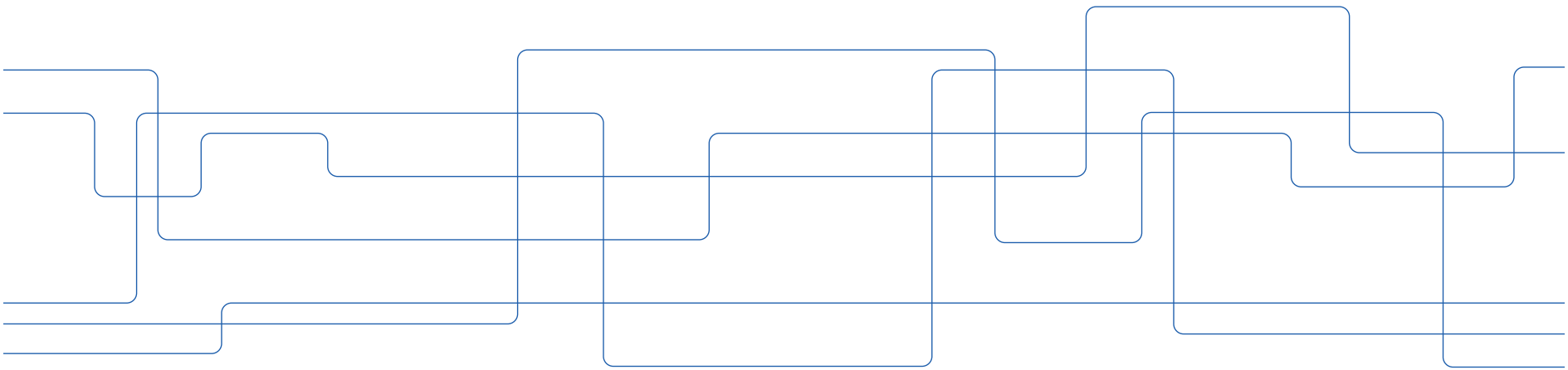
HPC Computer Architectures: Part I

AQTIVATE Training Workshop I

Dirk Pleiter

CST | EECS | KTH

November 2023



Overview

Introduction

Principles of Computer Architectures

Pipelining

Content

Introduction

Principles of Computer Architectures

Pipelining

Computational Science

- **Computational Science** = Multi-disciplinary field that uses advanced computing capabilities to understand and solve complex problems
- Key elements of computational science
  - Application science areas
  - Algorithms and mathematical methods
  - **Computer science**, including areas like
    - Computer architectures
    - Performance modelling

