INF2270 — Spring 2010

Philipp Häfliger

Lecture 5: Von Neuman Architecture

UNIVERSITETET I OSLO

content

Von Neumann Architecture

ALU

RAM
- SRAM
- DRAM

Register Transfer Language

Execution of Instructions
- Pipelining

content

Von Neumann Architecture

ALU

RAM
- SRAM
- DRAM

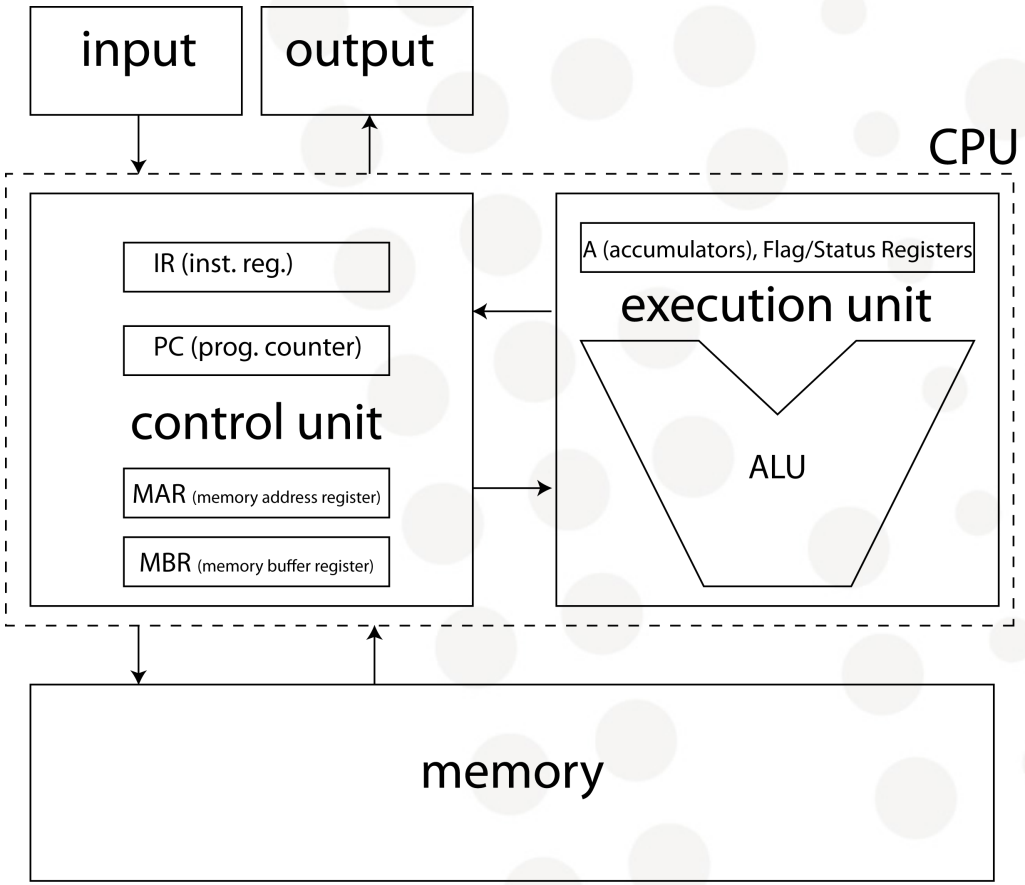
Register Transfer Language

Execution of Instructions
- Pipelining

Von Neumann Architecture

In 1945 John von Neumann published his reference model of a computer architecture that is still the basis of most computer architectures today. The main novelty was that a single memory was used for both, program *and* data.

Von Neumann Architecture Block Diagram



Typical Registers(1/3)

PC: (program counter, also called instruction pointer (IP)) the register holding the memory address of the next machine code instruction.

IR: (instruction register) the register holding the machine code of the instruction that is executed.

Typical Registers(2/3)

MAR: (memory address register) half of the registers dedicated as interface of the memory with the CPU, holding the memory address to be read or written to.

