INF2270 — Spring 2011

Lecture 7: Pipelining, I/O

UNIVERSITETET I OSLO

content

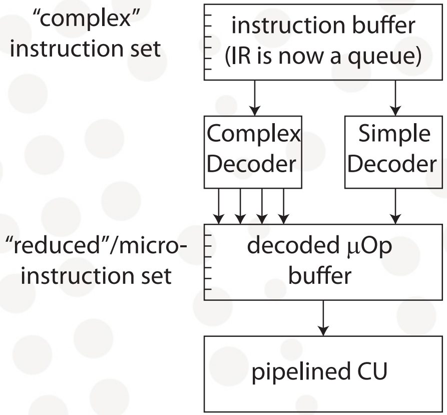
content

Pipelining Concept

CPUs today divide the execution of a single instruction into several uniform steps. The CPU is designed in a way that allows to execute these steps by independent subunits in a single clock cycles. All instructions go through the same steps. The first sub-unit can already fetch a new instruction, while the second sub-unit is still busy with the first, i.e. several instructions are already started before the first completes.
To achieve the *necessary uniformity* of the instructions, the set of instructions is kept small, which used to be known as reduced instruction set computer (RISC) architecture as opposed to complex instruction set computer (CISC).

Modern CPU “Hardware Compiler”

Today, however, the instruction sets tend to be quite complex again. Pipelining is still achieved by pipelining *micro-instructions* (different meaning than in micro-architecture!).

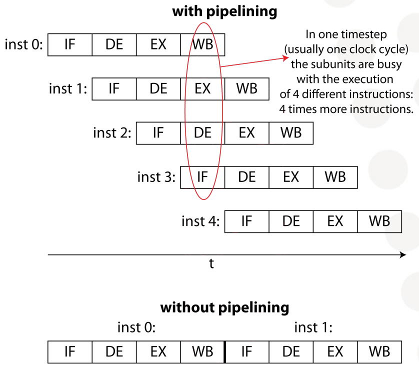

“complex” instruction set
instruction buffer (IR is now a queue)
Complex Decoder
Simple Decoder
“reduced”/micro-instruction set
decoded µOp buffer
pipelined CU

Pipelining Instruction Steps

An example of steps of instruction execution in pipelining are:

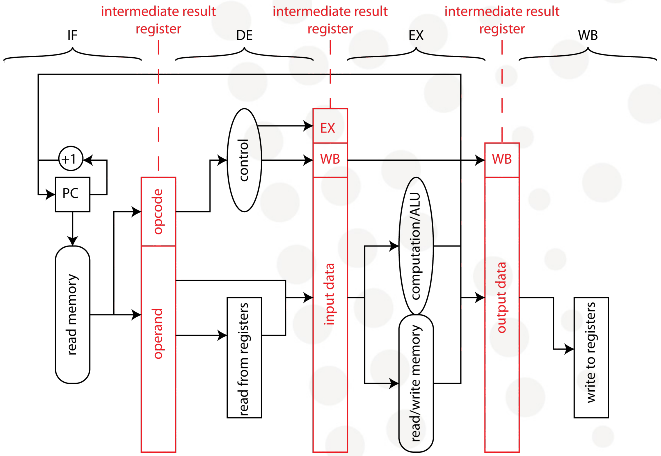
- IF: instruction fetch (get the instruction)
- DE: decode and load (from a register)
- EX: execute
- WB: write back (write the result to a register)

Pipelining Illustration



The pipeline in this example achieves a 4 times bigger instruction throughput. Note, though, that instruction 0 has the same *delay*. Problematic are e.g. jumps (see 'control hazards'): the execution of the pre-fetched instructions is interrupted and the pipeline restarted.

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 had 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

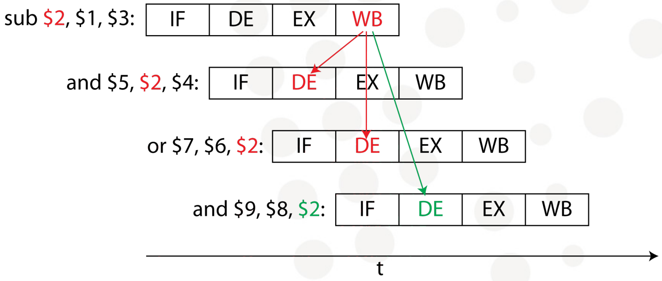
Pipelining
- Resource Hazards
- Data Hazards
- Control Hazards
- Conclusion

Input/Output (I/O)

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



sub $2, $1, $3: IF DE EX WB
and $5, $2, $4: IF DE EX WB
or $7, $6, $2: IF DE EX WB
and $9, $8, $2: IF DE EX WB
t

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

Pipeline

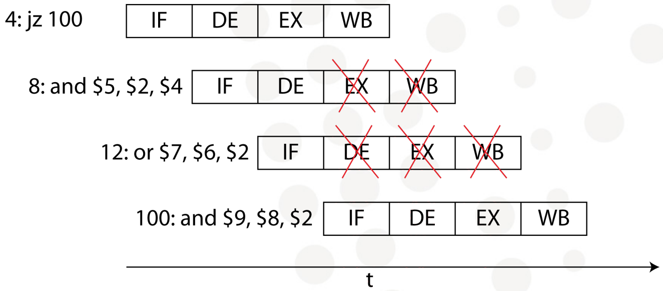
- Resource Hazards
- Data Hazards
- Control Hazards
- Conclusion

Input/Output (I/O)

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.

