SUPERSCALAR PROCESSORS

1. What is a Superscalar Architecture?
2. Superpipelining
3. Features of Superscalar Architectures
4. Data Dependencies
5. Policies for Parallel Instruction Execution
6. Register Renaming

What is a Superscalar Architecture?

- A superscalar architecture is one in which several instructions can be initiated *simultaneously* and executed *independently*.

What is a Superscalar Architecture?

- A superscalar architecture is one in which several instructions can be initiated *simultaneously* and executed *independently*.

As opposed to this:

- Pipelining allows several instructions to be executed at the same time, but they have to be in *different* pipeline stages at a given moment.

What is a Superscalar Architecture?

- A superscalar architecture is one in which several instructions can be initiated *simultaneously* and executed *independently*.

As opposed to this:

- Pipelining allows several instructions to be executed at the same time, but they have to be in *different* pipeline stages at a given moment.

Superscalar architectures include all features of pipelining but, in addition, there can be *several instructions executing simultaneously in the same pipeline stage*. They have the ability to initiate multiple instructions *during the same clock cycle*.

The Quest for Speed

There are two typical approaches today, in order to improve performance:

1. Superpipelining
2. Superscalar

Superpipelining

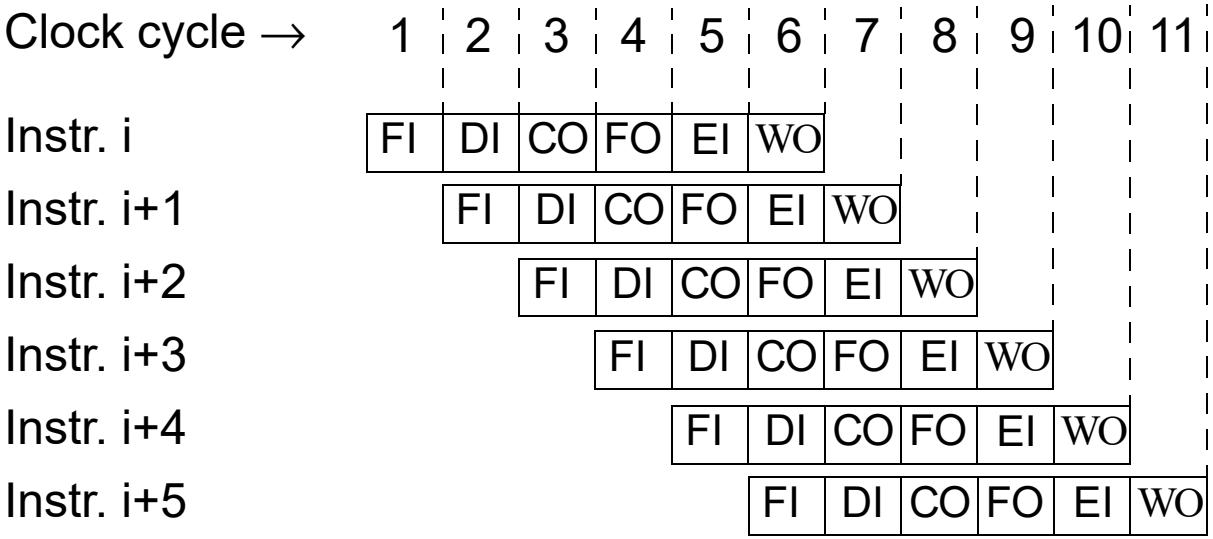
- Superpipelining is based on dividing the stages of a pipeline into substages and thus increasing the number of instructions which are active in the pipeline at a given moment.
  - By dividing each stage into two, the clock cycle period τ is reduced to the half, $\tau/2 \Rightarrow$ at maximum capacity, a result is produced every $\tau/2$ s.

Superpipelining

- Superpipelining is based on dividing the stages of a pipeline into substages and thus increasing the number of instructions which are active in the pipeline at a given moment.
  - By dividing each stage into two, the clock cycle period τ is reduced to the half, $\tau/2 \Rightarrow$ at maximum capacity, a result is produced every $\tau/2$ s.
- For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; *increasing the number of stages over this limit reduces the overall performance*.
- *A solution to further improve speed is the superscalar architecture*.

