardware speculation

- If the branch was CORRECTLY predicted, nothing special happens: simply, the branch instruction is removed from the ROB and the head of the queue is updated to the next-in-line entry.
- In some architectures, as soon as the CPU detects that a branch prediction is wrong, the branch is immediately removed from the ROB together with all following instructions (that have been unduly executed), while the preceding ones are preserved.

Hardware speculation

The basic structure for Tomasulo's scheme with speculation. Note the ROB and the absence of store buffers. Let us still assume that the "instruction queue" is actually the Instruction Memory (Hennessy-Patterson, Fig. 3.29):



Hardware speculation: example

- Let us consider a CPU with FP functional units that execute an ADD in 2 cycles, a MUL in 10 cycles and a DIV in 40 cycles. The following code is executed:

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

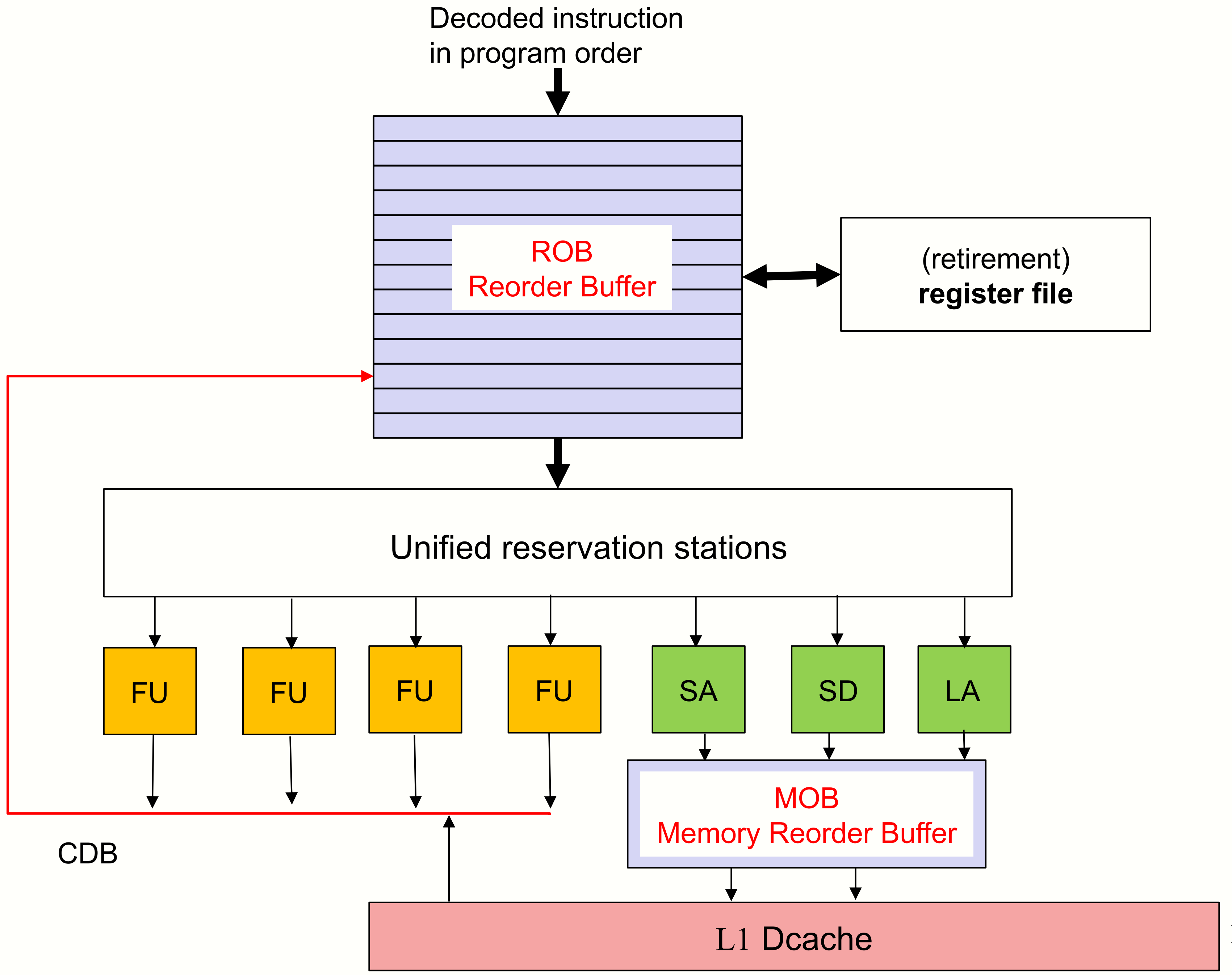
- Here follows the situation of the RS, RB, and FP registers when MUL is ready for commit (Hennessy-Patterson, Fig. 3.30):

