# Arquitecturas de Alto Desempenho

*Instruction-Level Parallelism (Complements)*

António Rui Borges

# Summary - 1

- *Efficiency of pipelining*
- *Data dependences and hazards*
  - *Data dependences*
  - *Name dependences*
  - *Data hazards*
  - *Control dependences*
- *Basic compiler techniques for exposing ILP*
- *Advanced branch prediction*
  - *Correlating branch predictors*
  - *Tournament predictors*
- *Dynamic scheduling*
  - *Scoreboarding*
  - *Tomasulo's algorithm*

Departamento de Electrónica, Telecomunicações e Informática

# Summary - 2

- *Hardware-based speculation*
- *Exploiting ILP using multiple issue*
- *Statically scheduled superscalar processor*
  - *ARM Cortex-A8*
- *Dynamically scheduled superscalar processor*
  - *Intel Core i7*
- *Suggested reading*

Departamento de Electrónica, Telecomunicações e Informática

# *Efficiency of pipelining - 1*

The simplicity of a instruction set is a fundamental property in building a pipeline for its implementation. Simple instruction sets offer yet another advantage as they are instrumental in making the scheduling of individual instructions more straight-forward. These advantages seem to be so significant that almost all recent pipelined implementations of complex instruction sets actually translate first the instructions into simple RISC-like operations and only then proceed to their pipelining and scheduling.

As a means to express the degree of success obtained by running a given program in a pipelined processor, the following equation can be used

$$\text{CPI}_{\text{prog}} \ = \ \text{CPI}_{\text{ideal}} + \text{structural stalls} + \text{data stalls} + \text{control stalls} \ ,$$

where the $\text{CPI}_{\text{ideal}}$, clock cycles per instruction in the ideal situation, is a measure of the maximum performance attainable by the pipelined implementation and the different types of stalls are assumed to be average values per instruction.

# *Efficiency of pipelining - 2*

The amount of parallelism available within the *basic block* of a program – a code segment with no branches in, except for the entry point, and no branches out, except for the exit point – is quite small. For a typical MIPS program, for instance, the average dynamic branch frequency is often between 15% and 25%, which means that as few as three to six instructions execute between a pair of branches. Furthermore, since these instructions are likely to depend upon one another, the amount of instruction overlap that can be exploited within a basic block, is likely to be less than the average block size. Therefore, to obtain substantial performance enhancements, ILP must be exploited across multiple basic blocks.

There are two largely distinct approaches to fulfill this goal

- an approach that relies on software technology to elicit parallelism statically at compile time
- an approach that relies on hardware to help to discover and exploit dynamically the inherent parallelism during execution.

# *Data dependences and hazards*

Determining how one instruction depends upon another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit ILP efficiently one must determine first which instructions can be executed in parallel. If two instructions are *independent* from one another, they can be executed without hindrance in a pipeline of arbitrary depth (no stalls need to be inserted), assuming there are sufficient resources (i.e. no structural hazards exist). If one instruction *depends* upon another, there will be time restrictions enforcing their execution, they must proceed in order and may often be only partially overlapped.

Three types of dependences are relevant
- *data* dependences, also called *true data* dependances
- *name* dependences
- *control* dependences.

An instruction $j$ is said to be *data dependent* upon an instruction $i$ if and only if either of the following conditions hold

- the instruction $i$ produces a value which is used by the instruction $j$
- the instruction $j$ is data dependent upon an instruction $k$ and the instruction $k$ is data dependent upon the instruction $i$
- the instructions $j$ and $i$ are connected by a chain of dependences of the second type.

Note that a dependence within an instruction, `DADD R1,R1,R1`, for instance, is not considered a data dependence.

Departamento de Electrónica, Telecomunicações e Informática

# Data dependences - 2

Consider the following code sequence

```
LOOP:     L.D      F0,0(R1)
          ADD.D    F4,F0,F2
          S.D      F4,0(R1)
          DADDUI   R1,R1,-8
          BNE      R1,R2,LOOP
```

For simplicity, the effects of delayed branches are ignored. Data dependencies between instructions are represented by vertical arrows.

The existence of dependences implies that there will be a chain of one or more data hazards between successive instructions. Executing the instructions in a pipelined processor with an interlocking unit and a pipeline depth longer than the distance between the instructions measured in clock cycles, causes the processor to detect a hazard and stall, if it cannot be resolved by forwarding. Thus, reducing the overlap. In a pipelined processor without an interlocking unit, on the other hand, it is the compiler responsibility to schedule dependent instructions in a manner that they will not overlap completely, since otherwise the program will not run correctly.

Departamento de Electrónica, Telecomunicações e Informática

# *Data dependences - 3*

The presence of a data dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved. Dependences are, therefore, a property of *programs*. On the other hand, whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall, are properties of the *pipeline organization*. This difference is crucial to understand how ILP can be exploited.

A data dependence conveys three ideas
- the possibility of a hazard
- the specification of the order the results must be computed
- an upper bound on how much parallelism can possibly be exploited.

A dependence can be overcome in two different ways
- maintaining a dependence, but avoiding a hazard
- eliminating a dependence by transforming the code.

Departamento de Electrónica, Telecomunicações e Informática

# Data dependences - 4

Scheduling the code is the primary method to avoid a hazard without altering a dependence and such scheduling can be done both by the compiler and by the hardware.

A data value is communicated between instructions either through a register of the register bank, or a memory location. When the flow of information occurs through a register, detecting the dependence is straightforward since the register names are fixed for each instruction; although it may become more complicated if branches intervene, since correctness concerns force the compiler or the hardware to be conservative. When the flow of information takes place through memory locations, the detection is more difficult since two addresses may refer to the same memory location, but look different. In addition, the effective address of a load or store instruction may change from one execution of the instruction to the next, further complicating the detection of a dependence.

Departamento de Electrónica, Telecomunicações e Informática

# Name dependences - 1

A *name dependence* is present when two instructions use the same register or memory location, called *name*, but without any flow of information taking place between them. There are two types of *name dependences*

- an *antidependence* between an instruction $i$ and an instruction $j$ occurs when the instruction $j$ writes to a register or memory location that the instruction $i$ reads – the original instruction ordering must be preserved to ensure that the correct value is read

- an *output dependence* between an instruction $i$ and an instruction $j$ occurs when both instructions $i$ and $j$ write a value to the same register or memory location – the original instruction ordering must be preserved to ensure that the value finally written is the correct one.

Departamento de Electrónica, Telecomunicações e Informática

# Name dependences - 2

Since a *name dependence* is not a true data dependence, as there is no value transmitted between the involved instructions, the instructions themselves may execute simultaneously, or be reordered, provided the *name* (register or memory location) used in the instructions is modified in one of them to remove the conflict.

This renaming can be more readily done for register operands, where it is called *register renaming*. Register renaming can be performed either statically by the compiler, or dynamically by the hardware.

Departamento de Electrónica, Telecomunicações e Informática

# Data hazards - 1

A *data hazard* exists whenever there is a data or name dependence between instructions and they are close enough to generate, by their overlap in the pipeline during execution, a change in the order of access to the operand involved in the dependence.

Because of the dependence, the *program order of execution* must be preserved, that is, the order of instruction execution if they were taken one at the time and executed sequentially as determined by the original source code. The goal of both software and hardware techniques is to exploit parallelism by preserving program order *only when it affects the outcome of the program*, and not in all circumstances.

Detecting and avoiding hazards ensures that the necessary program order is preserved.

*Data hazards* can be classified in three different categories depending on the combination of operand accesses present in the instructions. A name convention is used that portrays the instruction order that must be preserved by the pipeline.

Consider two instructions $i$ and $j$, with $i$ preceeding $j$ in the program, then the possible hazards are

- RAW (*read after write*) – $j$ tries to read the operand before $i$ writes a value to it, so $j$ gets a wrong value; it is the most common form of data hazard and corresponds to a true data dependence

- WAW (*write after write*) – $j$ tries to write a value to an operand before $i$ writes its value to it, so the final value is wrong; it arises in pipelines that allow writing in more than one pipe stage, or allow out of order instruction completion, and corresponds to an output dependence

- WAR (*write after read*) – *j* tries to write a value to an operand before *i* reads it, so *i* gets a wrong value; it cannot occur in most static issue pipelines, even deeper pipelines or floating point pipelines, because usually all reads are early and all writes are late; it arises when there are some instructions that write results early in the pipeline and other instructions that read operands late, or when out of order execution is allowed.

Notice that the RAR (*read after read*) is not a data hazard. The combination of operations is idempotent.

A *control dependence* determines the ordering of an instruction *i* with respect to a branch, so that the instruction *i* is executed in the correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches and, in general, these control dependences must be preserved to maintain the program order.

Consider the code segment

**if** (cond1) S1;
**if** (cond2) S2;

`S1` is control dependent on condition `cond1` and `S2` is control dependent on condition `cond2`, but not on condition `cond1`.

Two constraints are in general imposed by control dependencies

- an instruction that is control dependent on a branch, cannot be moved *before* the branch so that its execution is *no longer controlled* by the branch
- an instruction that is not control dependent on a branch, cannot be moved *after* the branch so that its execution is *controlled* by the branch.

# *Control dependences - 2*

When processors preserve strict program order, they ensure that control dependences are also preserved. One should point out, however, that it is permissible, although inefficient, to execute instructions that should not have been executed, thereby violating the control dependences, <u>if this can be done without affecting the program correctness</u>. Which means that control dependence is not *the critical* property that must be preserved. The two critical properties to program correctness – and normally preserved by maintaining both data and control dependences – are *exception behavior* and *data flow*.

# Control dependences - 3

Preserving the exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program. Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program.

Consider the code segment (again the effects of delayed branches are ignored)

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

It is clear that if the data dependence involving the register R2 is not maintained, the result of the program may be changed. Less obvious is the fact that if one ignores the control dependence and move the *load* instruction to a place just before the branch, this instruction may cause a memory protection violation.

To be able to reorder the instructions, and still preserve the data dependence, it would be necessary to ignore the exception when the branch is taken. Later on, a hardware technique, called *speculation*, will be studied and, as it will be seen, will allow the overcoming of this problem.

# Control dependences - 4

The *data flow* is the actual flow of data values among instructions that produce them and instructions that consume them. Branches make the data flow dynamic, since they allow the data for a given instruction to come from different source locations, thereby an instruction may be data dependent on several predecessor instructions. Program order is what in fact determines which predecessor will deliver the value at each moment and program order is ensured by maintaining the control dependences.

Consider the code segment (again the effects of delayed branches are ignored)

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

As it can be seen, the value of the register `R1` used by the *or* instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness, control dependence is also required. As for the exception problem, *speculation* will also help to lessen the impact of control dependence, while keeping the data flow.

Departamento de Electrónica, Telecomunicações e Informática

Sometimes it can be determined that violating the control dependence will neither affect the exception behavior, nor the data flow.

Consider the code segment (again the effects of delayed branches are ignored)

```
            DADDU    R1,R2,R3
            BEQZ     R12,SKIP
            DSUBU    R4,R5,R6
            DADDU    R5,R4,R9
                . . .
    SKIP:   OR       R7,R8,R9
```

Suppose it is known that the registers R4 and R5, destinations of the *subtract* and *add* instructions, respectively, are not used after SKIP – the property of whether a value will be used by an upcoming instruction is called *liveness*. Then, changing the values of R4 and R5 just before the branch would not affect the data flow, since R4 and R5 would be dead in the code region after SKIP.

This type of code scheduling is also a form of speculation, often called *software speculation*. The compiler is betting on the branch outcome, the bet being the branch is usually not taken.

# Basic compiler techniques for exposing ILP - 1

To keep a pipeline full, the parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall, the execution of a dependent instruction must be separated from the source instruction by a distance in clock cycles at least equal to the pipeline latency of that source instruction.

The compiler ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the pipeline functional units.

As an example, consider how the compiler can increase the amount of available ILP by transforming loops as the one bellow

$$\textbf{for } (i = 999; i >= 0; i{-}{-})$$
$$x[i] \mathrel{+}= s;$$

As it is immediately seen, the loop is highly parallel: each iteration is totally independent from any other one.

Departamento de Electrónica, Telecomunicações e Informática

# Basic compiler techniques for exposing ILP - 2

**Straightforward MIPS code of the loop (delayed branches are ignored)**

```
LOOP:      L.D       F0,0(R1)
           ADD.D     F4,F0,F2
           S.D       F4,0(R1)
           DADDUI    R1,R1,-8
           BNE       R1,R2,LOOP
```

It is assumed that the register `R1` contains initially the address of the element `x[999]` of the array and that the contents of the register `R2+8` is the address of the element `x[0]`.

# Basic compiler techniques for exposing ILP - 3

## Latencies of FP operations

Source: Computer Architecture: A Quantitative Approach

| Instruction<br>producing the result | Instruction<br>using the result | Latency<br>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 |

Departamento de Electrónica, Telecomunicações e Informática

# Basic compiler techniques for exposing ILP - 4

**MIPS code of the loop, with elimination of data dependencies**
**(delayed branches are ignored)**

|         |         |              | clock cycle |
|---------|---------|--------------|:-----------:|
| 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           |

The processing of each element takes 9 clock cycles (it assumes that there is no forwarding at the ID stage).

# *Basic compiler techniques for exposing ILP - 5*

**MIPS code of the loop, scheduled for the pipeline
(delayed branches are ignored)**

|              |        |            | *clock cycle* |
|--------------|--------|------------|:-------------:|
| LOOP:        | L.D    | F0,0(R1)   | 1             |
|              | DADDUI | R1,R1,-8   | 2             |
|              | ADD.D  | F4,F0,F2   | 3             |
|              | *stall* |           | 4             |
|              | *stall* |           | 5             |
|              | S.D    | F4,8(R1)   | 6             |
|              | BNE    | R1,R2,LOOP | 7             |

The processing of each element takes 7 clock cycles.

# *Basic compiler techniques for exposing ILP - 6*

**Straightforward MIPS code of the loop, with loop unrolled 4 times (delayed branches are ignored)**

```
LOOP:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)
        L.D     F6,-8(R1)
        ADD.D   F8,F6,F2
        S.D     F8,-8(R1)
        L.D     F10,-16(R1)
        ADD.D   F12,F10,F2
        S.D     F12,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F16,F14,F2
        S.D     F16,-24(R1)
        DADDUI  R1,R1,-32
        BNE     R1,R2,LOOP
```

In real programs, the loop upper bound is not usually known. Assuming it to be $n$ and unrolling the loop $k$ times, two consecutive loops are generated: the first iterates $n$ mod $k$ times and has a body equal to the original loop; the second iterates $n / k$ times and has the unrolled body.

Departamento de Electrónica, Telecomunicações e Informática

# *Basic compiler techniques for exposing ILP - 7*

**MIPS code of the loop, with loop unrolled 4 times and scheduled for the pipeline (delayed branches are ignored)**

```
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
```

The processing of each element takes 3.5 clock cycles.

# *Basic compiler techniques for exposing ILP - 8*

Decisions and transformations required to unroll a loop can be summarized in the following points

- determination if unrolling the loop is really useful by finding the degree of independence among its iterations
- use of different registers to avoid unnecessary constraints that would arise by the use of same registers for different computations
- elimination of the extra test and branch instructions and adjustment of the loop termination and iteration code
- determination if the loads and stores in the unrolled loop can be inter-changed – this transformation requires analyzing the memory addresses
- scheduling the code by preserving any dependencies needed to yield the same result as the original code.

The key requirement underlying all these transformations is the understanding of how one instruction depends upon another and how the different instructions can be changed and/or reordered, given the dependencies.

Departamento de Electrónica, Telecomunicações e Informática

Three different effects limit the gains from loop unrolling

- a decrease in the amount of overhead saved with each unroll – as predicted by the Amdhal's Law

- code size limitations – for large loops, the code size growth may lead to an increase in the instruction cache miss rate; furthermore, one has to be also concerned with the potential shortfall in registers, called *register pressure*, that is created by applying aggressive unrolling and scheduling strategies

- compiler limitations – the use of sophisticated high-level transformations, whose potential improvements are difficult to measure before detailed code generation, has led to significant increases in the complexity of modern compilers.