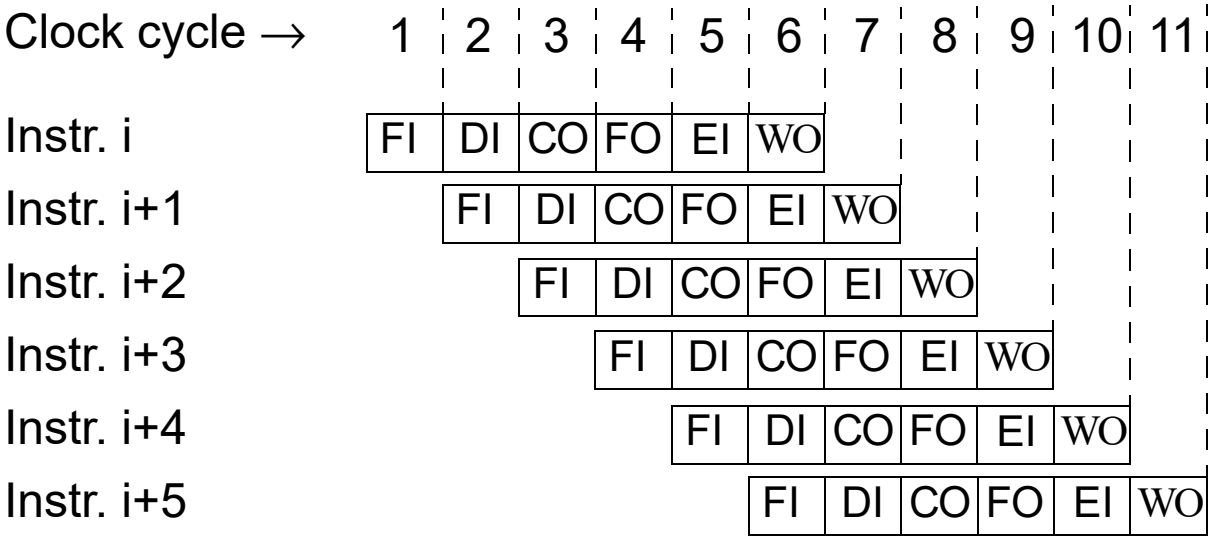
Superpipelining

Pipelined execution

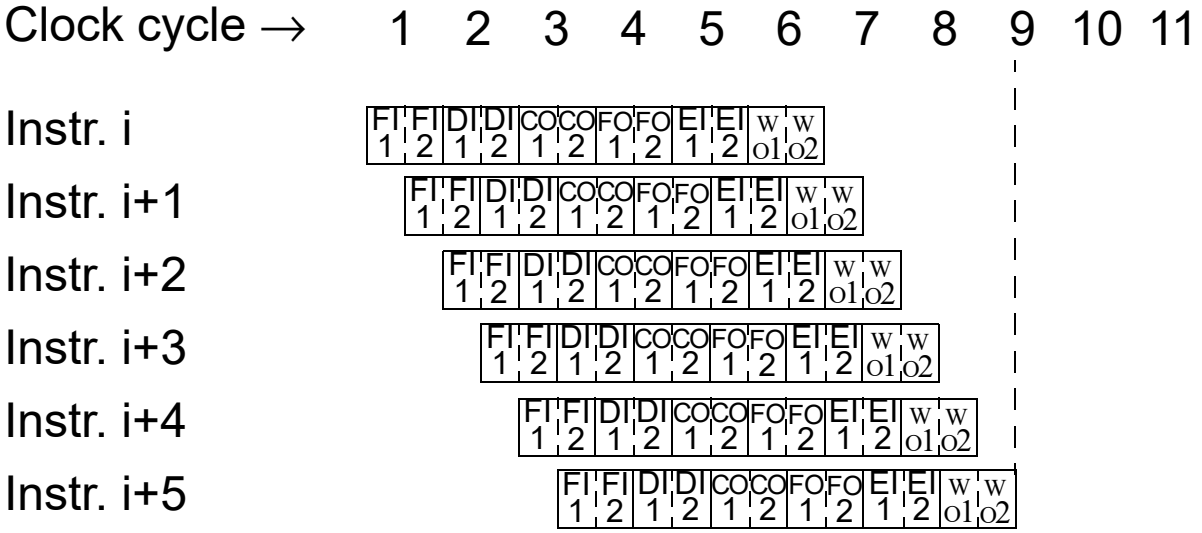


Superpipelining

Pipelined execution

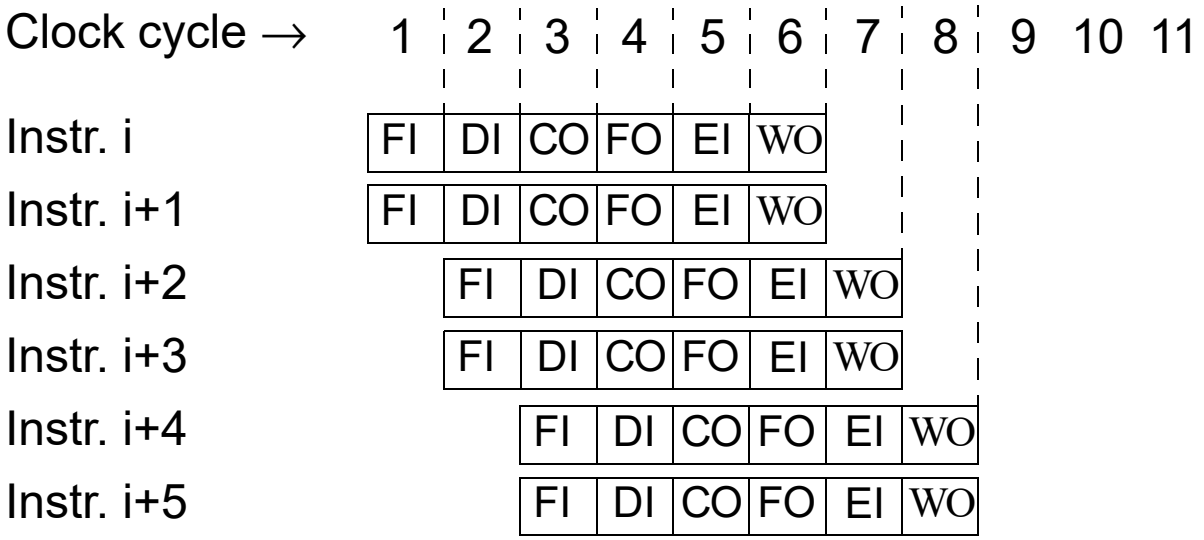


Superpipelined execution



Superscalar Architectures

Superscalar execution



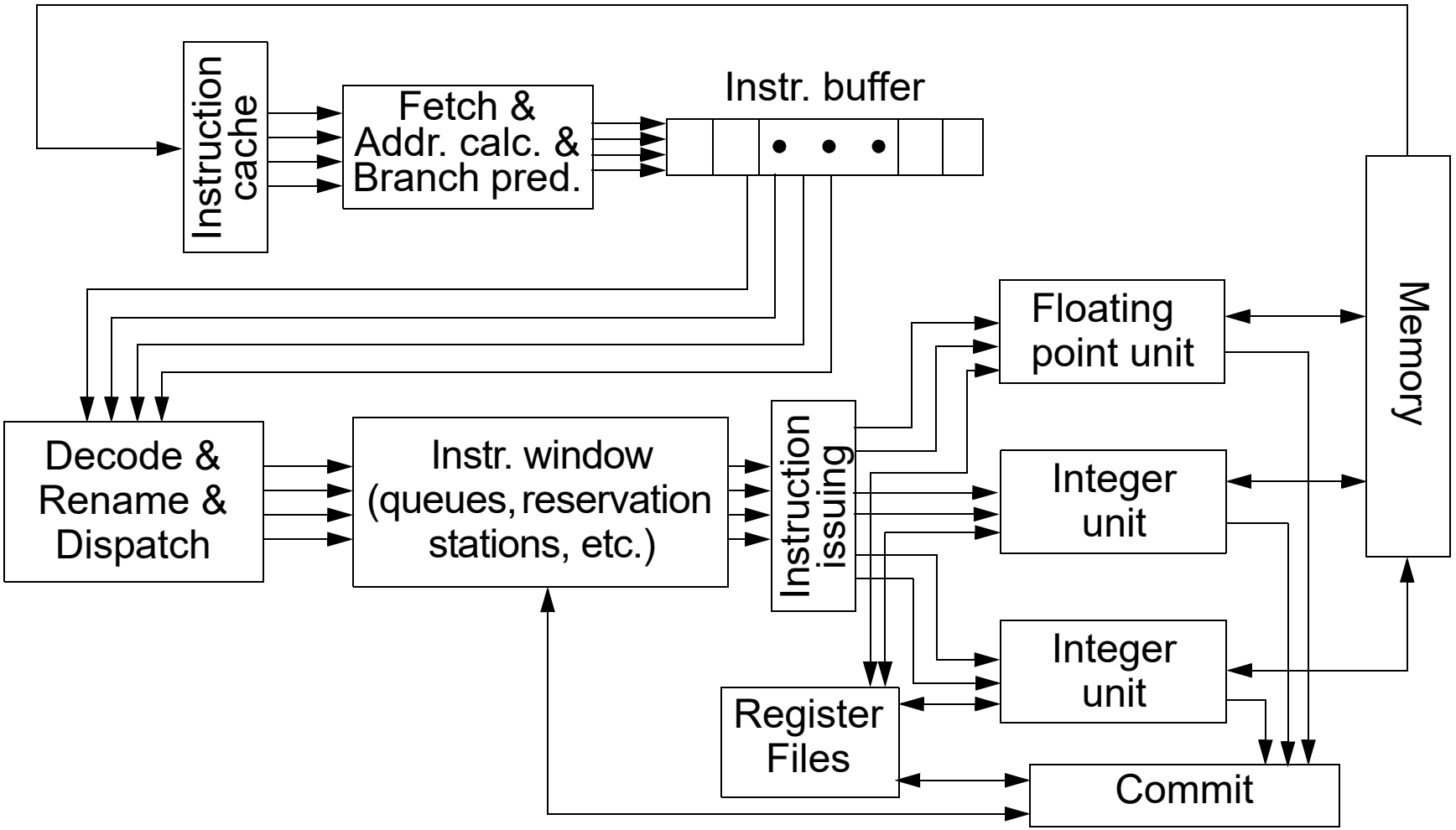
Superscalar Architectures

- Superscalar architectures allow several instructions to be issued and completed per machine cycle.
- A superscalar architecture consists of a number of pipelines that are working in parallel.
- Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel.

Superscalar Architectures

- Superscalar architectures allow several instructions to be issued and completed per machine cycle.
- A superscalar architecture consists of a number of pipelines that are working in parallel.
- Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel.
- In the following example a floating point and two integer operations can be issued and executed simultaneously; each unit is pipelined and can execute several operations in different pipeline stages.

Superscalar Architectures



Limitations on Parallel Execution

- The situations which prevent instructions to be executed in parallel by a superscalar architecture are very similar to those which prevent an efficient execution on any pipelined architecture (see *pipeline hazards*).

- The consequences of these situations on superscalar architectures are more severe than those on simple pipelines, because the potential of parallelism in superscalars is greater and, thus, a greater opportunity is lost.

Limitations on Parallel Execution

Three categories of limitations have to be considered:

Limitations on Parallel Execution

Three categories of limitations have to be considered:

1. **Resource conflicts:**
   - **Occur if several instructions compete for the same resource (register, memory, functional unit) at the same time; they are similar to *structural hazards* discussed with pipelines. Increasing the number of resources, superscalar architectures try to avoid resource conflicts.**

Limitations on Parallel Execution

Three categories of limitations have to be considered:

1. **Resource conflicts:**
   - **Occur if several instructions compete for the same resource (register, memory, functional unit) at the same time; they are similar to *structural hazards* discussed with pipelines. Increasing the number of resources, superscalar architectures try to avoid resource conflicts.**

2. **Control (procedural) dependency:**
