Computer architectures

How to increase the processor's performance

Általános INFORMATIKAI Tanszék

Today's topics

- CISC and RISC
- Parallelism
- Instruction level parallelism
- Pipeline processing
- Multiplication (superscalarity)
- Dealing with dependencies
- Maintaining the serial consistency

Performance enhancement

- **Non-structural methods**
  - Increase the clock frequency,
  - Reducing the number of instructions (optimization)
- **Structural methods**
  - Cycle number reduction: with RISC architectures ...
  - Cycle number reduction with parallelization

CISC and RISC

- CISC: Complex Instruction Set Computer
- RISC: Reduced Instruction Set Computer
  - (These are CPU characteristics)
- Historically, CISCs come first
  - the more you use the hardware,
  - complex instructions with micro-programs,
  - programming is easier with complex instructions (e.g. PUSHALL),
  - provide complex addressing modes.
  - The idea is very good, but ...

The RISC idea

- Statistics show that simple instructions are more common.
- Then let's “optimise" the CPU for them! (This is the new idea!)
- The simple instructions have the same logic:
  - simpler circuits are faster,
  - simpler, uniform decoding, which is also faster,
  - there can be more registers, this also speeds it up,
  - the addressing methods are also simpler.
- More complex tasks, on the other hand, require more instructions. Maybe the program will be longer.

Additional benefits

- The cycle time is the same (mostly 1 instruction / 1 cycle) This helps with super channeling (see later).
- Simple circuits (allowing higher frequency) allow multiple internal units. Superscalarity is possible.
- "Speculative execution" is also easier.
- The cache also fits in the chip, it is getting bigger.
- Matching to operating system and compiler.

Parallels

- **Inside CPU :**
  - Application of a pipeline, channel,
  - With multiplication: several instructions are processed in parallel
- **Apart from CPU:**
  - Fixed task distribution (co-processors)
    - for floating point arithmetic,
    - for graphics, image processing, etc.
  - Multiprocessor systems with variable task distribution (dual/quad systems).

Available and utilized parallelism

- **Concurrency is one of the best performance enhancing techniques**

- **The available parallelism: what arises from the task, from their solution, it is included in the solution of the problem**

- **Utilized parallelism: what we can enforce during execution**

Általános INFORMATIKAI Tanszék

The available and utilized parallelism

- It has two types: functional parallelism and data parallelism.
- Functional parallelism comes from the logic of the task solution. It is conceivable that even in an imperative program some threads could run in parallel.
  Functional parallelism is usually irregular (except cycle-level parallelism).
  The level of parallelism is not high (weak parallelism).
- Data parallelism comes from the use of data structures whose elements can be operated in parallel.
  Mostly regular parallelism.
  The parallelism can be strong (large, multi-digit sized).

Data parallelism

- **A data-parallel architecture is required.**
- **Vector processors.**

The levels of the available functional parallelism

- **Granulation can be different**
  - **Instruction level parallelism (fine granularity);**
    - **Instructions are executed in parallel**
  - **Cycle level parallelism (medium granularity);**
    - **Different consecutive iterations in parallel...**
  - **Process-level parallelism (medium granularity);**
    - **Procedures, function calls in parallel... Threads...**
  - **Program level parallelism (coarse granularity).**
    - **User level. Processes (tasks) in parallel.**
    - **You need the help of the operating system to use them. Also you need a multi-processor HW.**

Utilization of procedure-level parallelism

- **Procedures in parallel .**
  - **Threads must be applied**
  - **You can use a development system,**
  - **utilized with the help of the operating system**

Utilization of cycle-level parallelism

- **Iterations in parallel.**
  - **The compiler helps you discover this**

Utilization of instruction-level parallelism

- **Instructions are executed in parallel with instruction level parallel architectures (Instuction-Level Parallel, ILP processors)**
  - **In traditional "serial" programs, this remains hidden (transparent): the processor, or the compiler discovers the possibility of parallelization inherent in the program.**

- **Pipeline processing and**
- **with multiplication of the functional elements within the processor.**

Pipeline processing

- Processing of a single instruction goes through several stages. At least:
  - instruction retrieval (fetch),
  - decoding (decode) (and instruction „dispatch or allocate”),
  - the actual execution (commit),
  - writing back the result (retire).
- Each stages are carried out by different units, they can work in parallel:
  - during the execution of the i. instruction
  - i+1 can be decoded, instruction,
  - i+2 can be retrieved etc.