# Advanced branch prediction

Branches will hurt pipeline performance because enforcing control dependencies generate hazards which require stalling the pipeline progress in many situations. Loop unrolling is a method to reduce the number of control hazards during execution. The performance loss of branches, however, is more adequately dealt with if their behavior can be predicted.

Simple branch predictors that rely either on compile time information, or on the observed dynamic behavior of a branch in isolation, have already been studied. As the number of instructions in flight, that is, whose execution is taking place or are being considered for, has increased significantly in the past two decades, the need for accurate branch prediction has become more pressing.

Departamento de Electrónica, Telecomunicações e Informática

# *Correlating branch predictors - 1*

The *2-bit prediction* scheme takes into consideration the recent behavior of a single branch to predict its future behavior. The prediction accuracy is improved if one considers the recent behavior of *other* branches in the program as well.

To see why this idea is relevant, look at the following code segment where the effects of delayed branches are ignored.

```
        DADDIU  R3,R1,-2
        BNEZ    R3,L1           ; branch b1: R1 ≠ 2?
        DADD    R1,R0,R0        ; R1 = 0
L1:     DADDIU  R3,R2,-2
        BNEZ    R3,L2           ; branch b2: R2 ≠ 2?
        DADD    R2,R0,R0        ; R2 = 0
L2:     DSUBU   R3,R1,R2
        BEQZ    R3,L3           ; branch b3: R1 = R2?
```

Notice that the behavior of branch *b3* is correlated to the behavior of branches *b1* and *b2*: if both branches are *untaken*, then branch b3 is *taken*. A predictor that uses only the behavior of a single branch to predict what happens next, can never capture such a behavior.

Departamento de Electrónica, Telecomunicações e Informática
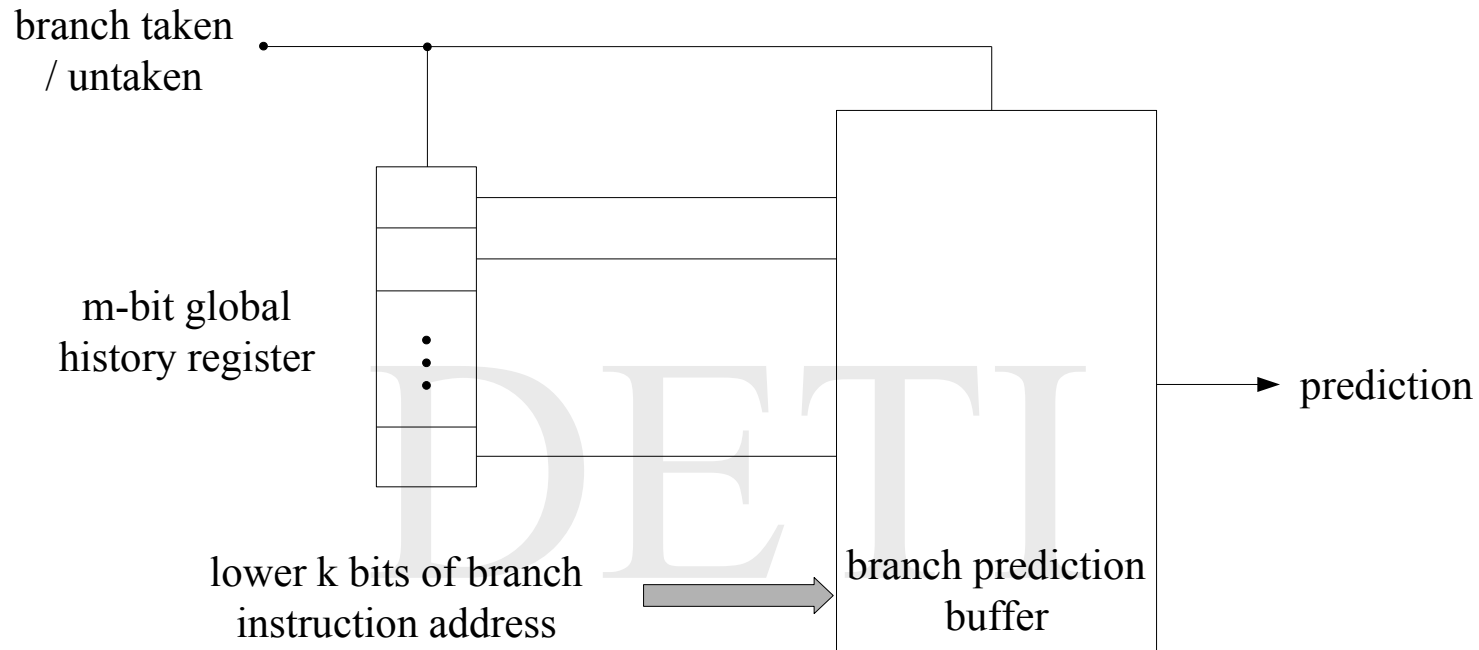
# *Correlating branch predictors - 2*

Branch predictors which add the execution behavior of other branches, besides their own, to make a prediction, are called *correlating predictors*.

A (1,2) *correlating branch predictor*, for instance, considers the execution behavior of the previous branch to choose among a pair of 2-bit branch predictors in predicting a particular branch. In general, a $(m,n)$ *branch predictor* adds the execution behavior of previous $m$ branches to choose among $2^m$ $n$-bit branch predictors in predicting a particular branch.

The popularity of this kind of branch predictors is that it can yield a higher prediction rate than the 2-bit prediction scheme, while requiring a trivial amount of additional hardware: the global history of the most recent $m$ branches can be recorded in a $m$-stage shift register, called the *global history register*, where the bit contents at each stage specifies whether the branch of the corresponding order was *taken*, 1, or *untaken*, 0; the *branch prediction buffer* is now indexed by the concatenation of the $m$-bit global history with the low-order $k$ address bits of the branch instruction.

# *Correlating branch predictors - 3*



A 2-bit predictor with no global history is simply a (0,2) correlating branch predictor and, in comparing branch predictors, the branch predictor buffer size should be kept invariant, that is,

$$2^m \cdot n \cdot 2^k \ = \ \text{constant} \ .$$

# *Correlating branch predictors - 4*

A (2,2) correlating branch predictor with 1K entries is compared with (0,2) simple predictors with 4K entries and an unlimited number of entries.

**Comparing 2-bit branch predictors of different types in SPEC89 benchmarks**
Source: Computer Architecture: A Quantitative Approach

Departamento de Electrónica, Telecomunicações e Informática
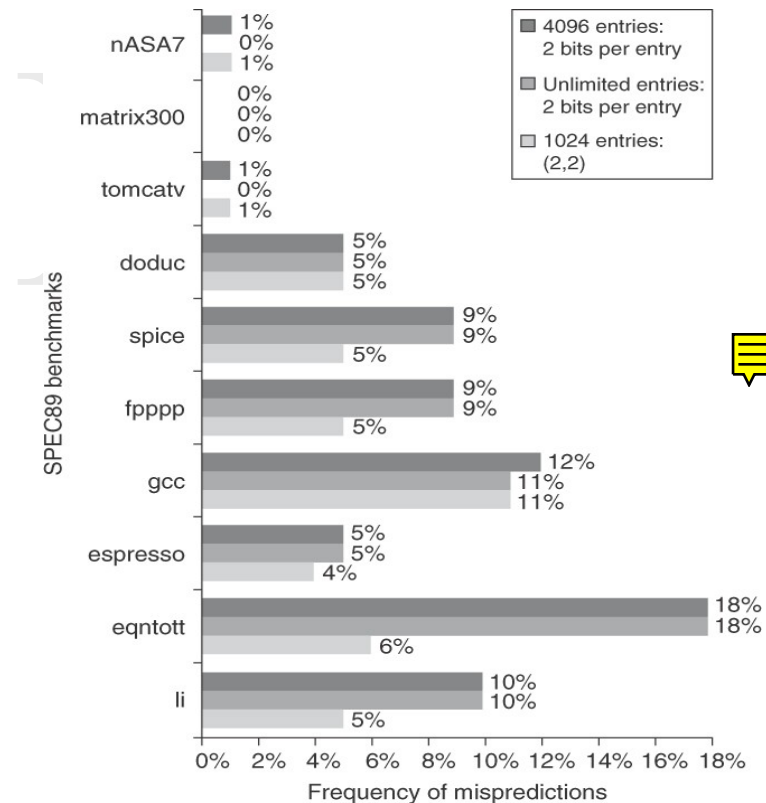
# Tournament predictors - 1

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that, by adding global information, the performance could be improved. *Tournament predictors* take this insight to the next level by using multiple predictors, usually one based on global information and one on local information, and combining them with a selector. Tournament predictors can achieve better accuracy at medium sizes of the branch predictor buffer and also make effective use of very large numbers of prediction bits.

Existing tournament predictors use a 2-bit hysteresis counter per branch to choose among two different predictors based on which predictor (local, global or a mix of the two) was the most effective in recent predictions. A 2-bit *hysteresis counter* requires two mispredictions in succession to change its state.

The advantage of tournament predictors is their ability to select the right predictor for a particular branch, which is specially crucial for integer programs. Typically, they select the global predictor almost 40% of the time for integer benchmarks and less than 15% of the time for floating point benchmarks.

# Tournament predictors - 2
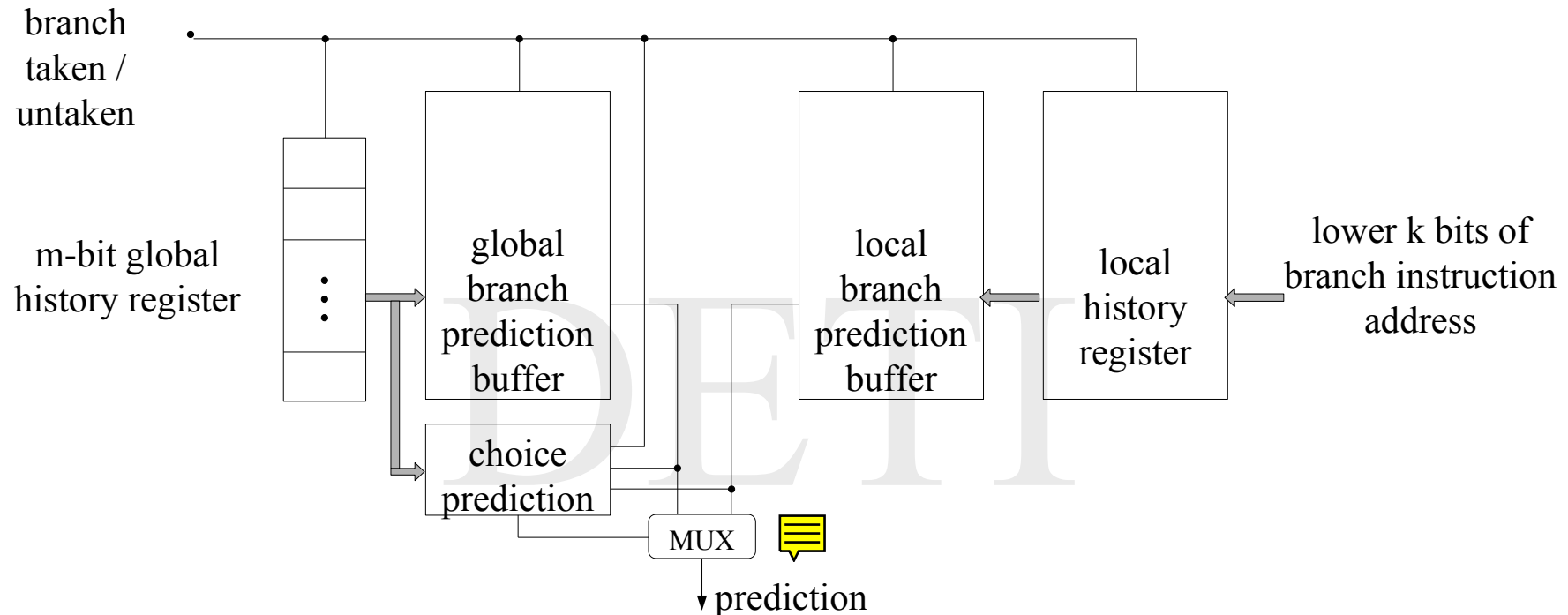


Both the *global branch prediction buffer* and the *choice prediction buffer* consist of $2^m$ $n$-bit predictors and the *local branch predictor buffer* of $2^u$ $v$-bit predictors. The *local history register*, on the other hand, consists of $2^k$ $u$-bit *local history registers*.

**Comparing the misprediction rate for three different predictors in SPEC89 benchmarks**

Source: Computer Architecture: A Quantitative Approach
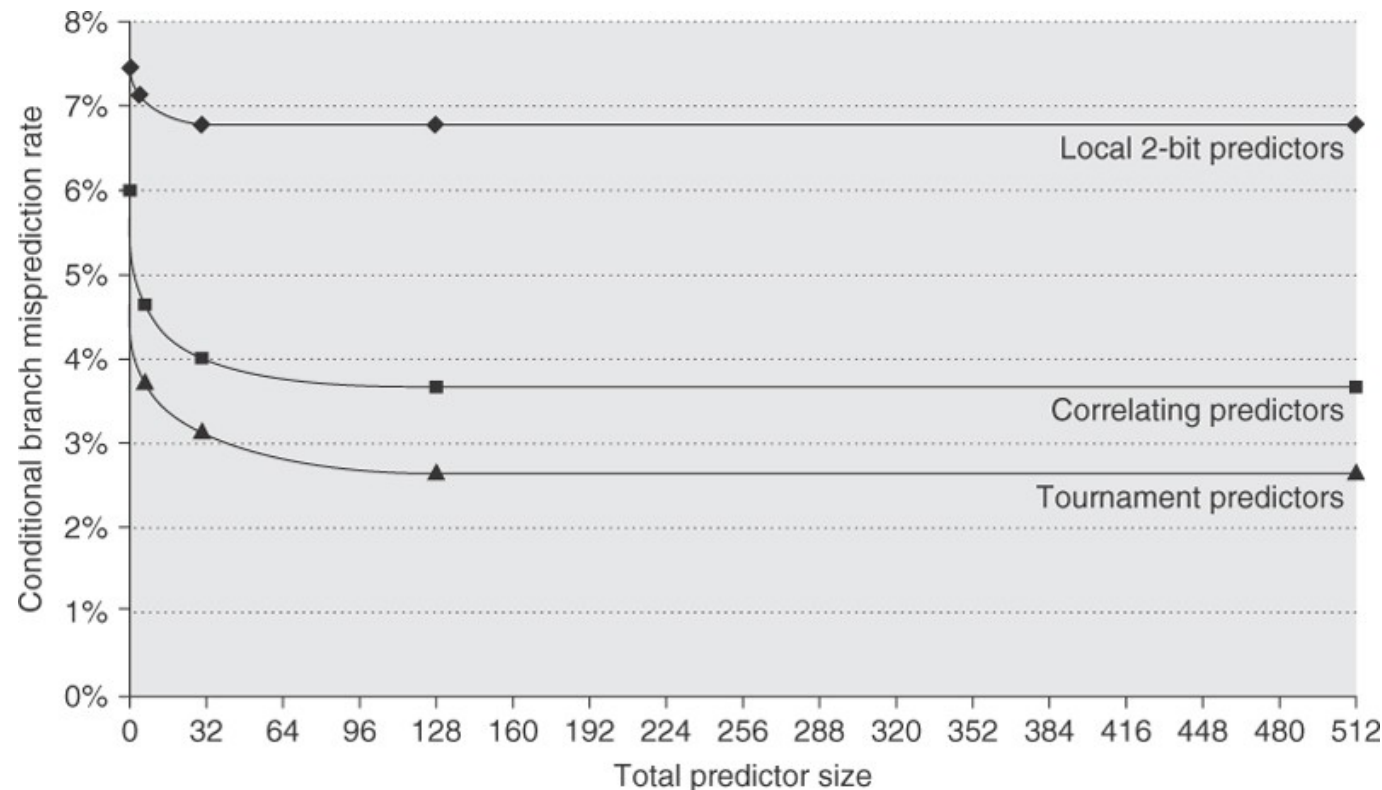


Although these data is from an older version of SPEC, recent SPEC benchmarks show similar behavior, perhaps converging to the asymptotic limits at slightly larger predictor sizes.