MBR: (memory buffer register) the other half of the CPU-memory interface, a buffer holding the data just read from the memory or to be written to the memory. Typically the MBR can be connected as one of the inputs to the ALU.

Typical Registers(2/3)

accumulator: a dedicated register that stores one operand and the result of the ALU. Several accumulators (or general purpose registers in the CPU) allow for storing of intermediate results, avoiding (costly) memory accesses.

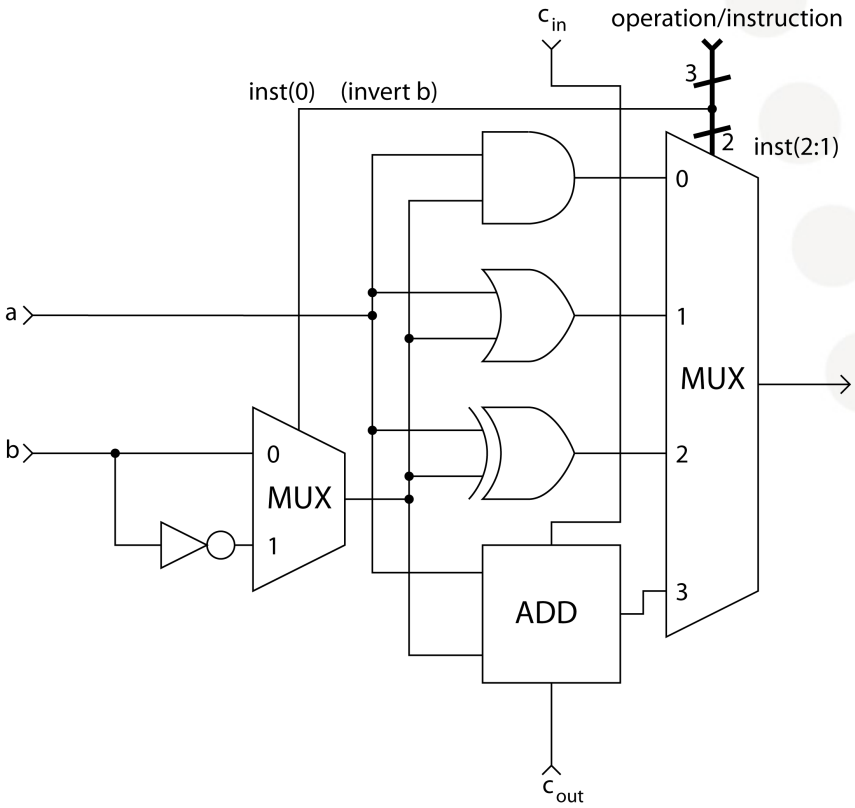
flag/status register: a register where each single bit stands for a specific property of the result from (or the input to) the ALU, like carry in/out, equal to zero, even/uneven ...

Data and Instruction Bus

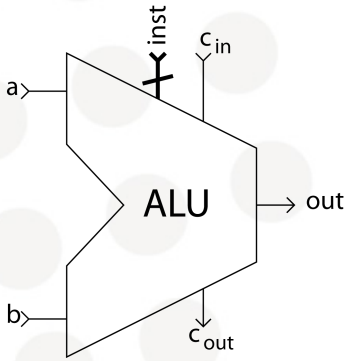
Buses are connections between registers, the functional units (such as the ALU), the memory, and I/O units. They are often shared by several of those units and usually only one unit sends data on the bus to one other at a time. Since there is only one bus between the memory and the CPU for both instruction and data transfer in a von Neumann architecture (actually two: one for the address and one for the data), this bus can be a main speed limiting factor (von Neumann bottleneck). Internally in the CPU there is one or several buses for data exchange among the registers.