   - **The presence of branches creates major problems in assuring an optimal parallelism. How to reduce branch penalties has been discussed in the lecture on pipelining.**

Limitations on Parallel Execution

Three categories of limitations have to be considered:

1. Resource conflicts:
   - Occur if several instructions compete for the same resource (register, memory, functional unit) at the same time; they are similar to *structural hazards* discussed with pipelines. Increasing the number of resources, superscalar architectures try to avoid resource conflicts.

2. Control (procedural) dependency:
   - The presence of branches creates major problems in assuring an optimal parallelism. How to reduce branch penalties has been discussed in the lecture on pipelining.

3. Data conflicts:
   - Data conflicts are produced by *data dependencies* between instructions. Because superscalar architectures provide a great liberty in the order in which instructions can be issued and completed, data dependencies have to be considered with much attention.

Limitations on Parallel Execution

Instructions have to be issued as much as possible in parallel!

Superscalar architectures have to exploit the potential of *instruction level parallelism* present in the program.

Limitations on Parallel Execution

Instructions have to be issued as much as possible in parallel!

Superscalar architectures have to exploit the potential of *instruction level parallelism* present in the program.

- **An important feature of modern superscalar architectures is dynamic instruction scheduling:**
  - **instructions are issued for execution in parallel and *out of order*.**
    - ***out of order issuing*: instructions are issued for execution independent of their sequential order, based only on dependencies and availability of resources.**
  - **All this is done dynamically, at run time, by the hardware.**

Limitations on Parallel Execution

Instructions have to be issued as much as possible in parallel!

Superscalar architectures have to exploit the potential of *instruction level parallelism* present in the program.

- **An important feature of modern superscalar architectures is dynamic instruction scheduling:**
  - **instructions are issued for execution in parallel and *out of order*.**
    - ***out of order issuing*: instructions are issued for execution independent of their sequential order, based only on dependencies and availability of resources.**
  - **All this is done dynamically, at run time, by the hardware.**

Results must be identical with those produced by strictly sequential execution.