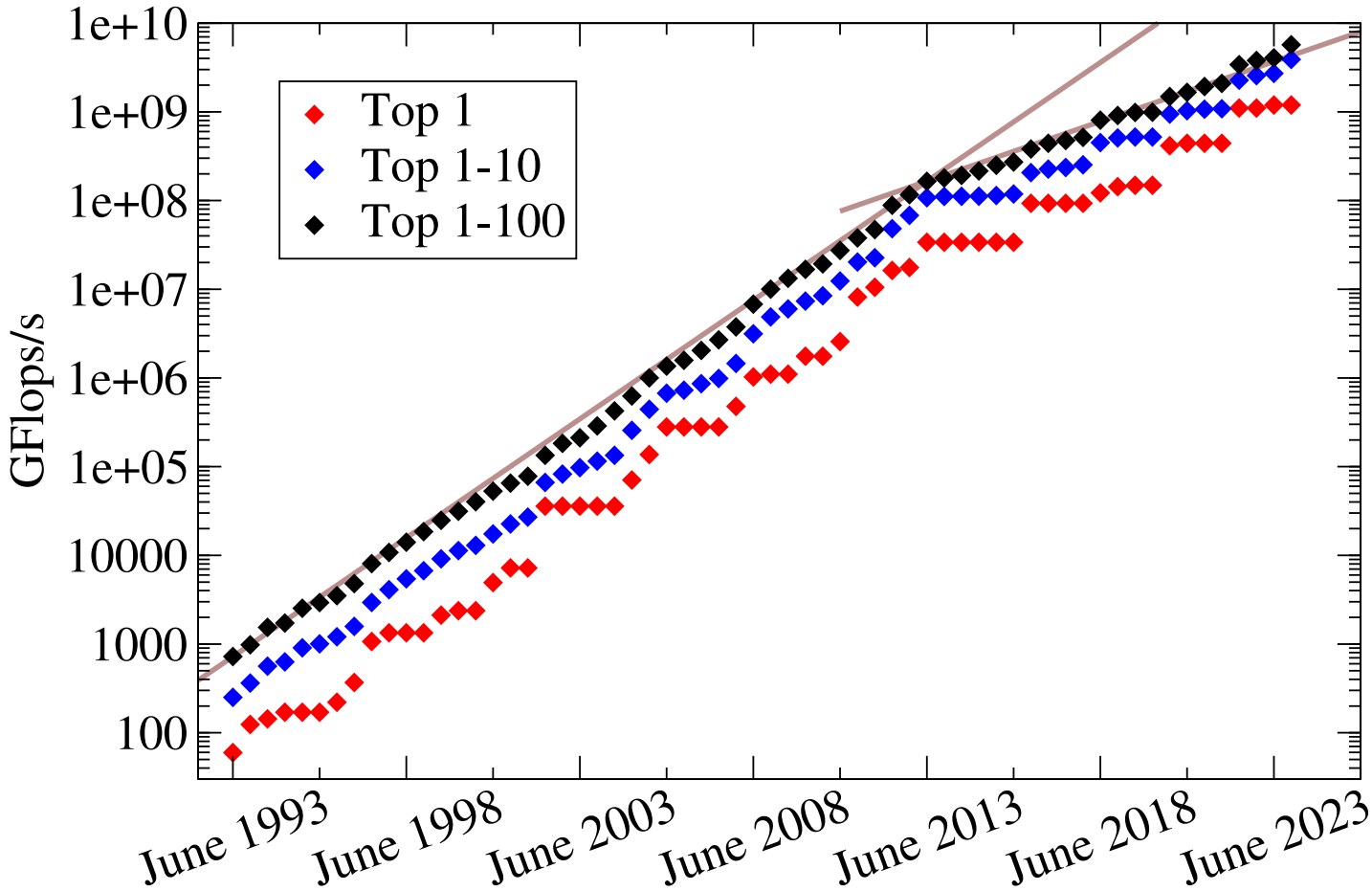
Motivation

- Know-how on computer architectures useful for application developer
  - How much could I optimize my application?
  - Which performance can I expect on a given architecture?
  - Which performance can I expect for a given algorithm?
- Deep knowledge on computer architectures is required to develop application optimized computing infrastructures
  - Which hardware features are required by a given application?
  - Which are the optimal design parameters?

Top500 List

- Performance metric
  - Floating-point operations per time unit while solving a dense linear set of equations
    ☞ High-Performance LINPACK (HPL) benchmark
- Criticism
  - Workload not representative
  - Problem size can be freely tuned
- But: Allows for long-term comparison
  - First list published in June 1993
- New, additional metric being established: HPCG
  - Workload: Sparse linear system solver based on CG

Top500 Trend: Rmax



Peak performance of top 100 systems doubled for many years every 13.5 months, now slowdown to about 26.8 months

Content

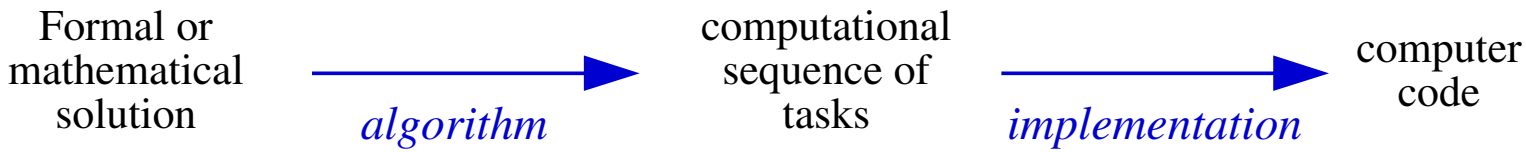
Introduction

Principles of Computer Architectures

Pipelining

Problem Solving Abstraction

- Numerical problem solving abstraction

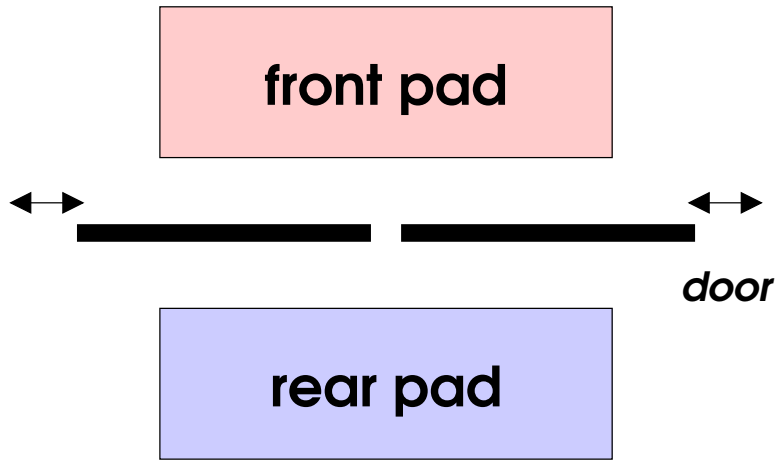


- Execution of computer code = computation
- What is computation?
  - Process of manipulating data according to simple rules
  - Let data be represented by $w \in \{0,1\}^n$

$$w^{(0)} \rightarrow w^{(1)} \rightarrow \cdots \rightarrow w^{(i)} \rightarrow \cdots$$

Computer Model: Deterministic Finite Automata (DFA)

- Informal definition:
  Perform automatic operations on some given sequence of inputs
- Example: Automatic door
- Inputs:
  - Sensor front pad
  - Sensor rear pad
- States:
  - Door open
  - Door closed



Computer Model: DFA (cont.)

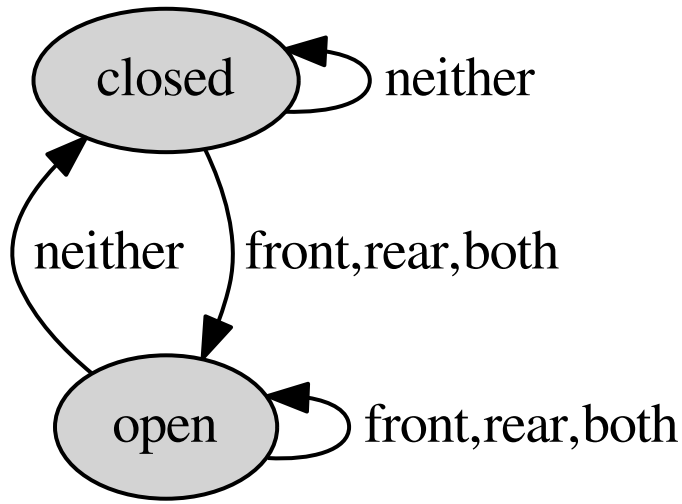
A Deterministic Finite Automaton (DFA) is defined by

- A finite set of **states** Q
  - Example: $Q = \{\text{door open, door closed}\}$
- An **alphabet** Γ
  - An alphabet is a finite set of letters
  - Example: $\Gamma = \{\text{front, rear, both, neither}\}$
- A **transition function** $\delta : Q \times \Gamma \to Q$
- A **start state** $q_0 \in Q$
- A set of **final states** $F \subseteq Q$
  - Allows to define when a computation is successful

Computer Model: DFA (cont.)

