Computer Hardware Engineering

IS1200, spring 2015

Lecture 9: Parallelism, Concurrency, Speedup, and ILP

David Broman

Associate Professor, KTH Royal Institute of Technology

Assistant Research Engineer, University of California, Berkeley

Slides version 1.0

Course Structure

Module 1: Logic Design

L → L → DCÖ1 → DCÖ2 → Lab:dicom

Module 4: I/O Systems

L7 → E6 → E7 → Lab: nios2io

E9 → Lab: nios2int

Module 2: C and Assembly Programming

L1 → L2 → E1

L3 → E2 → E3 → Lab: nios2time

L4 → E4 → Home lab: C

Module 5: Memory Hierarchy

L8 → E8 → Home Lab: cache

Module 3: Processor Design

L5 → L6 → E5

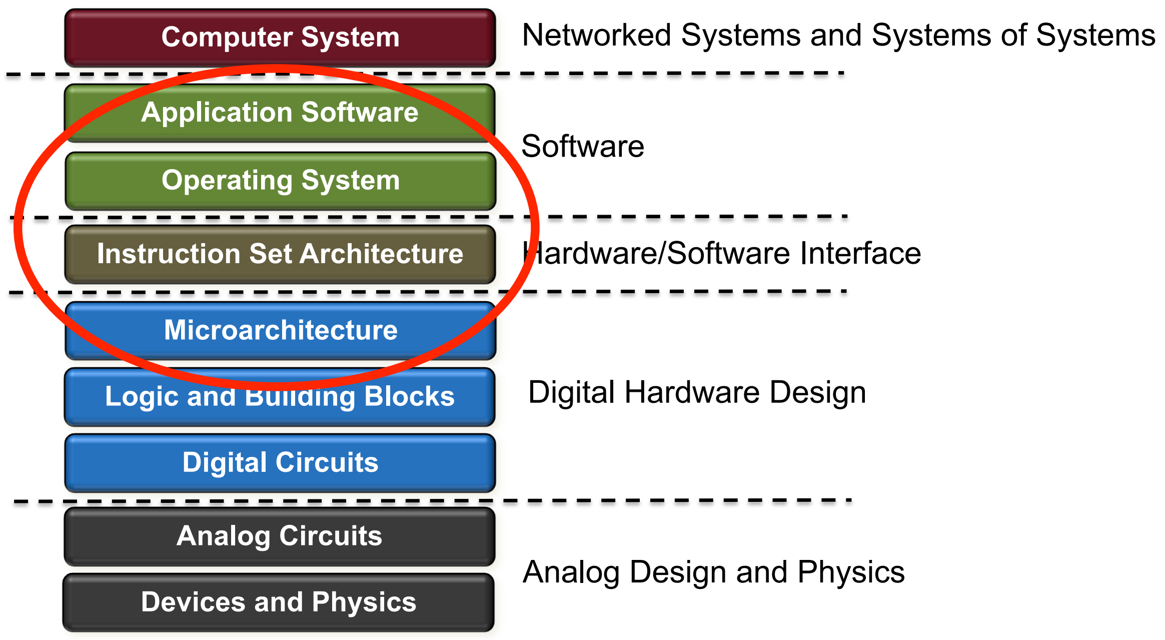
Module 6: Parallel Processors and Programs

L9 → L10 → E10

Part I
Multiprocessors, Parallelism, Concurrency, and Speedup

Part II
Instruction-Level Parallelism

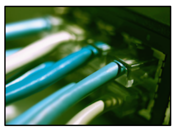
Abstractions in Computer Systems



Agenda

Part I

Multiprocessors, Parallelism, Concurrency, and Speedup

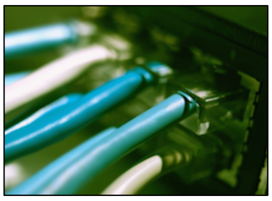


Part II

Instruction-Level Parallelism

Part I

Multiprocessors, Parallelism, Concurrency, and Speedup



Acknowledgement: The structure and several of the good examples are derived from the book "Computer Organization and Design" (2014) by David A. Patterson and John L. Hennessy

How is this computer revolution possible? (Revisited)

Moore's law:

- Integrated circuit resources (transistors) double every 18-24 months.
- By Gordon E. Moore, Intel's co-founder, 1960s.
- Possible because refined manufacturing process. E.g., 4th generation Intel Core i7 processors uses 22nm manufacturing.
- Sometimes considered a *self-fulfilling prophecy*. Served as a goal for the semiconductor industry.