name	busy	Op	Vj	Vk	Qj	Dest	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	no	MUL	Mem[45+Regs[R3]]	Regs[F4]		#3	
Mult2	yes	DIV		Mem[34+Regs[R2]]	#3	#5	

entry	busy	instruction	state	destination	value
1	no	LD F6,34(R2)	commit	F6	Mem[34+Regs[R2]]
2	no	LD F2,45(R3)	commit	F2	Mem[45+Regs[R3]]
3	yes	MUL F0,F2,F4	write result	F0	#2 x Regs[F4]
4	yes	SUB F8,F6,F2	write result	F8	#1 - #2
5	yes	DIV F10,F0,F6	execute	F10	
6	yes	ADD F6,F8,F2	write result	F6	#4 + #2

Field	F0	F1	F2	...	F5	F6	F7	F8	F10
reorder #	3			...		6		4	5
busy	yes	no	no	...	no	yes	...	yes	yes

Hardware speculation - INTEL

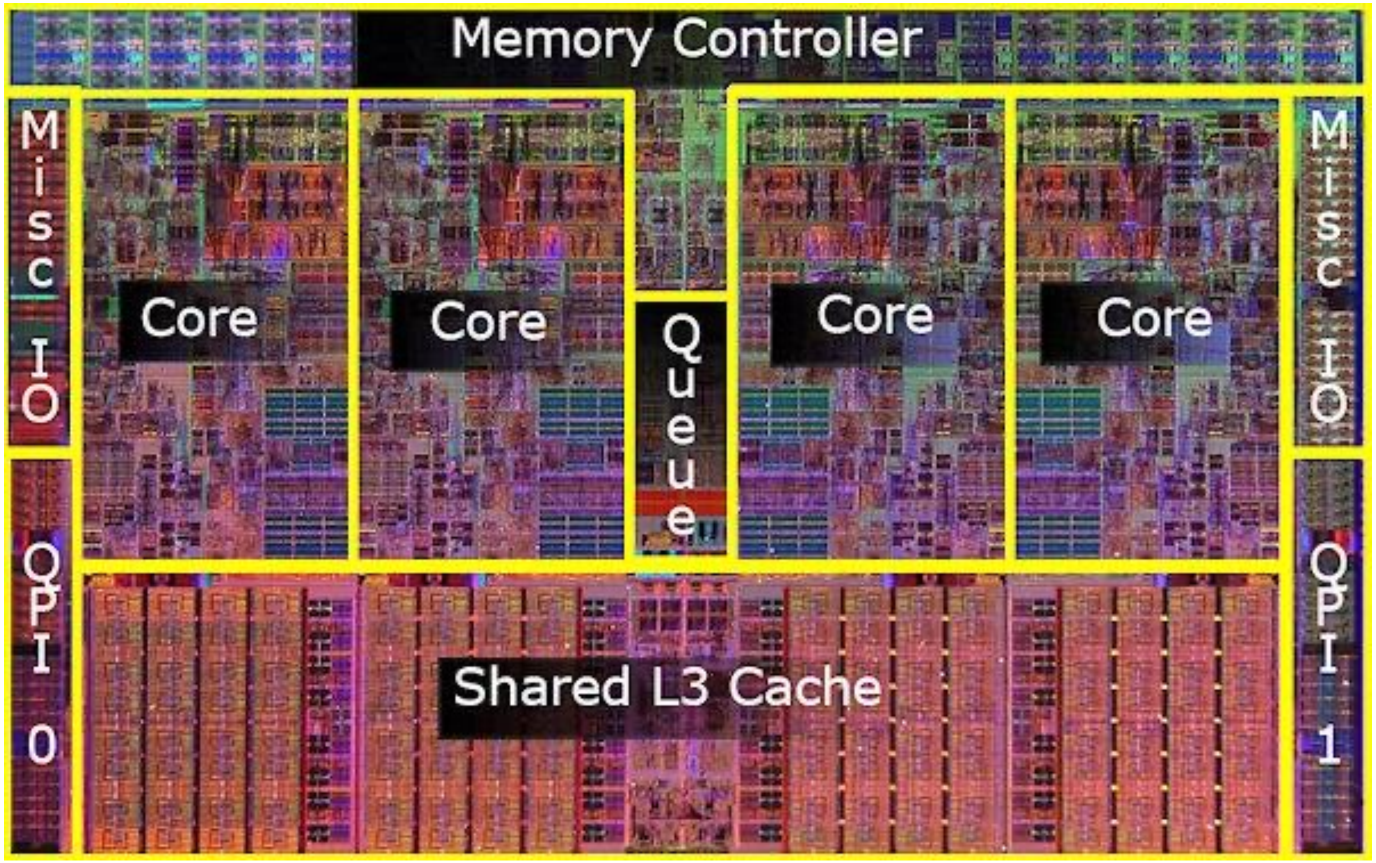

Decoded instruction in program order
ROB Reorder Buffer
(retirement) register file
Unified reservation stations
FU
FU
FU
FU
SA
SD
LA
CDB
MOB Memory Reorder Buffer
L1 Dcache