Deterministic Finite Automaton (DFA) for automatic door:

- $Q = \{\text{door open, door closed}\}$
- $\Gamma = \{\text{front, rear, both, neither}\}$
- Transistions:
  - $\{\text{closed,front}\} \rightarrow \text{open}$
  - $\{\text{open,neither}\} \rightarrow \text{closed}$
  - ...
- $q_0 = \text{closed}$
- $F = Q$



Graphical representation: **Finite State Machine (FSM)** diagram

Computer Model: Turing Machine

Turing Machine components: [Turing, 1936]

- **Tape** divided into cells
  - Each cell contains a symbol from some finite alphabet, which includes the special symbol "blank"
  - The tape is assumed to infinite in any direction
- **Head** that can
  - Read and write symbols on the tape, and
  - Move tape left and right one cell at a time
- **State register** that stores the state of the Turing Machine
  - The number of states is finite
  - There is a special start state
- **Table** including instructions
  - The number of instructions is finite
  - Chosen instruction depends on current state and input

Computer Model: Turing Machine (cont.)

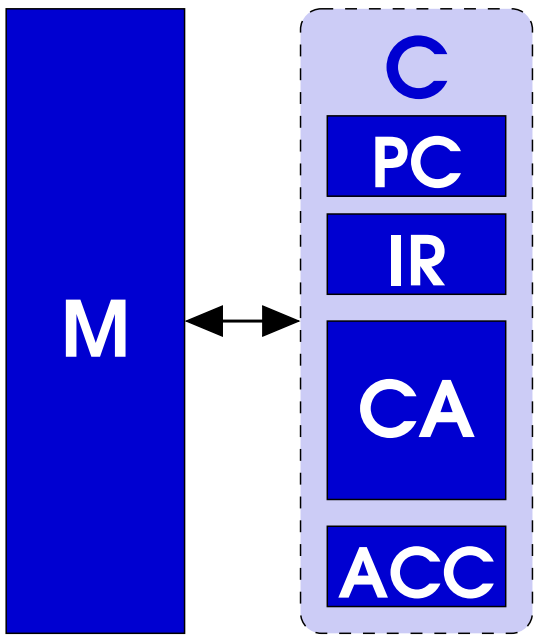
- Notation 1011 q_3 0101
  - Left-hand side tape is 1011
  - Current state is q_3
  - Head is right to state symbol, i.e. reading 0
  - Right-hand side of tape is 0101
- Transition $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$
  - Γ is the *tape alphabet*
- Example state table for $Q = \{\text{A,B,C,HALT}\}$:

Γ	$q = \text{A}$			$q = \text{B}$			$q = \text{C}$		
	Write	Move	State	Write	Move	State	Write	Move	State
0	1	R	B	1	L	A	1	L	B
1	1	L	C	1	R	B	1	R	HALT

- Differences compared to DFA
  - A Turing Machine performs write operations
  - The read-write head can move both to the left and right

Computer Model: Random-Access Machine (RAM)

- Components
  - CPU
    - `PC`: Program counter
    - `IR`: Instruction register
    - `ACC`: Accumulator
  - Randomly addressable memory
- Processing sequence
  - Load `MEM[PC]` to `IR`
  - Increment `PC`
  - Execute instruction in `IR`
  - Repeat



Computer Model: RAM (cont.)

Example set of instructions:

Instruction	Meaning	
`HALT`	Finish computations	
`LD <a>`	Load value from memory	`ACC` $\leftarrow$ `MEM[a]`
`LDI <i>`	Load immediate value	`ACC` $\leftarrow$ `i`
`ST <a>`	Store value	`MEM[a]` $\leftarrow$ `ACC`
`ADD <a>`	Add value	`ACC` $\leftarrow$ `ACC+MEM[a]`
`ADDI <i>`	Add immediate value	`ACC` $\leftarrow$ `ACC+i`
`JUMP <i>`	Update program counter	`PC` $\leftarrow$ `i`
`JZERO <i>`	Update PC if `ACC==0`	`PC` $\leftarrow$ `i`

Computer Model: RAM (cont.)