The benefits of the RISC

- **Same instructions - same execution times.**
- **One instruction can actually executed in one cycle!**

There are also problems

- **Timing risk: an instruction needs the result of the previous one. You have to wait for it. Dependency.**

The superpipe of the R3000

- It divides the execution of the instructions into 5 stages. Each stage divided to 2 phases.
- 1 stage/1 cycle
- The stages:
  - Instruction retrieval (Fetch) IF
  - Readings, inspection RD
  - ALU operations ALU
  - Data memory access MEM
  - Register write back WB

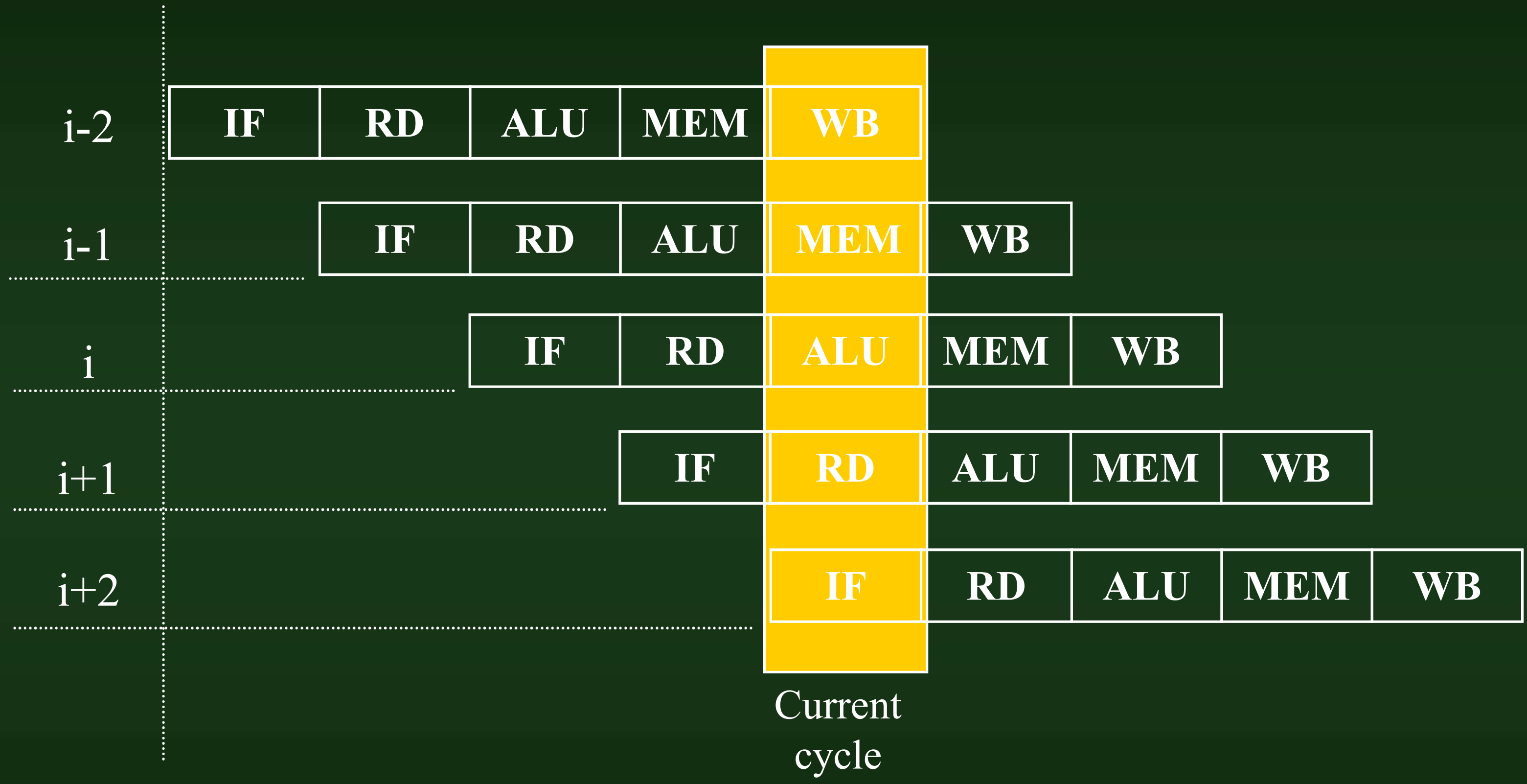
Used during the execution of the instruction

- Address translation is supported by an associative memory (TLB, Translation Lookaside Buffer),
- the instruction cache (I-Cache),
- the data cache (D-Cache),
- the register file (RF).

