

Computer Architectures



D. Pleiter

Jülich Supercomputing Centre and University of Regensburg

January 2016

Overview

Introduction

Principles of Computer Architectures

Processor Core Architecture

Memory Architecture

Processor Architectures

Network Architecture

Exascale Challenges

Content

Introduction

Principles of Computer Architectures

Processor Core Architecture

Memory Architecture

Processor Architectures

Network Architecture

Exascale Challenges

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

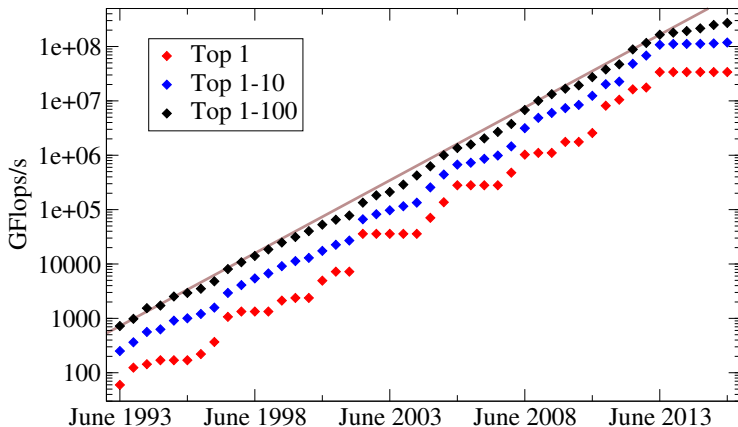
Motivation

- 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 Trends: Performance



Content

Introduction

Principles of Computer Architectures

Processor Core Architecture

Memory Architecture

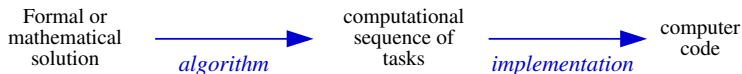
Processor Architectures

Network Architecture

Exascale Challenges

Problem Solving Abstraction

- Numerical problem solving abstraction

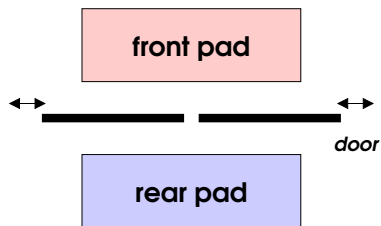


- Execution of computer code = computation
- What is computation?
 - Process of manipulating data according to simple rules
 - Let data be represented by $w \in \{0, 1\}^n$

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

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.)

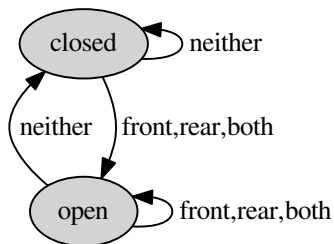
A Deterministic Finite Automaton (DFA) is defined by

- A finite set of **states** Q
 - Example: $Q = \{\text{door open, door closed}\}$
- An **alphabet** Σ
 - An alphabet is a finite set of letters
 - Example: $\Sigma = \{\text{front, rear, both, neither}\}$
- A **transition function** $\delta : Q \times \Sigma \rightarrow Q$
- 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:

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



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.)

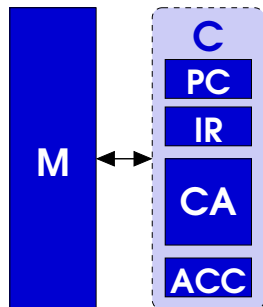
- Notation $1011 q_3 0101$
 - Left-hand side tape is 1011
 - Current state is q_3
 - Head is right to state symbol, i.e. reading 0
 - Right-hand side of tape is 0101
- Transition $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
 - Γ is the *tape alphabet*
- Example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

Γ	$q = A$			$q = B$			$q = C$		
	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 $\text{MEM}[\text{PC}]$ 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]