content

A 1-bit Arithmetic Logic Unit (ALU) Example

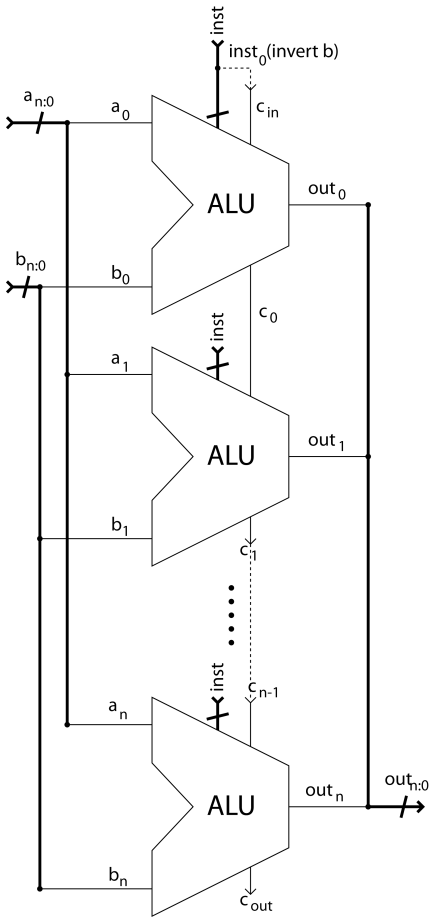


Symbol:

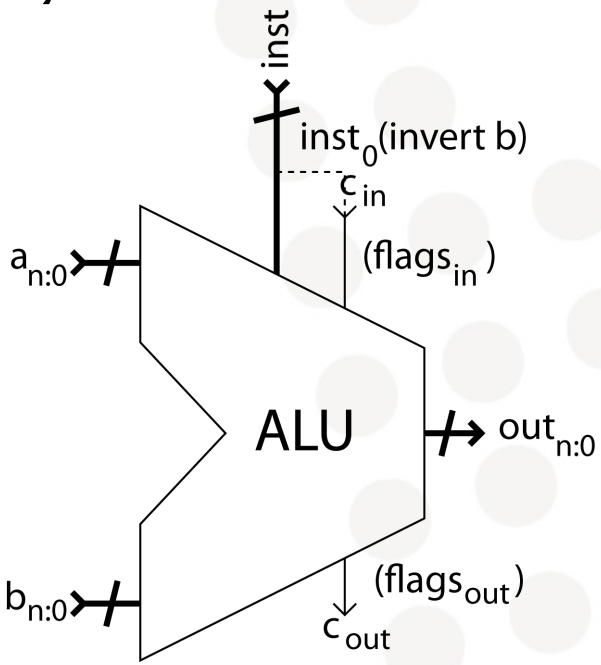


inst	computation
000	$a \wedge b$
001	$a \wedge \overline{b}$
010	$a \vee b$
011	$a \vee \overline{b}$
100	$a \oplus b$
101	$a \oplus \overline{b}$
110	$a + b$
111	$a - b^{(!)}$

n-bit ALU example



Symbol:



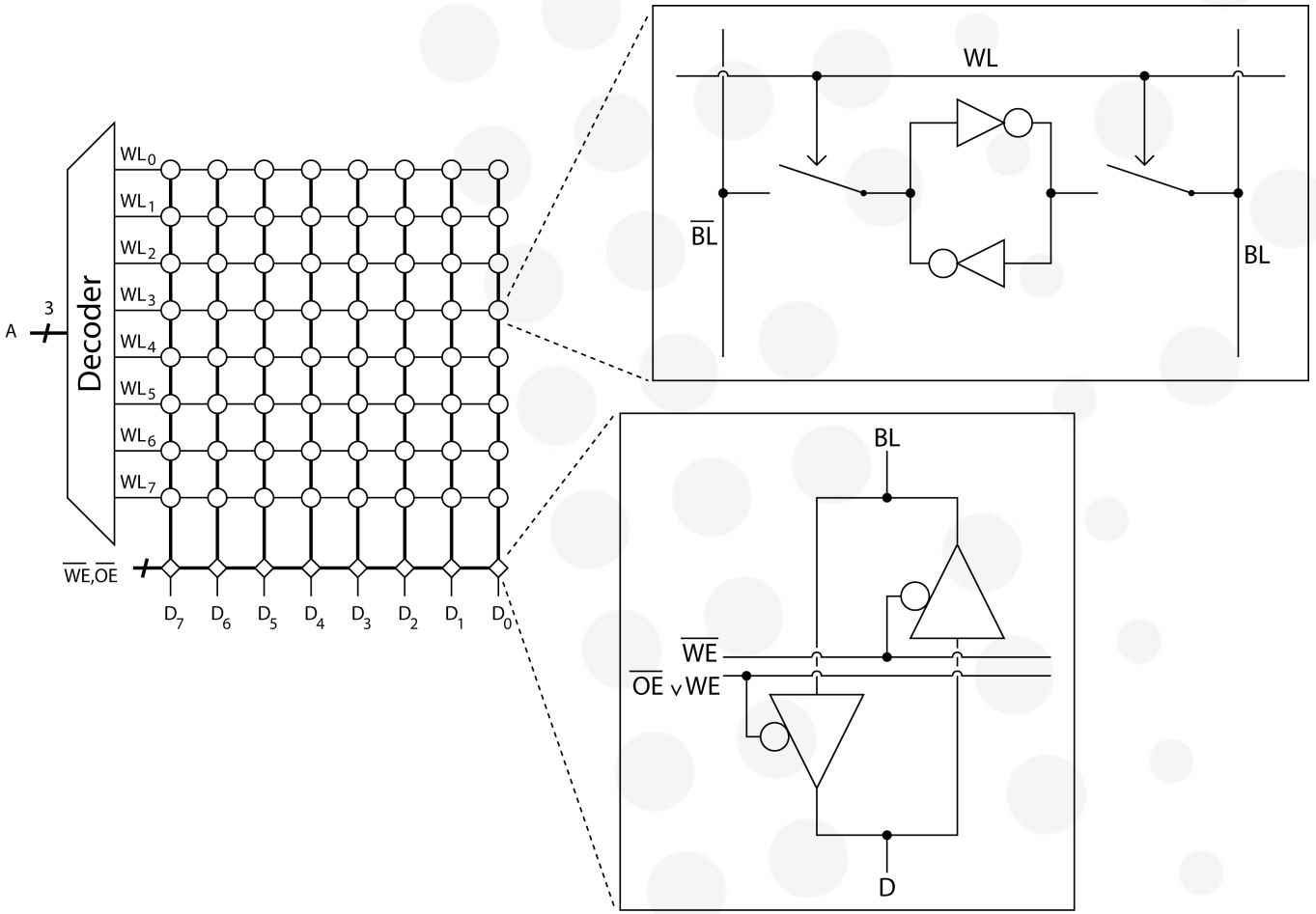
More complicated ALUs will have more flags, e.g. overflow, divide by zero ...

ALUs in CPUs

Modern CPUs contain several ALUs, e.g. one dedicated to memory pointer operations and one for data operations. ALUs can be much more complex and perform many more functions in a single step than the example shown here, but note that even a simple ALU can compute complex operations in several steps, controlled by the software. Thus, there is always a trade-off of where to put the complexity: either in the hardware or in the software. Complex hardware can be expensive in power consumption, chip area and cost. Furthermore, the most complex operation may determine the maximal clock speed. The design of the ALU is a major factor in determining the CPU performance!

content

Static Random Access Memory Principle



RAM terminology

address space: the number of words in a RAM. In the previous example its equivalent to the number of rows.

word length: The number of bits that can be accessed in a single read/write operation, i.e. the number of bits addressed with a single address. In the previous example the number of columns.

memory size: word length multiplied with address space.

Most Typical RAM Signals (1/3)

$\overline{\text{WE}}$, write enable (often active low): This signal distinguishes a read from a write access. If there is no $\overline{\text{RAS}}/\overline{\text{CAS}}$ input to the RAM, $\overline{\text{WE}}$ going low directly causes D to be stored into the RAM at address A.