Multiple Issue

- If dynamic ILP, considered so far, is augmented with **multiple issue**, namely the capability to **start the execution of more instructions in parallel** one comes close to a complete description of most modern processors.
- Multiple issue requires a “wider” datapath, to carry on from one pipeline stage to the following one all the informations associated to all instructions issued in parallel.
- But this costs little …. if one considers the actual usage of silicon die

Multiple Issue Drivers

- The die area is LARGELY devoted to caches, which implies that "CPUs" (aca pipeline stages) consume small areas.



- i7 die (2008), 263 mm^2 area, 731 millions transistors.

Multiple Issue Drivers

- Bringing data & instructions from RAM into the die IS THE COST (in time). So, busses allow for wide data transfers since dies have large numbers of pads
- Cache lines (LLC that is L3) are usually 32B or even larger, thus each line possibly accommodates for many instructions.
- The true penalty has been paid (miss), the silicon for the CPU is much less demanding than for caches, so
- Multiple pipelines in each core

Multiple Issue

a. There must be a sufficient number of functional units to execute in parallel multiple instructions. As an instance, at least a ALU, a multply unit for integers and fp, and so on (if these units are themselves pipelined, all the better !).

b. It must be possible to fetch multiple instructions from Instruction Memory, and multiple operands from Data Memory, within each clock cycle (cache memory for instructions and for data usually have enough “bandwidth” for this purpose)

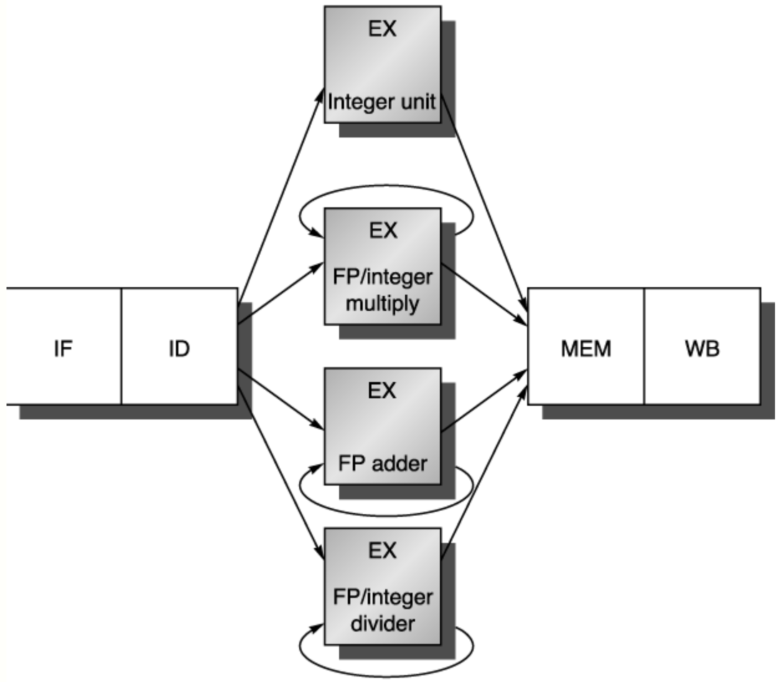
c. The register file must be multi-ported both for addressing and for reading/writing registers, to support multiple read/write accesses within the same clock cycle

Multiple Issue

- Any processor with these features is capable of issuing multiple instructions for execution in the same clock cycle, and it is therefore referred to as a **superscalar architecture**. (Hennessy-Patterson, Fig. A.29)

- A superscalar processor can be thought of as a set of pipelines working in parallel, each handling the execution of one instruction, as was the case with the first Pentium.

- Strictly speaking, a superscalar architecture need not support neither dynamic scheduling nor speculation.



Multiple Issue

- Nethertheless, if no dynamic scheduling is available, the number of instructions that can be effectively executed in parallel is strongly reduced:
  - an independent instruction C immediately following a couple of instructions A and B mutually dependent on one another is stopped any way, because of the stall caused by the couple A B.
- Therefore, it is really hopeless (at least inefficient) trying to issue multiple instructions in a statically scheduled pipeline.