The activities of the stages and phases

Stage	Phase	Arithmetic instruction	Store instruction	Load instruction	Jump instruction
IF	01	Mapping a virtual address to a physical one using a TLB			
	02	Sends the mapped address to I-Cache			
RD	01	Retrieves from the I-Cache, decodes, checks			
	02	Reading registry file			Address calculation
ALU	01	Arithmetic calculation	Data address calculation		Decision
	02		Data address mapping		
MEM	01		Send address to D-Cache		
	02		Moving data		
WB	01	Write registry file		Write registry file	

The 5-depth pipeline



PowerPC 601 assembly lines

- Branches
  - Extraction + Decoding-Dispatch-Execution-Estimation (2 stages)
- Fixed-point arithmetic
  - Fetch + Decode-dispatch + Execute + Writeback (4 stages)
- Load/Store instructions
  - Fetch + Decode-dispatch + Address calculation + Cache + Writeback (5 stages)
- Floating point arithmetic
  - Fetch + Decode + Dispatch + Execution1 + Execution2 + Writeback (6 stages)

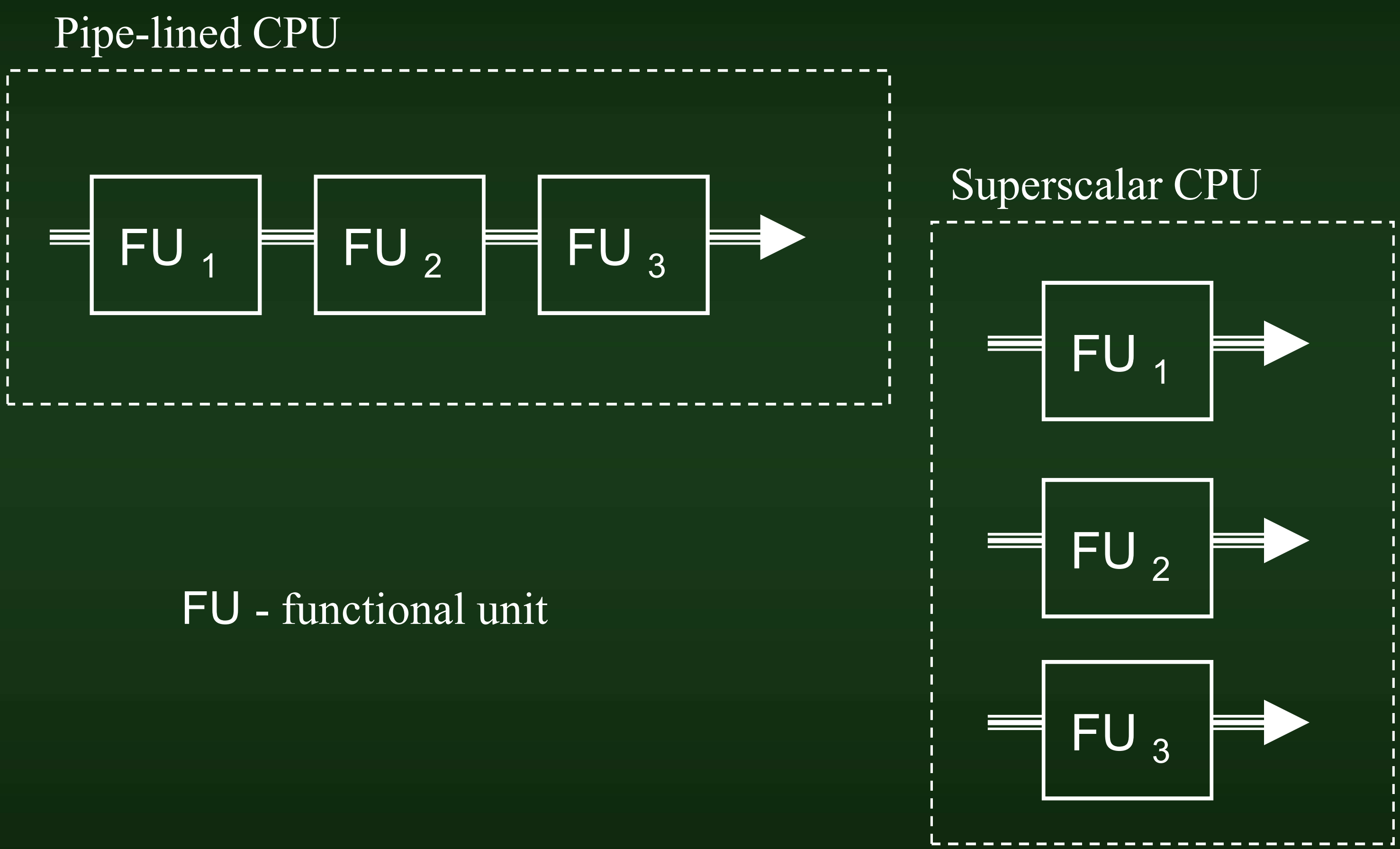
Comments

- The super-channel CPU: a lot of stages
- The ppipeline technique can be applied not only within the processor (at the micro level).
- It is also used at the macro level (several processors form a pipeline).
- Also on a logical level (pay attention to the shell pipeline)
