INF2270 — Spring 2010

Philipp Häfliger

Lecture 6: Microcode, Cache, Pipelining

UNIVERSITETET I OSLO

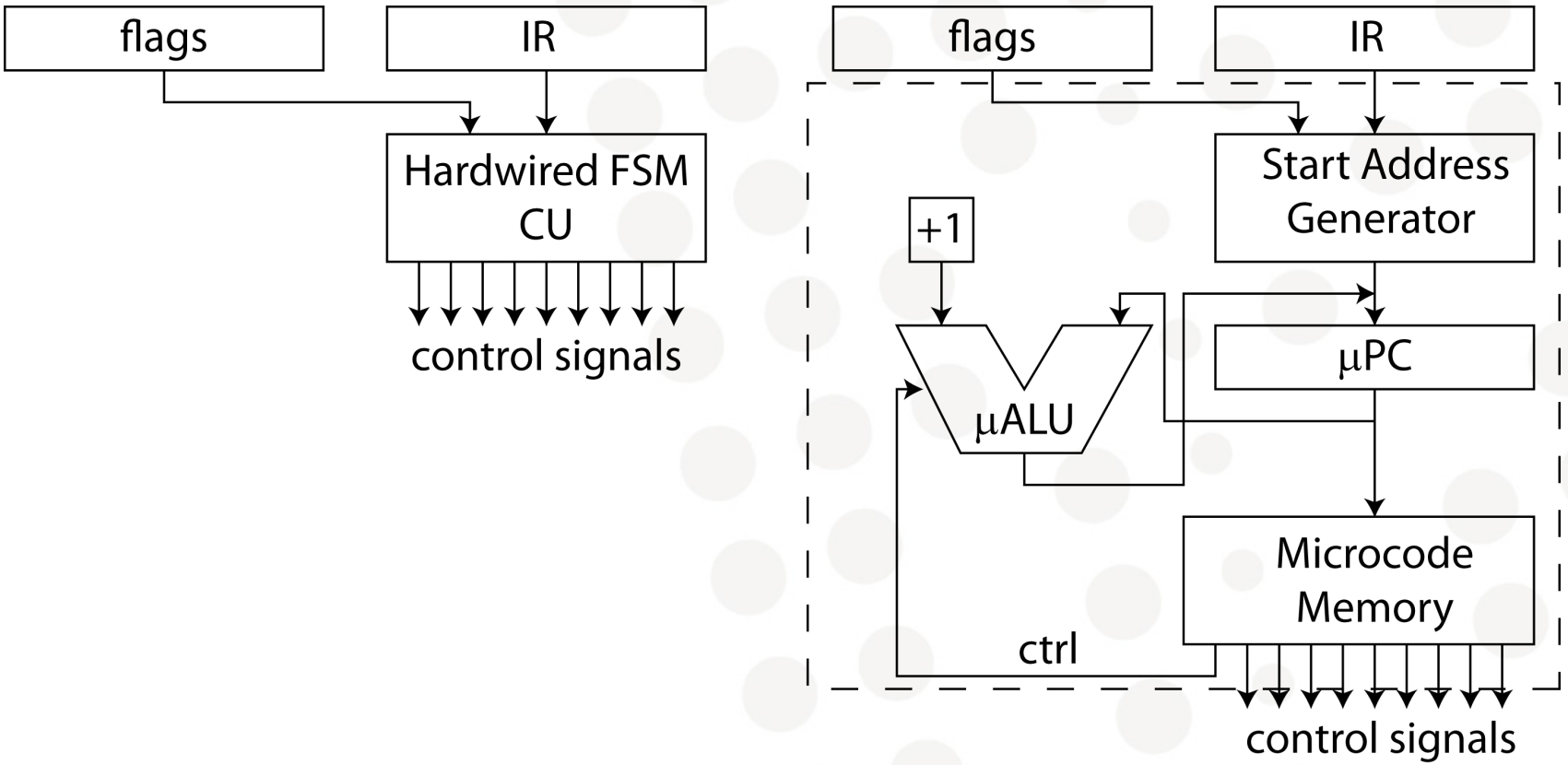
content

content

Microarchitecture

- So far we have considered a so called *hardwired* CU architecture, where a hardwired FSM issues the right sequence of control signals in response to a machine code in the IR.
- A more flexible alternative is to use *microcode* and a simple 'processor' within the processor that simply issues a sequence of control signals stored as *microinstructions* in the *microprogram memory*, typically a fast read only memory (ROM) but sometimes also a flash memory (i.e. electrically erasable programmable read only memory (EEPROM)).