Data dependencies have to be considered carefully

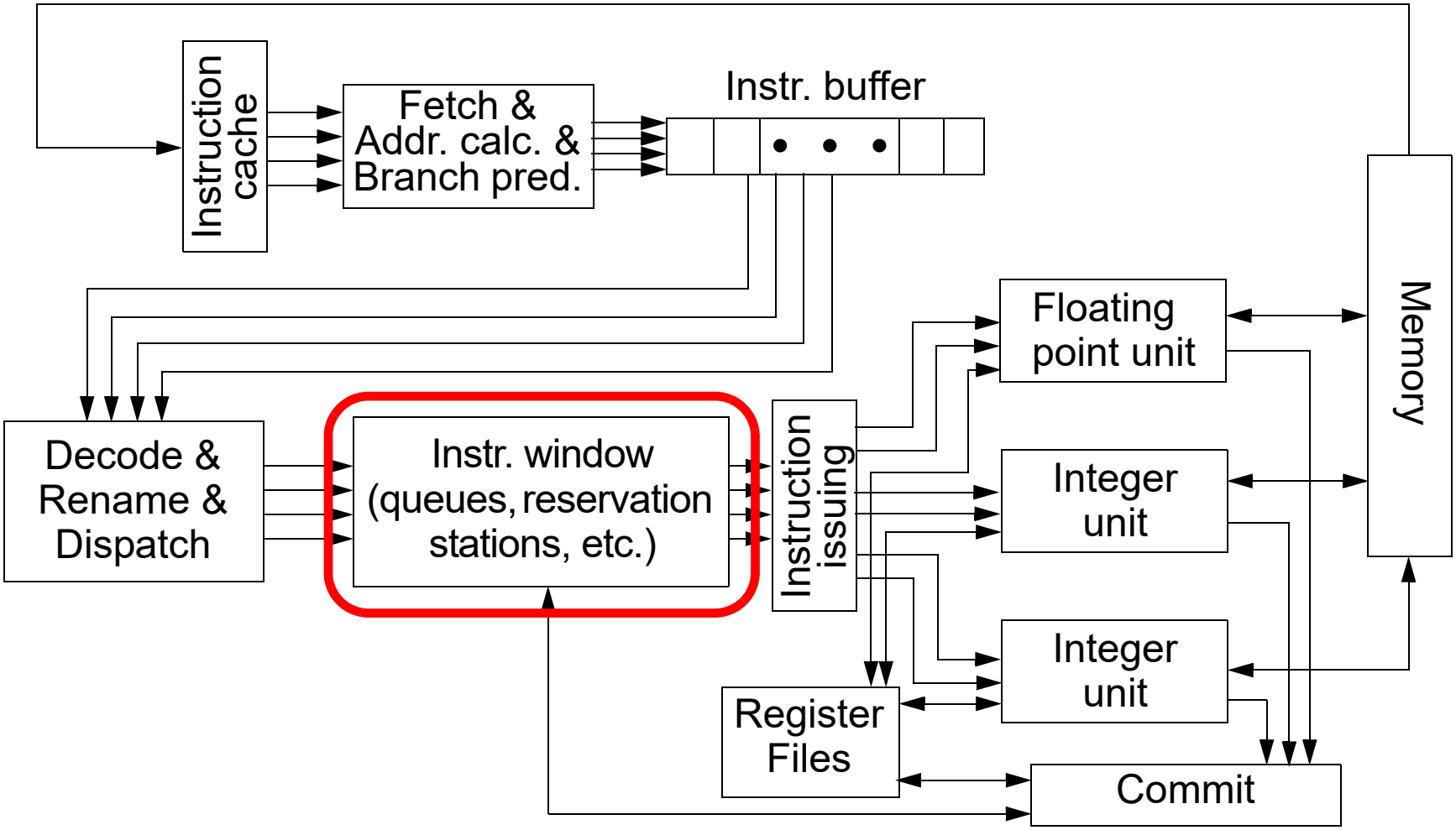
Limitations on Parallel Execution

- Because of data dependencies, only some of the instructions are potential subjects for parallel execution.

- In order to find instructions to be issued in parallel, the processor has to select from a sufficiently large instruction sequence.

A large *instruction window* is needed

Instruction Window



Instruction Window

- **Instruction Window**:

  Contains the set of instructions that is considered for execution at a certain moment. Any instruction in the window can be issued for parallel execution, subject to data dependencies and resource constraints.

- **The number of instructions in the window should be as large as possible.**

  Problems:

  - **Capacity to fetch instructions at a high rate**
  - **The problem of branches**

Instruction Window

```
for (i=0; i<last; i++) {
    if (a[i] > a[i+1]) {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
        change++;
    }
}
```

Instruction Window

```
for (i=0; i<last; i++) {
    if (a[i] > a[i+1]) {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
        change++;
    }
}
```

```
.....................................................................
r7: address of current element (a[i])
r3: address for access to a[i], a[i+1]
r5: change; r4: last; r6: i
.....................................................................

L2    move  r3,r7
      lw    r8,(r3)     r8 ← a[i]
      add   r3,r3,4
      lw    r9,(r3)     r9 ← a[i+1]
      ble   r8,r9,L3                    } basic block 1

      move  r3,r7
      sw    r9,(r3)     a[i] ← r9
      add   r3,r3,4
      sw    r8,(r3)     a[i+1] ← r8
      add   r5,r5,1     change++        } basic block 2

L3    add   r6,r6,1     i++
      add   r7,r7,4
      blt   r6,r4,L2                    } basic block 3
```

Instruction Window

- The instruction window is extended over basic block borders by *branch prediction* with *speculative execution*.

- With speculative execution, instructions of the predicted path are entered into the instruction window.

  Instructions from the predicted path are executed tentatively.
  If the prediction turns out to be correct the state change produced by these instructions will become permanent and visible (the instructions *commit*); if not, all effects are removed.

Data Dependencies

- All instructions in the instruction window may begin execution, subject to *data dependence* (and resource) *constraints*.

- Three types of data dependencies can be identified:

  1. True data dependency
  2. Output dependency
  3. Anti-dependency

  } artificial dependencies

True Data Dependency (read after write, RAW)

- *True data dependency* exists when the output of one instruction is required as an input to a subsequent instruction:

```
MUL   R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - - - - -
ADD   R2,R4,R5    R2 ← R4 + R5
```

- True data dependencies are intrinsic features of the user's program. They *cannot be eliminated* by compiler or hardware techniques.

True Data Dependency (read after write, RAW)

- *True data dependency* exists when the output of one instruction is required as an input to a subsequent instruction:

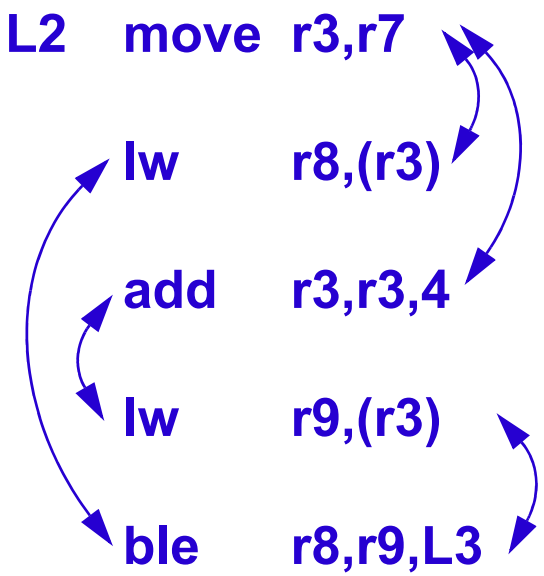
```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - - - -
ADD  R2,R4,R5    R2 ← R4 + R5
```

- True data dependencies are intrinsic features of the user's program. They *cannot be eliminated* by compiler or hardware techniques.