- Dataflow machines can be also considered pipeline

Multiplication of functional units

- Multiplication of functional units is a common parallelization technique
- Multiplication is also possible in instruction-level parallelization:
  - Multiple decoders
  - Multiple execution units (ALU/EU) etc.
- Multiplication is also natural at the macro level
  - See MIMD parallelism

Pipeline versus Multiplication



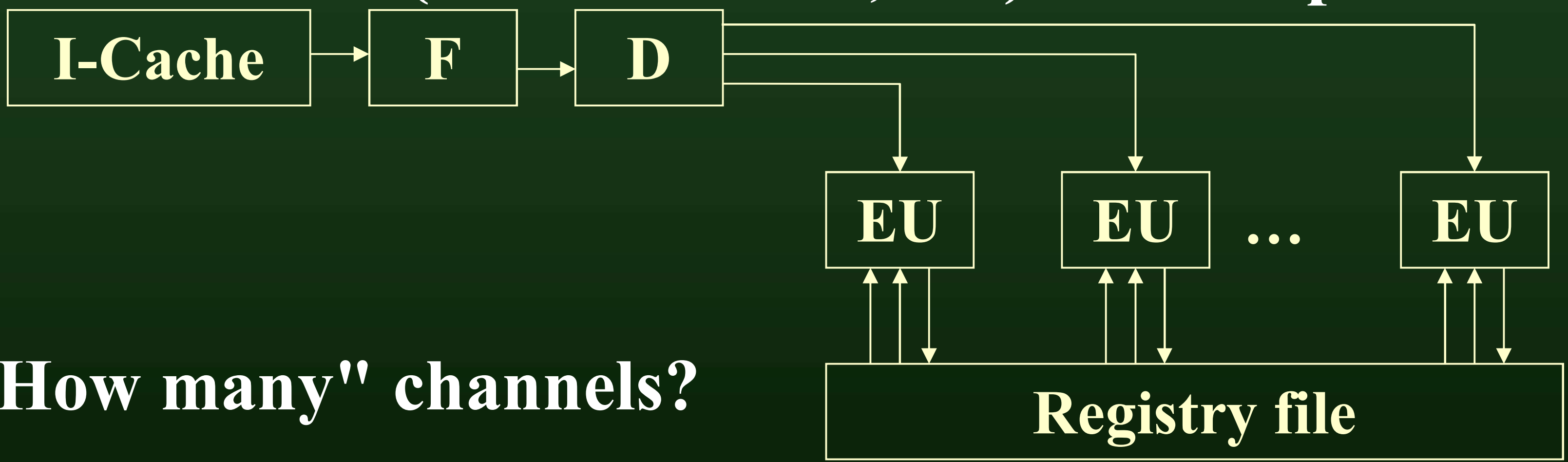
FU - functional unit

Multiplication within the processor

- One type: VLIW (Very Long Instruction Word) architectures
  - E.g. Trace, Intel IA64
  - A special compiler produces the long instruction (e.g. a floating-point and a fixed-point ADD or MUL in a long instruction, possibly even "wider")
  - Several ALUs, in parallel, the long instruction is "decomposed" by the decoder
  - Static dependency resolution (see dependency later)
- Another: superscalar processors

Multiplication within a processor, superscalar processors

- In one step (time window) several traditional instructions are fetched in
- Several traditional instructions are analyzed by (possibly several) decoders, and **multiple instructions are issued for execution**
- Several ALUs (execution unit, EU) work in parallel



- "How many" channels?