Multiple Issue

- This is why processors with a statically scheduled pipeline issue at most two instructions per clock cycle, since otherwise they could not sustain a higher degree of issue.
- Even so, they must resort to specific techniques for static ILP, most notably to a strong support from the compiler, to raise at the most CPI.
- A larger issue parallelism (4 or 5 instructions per clock cycle) requires either a dynamically scheduled pipeline, or a VLIW processor (next chapter)

Multiple Issue

- Let us examine the basic operation of a superscalar processor.
- It fetches from IM (that is, from first level instruction cache) from 0 to N instructions each cock cycle (*bundle*), being N the largest number of instructions that IM can provide in parallel.
- Instructions are forwarded (in *bundles*) to an Instruction Queue (IQ) so that the CU ("dispatch Control Unit") can analyze them and check for possible hazards and dependencies.
- This Instruction Queue is depicted in figures Henessy-Patterson 3.2 and 3.29. According to the type of processor, the IQ has a capacity of some tenths of entries (the actual value depends on processor model, with newer processor having larger and larger queues)

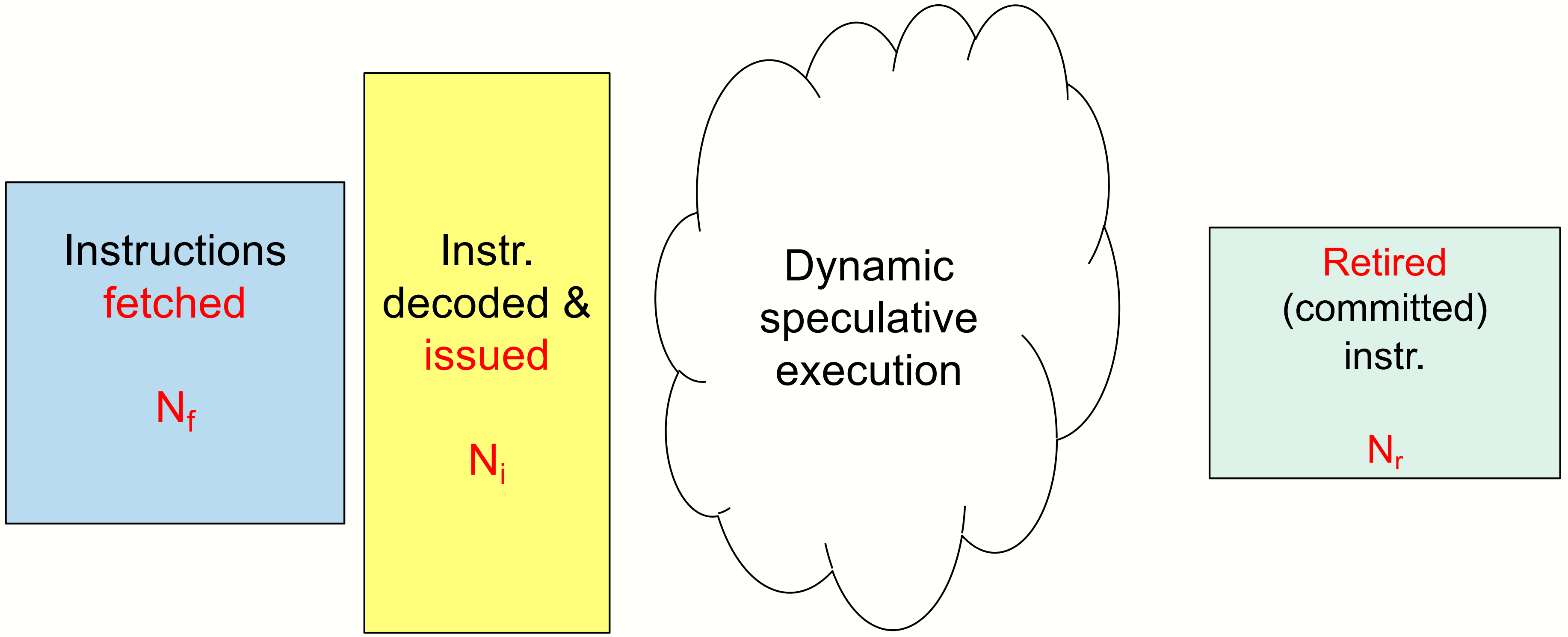
Multiple Issue

- The CU control logic checks for potential structural and data hazards among the instructions in a *bundle* and issues (that is, forwards to the reservation stations) some instructions, making room for more instruction in the queue.
- At the next clock cycle, another group (possibly the largest) of instructions is fetched from IM, and one more *bundle* is assembled.
- The number of instructions that the CPU can actually issue for execution is likely smaller to the number that can be fetched from IM.
- In the long run, the IQ gets filled up; if in a given clock cycle M instructions are issued for execution, a maximum of $M \leq N$ can be fetched from a bundle at the next clock cycle.