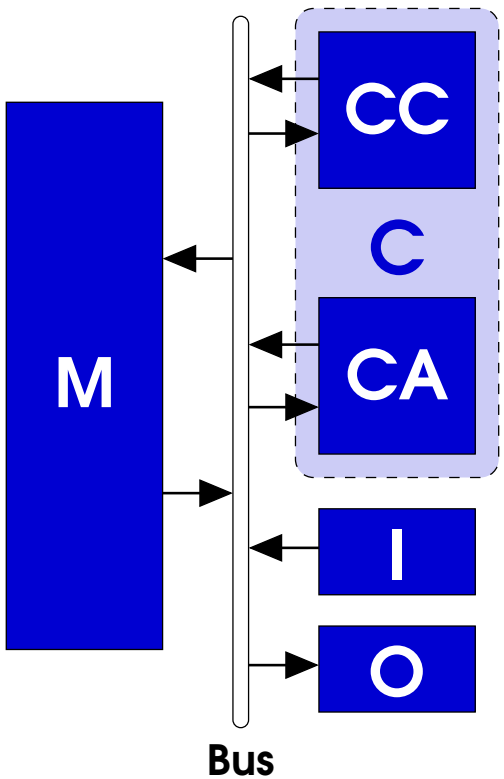
Example: Add numbers stored in address range `[0x100, 0x102]`

PC	Instruction	Meaning
`0x00`	`LD 0x100`	Initiate accumulator
`0x01`	`ADD 0x101`	Add second number
`0x02`	`ADD 0x102`	Add third number
`0x03`	`ST 0x103`	Store result

Benefits and limitations of RAM architecture:

- Memory is randomly addressible
- No direct support of memory address indirection
  - Work-around: Modify set of instructions during execution

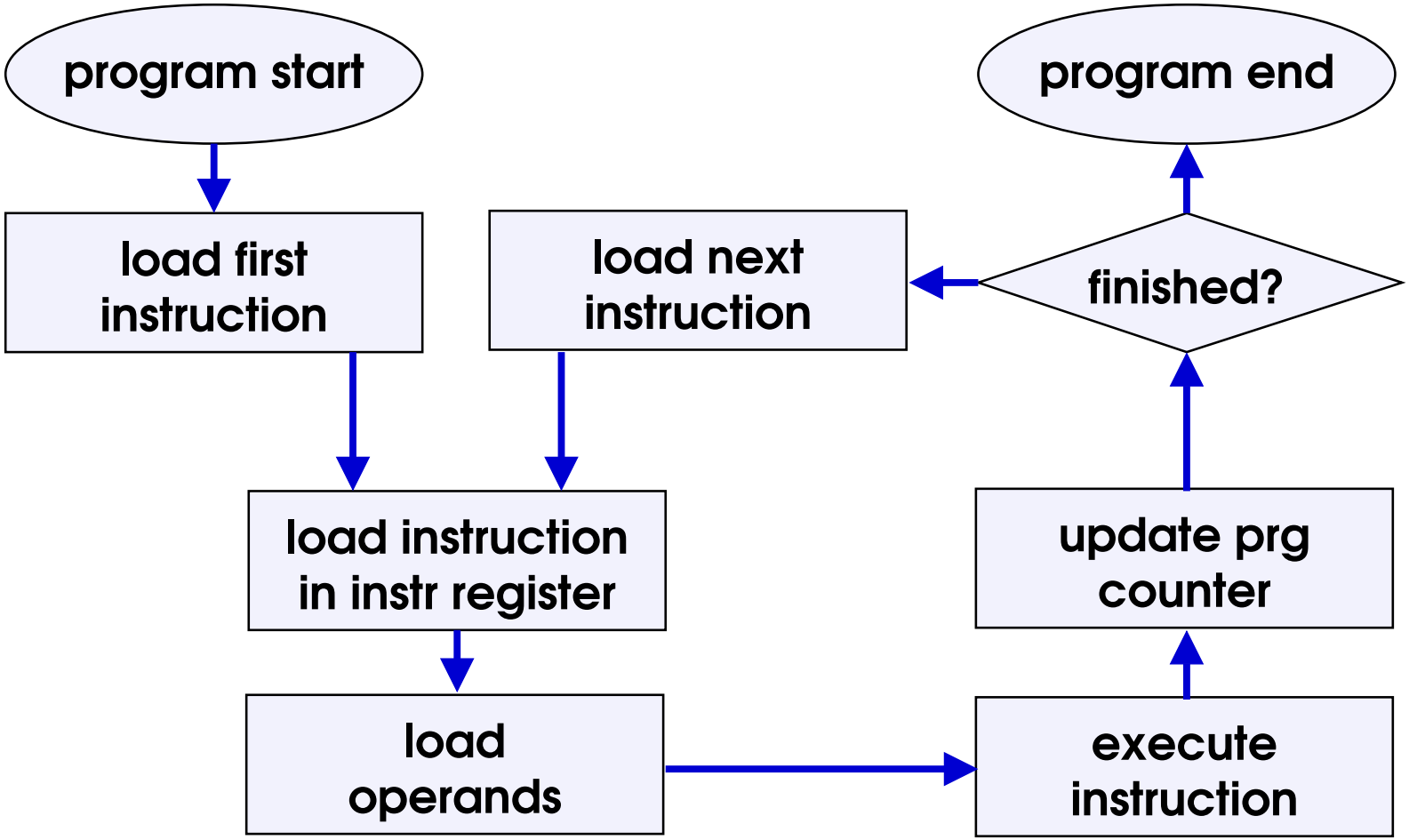
Von Neumann Architecture



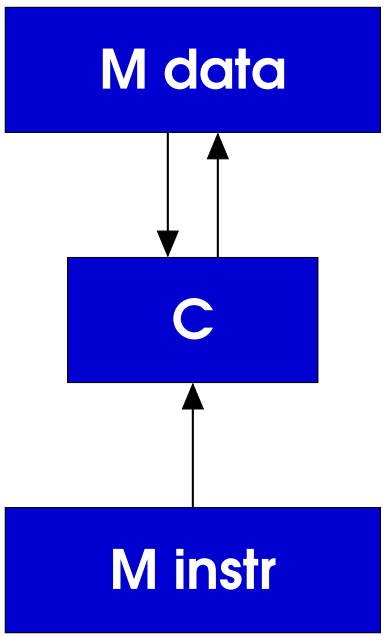
[Neumann, 1945]