Hardwired and Microprogrammed CU

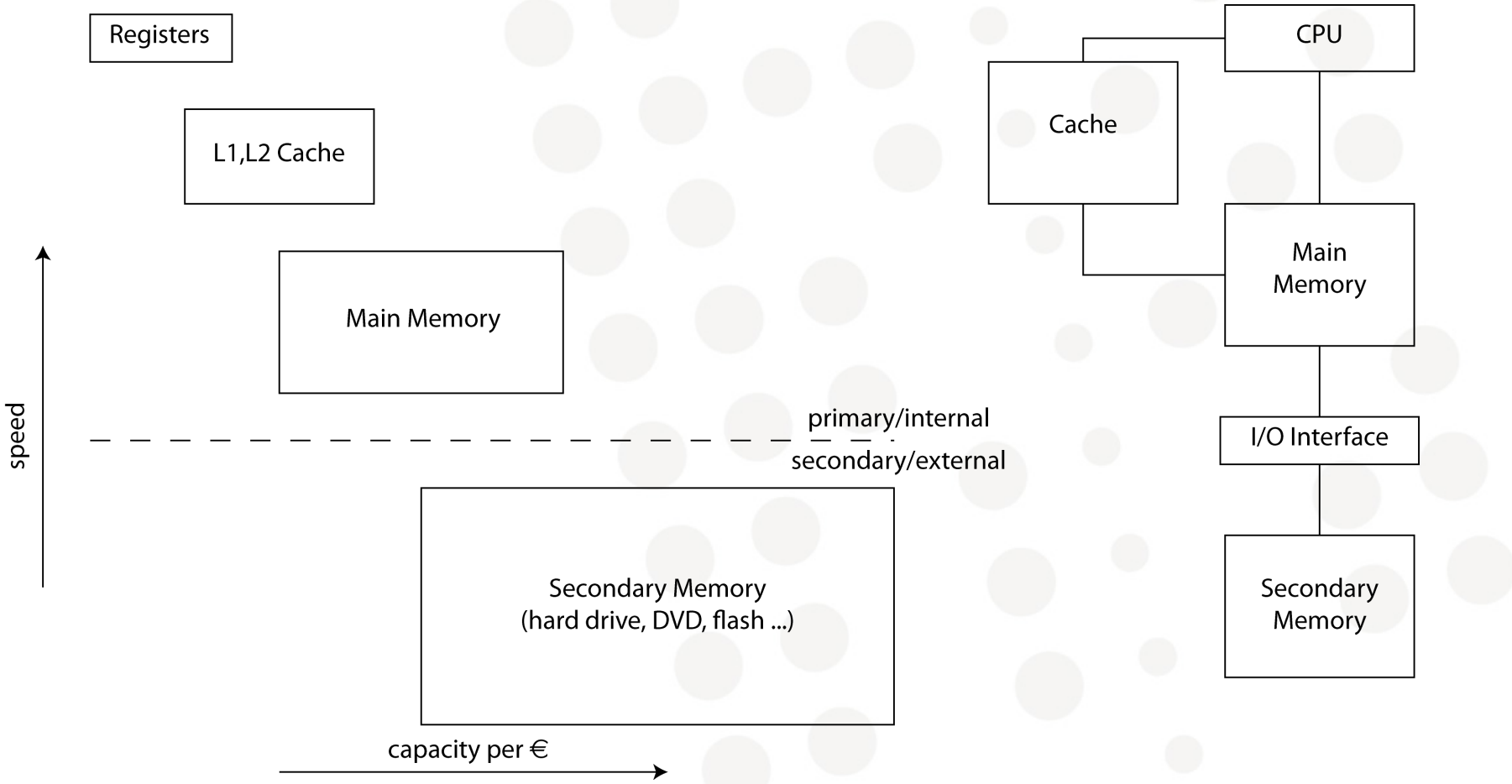


Pros and Cons

	Microarchitecture	Hardwired
Occurrence	CISC	RISC
Flexibility	+	-
Design Cycle	+	-
Speed	-	+
Compactness	-	+
Power	-	+

content

Memory Hierarchy



content

Cache

Cache is used to ameliorate the von Neumann memory access bottleneck. Cache refers to a small high speed RAM integrated into the CPU or close to the CPU. Access time to cache memory is considerably faster than to the main memory. Cache is small to reduce cost, but also because there is always a trade off between access speed and memory size. Thus, modern architectures include also several hierarchical levels of cache (L1, L2, L3 ...).

Locality of Code and Data

Cache uses the principle of *locality* of code and data of a program, i.e. that code/data that is used close in time is often also close in space (memory address). Thus, instead of only fetching a single word from the main memory, a whole block around that single word is fetched and stored in the cache. Any subsequent load or write instructions