- True data dependencies *have to be detected and treated*: the addition above cannot be executed before the result of the multiplication is available.
  - The simplest solution is to stall the adder until the multiplier has finished.
  - In order to avoid the adder to be stalled, the compiler or hardware can find other instructions which can be executed by the adder until the result of the multiplication is available.

True Data Dependency

For our previous example:



Output Dependency (write after write, WAW)

- An *output dependency* exists if two instructions are writing into the same location; if the second instruction writes before the first one, an error occurs:

```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R4,R2,R5    R4 ← R2 + R5
```

<u>For our previous example:</u>

```
L2   move   r3,r7
     lw     r8,(r3)
     add    r3,r3,4
     lw     r9,(r3)
     ble    r8,r9,L3
```

Anti-dependency (write after read, WAR)

- An *anti-dependency* exists if an instruction uses a location as an operand while a following one is writing into that location; if the first one is still using the location when the second one writes into it, an error occurs:

```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R3,R2,R5    R3 ← R2 + R5
```

For our previous example:

```
L2   move   r3,r7
     lw     r8,(r3)
     add    r3,r3,4
     lw     r9,(r3)
     ble    r8,r9,L3
```

The Nature of Output and Anti-dependency

- Output dependencies and anti-dependencies *are not intrinsic features of the executed program*; they are not *real* data dependencies but *storage conflicts*.
- Output dependencies and anti-dependencies are only the consequence of the manner in which the programmer or the compiler are using registers (or memory locations). They are produced by the competition of several instructions for the same register.
- In the previous examples the conflicts are produced only because:
  - the output dependency: R4 is used by both instructions to store the result;
  - the anti-dependency: R3 is used by the second instruction to store the result;

The Nature of Output and Anti-dependency

- Examples can be written without dependencies *by using additional registers*:

```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R4,R2,R5    R4 ← R2 + R5
```

Output dependency

```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R7,R2,R5    R7 ← R2 + R5
```

No dependency

The Nature of Output and Anti-dependency

- Examples can be written without dependencies *by using additional registers*:

```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R4,R2,R5    R4 ← R2 + R5
```

Output dependency

```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R7,R2,R5    R7 ← R2 + R5
```

No dependency

and

```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R3,R2,R5    R3 ← R2 + R5
```

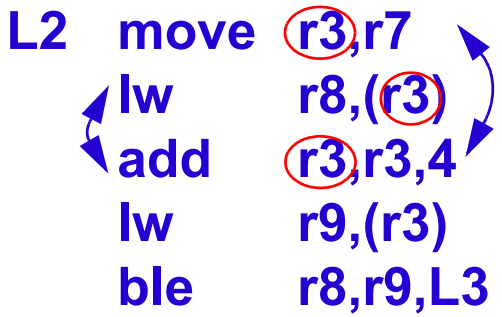
Anti-dependency

```
MUL  R4,R3,R1    R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R6,R2,R5    R6 ← R2 + R5
```

No dependency

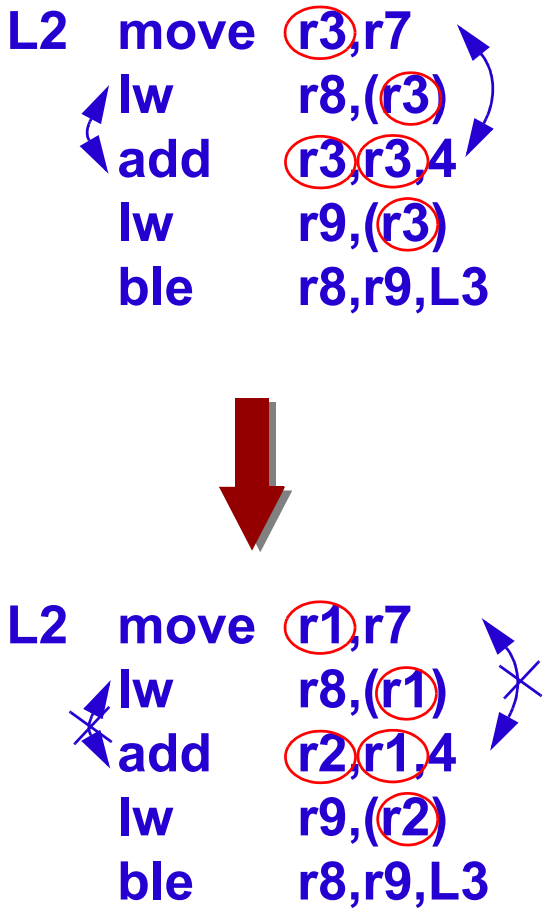
The Nature of Output and Anti-dependency

For our previous example:



The Nature of Output and Anti-dependency

For our previous example:



Policies for Parallel Instruction Execution

- The ability of a superscalar processor to execute instructions in parallel is determined by:
  - the number and nature of functional units (this determines the number and nature of instructions that can be executed at the same time);
  - the policies that the processor uses to find independent instructions (instructions that can be executed in parallel).

- The policies used for instruction execution are defined by two factors:
  1. the order in which instructions are issued for execution;
  2. the order in which instructions are completed (they write results into registers and memory locations).

Policies for Parallel Instruction Execution

- The simplest policy is to execute and complete instructions in their sequential order. This, however, gives little chances to find instructions which can be executed in parallel.
- In order to improve parallelism the processor has to look ahead and try to find independent instructions to execute in parallel.

Policies for Parallel Instruction Execution

- The simplest policy is to execute and complete instructions in their sequential order. This, however, gives little chances to find instructions which can be executed in parallel.
- In order to improve parallelism the processor has to look ahead and try to find independent instructions to execute in parallel.

Instructions will be executed in parallel, *in an order possibly different from the strictly sequential one*, with the restriction that the result must be correct.

- Execution policies:
  - In-order issue with in-order completion.
  - In-order issue with out-of-order completion.
  - Out-of-order issue with out-of-order completion.

In-Order Issue with In-Order Completion

- Instructions are issued in the exact order that would correspond to sequential execution; results are written (completion) in the same order.

  - An instruction cannot be issued before the previous one has been issued; but several *consecutive* instructions can be issued for *execution in parallel*!
  - Instruction are completed in the order of their issuing (in-order).

At issuing, *the processor has to detect and handle true data dependencies*; an instruction can be executed only once the data it needs is available!