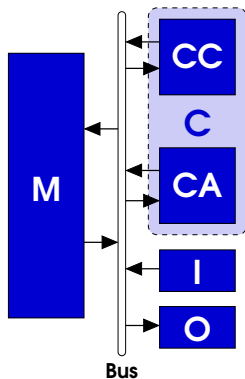
PC	Instruction	Meaning
0x00	LD 0x100	Initiate accumulator
0x01	ADD 0x101	Add second number
0x02	ADD 0x102	Add third number
0x03	ST 0x103	Store result

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

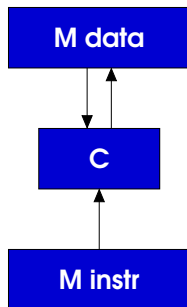
Von Neumann Architecture

[Neumann, 1945]



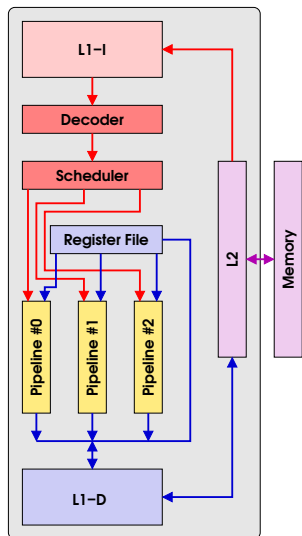
- Components defined by v. Neumann:
 - Central arithmetic CA
 - Central control CC
 - Memory M
 - Input I
 - Output O
- Simplified modern view:
 - Central Processing Unit
 - Memory
 - I/O chipset
 - Bus
- Memory used for instructions and data
 - ☞ Self-modifying code

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
- Independent data pathes
 - ☞ Concurrent access
- Different address spaces

Modern (Simple) Processor Architecture



- Components

- Instruction decoder and scheduler
- Execution pipelines
- Register file
- Caches (L1-I, L1-D, L2)
- External memory

- Memory architecture

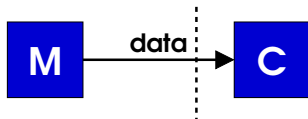
- From outside: von Neumann
 - General purpose memory
 - Common L2 cache
- From inside: Harvard
 - Data L1 cache (L1D)
 - Instruction L1 cache (L1P)

Digression on Terminology

- **Event** = Point of time t_i where something happens
 - An event has no duration
 - Examples
 - Instruction execution completed
 - State changed
 - Counter reaches/exceeds a given value
- **Latency** = Distance $\Delta t = t_j - t_i$ between two events
where $t_i < t_j$
 - Also: response time
 - Examples
 - Time between instruction issue and completion
 - Duration of a task execution

Digression on Terminology (cont.)

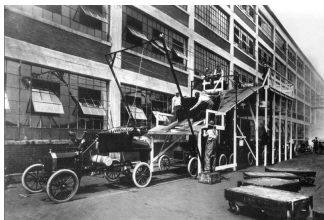
- **Bandwidth** = Amount of items passing a particular interface per unit time



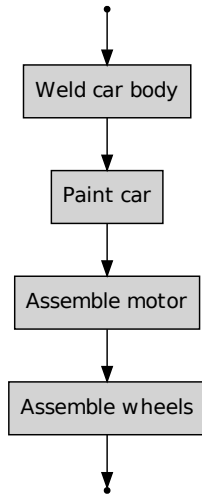
- If transfer of l items has a latency Δt then bandwidth is given by
$$b = \frac{l}{\Delta t}$$
- Equivalent to **throughput** = Amount of work W completed per unit time
- **Nominal bandwidth** = The maximal rate B at which data can be transferred over a data bus

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:**
 $t = t_0 \dots (t_0 + 3)$
- **pipeline draining:**
 $t = (t_0 + 6) \dots (t_0 + 9)$
- 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: $\lim_{n \rightarrow \infty} b(n, k) = B$

- Gain (or speed-up)