that fall within that block (a cache *hit*,) will not access the main memory but only the cache. If an access is attempted to a word that is not yet in the cache (a cache *miss*) a new block is fetched into the cache (paying a penalty of longer access time).

Checking for Hits

Checking for hits or misses quickly is a prerequisite for the usefulness of cache memory.

- associative cache
  Parallel search (extremely specialized HW) among memory block tags in the cache.
- direct mapped cache
  A hash-function assigns each memory block to only one cache slot, only one tag needs to be checked
- set-associative cache
  A combination of the previous two: each memory block is hashed to one block-set in the cache. Quick search for the tag needs only to be conducted within the set.

Cache Coherency

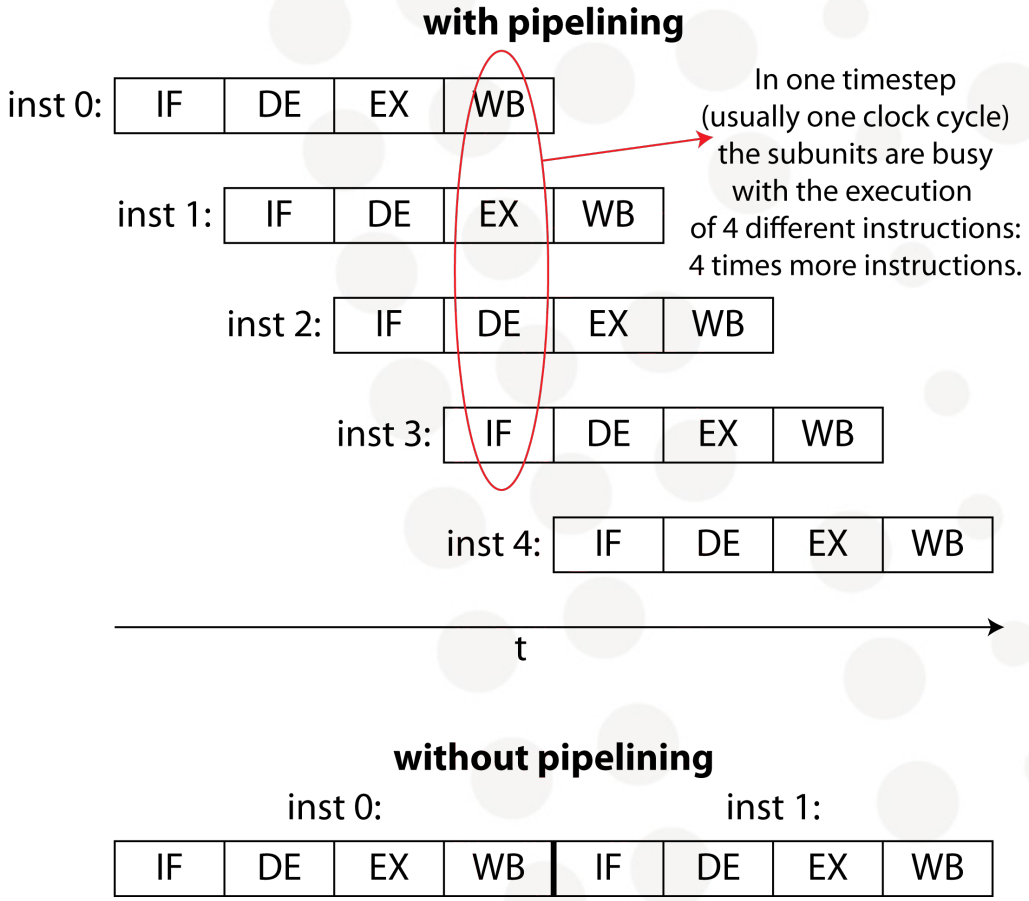
A write operation will lead to a temporary inconsistency between the content of the cache and the main memory. Several strategies are used in different designs to correct this inconsistency with varying delay. Major strategies:

write through : a simple policy where each write to the cache is followed by a write to the main memory. Thus, the write operations do not really profit from the cache.

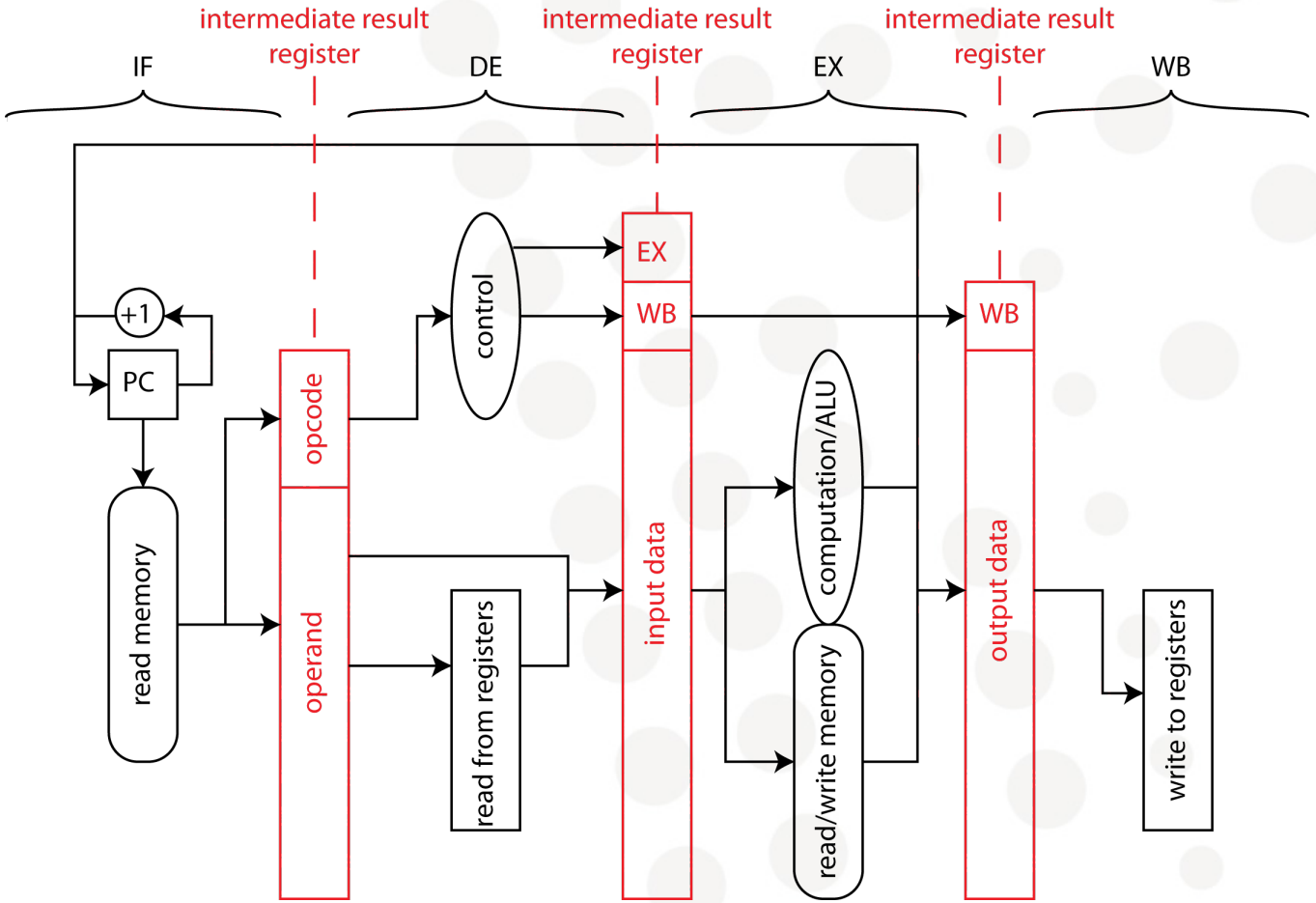
write back : delayed write back where a block that has been written to in the cache is marked as *dirty*. Only when dirty blocks are reused for another memory block will they be written back into the main memory.