Multiple Issue

- In the worst case, if IQ is filled up, and in the preceding clock cycle no instruction has been sent to the EXECUTE phase because of hazards, no further instruction can be fetched from IM.
- Should this happen even if there is room in IQ, since the processor might fetch a number of instructions smaller to the free entries in IQ?
- Also keep in mind that, in the long run, the CPU must check for dependences and hazards among some tenths of instructions, which requires thousands of cross checks (in one or two clock cycles!)
- If the processor supports speculation (the most common case indeed), the CPU must also be able to carry out the *commit* of multiple instructions in the same clock cycle, otherwise the ROB quickly becomes the system bottleneck.

Multiple Issue



Multiple Issue: example

```
Loop:   LD     F0, 0 (R1)
        FADD   F4, F0, F2
        SD     F4, 0 (R1)
        ADD    R1, R1, -8
        BNE    R1, R2, Loop;        //branches if R1 <> R2
```

- Let us use Tomasulo's scheme in a dynamically schedule *superscalar* version of MIPS featuring one ALU and one F.P. unit, capable of *issuing two* instructions per clock cycle, with *no speculation*. Let us assume that the BNE branch is correctly predicted thanks to a branch target buffer.
- In the following chart is depicted the situation of the first three iterations, scheduled dynamically. (I = Issue, X = Execute, M=Memory access, W=Write results to CDB)

Multiple Issue: example

- Further assumptions:
    - Phase X (execute) in FADD requires 3 clock cycles.
    - Optimal branch prediction, but instructions after the branch cannot proceed to X until the branch condition is evaluated (there is no speculation).
    - The computed value is written onto the CDB at the end of the clock cycle in which it is produced, and thus it is available to the various reservation stations waiting for it only at the end of the subsequent clock cycle
    - The ALU is used both for integer operations and for load and store address computation.

Multiple Issue: example

clck	LD	FADD	ST	ADD	BNE	LD	FADD	ST	ADD	BNE	LD	FADD	ST	ADD	BNE
1	I	I													
2	X		I	I											
3	M		X		I										
4	W			X		I	I								
5		X		W				I	I						
6		X			X					I					
7		X				X					I	I			
8		W				M		X					I	I	
9			M			W			X						I
10							X		W						
11							X			X					
12							X				X				
13							W				M		X		
14								M			W			X	
15												X		W	
16												X			X
17												X			
18												W			
19													M		

Multiple Issue: example

- 15 instructions (three iterations) are carried out in 19 clock cycles, with a CPI of 19/15 = 1,27
- *Can this performance be improved?*
- Note that the ALU, used both for integer operations and for addresses computation, becomes a bottleneck.
  - With two separate ALUS, the instructions would complete in 12 clock cycles, instead of 19.
- Furthermore, the FP unit is under-used (a single FP operation per clock cycle), and 2 of the 5 instructions are only for loop management and are repeated in each iteration (ADD and BNE)
  - It is possible to apply static loop unrolling, to increase the number of FP operations and to decrease loop management overhead (a feature to be discussed in the next chapter)

Why is dynamic ILP so good ?

- Processors with dynamic ILP try to minimize structural, data and control hazards at run time, and must carry out a very complex set of actions in a few clock cycles
- What about moving all work required to exploit parallelism embedded in instructions over to the compiler, that has much more time to analyze and solve (when possible) the various hazards in a program?
- The main reasons are three

1. **Cache miss cannot be foreseen statically**, and dynamic ILP can partially hide them by executing other instructions, while the instruction that caused the miss is waiting for the missing data to be fetched from RAM into the cache.

Why is dynamic ILP so good?

2. **Branches cannot be statically predicted with proper accuracy** and dynamic BP and speculation increase the probability to carry out useful work well in advance with respect to the moment when the outcome of the branch instruction is known.

3. **Static ILP works well only on a specific architecture**, as will be discussed in detail in the next chapter. With dynamic ILP programs can be distributed and get executed on different architectures (provided they support the ISA) having a different number of F.U., registers for renaming, pipeline stages, type of branch prediction (as an instance, many Pentium, Core duo, AMD and the like)

- Dynamic ILP works, but is it really good?