# *Dynamic scheduling - 1*

A major limitation of simple pipelining techniques is they use *in-order* instruction issue and execution. If a data dependence between an instruction in the pipeline and an incoming instruction occurs, which cannot be solved by the *forwarding unit*, the *interlocking unit* stalls the pipeline, starting at the instruction that uses the result. No new instructions are fetched or issued until the dependence is cleared.

*Dynamic scheduling* is another way of addressing the problem: the hardware rearranges instruction execution, while maintaining data flow and exception behavior, so that stalls are minimized. This entails, however, a significant increase in complexity.

There are several advantages resulting from the use of this technique
- it allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline, eliminating the need of multiple binaries
- it enables handling some cases where dependences are unknown at compile time, involving, for instance, memory references and data dependent branches or the use of dynamic linked libraries
- it allows the processor to cope with unpredictable delays, such as cache misses, by executing other code while waiting for the miss to resolve.

Consider the code segment.

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

The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet, SUB.D is not data dependent on the two preceding instructions. The performance limitation created by this hazard could be eliminated by <u>not</u> requiring instructions to execute in program order.

To fulfill this aim, the issue process could be decomposed in two parts: checking for structural hazards, and waiting for the absence of a data hazard. Thus, *in-order* instruction issue is still used, but an instruction may start execution as soon as its data operands are available. A pipeline with these features performs *out-of-order* execution, which also implies *out-of-order* completion.

*Out-of-order* execution introduces the possibility of WAR and WAW hazards, which did not exist in the classical 5-stage pipeline, the former, and only with multi-cycle floating-point operations that give rise to *out-of-order* completion, the latter.

Consider the code segment.

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

There is an *antidependence* between instructions ADD.D and SUB.D and an *output dependence* between instructions ADD.D and MUL.D. Both these hazards could be removed if *register renaming* is used.

# Dynamic scheduling - 4

*Out-of-order* completion also creates major complications in handling exceptions. Dynamic scheduling must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program were executed in strict program order, *actually* do arise. Dynamically scheduled processors preserve exception behavior by delaying the notification of an associated exception until the processor knows that the involved instruction is the next one completed.

Although exception behavior is to be preserved, dynamically scheduled processors could generate *imprecise* exceptions. *Imprecise* exceptions can occur because of the two facts bellow

- the pipeline may have *already completed* some instructions that succeed in program order the instruction causing the exception and whose result cannot be reversed
- the pipeline may have *not yet completed* some instructions that precede in program order the instruction causing the exception.

The way to solve this problem and get precise exceptions, will be discussed later on in the context of *speculative* processors.

Departamento de Electrónica, Telecomunicações e Informática

# *Dynamic scheduling - 5*

To allow *out-of-order* execution, the ID stage of the classical 5-stage pipeline is split into two stages
- *issue* – instruction decoding and checking for structural hazards
- *read operands* – waiting until all data hazards are cleared before reading the operands.

An *instruction fetch* stage precedes the *issue* stage and the instruction that has been fetched is placed either into an *instruction register*, or into a *queue of pending instructions*; instructions are then issued from the register or the queue if conditions allow. The *execution* stage follows the *read operands* stage. Depending on the operation, the execution stage may take a variable number of cycles.

In this sense, the moment an instruction *begins* execution must be distinguished from the moment the instruction *completes* execution; the instruction is *in execution* between these two times. Having multiple instructions *in execution* at the same time requires multiple functional units, pipelined functional units or both. Since these two capabilities are essentially equivalent on what concerns pipeline control, it will be assumed the processor has multiple functional units.

Departamento de Electrónica, Telecomunicações e Informática

# Dynamic scheduling - 6

In a dynamically scheduled pipeline, all instructions pass through the *issue* stage *in-order*. They can, nevertheless, be stalled and overtaken by others at the *read operands* stage and enter in execution *out-of-order*.

There are two basic techniques that allow instructions to execute *out-of-order* when there are sufficient resources and no data dependencies among them: *scoreboarding* was the first technique to be introduced, it came up in the design of the CDC 6600 supercomputer in the middle of 1960s; *Tomasulo's algorithm* was the second, developed by Robert Tomasulo in 1967 and applied to the floating point unit of IBM 360/91.

The primary difference between them is that the *Tomasulo's algorithm* handles antidependences and output dependences by effectively renaming dynamically the registers. Additionally, it can also be extended to handle *speculation*, a technique aimed to reducing the effect of control dependences by predicting the outcome of a branch through the execution of instructions at the predicted target address and taking corrective actions when the prediction is wrong.

# Scoreboarding - 1

The goal of a *scoreboard* is to try to maintain an execution rate of one instruction per clock cycle, provided there are no structural hazards. Thus, instructions are executed as early as possible. When an instruction which has been issued is stalled, other instructions in the processing table, that do not depend on any active or stalled instruction, are looked up and if one is found, it will be executed. The *scoreboard* takes full control for instruction issue and execution, including all hazard detection.

On a processor with MIPS architecture, scoreboards makes sense primarily on the floating point unit, since the latency of the other functional units is very small. It will be assumed there are two multipliers, one adder, one divider and a single integer unit for all memory references, branches and integer operations.

Departamento de Electrónica, Telecomunicações e Informática

# Scoreboarding - 2

**Basic organization of a MIPS processor with a scoreboard**

Source: Computer Architecture: A Quantitative Approach

Departamento de Electrónica, Telecomunicações e Informática
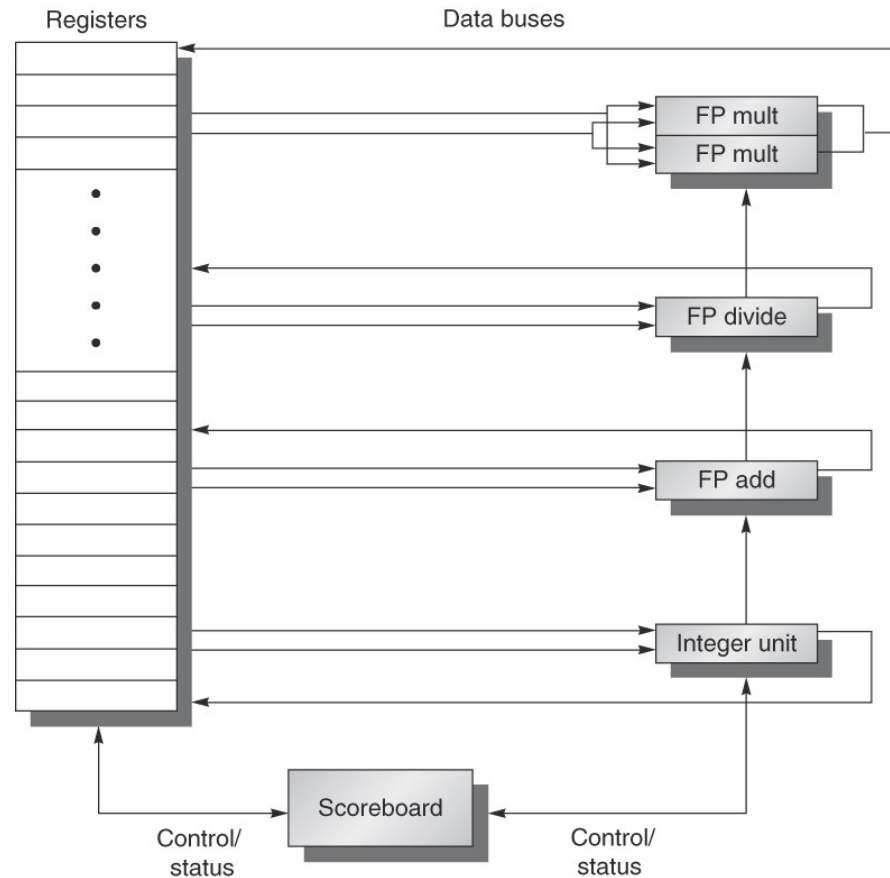
# *Scoreboarding - 3*

Every instruction goes through four processing steps while under scoreboard control. This is a simplification of the real situation actually, since memory access, necessary for load and store operations, is being disregarded. The four steps, which replace ID, EX and WB in the standard pipeline, are as follows

1. *Issue* – if a FP functional unit of the required type is free and no other active instruction has the same register as destination, the scoreboard issues the instruction to the unit and updates its internal data structure – this step replaces the first half of the MIPS pipeline ID stage; by ensuring that no other active functional unit writes its result into the same destination register, it is guaranteed that no WAW hazards are present; if a structural or a WAW hazard exists, instruction issue stalls and no further instructions will issue until these hazards are cleared; when the issue stage stalls, it causes the buffer between instruction fetch and issue to fill and instruction fetch to stall right away, if the buffer is a single register, or when the buffer is full, if it is a queue.

2. *Read operands* – the scoreboard monitors the availability of the source registers: a source register is available if no previous instruction, still in execution, is going to write it – this step replaces the second half of the MIPS pipeline ID stage; when the source operands are available, the scoreboard lets the functional unit to proceed to read the operands from the registers and begin execution; RAW hazards are resolved dynamically in this stage and instructions may be sent into execution out of order.

3. *Execution* – the functional unit starts execution upon receiving its operands; when the result is ready, it notifies the scoreboard that it has completed the operation – this step replaces the MIPS pipeline EX stage and takes a variable number of clock cycles in the MIPS FP pipeline.

4. *Write result* – once the scoreboard is aware the functional unit has terminated its operation, it checks for WAR hazards and, if required, prevents the instruction from completing – this step replaces the MIPS pipeline WB stage.

# Scoreboarding - 5

At first glance, it might appear the scoreboard has difficulty in distinguishing between RAW and WAR hazards. Because the operands for an instruction are read only when the contents of both source registers has the updated value, the scoreboard does not take advantage of forwarding. This is not a large penalty since the instructions write their result to the destination register as soon as they complete their execution (provided there are no WAR hazards). The consequence is a reduced pipeline latency and the benefits of forwarding are attained in this indirect manner. There is, however, an extra clock cycle added to the latency because *write the result* and *read the operand* operations cannot overlap.

Based on its own internal data structure, the scoreboard controls instruction progression from one step to the next through communication with the functional units. There is still a small complication. Since the number of buses connecting the register bank and the functional units are limited, leading to potentially structural hazards, the scoreboard must guarantee that the number of functional units allowed to proceed into steps 2 and 4 does not exceed the number of buses available.

# Scoreboarding - 6

The scoreboard *internal data structure* consists of three elements

- *instruction status* – it is a table with as many entries as the number of instructions under processing; for each instruction, it specifies which of the four states the instruction is in

- *functional unit status* – it is a table with as many entries as the number of functional units; each entry has nine fields

  · *busy* – it indicates whether the unit is busy or free
  · *op* – it indicates the operation to be performed in the unit
  · $F_i$ – number of the destination register
  · $F_j$, $F_k$ – numbers of the source registers
  · $Q_j$, $Q_k$ – identification of the functional units whose result is to be stored in the source registers $F_j$ and $F_k$
  · $R_j$, $R_k$ – flags signaling when the source registers are ready to be read and have not yet been read

- *register result status* – it is a single entry table with as many fields as there are registers in the register bank; it indicates which functional unit will write the register if an active instruction has the register as its destination.

# Scoreboarding - 7

Consider the code segment

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

There are true data dependences between the first `L.D` instruction and the `SUB.D` and `DIV.D` instructions, between the second `L.D` instruction and the `MUL.D`, `SUB.D` and `ADD.D` instructions, between the `MUL.D` and the `DIV.D` instructions and between the `SUB.D` and the `ADD.D` instructions, potentially leading to RAW hazards. There is also an antidependence between the `SUB.D` and `DIV.D` instructions and the `ADD.D` instruction and an output dependence between the first `L.D` instruction and the `ADD.D` instruction, potentially leading to WAR and WAW hazards, respectively.

Assume the following latencies: load – 1 clock cycle, addition – 2 clock cycles, multiplication – 6 clock cycles and division – 12 clock cycles.

# Scoreboarding - 8

## Components of the scoreboard internal data structure when the second load instruction is about to write the result to the destination register

| Instruction status | | | | |
|---|---|---|---|---|
| **Instruction** | **Issue** | **Read operands** | **Execution terminated** | **Write result** |
| L.D    F6,34(R2) | yes | yes | yes | yes |
| L.D    F2,45(R3) | yes | yes | yes | |
| MUL.D F0,F2,F4 | yes | | | |
| SUB.D F8,F6,F2 | yes | | | |
| DIV.D F10,F0,F6 | yes | | | |
| ADD.D F6,F8,F2 | | | | |

| Functional unit status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Name** | **busy** | **op** | **Fi** | **Fj** | **Fk** | **Qj** | **Qk** | **Rj** | **Rk** |
| integer | yes | load | F2 | R3 | | | | no | |
| mult1 | yes | mult | F0 | F2 | F4 | integer | | no | yes |
| mult2 | no | | | | | | | | |
| add | yes | sub | F8 | F6 | F2 | | integer | yes | no |
| divide | yes | div | F10 | F0 | F6 | mult1 | | no | yes |

| Register result status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **F0** | **F2** | **F4** | **F6** | **F8** | **F10** | **F12** | **…** | **F30** |
| mult1 | integer | | | add | divide | | | |

Departamento de Electrónica, Telecomunicações e Informática

**Components of the scoreboard internal data structure when the multiplication instruction is about to write the result to the destination register**

| Instruction status | | | | |
|---|---|---|---|---|
| **Instruction** | **Issue** | **Read operands** | **Execution terminated** | **Write result** |
| L.D   F6,34(R2) | yes | yes | yes | yes |
| L.D   F2,45(R3) | yes | yes | yes | yes |
| MUL.D F0,F2,F4 | yes | yes | yes | |
| SUB.D F8,F6,F2 | yes | yes | yes | yes |
| DIV.D F10,F0,F6 | yes | | | |
| ADD.D F6,F8,F2 | yes | yes | yes | |

| Functional unit status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Name** | **busy** | **op** | **Fi** | **Fj** | **Fk** | **Qj** | **Qk** | **Rj** | **Rk** |
| integer | no | | | | | | | | |
| mult1 | yes | mult | F0 | F2 | F4 | | | no | no |
| mult2 | no | | | | | | | | |
| add | yes | add | F6 | F8 | F2 | | | no | no |
| divide | yes | div | F10 | F0 | F6 | mult1 | | no | yes |

| Register result status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **F0** | **F2** | **F4** | **F6** | **F8** | **F10** | **F12** | **…** | **F30** |
| mult1 | | | add | | divide | | | |

**Components of the scoreboard internal data structure when the division instruction is about to write the result to the destination register**

| Instruction | Issue | Read operands | Execution terminated | Write result |
|---|---|---|---|---|
| L.D    F6,34(R2) | yes | yes | yes | yes |
| L.D    F2,45(R3) | yes | yes | yes | yes |
| MUL.D F0,F2,F4 | yes | yes | yes | yes |
| SUB.D F8,F6,F2 | yes | yes | yes | yes |
| DIV.D F10,F0,F6 | yes | yes | yes | |
| ADD.D F6,F8,F2 | yes | yes | yes | yes |

| Functional unit status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | busy | op | Fi | Fj | Fk | Qj | Qk | Rj | Rk |
| integer | no | | | | | | | | |
| mult1 | no | | | | | | | | |
| mult2 | no | | | | | | | | |
| add | no | | | | | | | | |
| divide | yes | div | F10 | F0 | F6 | mult1 | | no | no |

| Register result status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| F0 | F2 | F4 | F6 | F8 | F10 | F12 | … | F30 |
| | | | | | divide | | | |

**Required checks and bookkeeping actions for each step in instruction execution when a scoreboard is used**

| Instruction status | Wait until | Bookkeeping |
|---|---|---|
| Issue | (busy[FU] == no) && (regResult[DR] == empty) | busy[FU] = yes;  op[FU]  = op;<br>Fi[FU] = DR;  Fj[FU] = SR1;  FkFU] = SR2;<br>Qj[FU] = regResult[SR1]; Qk[FU] = regResult[SR2];<br>Rj[FU] = (Qj[FU] == empty) ? yes : no;<br>Rk[FU] = (Qk[FU] == empty) ? yes : no;<br>regResult[DR] = FU; |
| Read operands | (Rj[FU] == yes) && (Rk[FU] == yes) | Qj[FU] = empty; Qk[FU] = empty;<br>Rj[FU] = no; Rk{FU] = no; |
| Execution | FU operation terminated | |
| Write result | ∀f  ((Fj[f] != Fi[FU]) \|\| (Rj[f] == no)) &&<br>((Fk[f] != Fi[FU]) \|\| (Rk[f] == no)) | ∀f  **if** (Qj[f] == FU) Rj[f] == yes;<br>∀f  **if** (Qk[f] == FU) Rk[f] == yes;<br>regResult[Fi[FU]] = empty;<br>busy[FU] = no; |

where `FU` is the functional unit associated with the instruction, `SR1` and `SR2` its source registers and `DR` its destination register. The test at the write result stage is carried out to prevent WAR hazards.