content

Repetition 4-Stage Pipeline



4-Stage Pipeline Simplified Block Diagram



Pipelines with More Stages

The 4-stage pipeline is the shortest pipeline that has been used for CPU design and modern processors use generally more stages. The Pentium III had 16 and the Pentium 4 has 31 stages.

Effective Speed-Up (1/2)

The *speed-up* is the ratio of the time T needed to execute a specific program for pipelined and non-pipelined execution. The maximal speed-up of a 4-stage pipeline is not exactly a factor 4, since the pipeline first needs to be 'filled up', before it finishes 4 times more instructions than a purely sequential execution. For example, if a program contains only one single instruction the pipelined architecture is obviously no faster at all.

Effective Speed-Up (1/2)

In a k-stage pipeline requiring one clock cycle per stage the execution of n instructions with a clock cycle time t will be finished in:

$$T = (k + (n - 1))\, t \tag{1}$$

The speed-up is, thus:

$$\frac{knt}{(k + (n - 1))\, t} = \frac{kn}{k + n - 1} \tag{2}$$

It may approach k for very long programs according to this formula. Unfortunately there are other reasons why it never quite gets there: *pipelining hazards*

Pipelining Hazards

Other causes that limit the pipelining speed-up are called pipelining hazards. There are three major classes of these hazards:

- resource hazards
- data hazards
- control hazards

Hazards can be decimated by clever program compilation. In the following however, we will look at hardware solutions. In practice both are used in combination.

content

Resource Hazard Example: Memory Access

We have earlier referred to the von Neumann bottle neck as the limitation to only one memory access at a time. For pipelined operation, this means that only one instruction in the pipeline can have memory access at a time. Since always one of the instructions will be in the instruction fetch phase, a load or write operation of data to the memory is not possible without *stalling* the pipeline.

Improvement 1: Register File

To ameliorate the problem of the memory bottle neck, most instructions in pipelined architectures use local registers organized in a *register file* for data input and output. The register file is in effect a small RAM (e.g. with only a 3bit address space) with (commonly) two parallel read ports (addresses and data) and (commonly) one parallel write port. It does, thus, allow three parallel accesses at the same time. In addition it is a specialized very fast memory within the CPU allowing extremely short access times. Still, also registers in the register file can be cause for resource hazards if two instructions want to access the same port in different pipeline stages.