Most Typical RAM Signals (2/3)

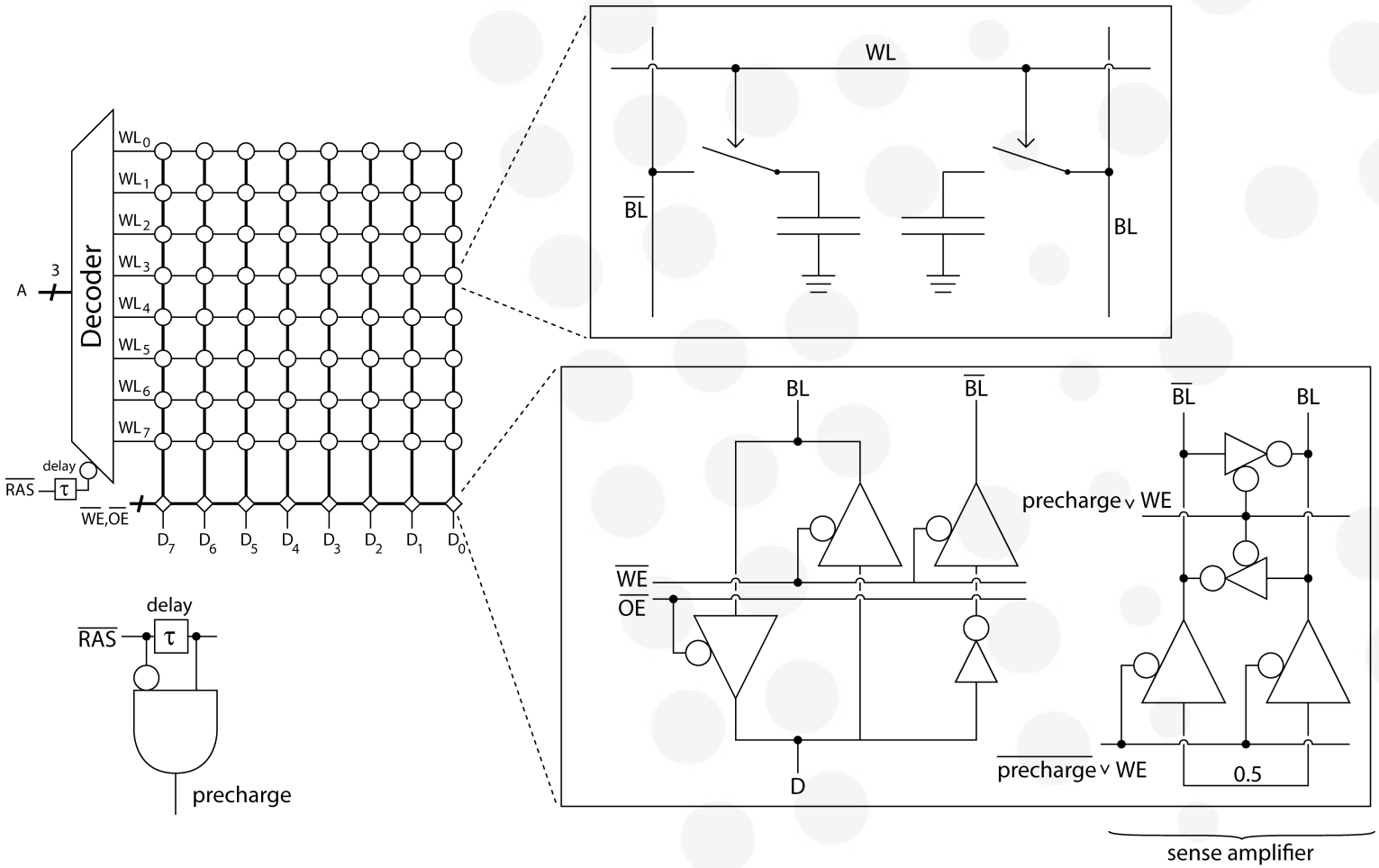
$\overline{\text{RAS}}/\overline{\text{CAS}}$, row/column access strobe: appears in DRAM that actually has a 3-D structure: one decoder for the row address, one for the column address and the word (conceptually) extends into the third dimension. The address bus is reused for both row and column address. First the row address is asserted on the address bus A and $\overline{\text{RAS}}$ is pulsed low, then the column address is asserted on the address bus and $\overline{\text{CAS}}$ is pulsed low. $\overline{\text{CAS}}$ is the final signal that triggers the either read or write. The other signals that are asserted *before*. Several column accesses can be made for a single row access for faster access times.