Since instructions are executed in order, output dependencies and anti-dependencies cannot create conflicts $\Rightarrow$ they can be ignored.

Out-of-Order Issue with Out-of-Order Completion

- With in-order issue, no new instruction can be issued when the processor has detected a conflict and is stalled, until after the conflict has been resolved.

The processor is not allowed to *look ahead* for further instructions, which could be executed in parallel with the current ones.

Out-of-Order Issue with Out-of-Order Completion

- With in-order issue, no new instruction can be issued when the processor has detected a conflict and is stalled, until after the conflict has been resolved.

  The processor is not allowed to *look ahead* for further instructions, which could be executed in parallel with the current ones.

- Out-of-order issue tries to resolve the above problem. Taking the set of decoded instructions *the processor looks ahead and issues any instruction, in any order, as long as the program execution is correct*.

Out-of-Order Issue with Out-of-Order Completion

- With *out-of-order issue&out-of-order completion* the processor has to bother about true data dependency and both about output-dependency and anti-dependency!

Output dependency can be violated (if addition completes before multiplication):

```
MUL  R4,R3,R1      R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R4,R2,R5      R4 ← R2 + R5
```

Anti-dependency can be violated (if operand in R3 is used after being over-written):

```
MUL  R4,R3,R1      R4 ← R3 * R1
- - - - - - - - - - - -
ADD  R3,R2,R5      R3 ← R2 + R5
```

Register Renaming

- Output dependencies and anti-dependencies can be treated similarly to true data dependencies as normal conflicts. Such conflicts are solved by delaying the execution of a certain instruction until it can be executed.

Register Renaming

- Output dependencies and anti-dependencies can be treated similarly to true data dependencies as normal conflicts. Such conflicts are solved by delaying the execution of a certain instruction until it can be executed.
- Parallelism could be improved by *eliminating* output dependencies and anti-dependencies, which are not real data dependencies.
- Output dependencies and anti-dependencies can be eliminated by automatically allocating new registers to values, when such a dependency has been detected. This technique is called *register renaming*.

Register Renaming

- Output dependencies and anti-dependencies can be treated similarly to true data dependencies as normal conflicts. Such conflicts are solved by delaying the execution of a certain instruction until it can be executed.
- Parallelism could be improved by *eliminating* output dependencies and anti-dependencies, which are not real data dependencies.
- Output dependencies and anti-dependencies can be eliminated by automatically allocating new registers to values, when such a dependency has been detected. This technique is called *register renaming*.

The *output dependency* is eliminated by allocating, R6 to the value R2+R5:

```
MUL  R4,R3,R1   R4 ← R3 * R1          MUL  R4,R3,R1   R4 ← R3 * R1
- - - - - - - - - - - -        ⇒      - - - - - - - - - - - -
ADD  R4,R2,R5   R4 ← R2 + R5          ADD  R6,R2,R5   R6 ← R2 + R5
```

The same is true for the *anti-dependency* below:

```
MUL  R4,R3,R1   R4 ← R3 * R1          MUL  R4,R3,R1   R4 ← R3 * R1
- - - - - - - - - - - -        ⇒      - - - - - - - - - - - -
ADD  R3,R2,R5   R3 ← R2 + R5          ADD  R6,R2,R5   R6 ← R2 + R5
```

Final Comments

- The following main techniques are characteristic for superscalar processors:
  1. additional pipelined units which are working in parallel;
  2. out-of-order issue&out-of-order completion;
  3. register renaming.

- All of the above techniques are aimed to enhance performance.

- Experiments have shown:
  - without the other techniques, only adding additional units is not efficient;
  - out-of-order issue is extremely important; it allows to look ahead for independent instructions;
  - register renaming can improve performance with more than 30%; in this case performance is limited only by *true dependencies*.
  - it is important to provide a fetching/decoding capacity so that the instruction window is sufficiently large.

Some Architectures

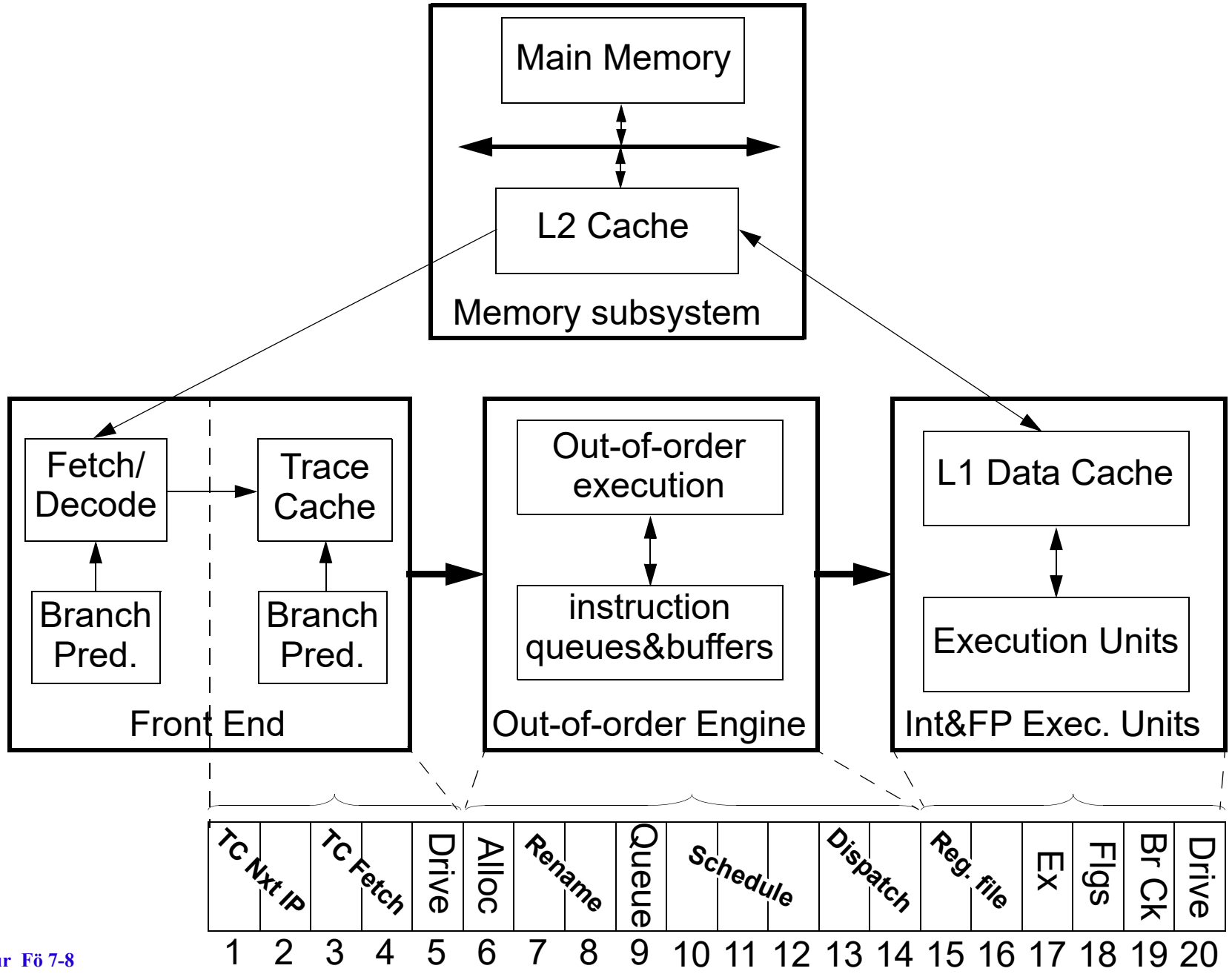
- PowerPC 604
  - six independent execution units:
    - Branch execution unit
    - Load/Store unit
    - 3 Integer units
    - Floating-point unit
  - in-order issue
- Power PC 620
  - provides in addition to the 604 *out-of-order issue*