- Components defined by v. Neumann:
  - Central arithmetic CA
  - Central control CC
  - Memory M
  - Input I
  - Output O
- Simplified modern view:
  - Central Processing Unit
  - Memory
  - I/O chipset
  - Bus
- Memory used for instructions and data ☞ Self-modifying code

V. Neumann Architecture: Instruction Processing

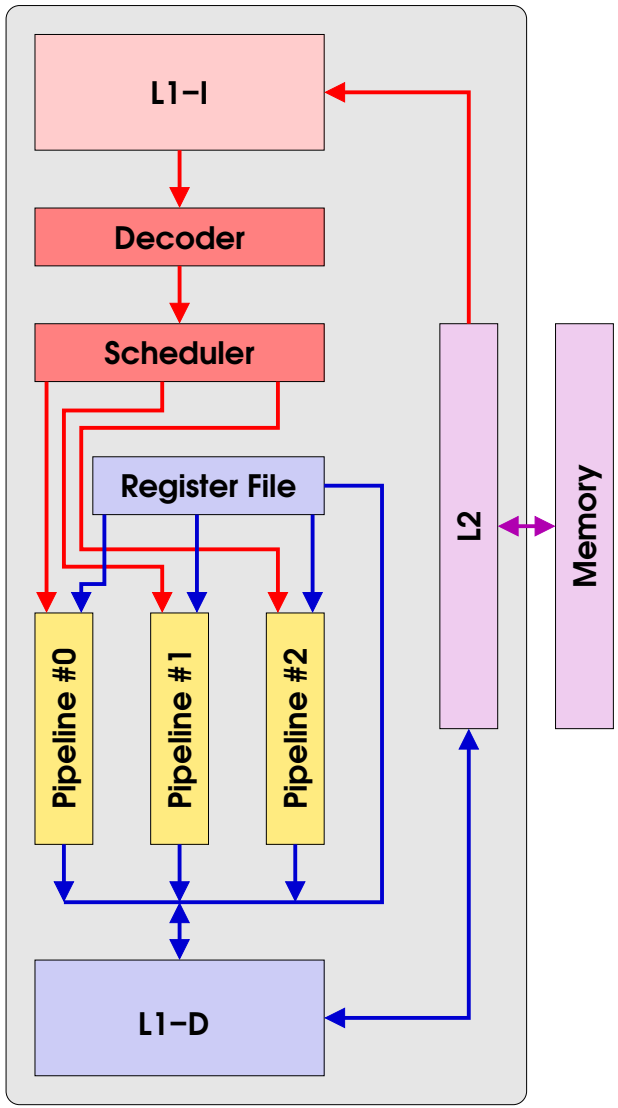


Harvard Architecture



- "Von Neumann bottleneck"
  - Only single path for loading instructions and data
    → Bad processor utilization
- General purpose memory replaced by
  - Data memory
  - Instruction memory
- Independend data pathes
  ☞ Concurrent access
- Different address spaces

Simple Processor Architecture



- Components
  - Instruction decoder and scheduler
  - Execution pipelines
  - Register file
  - Caches (L1-I, L1-D, L2)
  - External memory
- Memory architecture
  - From outside: von Neumann
    - General purpose memory
    - Common L2 cache
  - From inside: Harvard
    - Data L1 cache (L1-D)
    - Instruction L1 cache (L1-I)

Model

- **Model** = Simplification of another entity
  - Should contain the characteristics and properties of the modeled entity
  - It is expected to cover only those characteristics that are relevant to a particular task
- **Motivation for using models**
  - Performance modeling
  - Functional modeling and specification
  - Validation and verification
  - ...
- **Types of models**
  - Mathematical models
  - Event simulation
  - Signal level simulations (in soft- or hardware)
  - ...

Performance

- Reliable measure of **performance** $= \frac{\text{amount of work}}{\text{execution time}}$
- Execution time = time to execute a particular amount of work
- Ambigiouties in definition of execution time:
  - Wall-clock time = latency to complete task
  - Resource occupation time, e.g. CPU time

Architecture Performance Model

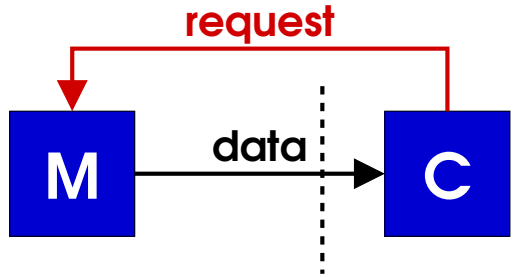
- Model of a computer architecture which allows to make performance predictions
- Cycle accurate, full system simulation not possible in practice
- Restrict model to performance relevant characteristics of the architecture
  - Model relevant system components only, e.g.:
    - CPU and Memory, no I/O sub-system
- Different levels of abstractions, e.g.:
  - Simplified data transport model

Events

- An event refers to a point in time where something happens
  - An event has no duration
- Examples:
  - Instruction is scheduled
  - Change of a signal value (e.g. indicating arrival of data)
  - Arrival of a message
  - Change of a state
  - A counter reaches/exceeds a given value