A scoreboard uses the available ILP to minimize the number of stalls arising from the program true data dependences. In eliminating stalls, a scoreboard is limited by several factors

- *the amount of parallelism inherent to the program* – this factor determines whether independent instructions can be found; if every instruction depends on its predecessor, no dynamic scheduling scheme can reduce the number of stalls

- *the number of scoreboard entries* – this factor determines how far ahead can the pipeline look for independent instructions

- *the number and types of functional units* – this factor determines how relevant the structural hazards are, which may increase when dynamic scheduling is used

- *the presence of antidependences and output dependences* – which leads to WAR and WAW hazards and can generate further stalls.

# Tomasulo's algorithm - 1

In the scheme developed by Robert Tomasulo, RAW hazards are avoided by executing an instruction only when its operands are available, which is exactly what the simpler scoreboard provides. WAR and WAW hazards, which arise from name dependencies, are eliminated by register renaming. *Register renaming*, in fact, eliminates these hazards by altering the name of all destination registers, including those with a pending read or write from an earlier instruction, so that the out-of-order write does not affect any instructions that depend on the older value of the operand.

To better understand how register renaming works, consider the following code sequence

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

There are antidependences between the ADD.D instruction and the SUB.D and between the S.D instruction and the MUL.D instruction, and an output dependence between the ADD.D and the MUL.D instruction, potentially leading to WAR and WAW hazards, respectively. There are also true data dependences between the DIV.D instruction and the ADD.D instruction, between the ADD.D instruction and the S.D instruction and between the SUB.D instruction and the MUL.D instruction.

The three name dependences can all be eliminated by register renaming. For simplicity, assume the existence of two temporary registers, `S` and `T`. The code sequence can be rewritten without any name dependences as

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

In addition, any subsequent uses of register F8 must be replaced by register T. In this code segment, the renaming process can be done statically by the compiler. Finding any uses of register F8 that appear later in the code, requires either sophisticated compiler analysis or hardware support, since there may be intervening branches between the code segment and a later use of F8.

Departamento de Electrónica, Telecomunicações e Informática

# *Tomasulo's algorithm - 4*

In Tomasulo's scheme, register renaming is provided by *reservation stations*, which buffer the operands of instructions waiting to be executed. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register of the register bank. Pending instructions designate the reservation buffer that will provide their input. Finally when successive writes to the same register overlap in execution, only the last one in program order will actually update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation stations which provide the updated values.

Since there can be more reservation stations than real registers, this approach can even eliminate hazards arising from name dependences that could not be taken care of by the compiler.

Departamento de Electrónica, Telecomunicações e Informática
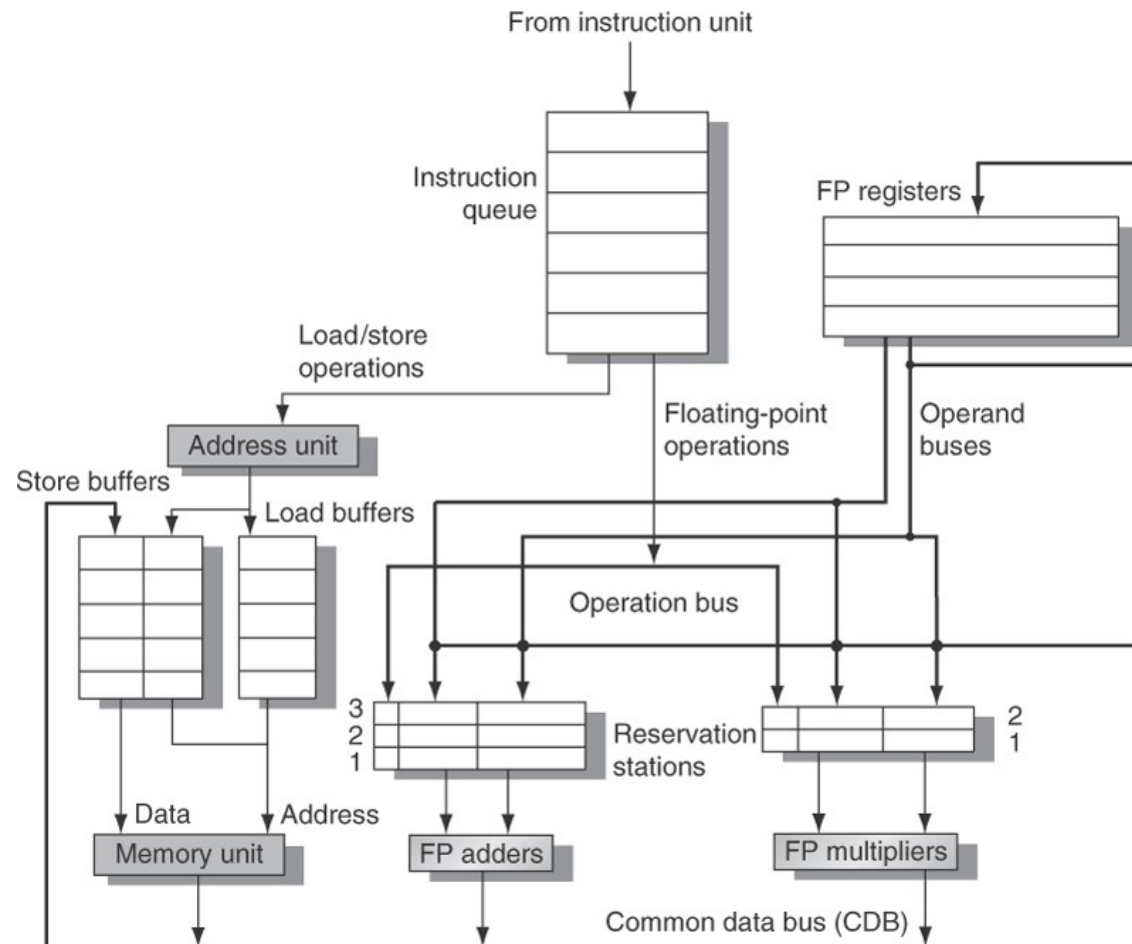
# Tomasulo's algorithm - 5

The use of reservations stations, rather than a centralized register bank, leads to two important properties

- *hazard detection and hazard control are distributed* – the information held in the reservation stations at each functional unit determines when an instruction can begin execution at that unit

- *results are passed directly to the functional units from the reservation stations where they are buffered, rather than going through the registers of the register bank* – this bypassing is done using a common result bus that allows all units waiting for an operand to be loaded simultaneously (the bus is called *common data bus* in the IBM 360/91); in pipelines with multiple execution units and issuing multiple instructions per clock cycle, more than one result bus is needed.

# *Tomasulo's algorithm - 6*

## Basic organization of a MIPS floating point unit using the Tomasulo's algorithm

Source: Computer Architecture: A Quantitative Approach

Departamento de Electrónica, Telecomunicações e Informática

Each reservation station holds both an instruction that has been issued and is waiting for execution at the associated functional unit, and either the operand values for that instruction, if they are available, or the names of the reservation stations that will provide the operand values once they are computed.

The load and store buffers hold addresses and data transferred from or to memory and behave almost exactly like the reservation stations, so they will be distinguished only when necessary. In particular, load and store buffers have three functions
- they hold the full source or destination address after its computation by the *address unit*
- they track outstanding loads that are waiting for transfer completion
- they control the results of complete loads that are waiting for the availability of the CDB, or hold the value to be stored until the *memory unit* is available.

All the results from the functional and the memory units are put in the common data bus, which connects everything but the load buffers. All reservation stations and load and store buffers have tag fields employed by the pipeline control.

# Tomasulo's algorithm - 8

Every instruction goes through three processing steps, although each one may take an arbitrary number of clock cycles. The three steps, which replace ID, EX and WB in the standard pipeline, are as follows

1. *Issue* – the next instruction is obtained from the head of the instruction queue, which is maintained in FIFO order to ensure the correct data flow. If a matching reservation station or buffer is free, the instruction is issued to the station or buffer together with the operand values, if they are currently saved in registers of the register bank. Otherwise, tracking of the functional units which will produce the operand values is performed. It is here that the registers are renamed, eliminating in consequence WAR and WAW hazards. This stage is sometimes called *dispatch*, within the context of dynamically scheduled processors.

2. *Execute* – If one or both operands are unavailable, the common data bus is monitored waiting for it or them to be computed. As soon as an operand becomes available, it is stored both at the register bank and at any station or store buffer requiring it. When all the operands are available, the operation can be executed at the corresponding functional or memory unit. By delaying instruction execution until all the operands are available, RAW hazards are avoided.

Several instructions can become ready in the same clock cycle. Although independent functional units can start execution at the same clock cycle, if more than one instruction is ready to execute at the same functional unit, a selection has to be made among them. Floating point reservation stations may be chosen arbitrarily.

Load and store instructions, however, are maintained in program order through the use of a load/store queue that contains the identification of the buffer being used. Their execution require a two-step process: in step 1, the effective address is computed; in step 2, the access to memory for data transfer is carried out.

# *Tomasulo's algorithm - 10*

To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede it in program order have completed. This restriction guarantees that an instruction that causes an exception during execution, will have been really executed. In a processor using branch prediction, as all dynamically scheduled processors do, this means that it must be known the branch prediction is correct, before allowing an instruction after the branch in program order to start execution. If the processor records the occurrence of an exception, but does not actually raise it, an instruction can start execution, but not stall until it reaches the write result step.

3. *Write result* – When the operation result is available, the CDB is accessed to save it in a register of the register bank and in the reservation stations, including store buffers, that are waiting for it.

# *Tomasulo's algorithm - 11*

The data structures that detect and eliminate hazards are attached to the reservation stations, to the register bank and to the load and store buffers, with slightly different information attached to each type. These tags are essentially names for the extended set of virtual registers used in the renaming process.

In the illustration shown, the tag field is a 4-bit quantity that denotes one of the five reservation stations (three FP adders for addition and subtraction and two FP multipliers for multiplication and division) or one of the five load buffers. This produces the equivalent of ten registers that can be designated as *result registers*.

The tag field describes which reservation station, or load buffer, contains the instruction that will produce a result needed as a source operand. Once an instruction is issued and is waiting for a source operand, it denotes the operand through the reservation station number, or load buffer number, where the instruction that will write the register has been assigned. Unused values, such as zero, indicate that the operand is already available in a register of the register bank. The fact that there are more reservation stations, or load buffers, than actual register numbers, makes the elimination of WAR and WAW hazards trivial.

# Tomasulo's algorithm - 12

In Tomasulo's scheme, results are broadcast in a bus which is monitored by the reservation stations and the store buffers for data retrieval. This kind of arrangement implements the forwarding and bypassing mechanisms used in a statically scheduled pipeline. In doing so, however, a dynamically scheduled scheme adds one latency clock cycle between source and result, since the matching of a result and its use cannot be done until the *write result* stage. Thus, in a dynamically scheduled pipeline, the effective latency between a producing and a consuming instruction is at least one clock cycle longer than the latency of the functional unit producing the result.

It is important to remember that the tags in the Tomasulo's algorithm refer to the unit or the buffer that produces a result: the register names are discarded once an instruction is issued. This is, in fact, a key difference between Tomasulo's scheme and scoreboarding. In scoreboarding, operands stay in the registers of the register bank and are only read after the producing instructions completes and the consuming instruction is ready to execute.

Each *reservation station* or *buffer* has seven fields

- *busy* – it indicates whether the reservation station or buffer is occupied or free
- *op* – it indicates the operation to be performed in the unit
- $Q_j$, $Q_k$ – identification of the reservation station, or buffer, that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in registers $V_j$ and $V_k$, or is unnecessary
- $V_j$, $V_k$ – the value of the source operands; notice that only one of the fields, Q or V, is valid for each operand; for load buffers, the $V_k$ field is used to hold the offset field
- A – used to hold the effective memory address for a load or store instruction.

Each *register* of the register bank has one field

- $Q_i$ – identification of the reservation station, or buffer, that will produce the the result to be stored in the register; a value of zero indicates that no active instruction is computing a value to be stored in the register.

Consider the code sequence

```
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
```

What is the execution state when only the first load instruction has completed and written its result?

Assume the following latencies: load / store – 1 clock cycle, addition – 2 clock cycles, multiplication – 6 clock cycles and division – 12 clock cycles.