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

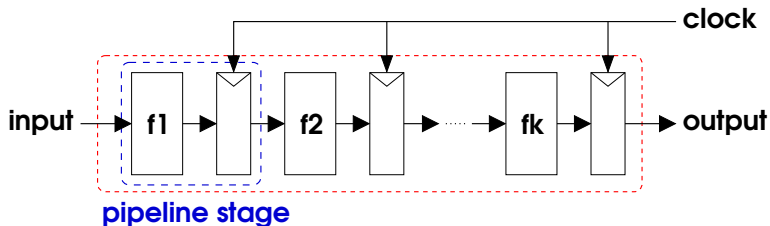
Note: $\lim_{n \rightarrow \infty} s(n, k) = k$

- Efficiency

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

Pipelined Design

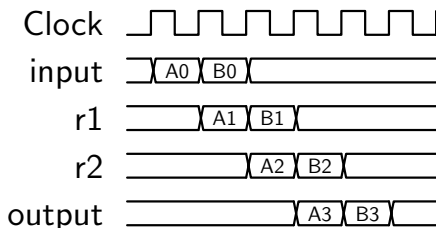
Digital pipeline with k stages:



- 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

Pipeline Design (2)

Waveform representation of pipeline:



- In each combinatorial step values change:
 $A_0 \rightarrow A_1 \rightarrow \dots \rightarrow A_3$
- Content of registers sample at rising clock edge

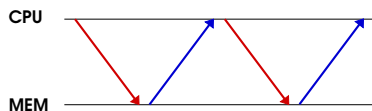
Pipeline: Loop Unroll

Consider copy of a vector:

$$y_i \leftarrow x_i \quad (i = 0, \dots, n-1)$$

Simple implementation:

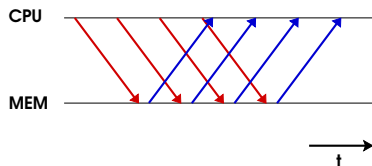
```
for(i = 0; i < n; i++) {
    y[i] = x[i];
}
```



Implementation with unrolling:

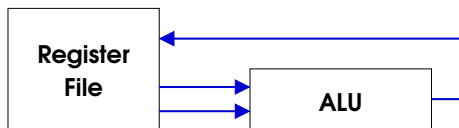
```
for (i = 0; i < n; i += 4) {
    y[i] = x[i];
    y[i+1] = x[i+1];
    y[i+2] = x[i+2];
    y[i+3] = x[i+3];
}
```

(assume n a multiple of 4)



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:
 - Nominal throughput $B = 1$
 - Depth $k = 4$

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:

```
1  mov 0xd, R0    !! Move constant value to R0
2  mov R0, R1     !! Move content register R0 to R1
3  add R0, R1, R2 !! Integer addition: R2=R0+R1
4  mul 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

```
mul R0, R1, R5 ! store  $a \times b$  in register 5
add R5, R2, R6 ! add  $c$ 
sub R6, R3, R7 ! subtract  $d$ 
```

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

Data hazards

- An instruction j is **data dependent** on 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:

```
mul R0, R1, R5 !  
sub R6, R3, R7 ! Register R6 not written, yet  
add 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: $B = 1$, $k = 4$
- Consider example: $z = a \times b + c - d$

1	mul R0, R1, R5	store $a \times b$ in register R5
2		stall
3		stall
4		stall
5	add R5 , R2, R6	add c
6		stall
7		stall
8		stall
9	sub R6 , R3, R7	subtract d

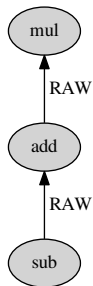
- Latency of operation: $3 \cdot 4$ clock cycles

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.)