Most Typical RAM Signals (3/3)

$\overline{\text{OE}}$, output enable: A signal that regulates the access if there are several devices using the bus. Only one of them should be allowed to drive the bus at anyone time.

$\overline{\text{CS}}$, chip select: A control line that allows to use several RAMs instead of just one on the same address bus and sharing all other control signals. If $\overline{\text{CS}}$ is not asserted all other signals are ignored and the output is not enabled. Extra address bits are used to address one specific RAM and a decoder issues the appropriate $\overline{\text{CS}}$ to just one RAM at a time. This extends the address space.

Dynamic Random Access Memory Principle



DRAM refresh

Capacitive storage is not self maintaining like flip flops. Memory content is lost over time. Thus, the sense amplifier has to be connected to every memory cell within a given time period, while the memory is idle. In modern DRAM internal state machines take care of this refresh cycle.

Static vs. Dynamic RAM

	SRAM	DRAM
access speed	+	-
memory density	-	+
no refresh needed	+	-
simple internal control	+	-
price per bit	-	+

content

Register Transfer Language (RTL)

expression	meaning
X	register X or unit X
[X]	the content of X
←	replace/insert or execute code
M()	memory M
[M([X])]	memory content at address [X]

RTL Examples

$[IR] \leftarrow [MBR]$ transfer the content of the MBR to the IR

content

A Simple Example CPU Executing Machine Code

At start-up of the CPU the program counter is initialized to a specific memory address. Here to address 0. The memory content is as follows:

mem adr	content
0	move 4
1	add 5
2	store 6
3	stop
4	1
5	2
6	0
⋮	⋮

Fetch

At the beginning of a instruction cycle a new instruction is fetched from the memory. A finite state machine in the control unit generates the right sequence of control signals. Actually the CU *is* nothing but a finite state machine controlled by the instructions.

$[MAR] \leftarrow [PC]$
$[PC] \leftarrow [PC] + 1$
$[MBR] \leftarrow [M([MAR])]$
$[IR] \leftarrow [MBR]$

Decode

As a last stage of the fetch phase the operation code ('move') of the instruction is decoded by the control unit (CU).

$$\text{CU} \leftarrow [\text{IR}(opcode)]$$

and triggers a cycle of the finite state machine with the appropriate signals to execute a sequence of operations specific to the instruction. The order, type and number of the individual operations may vary among different instructions and the set of instructions is specific to a particular CPU.

Machine Code

The other part of the 'machine code' instruction in our (16-bit) processor is the 'operand' 4. What we have written as 'move 4' is actually a bit pattern:

$$\underbrace{10110010}_{\text{opcode: move}} \quad \underbrace{00000100}_{\text{operand: 4}}$$

As mentioned before, the set of instructions and the machine codes are specific to a CPU. Machine code is *not portable* between different CPUs.

Execute

The data from memory location 4 (1) is moved to the accumulator A (often the accumulator is the implicit target of instructions without being explicitly defined):

$[\text{MAR}] \leftarrow [\text{IR}(operand)]$

$[\text{MBR}] \leftarrow [\text{M}([\text{MAR}])]$

$[\text{A}] \leftarrow [\text{MBR}]$

2nd Instruction (1/2)

A fetch and decode exactly like before:
[MAR] ← [PC] (now [PC]=1)
[PC] ← [PC] + 1
[MBR] ← M([MAR])
[IR] ← [MBR]
The instruction in the IR is now 'add 5'.
CU ← [IR($opcode$)]

2nd Instruction (2/2)

Again the accumulator is the implicit target of the instruction:
$[MAR] \leftarrow [IR(operand)]$
$[MBR] \leftarrow [M([MAR])]$
The ALU receives the appropriate instruction from the state machine triggered by the opcode, or sometimes parts of the opcode *are* the instruction for the ALU.
$ALU \leftarrow [A]; ALU \leftarrow [MBR]$
$[A] \leftarrow ALU$
The number from memory location 5 (2) has been added and the result (3) is stored in the accumulator.