# Tomasulo's algorithm - 15

**Reservation stations / buffers and register tags when only the first load instruction has completed and written its result**

| Instruction status (not part of the hardware) | | | |
|---|---|---|---|
| *Instruction* | *Issue* | *Execute* | *Write result* |
| L.D    F6,32(R2) | yes | yes | yes |
| L.D    F2,44(R3) | yes | yes | |
| MUL.D F0,F2,F4 | yes | | |
| SUB.D F8,F2,F6 | yes | | |
| DIV.D F10,F0,F6 | yes | | |
| ADD.D F6,F8,F2 | yes | | |

| Reservation stations / buffers | | | | | | | |
|---|---|---|---|---|---|---|---|
| *Name* | *busy* | *op* | *Vj* | *Vk* | *Qj* | *Qk* | *A* |
| load1 | no | | | | | | |
| load2 | yes | load | | | | | 44+reg[R3] |
| add1 | yes | sub | | mem[32+reg[R2]] | load2 | | |
| add2 | yes | add | | | add1 | load2 | |
| add3 | no | | | | | | |
| mult1 | yes | mult | | reg[F4] | load2 | | |
| mult2 | yes | div | | mem[32+reg[R2]] | mult1 | | |

| Register status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *Field* | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *…* | *F30* |
| Qi | mult1 | load2 | | add2 | add1 | mult2 | | | |

# Tomasulo's algorithm - 16

## Reservation stations / buffers and register tags when the multiplication instruction is ready to write its result

| Instruction status (not part of the hardware) | | | |
|---|---|---|---|
| Instruction | Issue | Execute | Write result |
| L.D   F6,32(R2) | yes | yes | yes |
| L.D   F2,44(R3) | yes | yes | yes |
| MUL.D F0,F2,F4 | yes | yes | |
| SUB.D F8,F2,F6 | yes | yes | yes |
| DIV.D F10,F0,F6 | yes | | |
| ADD.D F6,F8,F2 | yes | yes | yes |

| Reservation stations / buffers | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | busy | op | Vj | Vk | Qj | Qk | A |
| load1 | no | | | | | | |
| load2 | no | | | | | | |
| add1 | no | | | | | | |
| add2 | no | | | | | | |
| add3 | no | | | | | | |
| mult1 | yes | mult | mem[44+reg[R3]] | reg[F4] | | | |
| mult2 | yes | div | | mem[32+reg[R2]] | mult1 | | |

| Register status | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | F10 | F12 | … | F30 |
| Qi | mult1 | | | | | mult2 | | | |

# *Tomasulo's algorithm - 17*

**Required checks and bookkeeping actions for each step in instruction execution when Tomasulo's algorithm is used**

| *Instruction status* | *Wait until* | *Bookkeeping* |
|---|---|---|
| Issue<br>  FP operation | (RS[r].busy == no) | **if** (regStat[rs].Qi != 0)<br>   RS[r].Qj = regStat[rs].Qi;<br>   **else** { RS[r].Vj = reg[rs];  RS[r].Qj = 0; }<br>**if** (regStat[rt].Qi != 0)<br>   RS[r].Qk = regStat[rt].Qi;<br>   **else** { RS[r].Vk = reg[rt];  RS[r].Qk = 0; }<br>regStat[rd].Qi = r;<br>RS[r].busy = yes; |
| Issue<br>  load | (RS[r].busy == no) | **if** (regStat[rs].Qi != 0)<br>   RS[r].Qj = regStat[rs].Qi;<br>   **else** { RS[r].Vj = reg[rs];  RS[r].Qj = 0; }<br>RS[r].A= imm;<br>regStat[rt].Qi = r;<br>insert r into the load-store queue;<br>RS[r].busy = yes; |
| Issue<br>  store | (RS[r].busy == no) | **if** (regStat[rs].Qi != 0)<br>   RS[r].Qj = regStat[rs].Qi;<br>   **else** { RS[r].Vj = reg[rs];  RS[r].Qj = 0; }<br>RS[r].A= imm;<br>**if** (regStat[rt].Qi != 0)<br>   RS[r].Qk = regStat[rt].Qi;<br>   **else** { RS[r].Vk = reg[rt];  RS[r].Qk = 0; }<br>insert r into the load-store queue;<br>RS[r].busy = yes; |

Departamento de Electrónica, Telecomunicações e Informática

**Required checks and bookkeeping actions for each step in instruction execution when Tomasulo's algorithm is used (continuation)**

| | | |
|---|---|---|
| Execution<br>  FP operation | (RS[r].Qj == 0) && (RS[r].Qk == 0) | compute result – operands are in Vj and Vk |
| Execution<br>  load / store step 1 | (RS[r].Qj == 0) &&<br>r is the head of load-store queue | RS[r].A = RS[r].Vj + RS[r].A; |
| Execution<br>  load  step 2 | load step 1 is complete | read from mem[RS[r].A] |
| Write result<br>  FP operation<br>  or load | r has completed execution &&<br>CDB is available | ∀x ( **if** (regStat[x].Qi == r)<br>      { reg[x] = result;  regStat[x].Qi = 0; })<br>∀x ( **if** (RS[x].Qj == r)<br>      { RS[x].Vj = result;  RS[x].Qj = 0; })<br>∀x ( **if** (RS[x].Qk == r)<br>      { RS[x].Vk = result;  RS[x].Qk = 0; })<br>RS[r].busy = no; |
| Write result<br>  store | r has completed execution &&<br>(RS[r].Qk == 0) | mem[RS[r].A] = RS[r].Vk;<br>RS[r].busy = no; |

where `RS[r]` is the reservation station / buffer associated with the instruction and `regStat` is the register status.

# Tomasulo's algorithm - 19

A loop is considered next to illustrate the full power of eliminating WAW and WAR hazards through register renaming (the effects of delayed branches are ignored)

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

It is assumed that the register R1 contains initially the address of the last array element and that the contents of the register R2+8 is the address of the first array element.

If the branch is predicted *taken*, multiple executions of the loop can proceed in parallel through the use of multiple reservation stations and load and store buffers. This feature is gained without any code change since the loop is unrolled dynamically by the hardware. The reservation stations and load and store buffers play the role of additional registers.

Assume the same instruction latencies as in the last example.

# Tomasulo's algorithm - 20

**Reservation stations / buffers and register tags when two successive loop iterations have been issued but no instruction has yet completed**

| Instruction status (not part of the hardware) | | | | |
|---|---|---|---|---|
| **Instruction** | **Loop Iteration** | **Issue** | **Execute** | **Write result** |
| L.D    F0,0(R1) | i | yes | yes | |
| MUL.D F4,F0,F2 | i | yes | | |
| S.D    F4,0(R1) | i | yes | | |
| L.D    F0,0(R1) | i+1 | yes | yes | |
| MUL.D F4,F0,F2 | i+1 | yes | | |
| S.D    F4,0(R1) | i+1 | yes | | |

| Reservation stations / buffers | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Name** | **busy** | **op** | **Vj** | **Vk** | **Qj** | **Qk** | **A** |
| load1 | yes | load | | | | | reg[R1]+0 |
| load2 | yes | load | | | | | reg[R1]-8 |
| mult1 | yes | mult | | reg[F2] | load1 | | |
| mult2 | yes | mult | | reg[F2] | load2 | | |
| store1 | yes | store | reg[R1]+0 | | | mult1 | |
| store2 | yes | store | reg[R1]-8 | | | mult2 | |

| Register status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Field** | **F0** | **F2** | **F4** | **F6** | **F8** | **F10** | **F12** | **...** | **F30** |
| Qi | load2 | | mult2 | | | | | | |

# Tomasulo's algorithm - 21

When extended to multiple instruction issue, Tomasulo's approach can sustain more than one instruction per clock cycle. Loads and stores can be safely performed out-of-order provided they access different memory addresses. When they refer to the same address, then either

- the load is *before* the store in program order and interchanging their execution results in a WAR hazard, or

- the load is *after* the store in program order and interchanging their execution results in a RAW hazard.

Similarly, interchanging the execution of two stores to the same memory address results in a WAW hazard.

Hence, to determine if a load instruction can be executed, the processor must check whether there is any uncompleted store instruction preceding the load in program order and accessing the same address. Similarly, a store instruction must wait until there are no unexpected loads and stores that precede it and access the same address.

# Tomasulo's algorithm - 22

To prevent the data hazards to occur, the processor must have computed the effective address associated with any earlier memory operation still in progress. This can be done in a simple, but not necessarily optimal, way by performing all the effective address calculations in program order. (In fact, one only needs to keep the relative order between stores and other memory references, i. e., the load operations may be freely reordered).

If a load operation comes first, the existence of an address conflict can be readily checked by examining the A field of all active store buffers. A conflict being detected, the load instruction is not issued until the conflicting store completes.

In case of store instructions, the procedure is similar except that the processor must check for conflicts both the load and store buffers, since conflicting stores can not be reordered with respect to either loads and stores.

# *Tomasulo's algorithm – 23*

Tomasulo's scheme was not used for many years after its application to the IBM 360/91 floating point unit, but it became very popular for multiple-issue processors, starting in the 1990s, for several reasons

- although the algorithm was designed before the cache era, the presence of caches with inherently unpredictable delays, turned to be one of the major motivations for dynamic scheduling because *out-of-order* execution allows the processor to keep executing instructions while waiting for the completion of a cache miss, thus hiding all or part of the penalty

- as the processors become more aggressive on their issue capability and designers more concerned with the performance of difficult-to-schedule code, such as most non-numeric code is, the application of techniques like *register renaming*, *dynamic scheduling* and *speculation* become more relevant

- through its application, one can achieve high-performance without requiring the compiler to target the generated code to a specific pipeline structure.

# *Hardware-based speculation - 1*

Maintaining control dependences becomes an increasing burden when one tries to exploit instruction-level parallelism further. Branch prediction reduces the direct stalls attributable to branches, but just predicting branches accurately may not be sufficient to generate the desired amount of instruction-level parallelism for a processor executing multiple instructions per clock cycle. A wide-issue processor may need to execute a branch every clock cycle to keep performance at a peak value. Hence, exploiting more parallelism requires that the limitation of control dependence be overcome.

Overcoming control dependence is achieved by *speculating* on the outcome of branches and executing the code as if the guess were correct. This idea represents a subtle, but crucial, extension over branch prediction with dynamic scheduling: with *speculation*, instructions are fetched, issued and executed as if branch predictions were always right; dynamic scheduling only fetches and issues such instructions. Of course, mechanisms are needed to handle the situation where the speculation turns to be incorrect.

Departamento de Electrónica, Telecomunicações e Informática

# *Hardware-based speculation - 2*

Hardware-based speculation combines three key ideas

- *dynamic branch prediction*, to select which instructions to execute

- *speculation*, to allow the execution of instructions before the control dependences are resolved (with the understanding that the effects produced by an incorrect speculated sequence can be undone)

- *dynamic scheduling*, to deal with the issuing of different combinations of basic blocks.

In contrast, dynamic scheduling [without speculation] overlaps basic blocks only partially, because it requires a branch to be resolved before actually executing any instructions in the successor block.

Hence, hardware-based speculation follows the predicted flow of data values to determine when to execute the instructions. This method of execution is essentially a *data flow execution*: operations are carried out as soon as their source operands become available.

Departamento de Electrónica, Telecomunicações e Informática

# Hardware-based speculation - 3

In order to extend Tomasulo's algorithm to support speculation, the bypassing of results among instructions, which is needed to execute instructions speculatively, has to be separated from the actual completion of instructions. If such a mechanism is accomplished, an instruction can be allowed to execute and bypass its result to others, without allowing the instruction to perform any updates that can not be reversed until it is established that the instruction is no longer speculative.

Using a bypassed value is like performing a speculative register read, since it may not be known at that moment whether the instruction providing the value for the [source] register is providing a *real* result. Only when the instruction is no longer speculative, the specific register of the register bank, or the referred memory location, may be updated – this additional step in instruction execution is called *instruction commit*.

Therefore, instructions can execute *out-of-order*, but are forced to commit *in-order* so that any irreversible action, such as state updating or exception taking, is prevented. The separation of instruction completion from instruction commit is essential because instructions may finish execution considerably before they are ready to commit.

Departamento de Electrónica, Telecomunicações e Informática

# *Hardware-based speculation - 4*

The introduction of the commit phase to instruction execution requires an additional set of hardware buffers which hold the results of completed instructions that have not committed yet. This buffer bank, called the *reorder buffer* (ROB), is also used to pass results among instructions.

The *reorder buffer* provides additional registers in just the same way as the reservation stations and the load and the store buffers extend the register set in Tomasulo's algorithm. The key difference is that in Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will get the value from the register bank, while with speculation, the register bank is not updated until the instruction commits (that is, when it is definitively established that the instruction should execute), which means that ROB supplies operands to other instructions in the interval between instruction completion and instruction commit. ROB is similar to the store buffers in Tomasulo's algorithm, so the functionality of store buffers is integrated into ROB for simplicity.
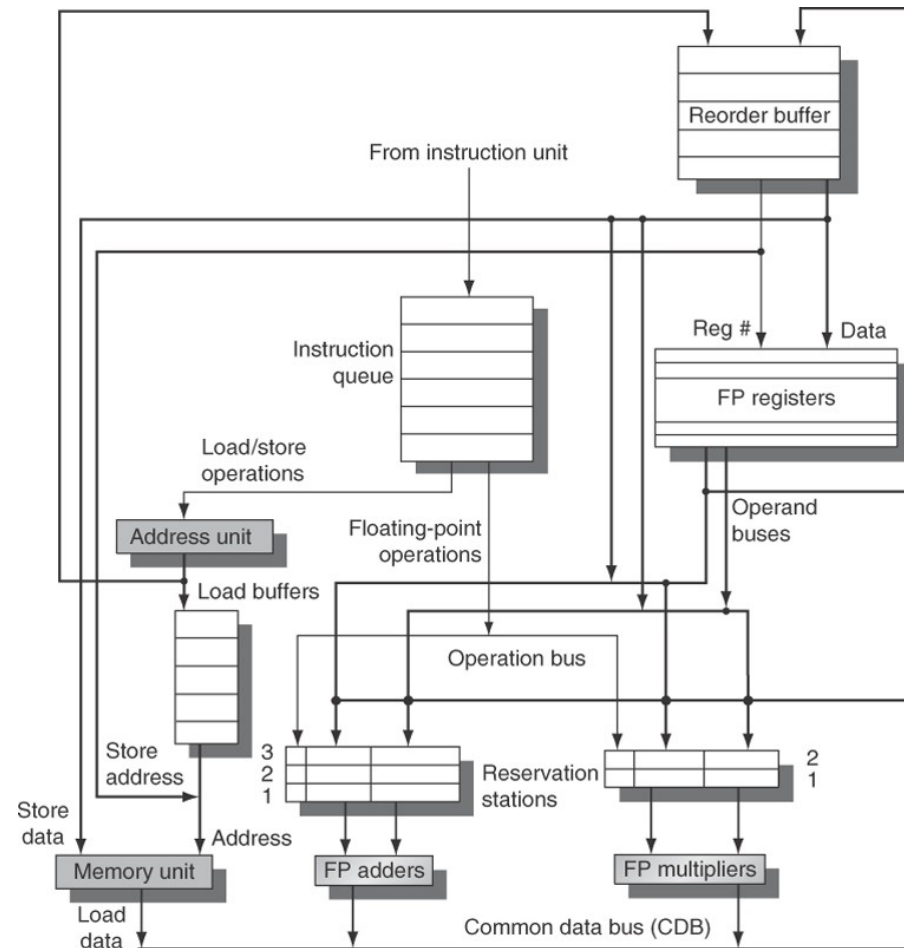
# *Hardware-based speculation - 5*

Each ROB entry contains five fields

- *busy* – it indicates whether the entry is occupied or free

- *instruction type* – it signals whether the instruction is a branch, with no destination result, a store, with a memory destination, or a load or ALU operation, with a register bank destination

- *destination location* -  it supplies the memory address, for stores, or the register number, for loads and ALU operations, where the instruction result should be written to

- *value* – it holds the value of the instruction result between instruction completion and instruction commit

- *ready* -  it indicates whether the instruction has completed execution.

Departamento de Electrónica, Telecomunicações e Informática

# *Hardware-based speculation - 6*

**Basic organization of a MIPS floating point unit using the Tomasulo's algorithm extended to handle speculation**

Source: Computer Architecture: A Quantitative Approach

Departamento de Electrónica, Telecomunicações e Informática

Stores still execute in two steps, but the second step is included in instruction commit.

Although the renaming function of the reservation stations is replaced by the ROB, a place is still needed to buffer operations and operands between the time they issue and the time they end execution. This function is carried out by the reservation stations and the load buffers. On the other hand, since every instruction has an entry in ROB until it commits, the operation result is tagged using the ROB entry rather than the reservation station ot the load buffer number. This tagging requires that the ROB entry assigned to the instruction is tracked at the reservation station or load buffer. Later on, an alternative implementation, which uses extra registers for renaming and a queue to replace ROB, is described and shown how it can decide when the instructions can commit.

The four steps involved in instruction execution are as follows
1. *Issue* – the next instruction is obtained from the head of the instruction queue, which is maintained in FIFO order to ensure the correct data flow. If both a matching reservation station or buffer *and* a ROB slot are free, the instruction is issued to the station or buffer together with the operand values, if they are currently saved in registers of the register bank or in ROB entries. Otherwise, tracking of the ROB entries which will eventually contain the operand values is performed. The number of the ROB entry allocated to the instruction is also sent to the reservation station or buffer so that the number can be used to tag the result when it is placed on the CDB.
2. *Execute* – the CDB is monitored while waiting for the operands to be computed so that RAW hazards are prevented. When the operands are available, the operation is executed. Instruction execution may take multiple clock cycles. Loads still require two steps. Stores, on the other hand, only compute the effective address.

3. *Write result* – when the instruction completes, the result is placed on the CDB, together with the ROB entry tag, to be written into the ROB and into all the reservation stations and buffers waiting for the value. The reservation station or buffer is marked free and the *ready* field of the ROB entry is set. Special actions are required for store instructions: if the value to be stored is available, it is written into the *value* field of the ROB entry; if the value is unavailable, the CDB is monitored for the broadcast of the value, it is saved in the buffer and then the *value* field of the ROB entry is updated.