Superscalar processors

- It is characterized by **dynamic dependency resolution**
- The pipeline technique is also common
- Typical tasks in superscalar processing
  - Parallel decoding
  - Superscalar (multipath) instruction issue
  - Parallel execution
  - Maintaining serial consistency of execution
  - Maintaining serial consistency of exception

(The idea came up as early as 1970 [Tjaden and Flynn]. The first superscalar processor Released in 1982-83 [IBM]. The name superscalar has been used since 1989.)

Dependencies between instructions

- **Dependencies are the fundamental limitation of parallel execution**
- **Data dependency: an instruction uses the result of the previous one**
- **Control dependency: conditional jump (control transfer) the control branches are depending on the result of the instruction**
- **Resource dependency: instructions require the same resource (e.g. some execution unit, ALU)**

Data dependencies

- **A real dependency** is the RAW (Read after Write) dependency

  i1: load r1, a // r1 ← (a)

  i2: add r2, r1, r1 // r2 ← (r1) + (r1)

- **False dependencies** are WAR (Write after Read) and WAW (Write after Write) dependencies,

  i1: mul r1, r2, r3 // r1 ← (r2) * (r3)

  i2: add r2, r4, r5 // r2 ← (r4) + (r5)

They **can be resolved by registry renaming**

  i1: mul r1, r2, r3 // r1 ← (r2) * (r3)

  i2: add **r36** , r4, r5 // r36 ← (r4) + (r5)

Additional data dependencies, dependency graph

- Cycle dependency (repetition dependency)
  - In the case of k-th order cycle dependency, the instruction in question is depending on the value calculated in the k. previous cycles
- Data and control dependencies can be discovered and recorded in a dependency graph.
  - Directed graph: nodes are instructions, edges are dependencies
- A dependency graph can help reorder instructions to resolve true dependencies.