Some Architectures

- PowerPC 604
  - six independent execution units:
    - Branch execution unit
    - Load/Store unit
    - 3 Integer units
    - Floating-point unit
  - in-order issue
- Power PC 620
  - provides in addition to the 604 *out-of-order issue*
- Pentium
  - three independent execution units:
    - 2 Integer units
    - Floating point unit
  - in-order issue; two instructions issued per clock cycle.
- Pentium II to 4
  - provide in addition to the Pentium *out-of-order execution*
  - five to seven independent execution units

Pentium 4 Basic Block Diagram

Intel calls this architecture *Netburst*



Pentium 4

- The *fetch unit* loads x86 instructions form the L2 cache which, then, are decoded and translated into microoperations that are stored in the *trace cache* (the L1 instruction cache).
  These (RISC-like) microoperations are executed by the Pentium 4 hardware.
- Branch predictions
  - Based on two branch history tables (called BTB - branch target buffer):
    - a 4K-entries BTB for the fetch unit;
    - a 512 -entries BTB for the trace cache.
  - Exceptional: 4 bits branch prediction (very rare to have more than 2 bits prediction!) - this is due to the very long pipeline which comes with a huge penalty for misprediction.
- The out-of-order execution mechanism uses several buffers (which together build the instruction window) to reorder the flow of instructions.
  These buffers store up to 126 microoperations at a given time.
- 20 stage pipeline; the instruction fetch and decode are considered outside (and before) this pipeline.

Pentium 4

- The trace cache stages (1-5):
  - *Trace cache next instruction pointer* (TC Nxt IP): Determines, using branch prediction, the next microoperation to be executed.
  - *Trace cache fetch*: The microoperation is fetched from the trace cache.
  - *Drive*: Delivers the microoperation to the rename/allocator module.

- The out of order engine stages (6-14):
  - *Allocate*: Allocates buffer space in the out of order engine to the new microoperation.
  - *Rename*: Renames the 8 (in 32-mode) registers visible by the programmer using the 128-entry physical register file.
    Removes output dependencies and anti-dependencies.
  - *Queue*: Microoperations are placed into queues where they wait to be scheduled.
  - *Scheduling*: Microoperation schedulers determine when a microoperation is ready to execute (based on dependencies).
  - *Dispatching*: Microoperations ready to be executed are fetched and dispatched to the corresponding functional units (when such a unit is available).
    Six microoperations can be dispatched for execution in one cycle.

Pentium 4

- The execution stages (15-20):
  - *Register file*: Operands are fetched from the register files and L1 data cache.
  - *Execute* (Ex): The microoperation is executed.
  - *Flags*: Flags (e.g. zero, negative) are computed and set.
  - *Branch checking* (Br Ck): The actual branch result is compared with the prediction. If there has been a misprediction the pipeline is cleaned.
  - *Drive*: The branch check result is registered in the BTB for further branch prediction.
    The whole pipeline is restarted.

Pentium 4

- **Execution units**:
  - Addressing units:
    - One address generation unit for memory loads; it also executes the memory loads.
    - One address generation unit for stores.
  - Integer units:
    - Two low latency ALUs; they execute very fast simple operations (e.g. add, subtract, logic operations, integer store).
    - One complex integer unit (for e.g. multiply, divide, shift).
  - Floating point units:
    - One FP execution unit; it executes FP operations, multimedia instruction set (MMX), Streaming SIMD extension (SSE).
    - One FP unit for FP (128-bit) register-to-register moves and memory stores.

The End of the Lonely Big

The Pentium 4 was the last of its kind!

The End of the Lonely Big

The Pentium 4 was the last of its kind!

- It delivered on performance *but*
  - It was running at very high frequency

Very high power consumption
and temperature dissipation!

  - The basic philosophy, to deliver, generation by generation, increased performance by more complex architectures and running at higher frequency was no longer sustainable!

The End of the Lonely Big

The Pentium 4 was the last of its kind!

- It delivered on performance *but*
  - It was running at very high frequency

Very high power consumption
and temperature dissipation!

  - The basic philosophy, to deliver, generation by generation, increased performance by more complex architectures and running at higher frequency was no longer sustainable!

- A radically new approach has been adopted:
  - Replace one large and complex processor running at high frequency with *several simpler processors running at lower frequency*!