Theoretical limits of dynamic ILP

- We have discussed many complex techniques to exploit at run-time the parallelism embedded in a program instructions.
- But, set aside the practical limitations (due to effective availability of hardware resources), how much room is there for parallelism?
- The only limitations than cannot be overcome are those due to real data dependences:

    LD **F0**, 0 (R1)

    ADD F4, **F0**, F2

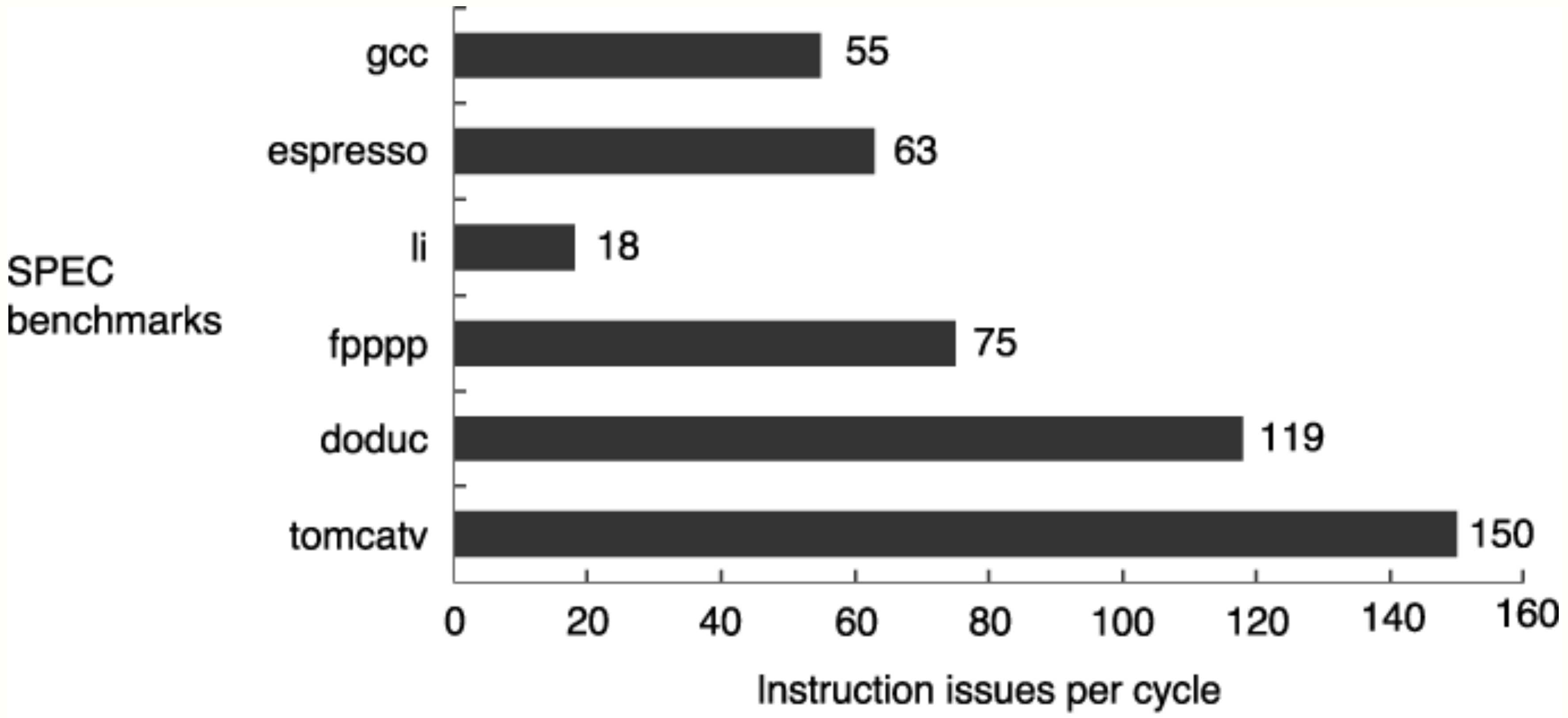
- all other limitations can be overcome with enough hardware and enough info about them

Theoretical limits of dynamic ILP

- Let us do the following assumption on a theoretical CPU:
  - **register renaming**: the CPU has an infinite number of registers for renaming. So all WAW and WAR are eliminated and an arbitrary number of instructions can be executed concurrently
  - **Branch prediction**: optimal
  - **Memory-address alias analysis**: all RAM addresses are known, so that RAM based name dependences can always be avoided. For example, it is known if #57 (R5) = #10 (R1)
  - **Multiple issue**: unlimited
  - **Cache memory**: no miss

Theoretical limits of dynamic ILP

- Here are the results for a few benchmarks. (Hennessy-Patterson, Fig. 3.35)