4. *Commit* – the associated set of actions depend upon whether the committing instruction is a branch with a wrong prediction, a store or any other case (*normal commit*). The *normal commit* occurs when the instruction attains the head of ROB and the *ready* field is set, the processor then updates the register of the register bank, if any, and finishes. Committing a store is similar, except that a memory location is updated instead of a register. Finally, for a branch with a wrong prediction, the speculation is found to be incorrect, ROB is flushed and fetching is restarted at the proper address defined by the branch behavior.

Departamento de Electrónica, Telecomunicações e Informática

Consider the code sequence (identical to the one used in the example for the Tomasulo's algorithm and assume the same instruction latencies)

```
L.D     F6,32(R2)
L.D     F2,44(R3)
MUL.D   F0,F2,F4
S.D     F8,F2,F6
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

What is the execution state when the multiplication instruction is ready to commit, that is, the MULT instruction is at the head of ROB?

Departamento de Electrónica, Telecomunicações e Informática

# *Hardware-based speculation - 11*

**Reorder buffer, reservation stations / buffers and register tags when the multiplication instruction is ready to commit**

| | | | Reorder buffer | | | |
|---|---|---|---|---|---|---|
| entry | state | busy | instruction type | destination location | value | ready |
| 1 | commit | no | L.D    F6,32(R2) | F6 | mem[reg[R2]+32] | yes |
| 2 | commit | no | L.D    F2,44(R3) | F2 | mem[reg[R3]+44] | yes |
| 3 | write result | yes | MUL.D F0,F2,F4 | F0 | #2 x reg[F4] | yes |
| 4 | write result | yes | SUB.D F8,F2,F6 | F8 | #2 - #1 | yes |
| 5 | execute | yes | DIV.D F10,F0,F6 | F10 | | no |
| 6 | write result | yes | ADD.D F6,F8,F2 | F6 | #4 + #2 | yes |

| | | | | Reservation stations / buffers | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | busy | op | Vj | Vk | Qj | Qk | dest | A |
| load1 | no | | | | | | | |
| load2 | no | | | | | | | |
| add1 | no | | | | | | | |
| add2 | no | | | | | | | |
| add3 | no | | | | | | | |
| mult1 | no | mult | mem[reg[R3]+44] | reg[F4] | | | #3 | |
| mult2 | yes | div | | mem[reg[R2]+32] | #3 | | #5 | |

| | | | | Register status | | | | |
|---|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | F10 | F12 | … | F30 |
| busy | yes | no | no | yes | yes | yes | no | … | no |
| ROB entry | 3 | | | 6 | 4 | 5 | | … | |

Departamento de Electrónica, Telecomunicações e Informática

Consider the following code sequence, identical to the one used in the example for the Tomasulo's algorithm (the effects of delayed branches are ignored)

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

It is assumed that the register `R1` contains initially the address of the last array element and that the contents of the register `R2+8` is the address of the first array element.

If the branch is predicted *taken*, multiple executions of the loop can proceed in parallel. Assume that all the instructions in the loop have been issued twice, the load and the multiplication instructions of the first iteration have committed and all the others have completed execution.

Also assume the same instruction latencies as in the last example.

Departamento de Electrónica, Telecomunicações e Informática

# *Hardware-based speculation - 13*

**Reorder buffer and register tags when two successive loop iterations have been issued, the load and multiplication instructions of the first iteration have committed and all the others have completed execution**

| Reorder buffer | | | | | | |
|---|---|---|---|---|---|---|
| entry | state | busy | instruction type | destination location | value | ready |
| 1 | commit | no | L.D    F0,0(R1) | F0 | mem[reg[R1]+0] | yes |
| 2 | commit | no | MUL.D  F4,F0,F2 | F4 | #1 x reg[F2] | yes |
| 3 | write result | yes | S.D    F4,0(R1) | reg[R1]+0 | #2 | yes |
| 4 | write result | yes | DADDIU R1,R1,-8 | R1 | reg[R1] − 8 | yes |
| 5 | write result | yes | BNE    R1,R1,Loop | | | yes |
| 6 | write result | yes | L.D    F0,0(R1) | F0 | mem[#4] | yes |
| 7 | write result | yes | MUL.D  F4,F0,F2 | F4 | #6 x reg[F2] | yes |
| 8 | write result | yes | S.D    F4,0(R1) | #4+0 | #7 | yes |
| 9 | write result | yes | DADDIU R1,R1,-8 | R1 | #4 − 8 | yes |
| 10 | write result | yes | BNE    R1,R1,Loop | | | yes |

| Register status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Field | F0 | F2 | F4 | F6 | F8 | F10 | F12 | … | F30 |
| busy | yes | no | yes | no | no | no | no | … | no |
| ROB entry | 6 | | 7 | | | | | … | |

Departamento de Electrónica, Telecomunicações e Informática

# *Hardware-based speculation - 14*

Since neither register values, nor memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. The instructions prior to the branch will commit in succession when each reaches the head of ROB. When the turn of the branch comes, ROB is flushed and the processor starts to fetch instructions from the correct path.

In speculative processors, performance is very sensitive to branch prediction. Thus, all aspects of handling branches such as prediction accuracy, latency of misprediction detection and misprediction recovery time, become very important.

Exceptions are handled by not recognizing the exception until the instruction that produced it is ready to commit. If a speculated instruction raises an exception, the exception is recorded in the associated ROB entry. When a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with ROB. If the instruction reaches the head of ROB, it is no longer speculative and the exception is now taken.

## Required checks and bookkeeping actions for each step in instruction execution when hardware-based speculation is used

| Instruction status | Wait until | Bookkeeping |
|---|---|---|
| Issue<br>  FP operation | (RS[r].busy == no) && (ROB[b].busy == no) | **if** (regStat[rs].busy == yes)<br>  { x = regStat[rs].reorder;<br>   **if** (ROB[x].ready == yes)<br>    { RS[r].Vj = ROB[x].value;  RS[r].Qj = 0; }<br>    **else** RS[r].Qj = x;<br>  }<br>  **else** { RS[r].Vj = reg[rs];  RS[r].Qj = 0; }<br>**if** (regStat[rt].busy == yes)<br>  { x = regStat[rt].reorder;<br>   **if** (ROB[x].ready == yes)<br>    { RS[r].Vk = ROB[x].value;  RS[r].Qk = 0; }<br>    **else** RS[r].Qk = x;<br>  }<br>  **else** { RS[r].Vk = reg[rt];  RS[r].Qk = 0; }<br>regStat[rd].reorder = b; regStat[rd].busy = yes;<br>RS[r].dest = b; RS[r].busy = yes;<br>ROB[b].inst = opcode; ROB[b].dest = rd; ROB[b].ready = no; ROB[b].busy = yes; |
| Issue<br>  load | (RS[r].busy == no) && (ROB[b].busy == no) | **if** (regStat[rs].busy == yes)<br>  { x = regStat[rs].reorder;<br>   **if** (ROB[x].ready == yes)<br>    { RS[r].Vj = ROB[x].value;  RS[r].Qj = 0; }<br>    **else** RS[r].Qj = x;<br>  }<br>  **else** { RS[r].Vj = reg[rs];  RS[r].Qj = 0; }<br>regStat[rt].reorder = b; regStat[rt].busy = yes;<br>RS[r].A= imm; RS[r].dest = b; RS[r].busy = yes;<br>ROB[b].inst = opcode; ROB[b].dest = rt; ROB[b].ready = no; ROB[b].busy = yes; |

## Required checks and bookkeeping actions for each step in instruction execution when hardware-based speculation is used (continuation)

| | | |
|---|---|---|
| Issue<br>    store | (RS[r].busy == no)  &&  (ROB[b].busy == no) | **if** (regStat[rs].busy == yes)<br>  { x = regStat[rs].reorder;<br>    **if** (ROB[x].ready == yes)<br>      { RS[r].Vj = ROB[x].value;  RS[r].Qj = 0; }<br>      **else** RS[r].Qj = x;<br>  }<br>  **else** { RS[r].Vj = reg[rs];  RS[r].Qj = 0; }<br>**if** (regStat[rt].busy == yes)<br>  { x = regStat[rt].reorder;<br>    **if** (ROB[x].ready == yes)<br>      { RS[r].Vk = ROB[x].value;  RS[r].Qk = 0; }<br>      **else** RS[r].Qk = x;<br>  }<br>  **else** { RS[r].Vk = reg[rt];  RS[r].Qk = 0; }<br>RS[r].A= imm; RS[r].busy = yes;<br>ROB[b].inst = opcode; ROB[b].ready = no; ROB[b].busy = yes; |
| Execution<br>    FP operation | (RS[r].Qj == 0)  &&  (RS[r].Qk == 0) | compute result – operands are in Vj and Vk |
| Execution<br>    load  step 1 | (RS[r].Qj == 0)  &&<br>there are no previous stores in ROB queue | RS[r].A= RS[r].Vj + RS[r].A; |
| Execution<br>    load  step 2 | load step 1 is complete &&<br>all previous stores in ROB queue have different<br>effective addresses | read from mem[RS[r].A] |
| Execution<br>    store | (RS[r].Qj == 0)  &&<br>b is the head of ROB queue | ROB[b].dest = RS[r].Vj + RS[r].A; |

# *Hardware-based speculation - 17*

**Required checks and bookkeeping actions for each step in instruction
execution when hardware-based speculation is used (continuation)**

| | | |
|---|---|---|
| Write result<br>  FP operation<br>  or load | r has completed execution  &&<br>CDB is available | b = RS[r].dest;<br>∀x  ( **if** (RS[x].Qj == b)<br>          { RS[x].Vj = result;  RS[x].Qj = 0; })<br>∀x  ( **if** (RS[x].Qk == b)<br>          { RS[x].Vk = result;  RS[x].Qk = 0; })<br>ROB[b].value = result; ROB[b].ready = yes;<br>RS[r].busy = no; |
| Write result<br>  store | r has completed execution  &&<br>(RS[r].Qk == 0) | ROB[b].value = RS[r].Vk;<br>RS[r].busy = no; |
| Commit | b is the head of ROB queue &&<br>(ROB[b].ready == yes) | **if** (ROB[b].inst == branch)<br>   { **if** (mispredicted branch)<br>      { flush ROB; reset reservation stations / load buffers;<br>        reset register status; fetch at branch destination;<br>      }<br>   }<br>   **else if** (ROB[b].inst == store)<br>          mem[ROB[b].dest] = [ROB[b].value;<br>          **else** { reg[ROB[b].dest] = ROB[b].value;<br>                 **if** (regStat[ROB[b].dest].reorder == b) regStat[ROB[b].busy = no;<br>          }<br>ROB[b]..busy = no; |

where `ROB[b]` is the ROB entry and `RS[r]` the reservation station / buffer associated
with the instruction and `regStat` is the register status.

# *Exploiting ILP using multiple issue - 1*

The techniques just described can be used to eliminate stalls resulting from data and control dependences and approach an ideal CPI of one when running a given program

$$CPI_{prog} = CPI_{ideal} + \text{structural stalls} .$$

To improve performance further, one needs to decrease the ideal CPI to a value less than one, but this can not be done if only one instruction is issued per clock cycle.

*Multiple issue processors* are of three basic types

- *statically scheduled superscalar processors* – a variable number of instructions are issued together in multiple pipelines, each statically scheduled
- VLIW (*very long instruction word*) *processors* – a fixed number of instructions formatted either as one large instruction, or a fixed instruction packet, are issued together
- *dynamically scheduled superscalar processors* – a variable number of instructions are issued together in multiple pipelines, each dynamically scheduled.

Departamento de Electrónica, Telecomunicações e Informática