Bandwidth or Throughput vs. Latency

- Definition **bandwidth or throughput**
  - Bandwidth = amount of items passing a particular interface per time unit
  - Throughput = amount of work done in a given time
- Definition **latency (or response time)** Δt = Time between events indicating start of a task/operation and the event signaling its completion
- Example: Memory read operation



  - Bandwidth: Amount of data passing memory interface per time unit
  - Latency: Time from starting memory read operation until last data item arrived in register file

Content

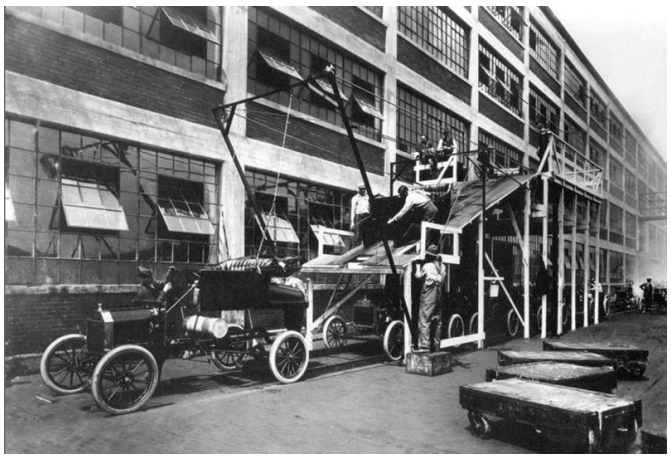
Introduction

Principles of Computer Architectures

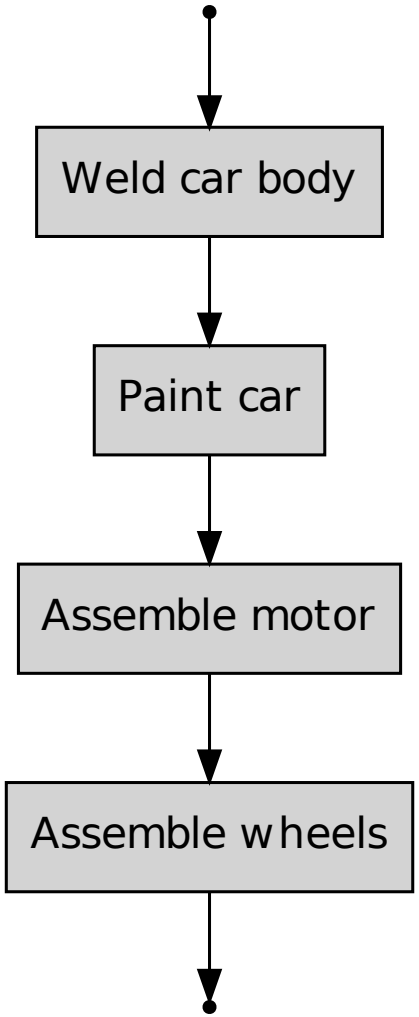
Pipelining

Pipelining

Optimize work using an assembly line:



Works only well if time needed at all stations is similar



Pipeline: Time Diagram

	P_0	P_1	P_2	P_3	P_4
t_0	C_0				
$t_0 + 1$	C_1	C_0			
$t_0 + 2$		C_1	C_0		
$t_0 + 3$			C_1	C_0	
$t_0 + 4$				C_1	C_0
$t_0 + 5$					C_1
$t_0 + 6$					

Without pipeline: Need 5 time units to finish 1 car
With pipeline: Need 6 time units to finish 2 cars

Pipeline: Time Diagram (cont.)

	P_0	P_1	P_2	P_3	P_4
t_0	C_0				
t_0+1	C_1	C_0			
t_0+2	C_2	C_1	C_0		
t_0+3	C_3	C_2	C_1	C_0	
t_0+4	C_4	C_3	C_2	C_1	C_0
t_0+5	C_5	C_4	C_3	C_2	C_1
t_0+6		C_5	C_4	C_3	C_2
t_0+7			C_5	C_4	C_3
t_0+8				C_5	C_4
t_0+9					C_5

- **pipeline filling**: $t = t_0 ... (t_0 + 3)$
- **pipeline draining**: $t = (t_0 + 6) ... (t_0 + 9)$
- Time to perform n operations in k **pipeline stages** assuming a nominal throughput $B = 1$:

$$\Delta t_p(n, k) = n + (k - 1)$$

Here: $\Delta t_p(6, 5) = 10$

Pipeline Costs

- In our example: throughput $B = 1$
- Throughput:

$$b(n,k) = \frac{\text{number of operations}}{\Delta t_p(n,k)} = \frac{n}{n+(k-1)}$$

  Note: $\lim_{n\to\infty} b(n,k) = B$
- Gain (or speed-up)