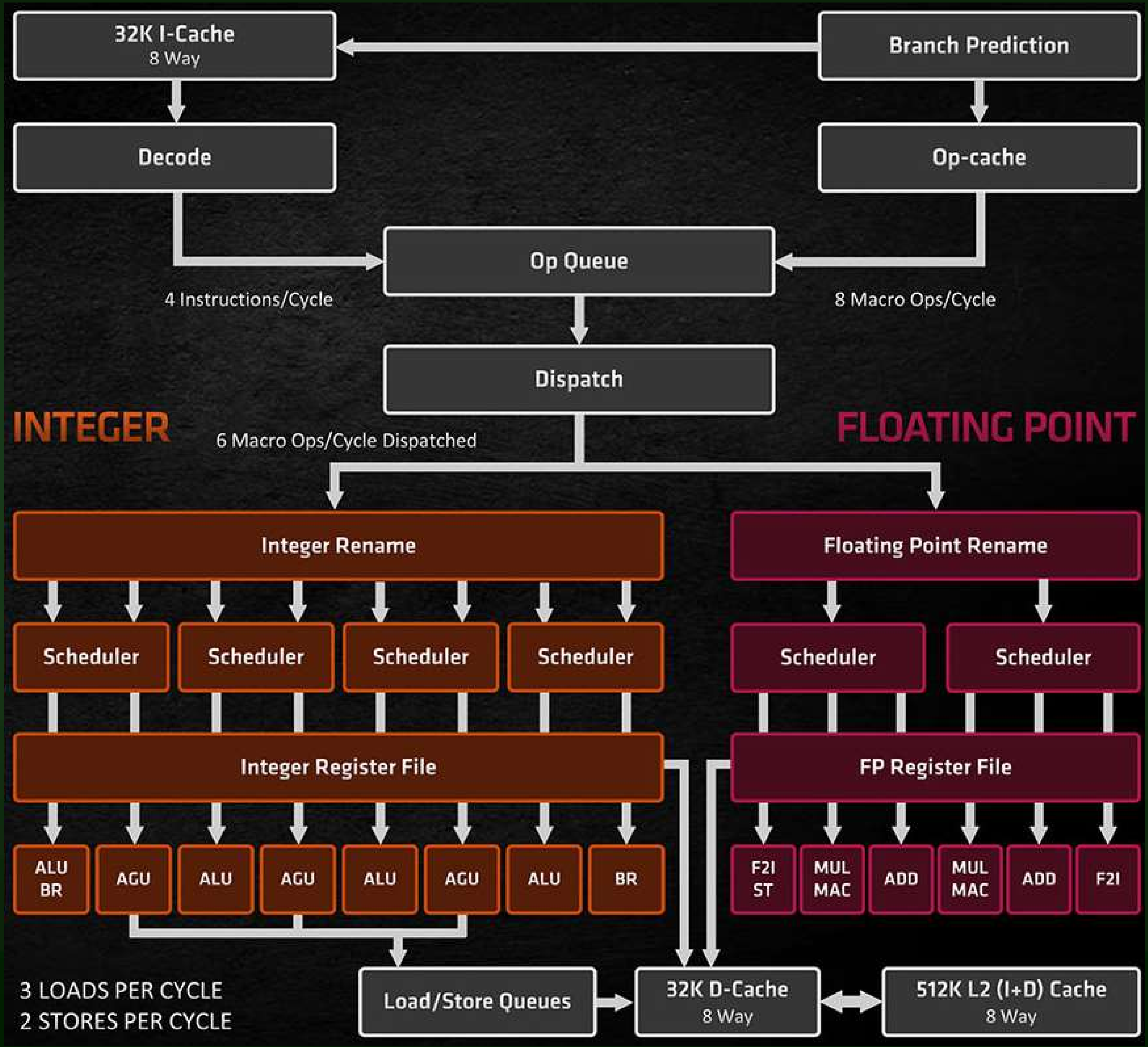
Detecting and resolving dependencies

- **Detection and resolution of dependencies can be static or dynamic**
- **Static: the compiler detects and resolves it: it generates a reordered sequence of instructions** (code optimization)
  - **VLIW (Very Long Instruction Word) processors expect a sequence of instructions without dependencies**
  - **It can also be used for superscalar and pipeline processors**
- **Dynamic: detection and management of dependencies is the task of the processor**
  - **Most superscalar processors are use it**

Dynamic dependency handling

- The processor uses two sliding windows
  - Instruction window, in which
    - there are the instructions that you would issue in the next cycle ;
  - Execution window, in which
    - The instructions are still being executed (there is no result yet).
- In every step, it checks if in the Instruction window
  - there is an instruction dependent on the instructions of the execution window (not ready yet) ,
  - respectively is there a dependency between the instructions of the instruction window.
  - The dispatch depends on these (the independents can be dispatched for execution) and the dispatch policy

AMD " Zen 3 " Microarchitecture



Dispatch policies

- **Blocking execution**
  - It blocks an instruction until its dependency is removed
- **Out of order execution**
  - dispatch independents out of sequence after the blocked one
- **Speculative execution**
  - It issues both branches to handle control dependency

Speculative execution

- Each instruction (operation, elementary instruction) is executed as soon as possible, and regardless of whether its result will be needed or not ...
  (as soon as possible + regardless of its necessity.
  If it is unnecessary, it is needed to be ensured that it does not cause an error!).
- The "load" instructions (are quite frequent and quite expensive) e.g. it is advisable to perform it speculatively (as soon as possible and in any case).

Maintaining serial consistency

- **Consistency here: free from contradiction**
  - **If the order is "overturned" because of the static or dynamic dependency management, or code optimization? The programmer's intention? Logical integrity?**
- **Even with parallel execution, the logic of serial execution must be maintained!** **(Instruction level parallelism)**
- **Serial consistency can be**
  - **Serial consistency of instruction processing,**
    - **Processor consistency (order of instructions)**
    - **Memory consistency (order of memory accesses)**
  - **Serial consistency of exception processing**

Processor consistency

- **The order in which the instructions are completed is the question**
- **In case of “Weak consistency”, the execution order may differ from the programmed one, but this should not cause an integrity error.**
- **In case of “Strong consistency”, the execution order must be the programmed one**
  - **Most of the time this is done by a reordering buffer (ROB , ReOrder Buffer)**

ROB

- **ROB is a circular buffer with start and end pointers. The start pointer indicates the location of the next free entry. The status of the instruction associated with the entry (issued, in progress, completed) is recorded for each entry. An instruction can only be written back, when it has finished and all the instructions before it has already been written back.**
- **ROB also supports the validation of instructions resulting from speculative execution** (if it turns out that it is really necessary, or not) **and non-validation** (with additional status indicator)

The interrupt handling sequence

- **This consistency is also supported by ROB**
- **interruptions and exceptions are accepted by the processor when the instruction is written back (validated) from the ROB**

Memory consistency

- **In case of weak memory consistency, there may be a deviation from the programmed order**
  - **where the programmer's intention is not violated,**
  - **Speculative execution is also possible with load-store instructions: usually loads can precede stores**

Computer architectures

How to increase the processor's performance

End