# Exploiting ILP using multiple issue - 2

## Primary approaches in use for multiple issue processors

Source: Computer Architecture: A Quantitative Approach

| Common name | Issue structure | Hazard detection | Scheduling | Distinguishing characteristic | Examples |
|---|---|---|---|---|---|
| superscalar (static) | dynamic | hardware | static | in-order execution | mostly in embedded environments: MIPS and ARM (including ARM Cortex-A8) |
| superscalar (dynamic) | dynamic | hardware | dynamic | some out-of-order execution, but no speculation | none at the present time |
| superscalar (speculative) | dynamic | hardware | dynamic with speculation | out-of-order execution with speculation | Intel Core i3, i5, i7, AMD Phenom and IBM POWER 7 |
| VLIW / LIW | static | primarily sofware | static | all hazards determined and indicated by the compiler (often implicitly) | mostly in signal processing: TI C6x |
| EPIC | primarily static | primarily sofware | mostly static | all hazards determined and indicated by the compiler (explicitly) | Itanium |

Departamento de Electrónica, Telecomunicações e Informática

# *Statically scheduled superscalar processor*

A *statically scheduled superscalar processor* typically issues in-order a variable number of instructions per clock cycle up to an upper limit that corresponds to the number of parallel pipelines which are implemented.

. The reason why the number of instructions issued per clock cycle is variable has to do mainly with two factors

- the multiple pipelines are not exactly alike, which may lead to structural hazards if any combination of instructions is considered
- although forwarding is used in an extensive way across the different pipelines, it is not possible, even with compiler's help, to prevent stalls due to data hazards among successive instructions.

Thus, there are in fact diminishing advantages for a statically scheduled superscalar architecture as the instruction issue width increases. This is why the issue width is normally just two.

The A8 is a dual-issue, statically scheduled superscalar processor with dynamic issue detection, which enables it to issue one or two instructions per clock cycle.
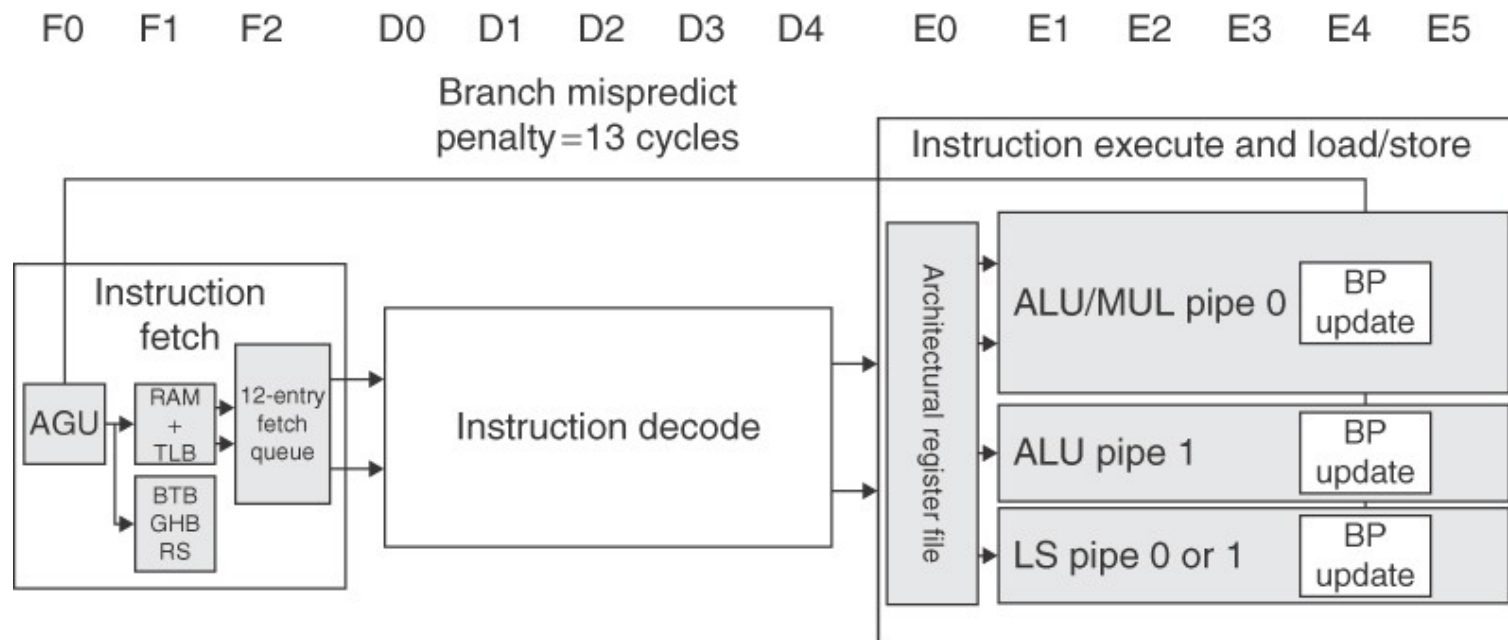


**A8 basic 13-stage pipeline structure**
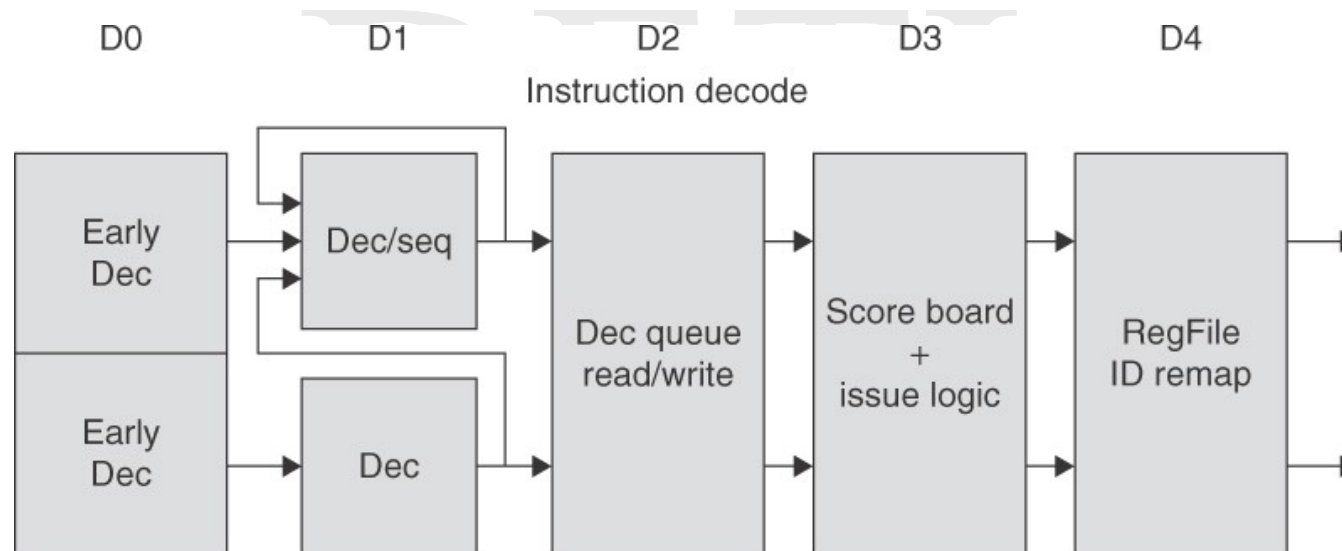Source: Computer Architecture: A Quantitative Approach

Departamento de Electrónica, Telecomunicações e Informática

# ARM Cortex-A8 - 2

The A8 uses a dynamic branch predictor with a 512-entry 2-way set associative branch target buffer and a 4K-entry global history buffer, which is indexed by the branch history and the current PC. In the event that the branch target buffer misses, a prediction is obtained from the global history buffer, which is then used to compute the branch address.

In addition, an 8-entry return stack is kept to track return addresses. An incorrect prediction results in a 13-cycle penalty as the pipeline is flushed.

# ARM Cortex-A8 - 3

Up to two instructions per clock cycle can be issued using an in-order issue mechanism. A simple scoreboard structure is used to track when an instruction can issue. A pair of dependent instructions can be processed through the issue logic, but they will be serialized at the scoreboard, unless the forwarding paths can resolve the dependence.
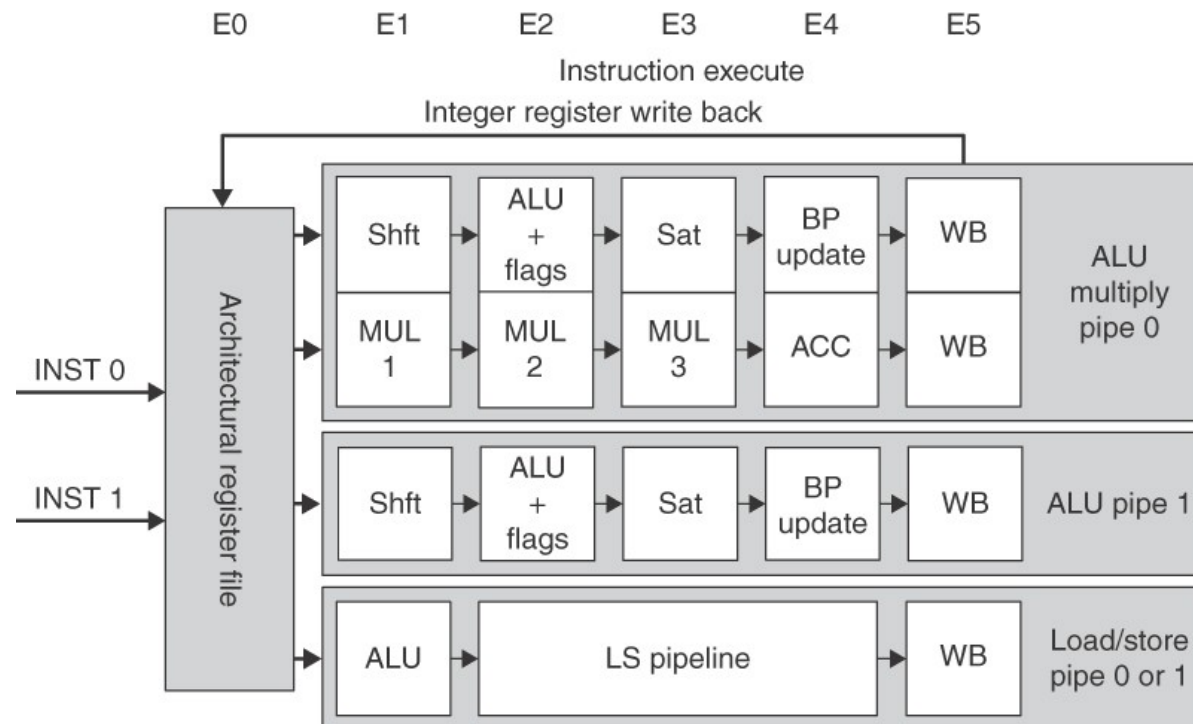


**A8 5-stage instruction decode**
Source: Computer Architecture: A Quantitative Approach

Departamento de Electrónica, Telecomunicações e Informática

# ARM Cortex-A8 - 4

Either instruction 1 or 2 can go to the load / store pipeline. Fully bypassing is supported between the pipelines in order to minimize stalling at the scoreboard.



**A8 execution pipeline**
Source: Computer Architecture: A Quantitative Approach

The A8 has an ideal CPI of 0.5 due to its dual-issue structure. Pipeline stalls can arise from three sources

- *structural hazards* – they occur when two adjacent instructions selected for issue simultaneously require the same functional pipeline; since A8 is statically scheduled, it is the compiler duty to try to avoid such conflicts; when they can not be avoided, the A8 can issue at most one instruction in that clock cycle
- *data hazards* – they are detected early in the pipeline, at the scoreboard, and may stall either both instructions (if the first instruction can not issue, the second is always stalled) or just the second of a pair; again it is the compiler duty to try to prevent such stalls whenever possible
- *control hazards* -  they only arise when branches are mispredicted.

In addition to stalls due to hazards, L1 and L2 cache misses at the load / store pipeline will also produce stalls.

**Estimated composition of CPI of ARM A8 when running the Minnespec benchmark suite**

Source: Computer Architecture: A Quantitative Approach
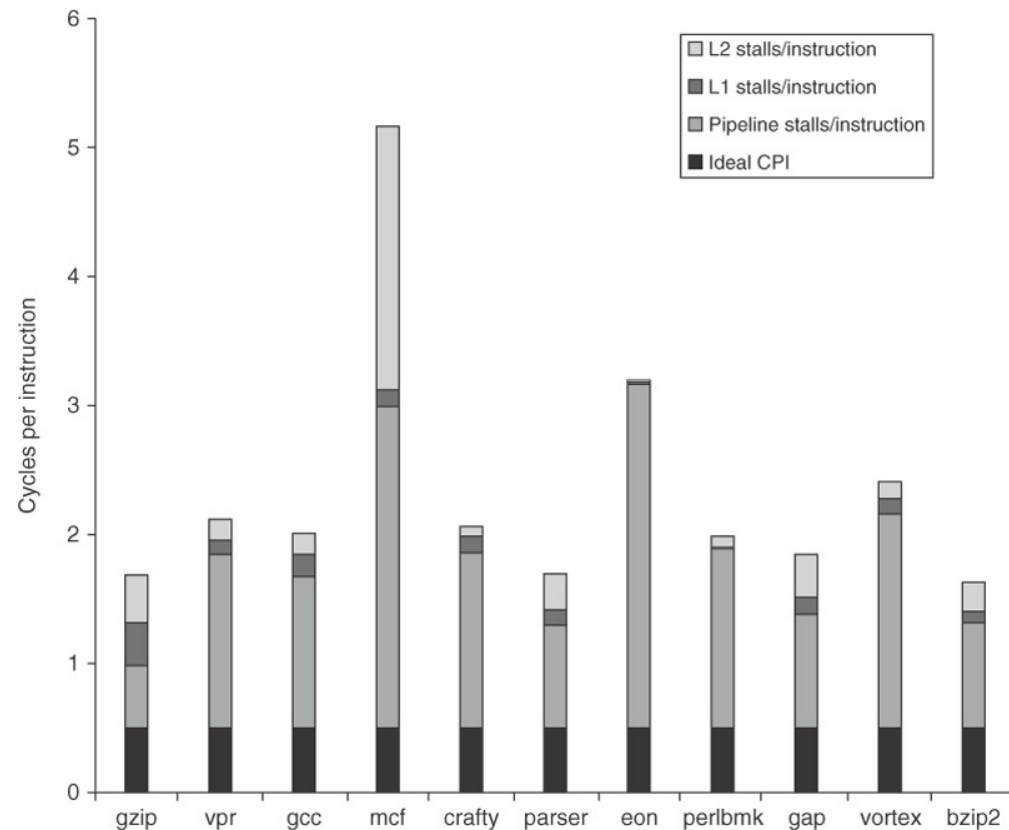
# *Dynamically scheduled superscalar processor - 1*

For simplicity, an issue rate of two instructions per clock cycle will be assumed. The key concepts are not different from those found in modern processors that issue three or more instructions per clock cycle.

Tomasulo's algorithm will be extended to support a multiple issue speculative superscalar pipeline with separate integer, load / store and floating point functional units, each of which can initiate an operation every clock cycle. To gain full advantage of dynamic scheduling, the pipeline is allowed to issue any combination of two instructions.

Because the interaction of integer and floating point instructions is determinant, Tomasulo's scheme is also extended to deal with both integer and floating point functional units and registers.

# *Dynamically scheduled superscalar processor - 2*

**Basic organization of a multiple issue processor with speculation**
Source: Computer Architecture: A Quantitative Approach
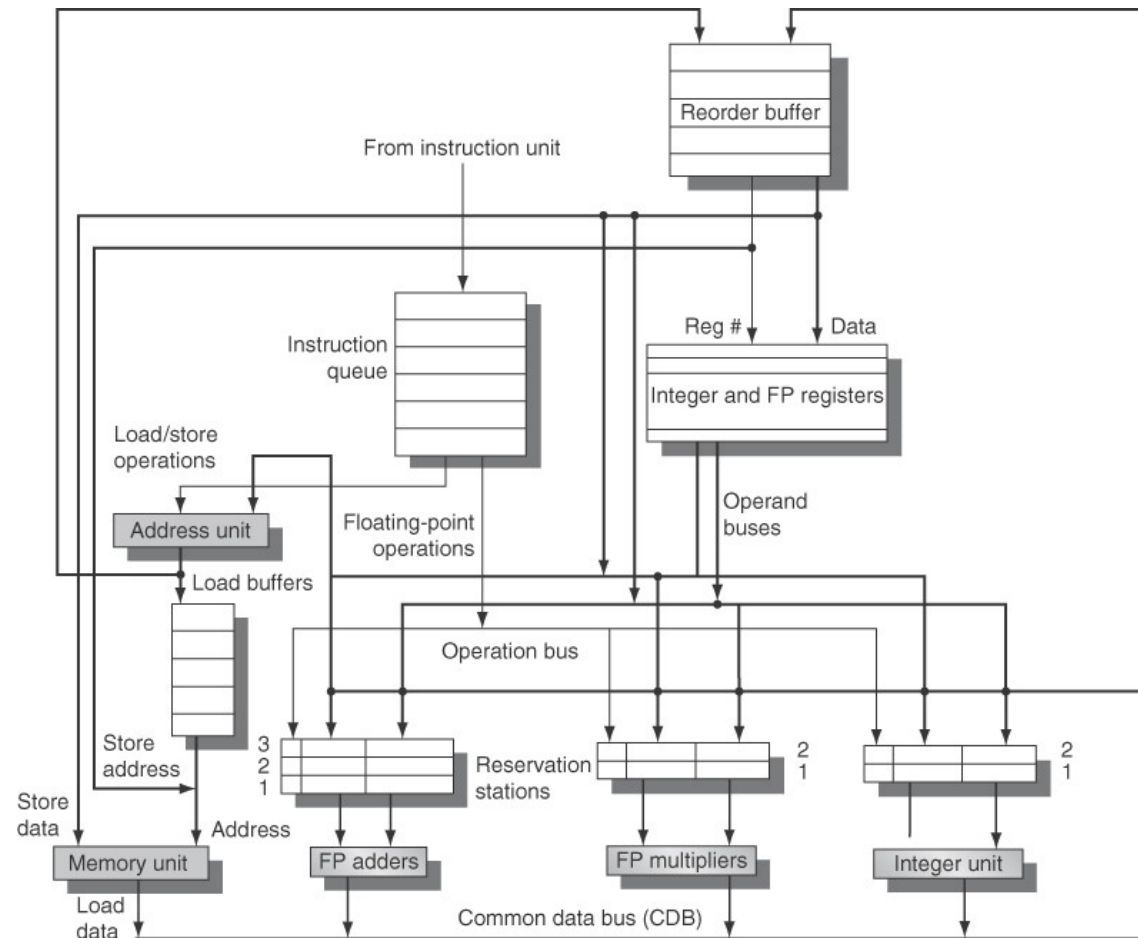
# *Dynamically scheduled superscalar processor - 3*

Issuing multiple instructions in the same clock cycle in a dynamically scheduled processor, with or without speculation, is a very complex task as the instructions may depend upon one another. Due to this fact, the control tables must be updated in parallel; otherwise, the values will be incorrect or the dependence may be  lost.

Two different approaches have been used. The first is to run the step in a fraction of the whole clock cycle for each instruction. For instance, when the issue width is two, this means that it is run in one half of the clock cycle. Unfortunately, it can not be extended in a straightforward manner to handle four instructions! The second is to build the necessary logic to run simultaneously two or more instructions, including any possible dependences among them.

Modern superscalar processors that issue four or more instructions per clock cycle may include both: they pipeline and wide the issue logic.

This issue step is one of the most fundamental bottlenecks in developing dynamically scheduled superscalar processors.

# Dynamically scheduled superscalar processor - 4

**Required checks and bookkeeping actions for the issue of a bundle of two instructions where inst1 is a FP load and inst2 is a FP operation**

| Instruction status | Wait until | Bookkeeping |
|---|---|---|
| Issue<br>load<br>(first instruction<br>of the bundle) | (RS[r1].busy == no) && (ROB[b1].busy == no) | **if** (regStat[rs1].busy == yes)<br>   { x = regStat[rs1].reorder;<br>    **if** (ROB[x].ready == yes)<br>      { RS[r1].Vj = ROB[x].value; RS[r1].Qj = 0; }<br>     **else** RS[r1].Qj = x;<br>  }<br>  **else** { RS[r1].Vj = reg[rs1]; RS[r1].Qj = 0; }<br>regStat[rt1].reorder = b1; regStat[rt1].busy = yes;<br>RS[r1].A= imm; RS[r1].dest = b1; RS[r1].busy = yes;<br>ROB[b1].inst = load; ROB[b1].dest = rt1; ROB[b1].ready = no; ROB[b1].busy = yes; |
| Issue<br>FP operation<br>(second instruction<br>of the bundle)<br>the first operand<br>comes from load | (RS[r2].busy == no) && (ROB[b2].busy == no) | RS[r2].Qj = b1;<br>**if** (regStat[rt2].busy == yes)<br>   { x = regStat[rt2].reorder;<br>    **if** (ROB[x].ready == yes)<br>      { RS[r2].Vk = ROB[x].value; RS[r2].Qk = 0; }<br>     **else** RS[r2].Qk = x;<br>  }<br>  **else** { RS[r2].Vk = reg[rs2]; RS[r2].Qk = 0; }<br>regStat[rd2].reorder = b2; regStat[rd2].busy = yes;<br>RS[r2].dest = b2; RS[r2].busy = yes;<br>ROB[b2].inst = FP op; ROB[b2].dest = rd2; ROB[b2].ready = no; ROB[b2].busy = yes; |

where `ROB[b1]` and `ROB[b2]` are ROB entries and `RS[r1]` and `RS[r2]` the reservation stations / buffers associated with the two instructions and `regStat` is the register status.

# Dynamically scheduled superscalar processor - 5

In a real situation, every possible combination of dependent instructions allowed to issue in the same clock cycle must be considered. Since the number of possibilities increases as the square of the number of instructions issued in a clock cycle, this turns out to be a pressing concern for a large issue width (beyond four, for instance).

# *Dynamically scheduled superscalar processor - 6*

The basic strategy to update the issue logic in a dynamically scheduled superscalar processor with up to n issues per clock cycle is as follows

1. Assign a reorder buffer and a reservation station / buffer for every instruction that might be issued in the next bundle. This assignment can be done before the instruction types are known by preallocating the reorder buffer entries sequentially to the instructions in the bundle and by ensuring that there are enough reservation stations / buffers available, independent of the bundle content. Should not sufficient reservation stations / buffers be available, the bundle has to be broken and only a subset of these instructions, in the original program order, is issued. The remaining instructions will join the next bundle.

2. Find all dependences among all the instructions in the bundle.

3. If a dependence of an instruction in the bundle is found to a preceding one in the bundle, the assigned reorder buffer entry number should be used to update the reservation table for the dependent instruction; otherwise, the existing reorder buffer and reservation table information should be used to update the reservation table for the issuing instruction.

At the back end of the pipeline, it is also required to complete and commit multiple instructions per clock cycle. The steps here are, however, somewhat simpler than the issue problem, since the instructions which can actually commit in the same clock cycle must have already dealt with and resolved the dependences.

# Dynamically scheduled superscalar processor - 8

Consider the code sequence bellow and analyze its execution on a 2-issue processor without and with speculation (the effects of delayed branches are ignored)