Test Yourself (Exam 2010)

Riktig eller feil?

- Problemet med “resource hazard”-er en del redusert i en Harvard arkitektur sammenlignet med en von Neumann arkitektur.
- En “pipelined” arkitektur med 4 “pipelining stages” vil eksekvere et program 4 ganger fortere enn en tilsvarende arkitektur uten “pipelining”.
- Uten “jump prediction” vil eksekvering med “pipelining” av et program alltid være like langsom som uten “pipelining”.
- “Resource hazard”-er skjer utelukkende når to instruksjoner i “pipelinen” vil ha tilgang til hovedminnen samtidig.

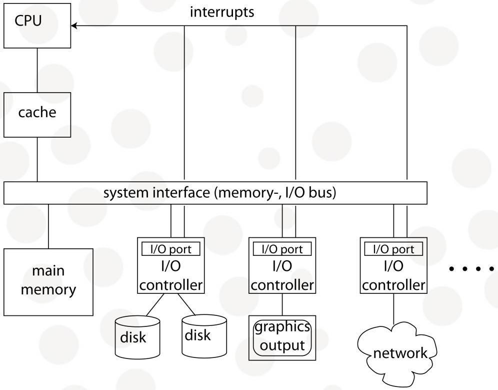
content

Pipelining
- Resource Hazards
- Data Hazards
- Control Hazards
- Conclusion

Input/Output (I/O)

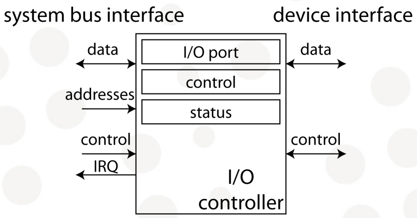
I/O Block Diagram

A computer is connected to various devices transferring data to and from the main memory. This is referred to as Input/output (I/O). Examples: Keyboard, Graphics, Mouse, Network (Ethernet, Bluetooth ...), USB, Firewire, PCI, PCI-express, SATA ...



I/O Controller Principle

An I/O controller translates and synchronizes a peripheral device *protocol* (communication language) for the system bus. It normally has at least one data buffer referred to as *I/O port*, a control register that allows some SW configuration and a status register with information for the CPU.



I/O Addressing (1/2)

Memory mapped I/O is to access I/O ports and I/O control- and status registers (each with its own address) with the same functions as the memory. Thus, in older systems, the system interface might simply have been a single shared I/O and memory bus. A disadvantage is that the use of memory addresses may interfere with the expansion of the main memory.

I/O Addressing (2/2)

Isolated I/O (as in the 80x86 family) means that separate instructions accessing an I/O specific address space are used for I/O access. An advantage can be that these functions can be made *privileged*, i.e. only available in certain modes of operation, e.g. only to the operating system.

Modes of Transfer(1/3)

Programmed/Polled: The processor is in full control of all aspects of the transfer. It *polls* the I/O status register in a loop to check if the controller has data to be collected from the port or is ready to receive data to the port. Polling uses up some CPU time and prevents the CPU from being used for other purposes while waiting for I/O.

Modes of Transfer(2/3)

Interrupt Driven: The I/O controller signals with a dedicated 1bit data line (*interrupt request* (IRQ)) to the CPU that it needs servicing. The CPU is free to run other processes while waiting for I/O. If the interrupt is not *masked* in the corresponding CPU status register, the current instruction cycle is completed, the processor status is saved (PC and flags pushed onto stack) and the PC is loaded with the starting address of an *interrupt handler*. the start address is found, either at a fixed memory location specific to the interrupt priority (*autovectored*) or stored in the controller and received by the CPU after having sent an *interrupt acknowledge* control signal to the device (*vectored*)

Modes of Transfer(3/3)

Direct Memory Access (DMA): The processor is not involved, but the transfer is negotiated directly with the memory, avoiding copying to CPU registers first and the subroutine call to the interrupt handler. DMA is used for maximum speed usually by devices that write whole blocks of data to memory (e.g. disk controllers). The CPU often requests the transfer but then relinquishes control of the system bus to the I/O controller, which only at the completion of the block transfer notifies the CPU with an interrupt.
(DMA poses another challenge to the cache as data can now become *stale*, i.e. invalid in the cache)

Test Yourself (Exam 2010)

Riktig eller feil?

- “Stale” data betyr at data i hovedminnen ikke er oppdatert.
- “Direct memory access” (DMA) er en metode som gir som regel fort transfer av inngangs- og utgangsdata