Write Back: 3rd Instruction

Fetch and decode like before (not shown).

⋮

and then a 'write back':

$[\text{MBR}] \leftarrow [\text{A}]$

$[\text{MAR}] \leftarrow [\text{IR}(operand)]$

$[\text{M}([\text{MAR}])] \leftarrow [\text{MBR}]$

4th Instruction

The forth instruction is a stop instruction which halts the execution of the program. The memory content is now changed to:

mem adr	content
0	move 4
1	add 5
2	store 6
3	stop
4	1
5	2
6	3
⋮	⋮

content

Von Neumann Architecture

ALU

RAM
- SRAM
- DRAM

Register Transfer Language

Execution of Instructions
- Pipelining

Pipelining Concept

To accelerate the execution of instructions computer architectures today divide the execution into several steps. The CPU is designed that in a way that allows to execute these steps by independent subunits and such that each step needs the same number of clock cycles for all instructions. Thus, the first step's sub-unit can already fetch a new instruction, while the second step's sub-unit is still busy with the first.

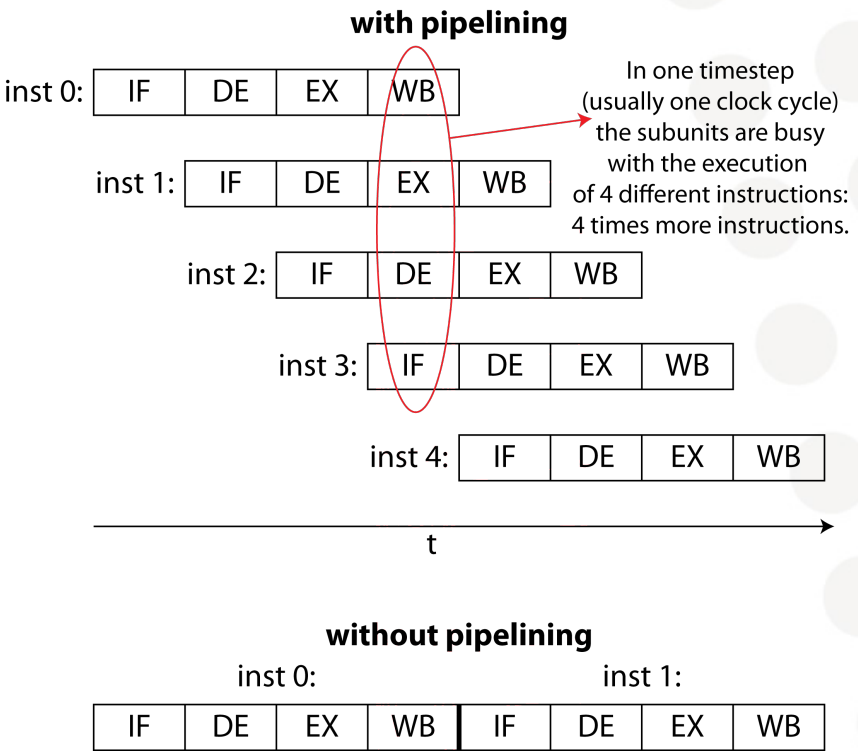
To achieve this, the set of instructions is kept small, which used to be known as reduced instruction set computer (RISC) architecture as opposed to complex instruction set computer (CISC). Today, however, the instruction sets tend to become more complex again, still allowing pipelining.

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*. An 'annoyance' are instructions that cause program counter jumps: then the execution of the pre-fetched instructions is interrupted and the pipeline restarted.