Improvement 2: Separate Data and Instruction Cache

Another improvement is the so called Harvard architecture, different from the von Neumann model insofar as there are two separate memories again for data and instructions, on the level of the cache memory. Thus, the instruction fetch will not collide with data access unless there is a cache miss of both.

About Memory Access

Memory access still constitutes a hazard in pipelining. E.g. in the first 4-stage SPARC processors memory access uses 5 clock cycles for reading and 6 for writing, and thus impede pipe-line speed up.

Other Resource Hazards

Dependent on the CPU architecture a number of resources may be used by different stages of the pipeline and may thus be cause for resource hazards, for example:

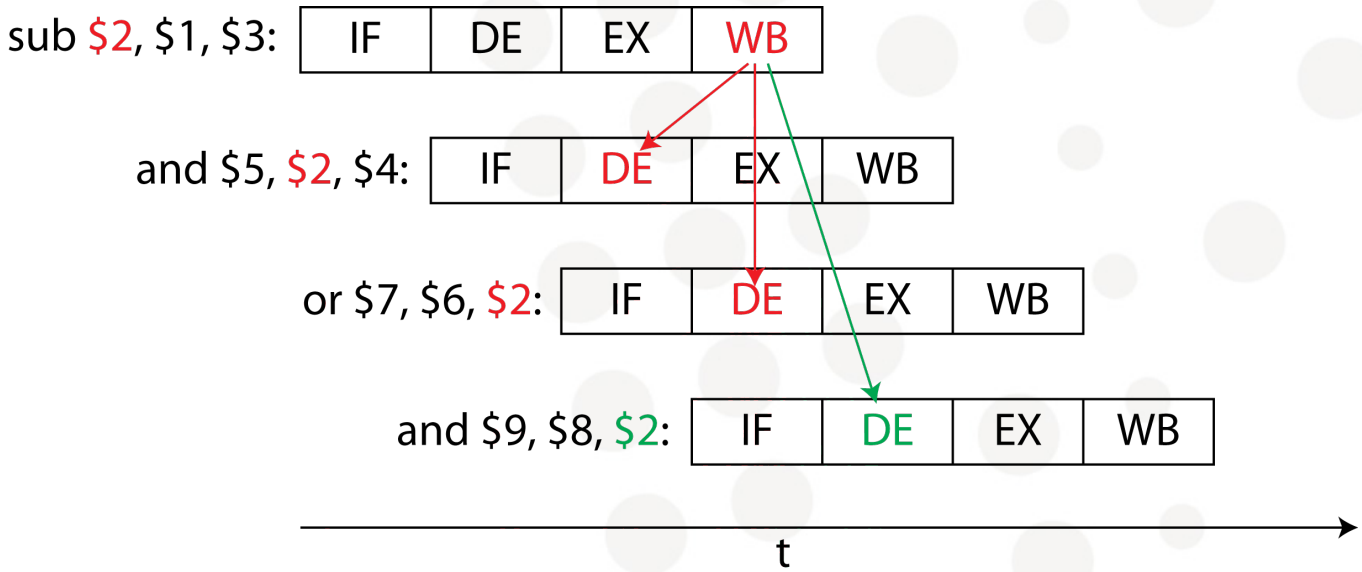
- memory, caches,
- register files
- buses
- ALU
- ...

content

Data Hazards

Data hazards can occur when instructions that are in the pipeline simultaneously access the same data (i.e. register). Thus, it can happen that an instruction reads a register, before a previous instruction has written to it.

Data Hazard Illustration



A Solution: Stalling

A simple solution is to detect a dependency in the IF stage and stall the execution of subsequent instructions until the crucial instruction has finished its WB