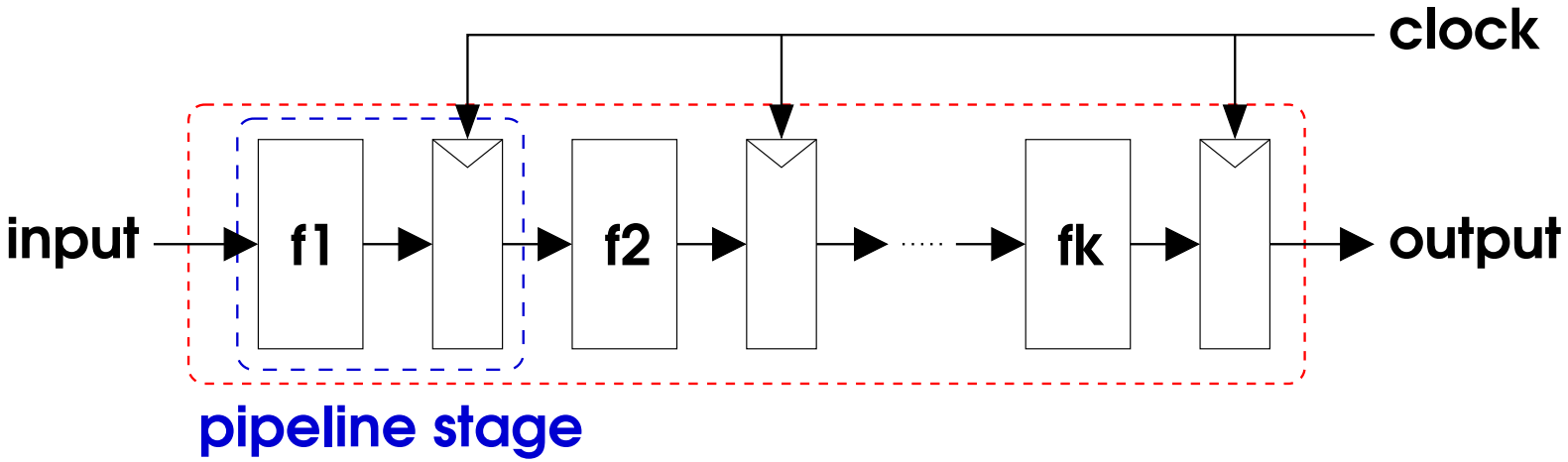
$$s(n,k) = \frac{\text{scalar execution time}}{\text{pipelined execution time}} = \frac{n\,k}{n+(k-1)}$$

  Note: $\lim_{n\to\infty} s(n,k) = k$
- Efficiency

$$\epsilon(n,k) = \frac{1}{k}s(n,k) = \frac{n}{n+(k-1)}$$

Pipelined Design

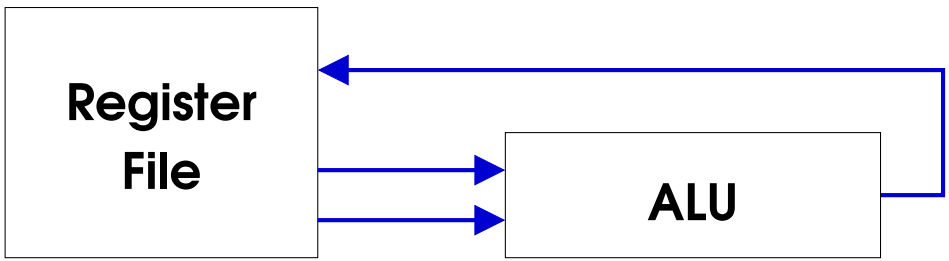
Digital pipeline with k stages:



- Each pipeline stage consists of a
  - Functional/combinatorial step
  - Register
- Execution of a pipeline stage takes a fixed time unit
  - Pipeline synchronized by a **clock**
  - Synchronous vs. asynchronous pipelines

Pipelined Architecture Example

- Consider the following simple architecture:



- Components:
  - **Register file** = collection of **registers** with 2 **output ports** and 1 **input port**
  - Pipelined arithmetic and logic unit **ALU**
- Instructions are executed **in order**
- Performance parameters of arithmetic unit:
  - Throughput $B = 1$
  - Start-up latency $\lambda = 4$ clock cycles

Digression: Pseudo-Assembler

- Pseudo → Not an existing Instruction Set Architecture
- General syntax (to be extended later)

  `<instr> imm <target>`

  `<instr> <src_1>, [<src_2>, ...] <target>`

  where

`<instr>`	...	instruction
`<target>`	...	destination register
`<imm>`	...	immediate value
`<src_i>`	...	source register

- Examples:

```
mov 0xd, R0       !! Move constant value to R0
mov R0, R1        !! Move content register R0 to R1
iadd R0, R1, R2   !! Integer addition: R2=R0+R1
imul R0, R1, R2   !! Integer multiplication: R2=R0*R1
```

Pipelined Architecture Example (2)

- Consider the following operation: $z = a \times b + c - d$ (a, b, c, d integer values)
- Pseudo-assembler
  - Assume a, b, c, d to be already loaded in register `R0`, `R1`, `R2`, `R3`

```
imul R0, R1, R5 ! store a × b in register 5
iadd R5, R2, R6 ! add c
isub R6, R3, R7 ! subtract d
```

- Instructions are not independent: **hazard** due to data dependence
- **Scheduling of instructions** must be aware of hazards

Data hazards

- Assume that in program order instruction i occurs before instruction j
- The instruction j is **data dependent** on the instruction i if either
  - Instruction i produces a result that may be used by instruction j
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i
- **Read-after-write (RAW) hazard**

  Instruction j tries to read a source before it is updated by instruction i

  Example for wrong scheduling of instructions:

```
imul R0, R1, R5 !
isub R6, R3, R7 ! Register R6 not written, yet
iadd R5, R2, R6 !
```

Data Hazards (2)

Other data hazards can occur in more complicated architectures:

- **Write after write (WAW) hazard**:
  Instruction j tries to write to a destination before it is written by instruction i
  ☞ Result of instruction i remains in destination
  - Cannot occur in our simple architecture with single pipeline of fixed depth
- **Write after read (WAR) hazard**:
  Instruction j tries to write a destination before it is read by i
  ☞ Instruction i will read already updated value
  - Cannot occur in our simple architecture with single pipeline of fixed depth