Multicore architectures

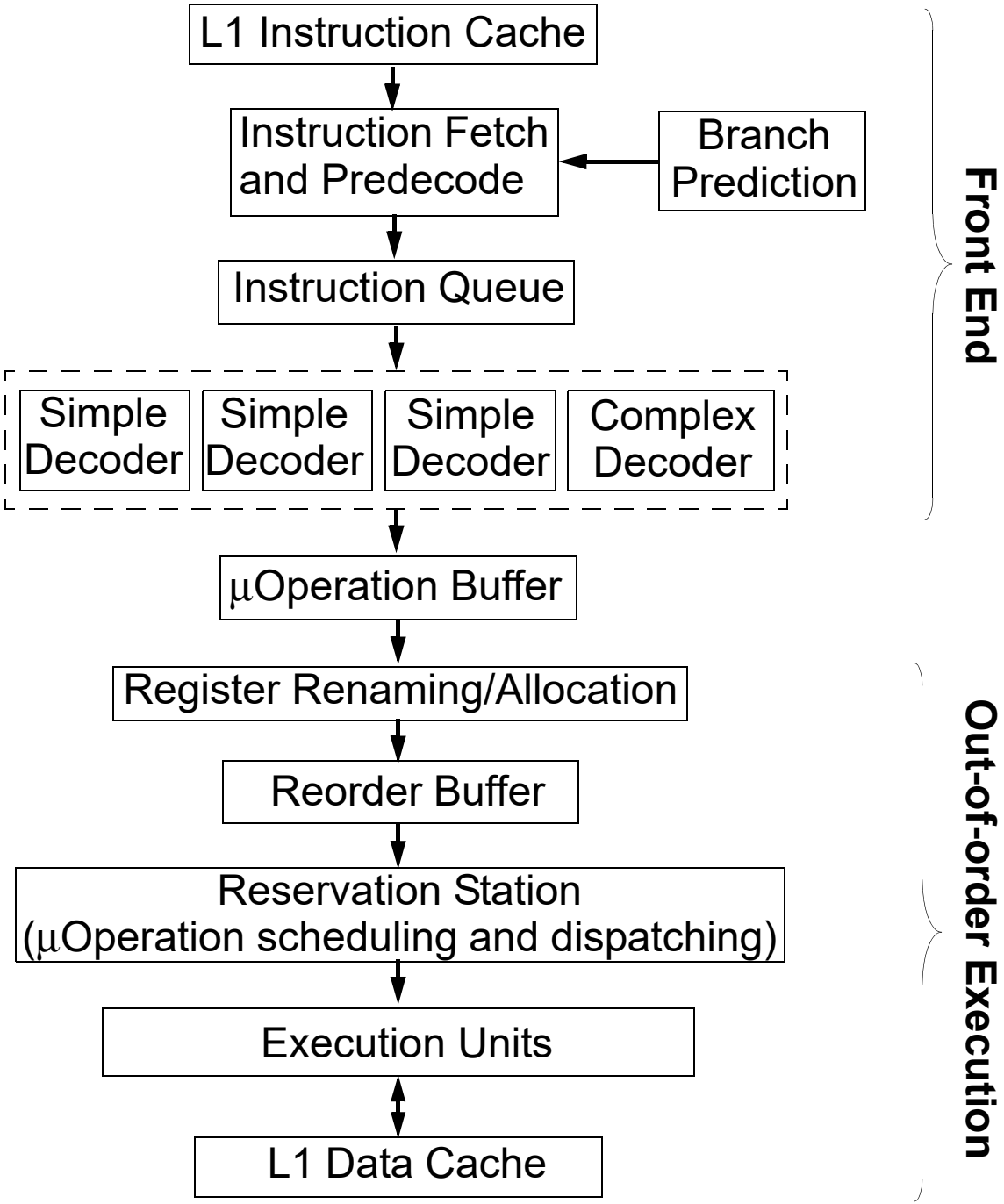
The End of the Lonely Big

- Intel's multicore architectures:
  - Core Duo, Core 2 Duo, Core 2 Quad, Core i3, Core i5, Core i7 ...

- A new architecture has been developed *for the individual processors* on the multicores:
  - the *Intel Core* architecture (in the Core 2).
  - the *Nehalem* architecture, a further development of *Intel Core* (in Core i5, i7).

- Basic principle: keep the power consumption of each core down and increase the power/performance efficiency:
  - Overall processor complexity is reduced, compared to e.g. Pentium 4 (e.g. the number of pipeline stages is 14-16).
  - Processors run at lower frequency.

The *Nehalem* Basic Block Diagram



The *Nehalem* Architecture

Front-End

- The *Front-End* is responsible for loading x86 instructions from the *instruction cache* and translating them into microoperations that are buffered for further execution by the *Out-of-order Execution Back-End*.

- You remember the *trace cache* of the Pentium 4 and the motivation behind it! *Nehalem* is back to the traditional *L1 instruction cache* placed before the *fetch unit*. The *microoperation buffer* (see below) can, however, act like a *trace cache*.

- The *fetch unit* fetches 128 instruction bits every cycle. The *predecoder* takes these 128 bits and prepares the X86 instructions found, for further decoding; the *predecoder* determines instruction length and identifies branch instructions; it writes, up to 6 instructions per cycle, into the *instruction queue* (which stores up to 18 instructions).

The *Nehalem* Architecture

- The *Instruction Decoding Unit* translates the X86 instructions into a stream of RISC-like microoperations; the unit consists of 4 decoders that work in parallel: 3 simple (translate simpler X86 instructions) and one complex (translates X86 instructions that result in longer sequences of microoperations). The generated microoperations are stored in the *microoperation buffer* (capacity of 28 microoperations).

- *Branch prediction* is based (like in Pentium 4) on a large branch target buffer (BTB).

- A *Loop Stream Detector* detects short loops, such that the whole loop fits into the *microoperation buffer*; when such a loop is executed, the microoperations are streamed into the *Execution Back-End* from the *microoperation buffer*, without using the whole *Front-End* (thus, avoid fetch, decode etc.); this works similar to the *trace cache* in the Pentium 4.

The *Nehalem* Architecture

Out-of-order Execution Back-End

- ***Register Renaming & Allocation***: detects dependencies; allocates slots to the microoperation in the *reorder buffer* and *reservation station*; does register renaming.
- The *reorder buffer* (128 entries) and *reservation stations* (36 entries) act like an instruction window.
- The scheduler can dispatch for execution maximum 6 microoperations per clock cycle: 3 arithmetic/logic operations and 3 memory operations.

L1 caches

- L1 Instruction cache: 32 KB, 4-way associative.
- L1 data cache: 32 KB, 8 way-associative.
- Level 2&3 caches are shared by several cores (see lecture on Parallel architectures).

The *Nehalem* architecture (like its predecessor, *Intel Core*) is implemented in Intel's multicore chips (e.g. Core i5, i7) - (see lecture on Parallel architectures).

ARM Cortex-A8

An embedded RISC processor from the ARM family for complex applications (wireless, imaging, gaming, etc.).

- **13-stage pipeline**
- **In-order issue (to keep power consumption reduced).**
  Two instructions issued per clock cycle.
- **2 integer ALUs, one integer multiplier, one load/store unit + the *NEON* unit.**

  The *NEON* unit implements packed SIMD (see lecture on parallel architectures) instructions. It can handle both integer and single precision floating-point values.