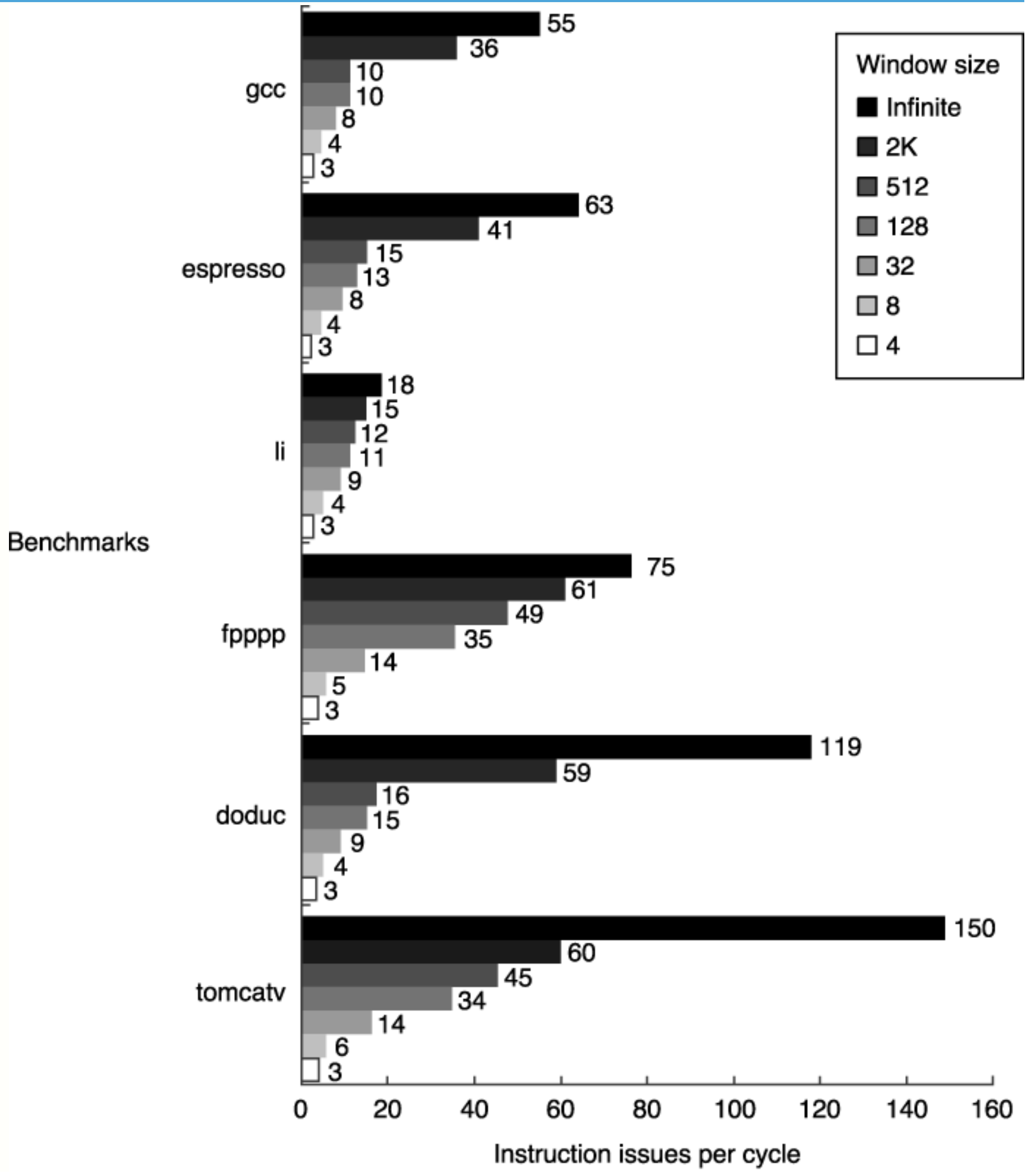
- note that FP programs usually have more parallelism to be exploited at the loop level

Theoretical limits of dynamic ILP

- So, what if one limits the number of consecutive instructions that can be examined for dependences analysis? (still maintaining optimal branch prediction)
- The amount of work to be done (quickly ! in one or two clock cycles) can be enormous. Some estimations:
  - 2000 instructions: 4 millions of comparisons
  - 50 instructions: 2500 comparisons
- The last figure is the average number of comparisons affordable in a modern CPU (in one or two clock cycles!)