Improvement: Shortcuts/Forwarding

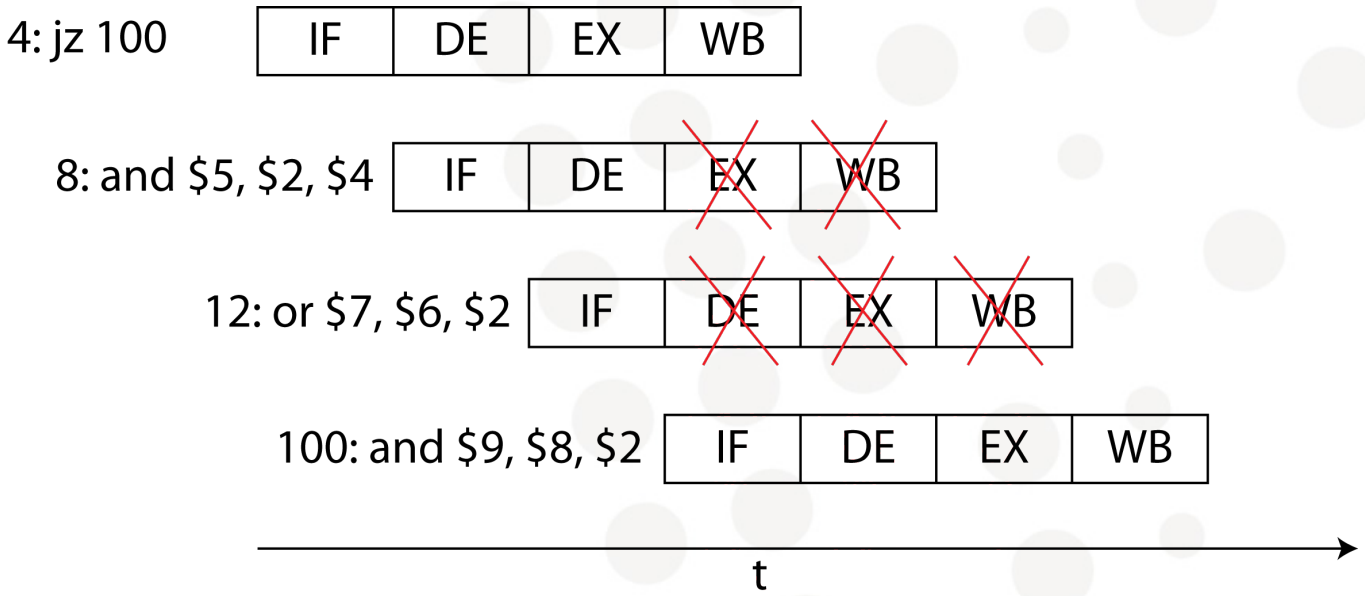
In this solution there is a direct data path from the EX/WB intermediate result register to the execution stage input (e.g. the ALU). If a data hazard is detected this direct data path supersedes the input from the DE/EX intermediate result register.

content

Control Hazards

Pipelining assumes in a first approximation that there are no program jumps and 'pre-fetches' always the next instruction from memory into the pipeline. The target of jump instructions, however, is usually only computed in the EX stage of a jump instruction. A this time, two more instructions have already entered the pipeline and are in the IF and DE stage. If the jump is taken these instructions should not be executed and be prevented from writing their results in the WB stage or accessing the memory in the EX stage.

Control Hazard Illustration



A solution: Always Stall

Simply do not fetch any more instructions until it is clear if the branch is taken or not.

Improvement 1: Jump Prediction

Make a prediction and fetch the predicted instructions.
Only if the prediction is proven wrong, flush the pipeline.
Variants:

- assume branch not taken (also a static prediction)
- static predictions
- dynamic predictions

Improvement 2: Hardware Doubling

By doubling the hardware of some of the pipeline stages one can continue two pipelines in parallel for both possible instruction addresses. After it is clear, if the branch was taken or not, one can discard/flush the irrelevant pipeline and continue with the right one.
Of course, if there are two jumps or more just after each other, this method fails on the second jump and the pipeline needs to stall.

Pipelining Conclusion

Pipelining speeds up the instruction throughput (although the execution of a single instruction is not accelerated). The ideal speed-up cannot be reached, because of this, and because of instruction inter-dependencies that sometimes require that an instruction is finished before another can begin. There are techniques to reduce the occurrence of such hazards, but they can never be avoided entirely.