```
Loop:    L.D      R2,0(R1)
         DADDIU   R2,R2,1
         S.D      R2,0(R1)
         DADDIU   R1,R1,8
         BNE      R1,R3, Loop
```

It is assumed that the register `R1` contains initially the address of the first array element and that the contents of the register `R3-8` is the address of the last array element.

It is also assumed that there are separate functional units for effective address calculation, for ALU operations and for branch condition evaluation. The first three iterations should be analysed.

Departamento de Electrónica, Telecomunicações e Informática

## Code execution on a 2-issue processor without speculation

| iteration number | instruction | issues at clock cycle number | executes at clock cycle number | memory access at clock cycle number | write CDB at clock cycle number | comment |
|---|---|---|---|---|---|---|
| 1 | L.D    R2,0(R1) | 1 | 2 | 3 | 4 | first issue |
| 1 | DADDIU R2,R2,1 | 1 | 5 | | 6 | wait for L.D |
| 1 | S.D    R2,0(R1) | 2 | 3 | 7 | | wait for DADDIU |
| 1 | DADDIU R1,R1,8 | 2 | 3 | | 4 | execute |
| 1 | BNE    R1,R3,Loop | 3 | 7 | | | wait for DADDIU |
| 2 | L.D    R2,0(R1) | 4 | 8 | 9 | 10 | wait for BNE |
| 2 | DADDIU R2,R2,1 | 4 | 11 | | 12 | wait for L.D |
| 2 | S.D    R2,0(R1) | 5 | 9 | 13 | | wait for DADDIU |
| 2 | DADDIU R1,R1,8 | 5 | 8 | | 9 | execute |
| 2 | BNE    R1,R3,Loop | 6 | 13 | | | wait for DADDIU |
| 3 | L.D    R2,0(R1) | 7 | 14 | 15 | 16 | wait for BNE |
| 3 | DADDIU R2,R2,1 | 7 | 17 | | 18 | wait for L.D |
| 3 | S.D    R2,0(R1) | 8 | 15 | 19 | | wait for DADDIU |
| 3 | DADDIU R1,R1,8 | 8 | 14 | | 15 | execute |
| 3 | BNE    R1,R3,Loop | 9 | 19 | | | wait for DADDIU |

Departamento de Electrónica, Telecomunicações e Informática

# *Dynamically scheduled superscalar processor - 10*

## Code execution on a 2-issue processor with speculation

| iteration number | instruction | issues at clock cycle number | executes at clock cycle number | memory access at clock cycle number | write CDB at clock cycle number | commit at clock cycle number | comment |
|---|---|---|---|---|---|---|---|
| 1 | L.D     R2,0(R1) | 1 | 2 | 3 | 4 | 5 | first issue |
| 1 | DADDIU R2,R2,1 | 1 | 5 | | 6 | 7 | wait for L.D |
| 1 | S.D     R2,0(R1) | 2 | 3 | | | 7 | wait for DADDIU |
| 1 | DADDIU R1,R1,8 | 2 | 3 | | 4 | 8 | commit in order |
| 1 | BNE     R1,R3,Loop | 3 | 7 | | | 8 | wait for DADDIU |
| 2 | L.D     R2,0(R1) | 4 | 5 | 6 | 7 | 9 | no execute delay |
| 2 | DADDIU R2,R2,1 | 4 | 8 | | 9 | 10 | wait for L.D |
| 2 | S.D     R2,0(R1) | 5 | 6 | | | 10 | wait for DADDIU |
| 2 | DADDIU R1,R1,8 | 5 | 6 | | 7 | 11 | commit in order |
| 2 | BNE     R1,R3,Loop | 6 | 10 | | | 11 | wait for DADDIU |
| 3 | L.D     R2,0(R1) | 7 | 8 | 9 | 10 | 12 | earliest possible |
| 3 | DADDIU R2,R2,1 | 7 | 11 | | 12 | 13 | wait for L.D |
| 3 | S.D     R2,0(R1) | 8 | 9 | | | 13 | wait for DADDIU |
| 3 | DADDIU R1,R1,8 | 8 | 9 | | 10 | 14 | executes earlier |
| 3 | BNE     R1,R3,Loop | 9 | 13 | | | 14 | wait for DADDIU |

Departamento de Electrónica, Telecomunicações e Informática

The example shows how speculation can be advantageous when there are data dependent branches, which otherwise would limit performance. This advantage depends, however, on accurate branch prediction. It is important to note that incorrect speculation harms performance and dramatically lowers energy efficiency!

Why is it so?

# *Intel Core i7 - 1*

The Intel Core i7 uses an aggressive out-of-order speculative microarchitecture with reasonably deep pipelines having as goal the attaining of high instruction throughput by combining multiple issue and high clock rates.

**Intel Core i7 pipeline structure with memory system interface**
Source: Computer Architecture: A Quantitative Approach

Departamento de Electrónica, Telecomunicações e Informática
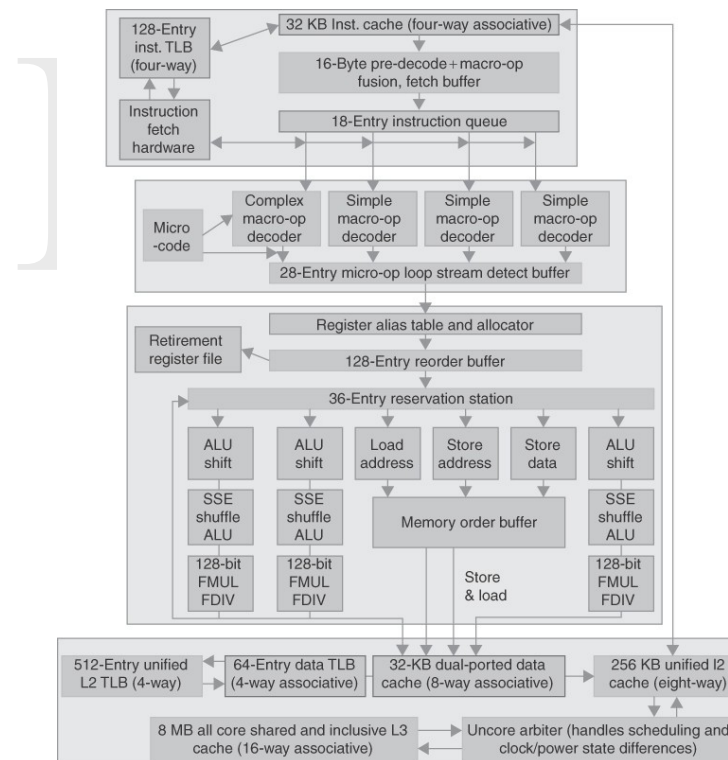
# Intel Core i7 - 2

Some features of Intel Core i7 pipeline are next presented

1. The processor uses a multilevel branch target buffer, located at the instruction fetch stage, to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 15 clock cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.

2. These 16 bytes are placed in the pre-decode instruction buffer, where a procedure called macro-op fusion is executed. *Macro-op fusion* takes instruction combinations such a compare followed by a branch and generates a single operation. The pre-decode stage also breaks the 16 bytes into individual x86 instructions. Individual x86 instructions, including some fused instructions, are placed in the 18-entry instruction queue.

3. Individual x86 instructions are translated into micro-ops which are simple MIPS-like instructions executed directly by the pipeline. This approach was introduced in the Pentium Pro and has been used ever since. Three of the decoders handle x86 instructions that translate directly into one micro-op. For x86 instructions that have more complex semantics, there is a microcode engine that produces the corresponding micro-ops sequence: it can generate up to four micro-ops per clock cycle and goes on until the whole sequence is produced. The micro-ops are placed in the 28-entry micro-op buffer according to the order of the x86 instructions.

4. The micro-op buffer performs *loop stream detection* and *microfusion*. If there is a small sequence of instructions, less than 28 or 256 bytes in length that comprises a loop, the loop stream detector will find the loop and directly issue mico-ops from the buffer, eliminating the need for the instruction fetch and the instruction decode stages to be activated. On the other hand, microfusion combines instructions pairs such as load / ALU operation and ALU operation / store and issues them to a single reservation station, where they can still issue independently.

3. The basic instruction issue comprises looking up the register location in the register tables, renaming the registers, allocating a reorder entry and fetching any results from the registers or the reorder buffer, before sending the micro-ops to the reservation stations.

6. The i7 uses a 36-entry centralized reservation station shared by six function units. Up to six micro-ops may be dispatched to the functional units every clock cycle.

7. Micro-ops are executed by the individual functional units and the results are sent back to any waiting reservation station as well as to the register retirement unit, where the register state is updated once it is asserted that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.

8. When one or more instruction at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed and the instructions are removed from the reorder buffer.

## Performance of Intel Core i7 CPI for the SPECCPU2006 benchmark suite

Source: Computer Architecture: A Quantitative Approach

Departamento de Electrónica, Telecomunicações e Informática
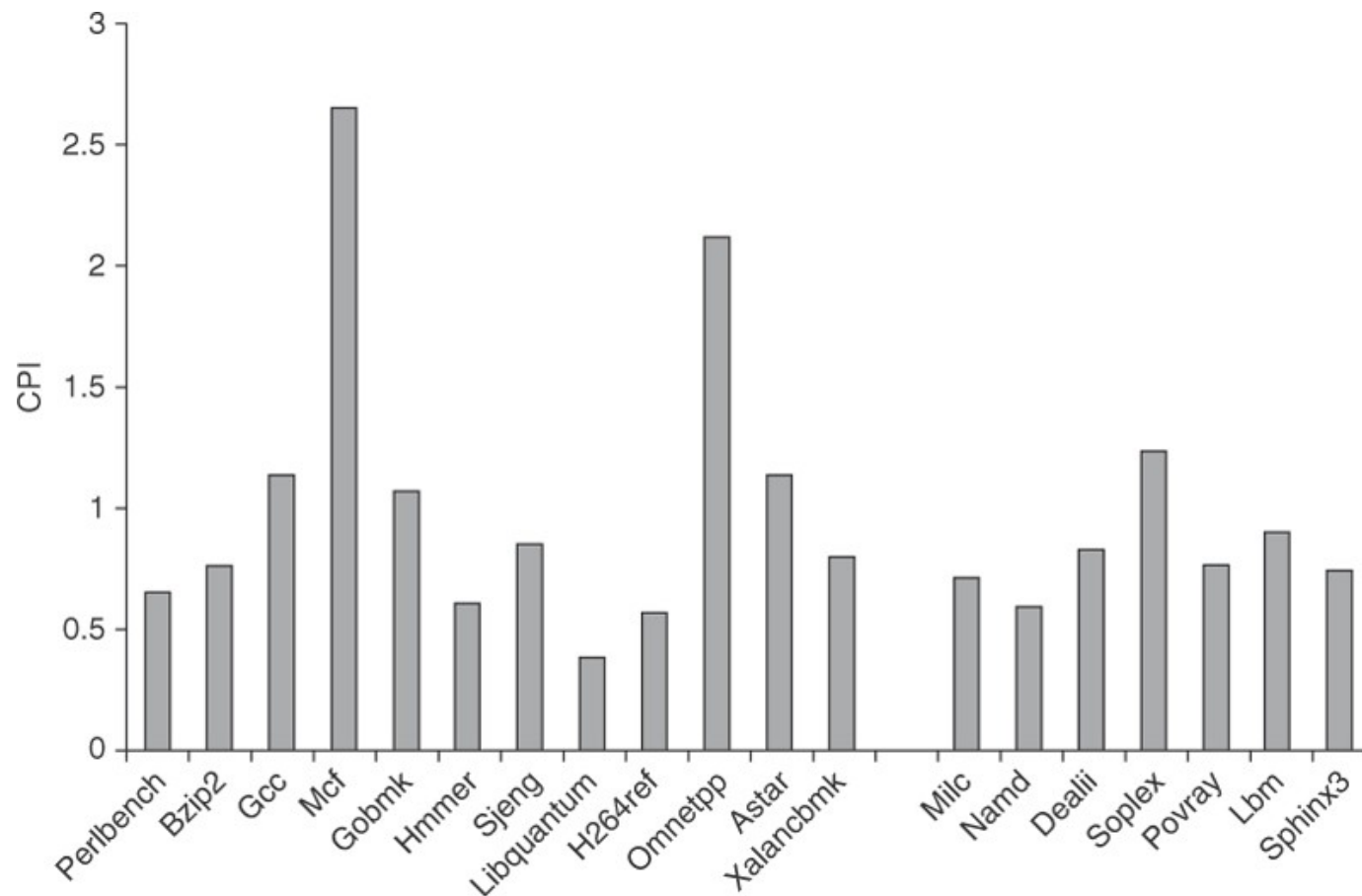
# *Suggested reading*

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
  - Chapter 3: *Instruction-Level Parallelism and its Exploitation* (Sections 1 to 12)
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
  - Chapter 16: *Instruction-Level Parallelism and Superscalar Processors*

Departamento de Electrónica, Telecomunicações e Informática