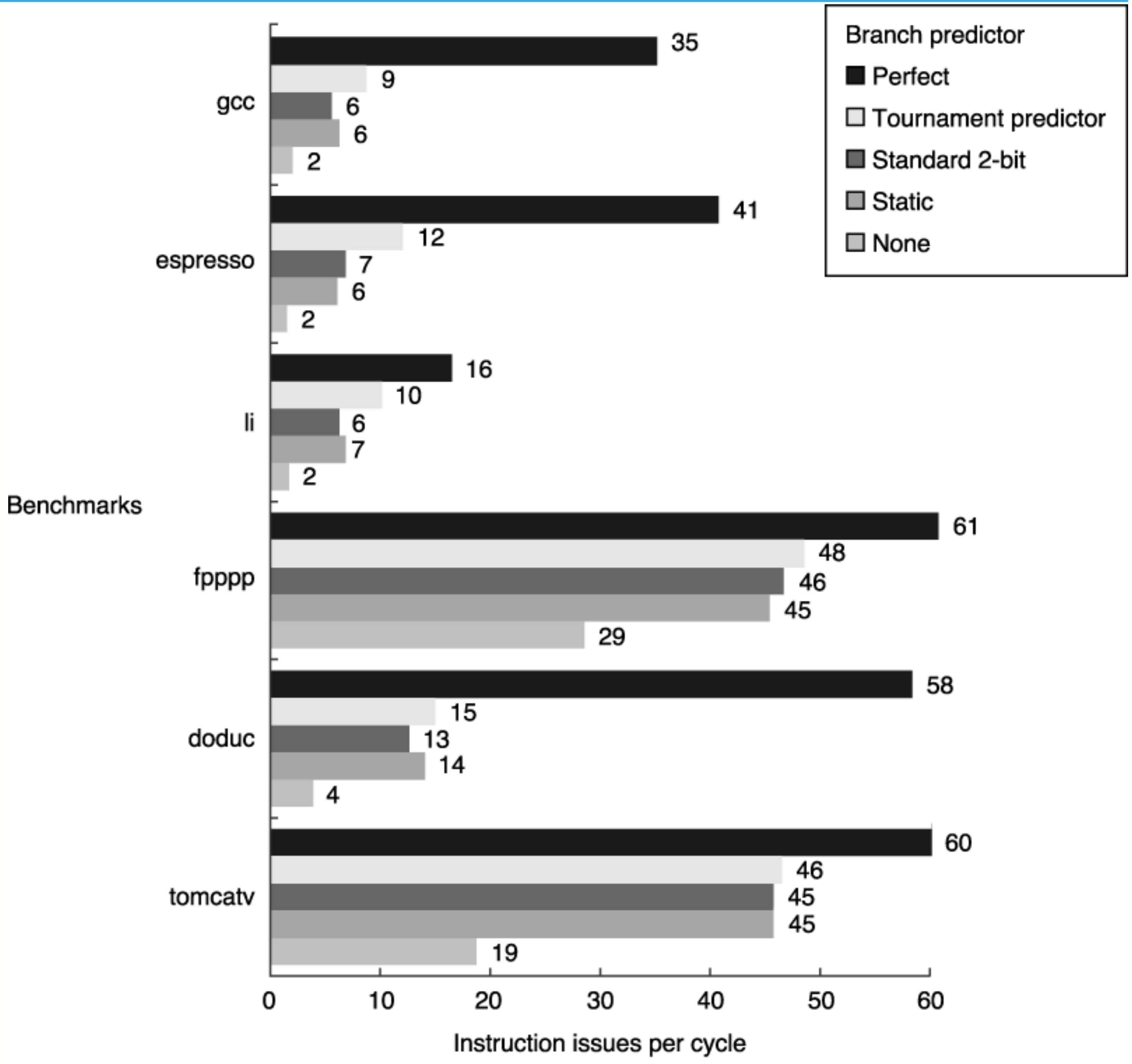
Theoretical limits of dynamic ILP

Here is the decrease in ILP actually available (Hennessy -Patterson, Fig 3.37):



Theoretical limits of dynamic ILP

Let us now add a limitation on branch prediction, by using different actual modes (Hennessy-Patterson, Fig. 3.39)



Theoretical limits of dynamic ILP

- Clearly, by introducing just a few real limitations, actually available ILP diminishes quickly.
- If one takes into account other factors, such as a limited number of registers for renaming, non-perfect RAM references analysis, cache miss, and so on, true ILP is further limited.

Some science(fiction)

- Clearly, it is possible to extract more and more ILP by increasing the number of avalialble resources (cache, registers, circuitry for dependencs analysis). Is there anything else?
- Some research issue hint to some form of **value prediction** (VP).
- VP consists of trying to predict vales produced by instructions, and effective addresses used in LOAD and STORE

Some science(fiction)

- If the result of one operation can be predicted, it can also be forwarded to dependent instructions that wait for it as their operand.
- This is a form of instruction speculation, not branch speculation, and it would allow for the concurrent execution of mutually dependent instructions.
- Thus is useful only if the prediction has good chances of being correct, which can indeed happen in some situations.
- What would be the benefit of a perfect prediction capability?

Some science(fiction)

- Another form of prediction can be tried on LOAD and STORE addresses, allowing to re-order such memory operations without incurring in WAW or WAR hazards.
- Finally, some studies hint to a possible branch speculation involving multiple branches, possibly nested, up to a number of 8 consecutive branches...