Scheduling

- To resolve data hazards the processor may have to **stall**
- Assume the following hardware parameters:
  $\beta_{\text{int}} = 1$, $\lambda_{\text{int}} = 4\,\text{clock cycles}$
- Consider example: $z = a \times b + c - d$

1	`imul R0, R1` **R5**	store $a \times b$ in register 5
2		stall
3		stall
4		stall
5	`iadd` **R5**`, R2,` **R6**	add c
6-8	...	stall
9	`isub` **R6**`, R3, R7`	subtract d

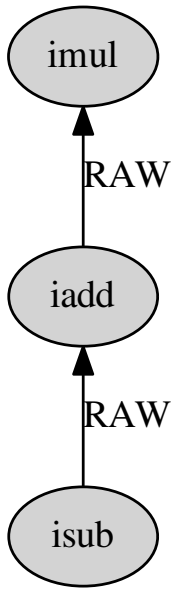
- Latency of operation: $3 \cdot 4$ clock cycles

Data Hazards and Optimization

- Consider the following example:

$$z = a \times b + c - d$$

- Data dependence graph:



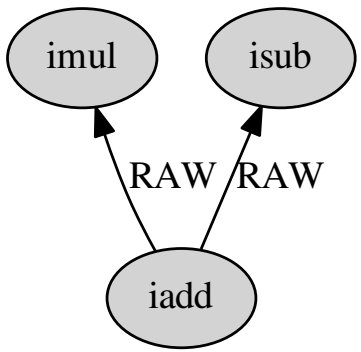
- Latency of operation: $3 \cdot \Delta t_p(1,4) = 12$ clock cycles

Data Hazards and Optimization (cont.)

- Rewrite the example as follows:

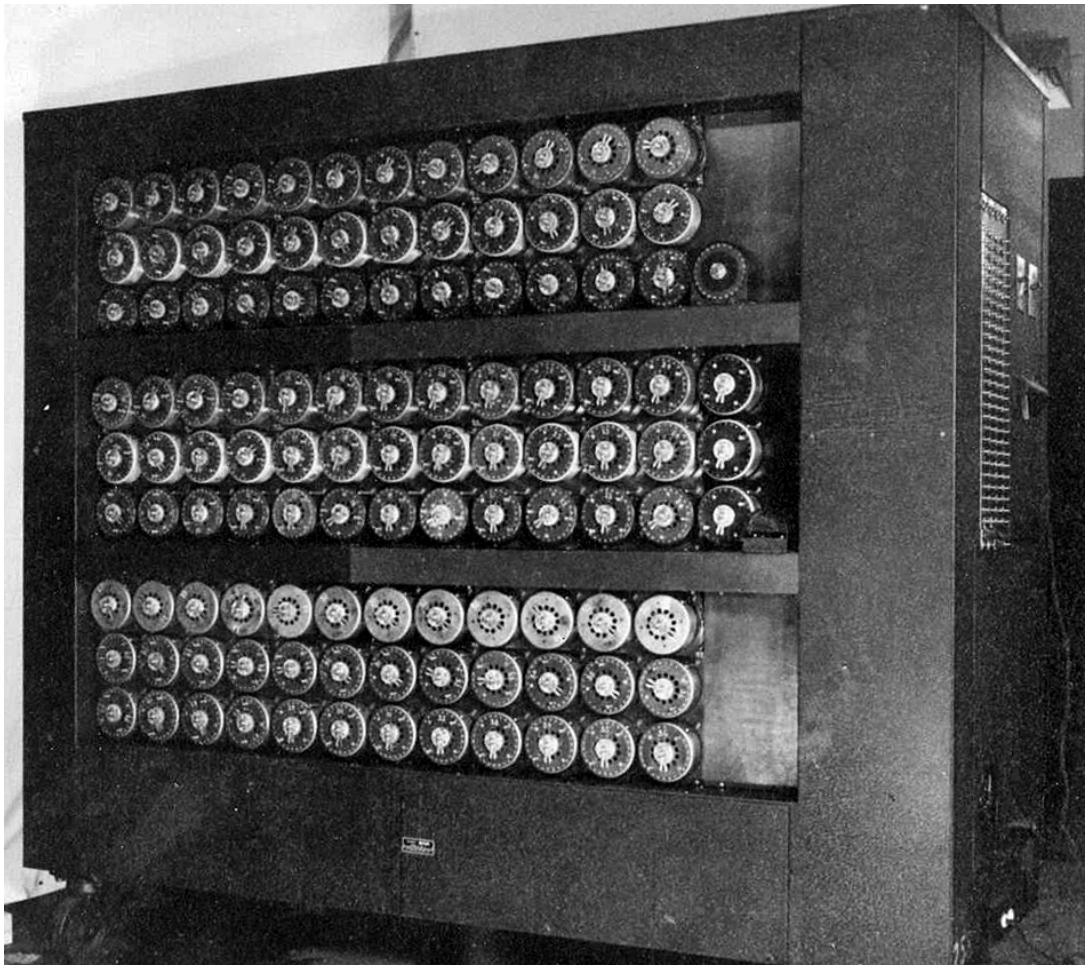
$$\begin{aligned} x &= a \times b \\ y &= c - d \\ z &= x + y \end{aligned}$$

- Data dependence graph:



- Latency:
  $\Delta t_p(2,4) + \Delta t_p(1,4) = (5+4)\,\text{clock cycles} = 9\,\text{clock cycles}$

Finish with an Architecture from Turing: Bombe



[United Kingdom Government, 1945]