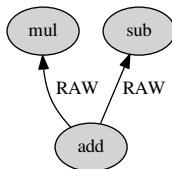
- Rewrite the example as follows:

$$x = a \times b$$

$$y = c - d$$

$$z = x + y$$

- Data dependence graph:



- Latency:

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

Parallel Execution

- Taking advantage of parallelism is one of the most important methods for improving performance
- Today: several levels of parallelism
 - **Instruction Level Parallelism (ILP)**
Multiple instruction pipelines
 - **Vector/SIMD instructions**
Single Instruction performing same operation on **Multiple Data**
 - **Simultaneous Multi-Threading (SMT)**
Concurrent execution of multiple instruction streams within same processor core
 - **Multiple cores per processor**
Processors with multiple cores
 - **Multiple processors**
Multiple processors per nodes, multiple nodes per system
- Problem: Efficient exploitation of parallelism

Performance Metric

- Parallel speedup** using P processes

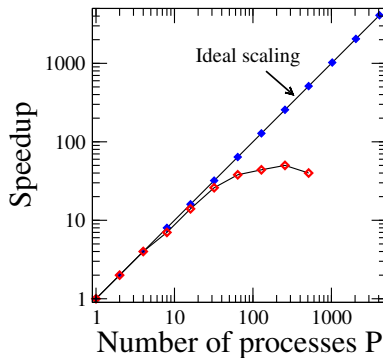
$$S(P) = \frac{\Delta t_{\text{sequential}}}{\Delta t_{\text{parallel}}(P)}$$

- Parallel efficiency**

$$\epsilon = \frac{S(P)}{P}$$

Example: $S = 26$ using $P = 32$
processing elements
 $\Rightarrow \epsilon = 81\%$

- Super-linear speedup**: $S > P$



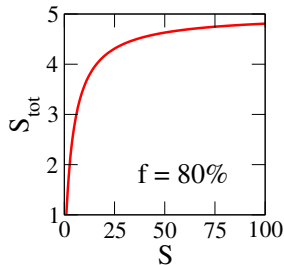
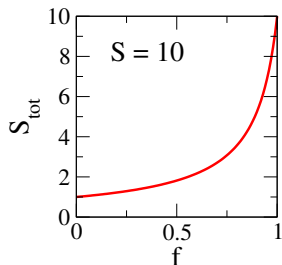
Amdahl's Law

- Consider the following case:
 - Fraction f of application can be parallelized resulting in speed-up by factor S
 - The remaining fraction $1 - f$ remains serial
- Execution time after acceleration:

$$\Delta t' = (1 - f)\Delta t + \frac{f \Delta t}{S}$$

- Total speed-up

$$S_{\text{tot}} = \frac{\Delta t}{\Delta t'} = \frac{1}{(1 - f) + f/S}$$



Gustafson's Law

- Amdahl's law: Only considers scaling with fixed workload
 - ☞ **strong scaling**
- Different ansatz by Gustafson:
 - Increase workload proportional to number of processing elements P ☞ **weak scaling**
 - Assume non-parallelized code part to be independent of workload
- Execution times
 - Serial execution
 - Execution time for original problem: Δt
 - Execution time for scaled problem size: $(1 - f)\Delta t + f P \Delta t$
 - Parallel execution
 - Execution time for scaled problem size: $(1 - f)\Delta t + f \Delta t$
- Speed-up:

$$S = \frac{(1 - f)\Delta t + f P \Delta t}{(1 - f)\Delta t + f \Delta t} = P - (P - 1)(1 - f)$$

Parallel Efficiency: Amdahl vs. Gustafson

- Parallel efficiency according to Amdahl:

$$\epsilon_{\text{Amdahl}}(P) = \frac{1}{P} \cdot \frac{1}{(1-f) + f/P} = \frac{1}{P(1-f) + f}$$

- Parallel efficiency according to Gustafson:

$$\epsilon_{\text{Gustafson}}(P) = 1 - \frac{P-1}{P}(1-f) = f + \frac{1-f}{P}$$

- Limit of large number of processing elements P :

$$\lim_{P \rightarrow \infty} \epsilon_{\text{Amdahl}}(P) = 0$$
$$\lim_{P \rightarrow \infty} \epsilon_{\text{Gustafson}}(P) = f$$