Have we reached the limit? (Revisited)

Why?

The Power Wall

http://www.publicdomainpictures.net/view-image.php?image=1281&picture=tegelvagg

During the last decade, the clock rate has increased dramatically.

• 1989: 80486,	25MHz
• 1993: Pentium,	66Mhz
• 1997: Pentium Pro,	200MHz
• 2001: Pentium 4,	2.0 GHz
• 2004: Pentium 4,	3.6 GHz

2013: Core i7, 3.1 GHz - 4 GHz

Increased clock rate implies increased power

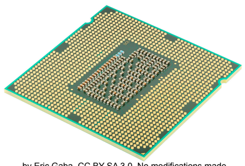
We cannot cool the system enough to increase the clock rate anymore…

"New" trend since 2006: Multicore
- Moore's law still holds
- More processors on a chip: multicore
- "New" challenge: parallel programming

What is a multiprocessor?

A **multiprocessor** is a computer system with two or more processors.

By contrast, a computer with one processor is called a **uniprocessor**.

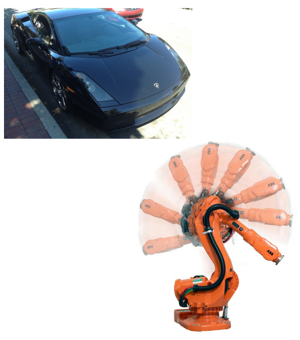

by Eric Gaba, CC BY-SA 3.0. No modifications made.

Multicore microprocessors are multiprocessors where all processors (cores) are located on a single integrated circuite

Photo by Robert Harker

A **cluster** is a set of computers that are connected over a local area network (LAN). May be viewed as one large multiprocessor.

Different Kinds of Computer Systems (Revisited)



Embedded
Real-Time Systems

Photo by Kyro

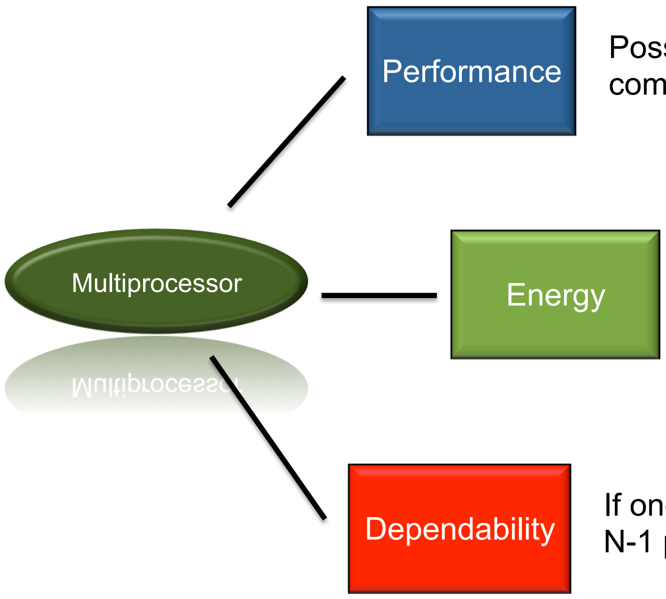
Personal Computers and
Personal Mobile Devices

Photo by Robert Harker

Warehouse
Scale Computers

Dependability | Energy | Performance

Why multiprocessors?



Possible to execute many computation tasks in parallel.

Replace energy inefficient processors in data centers with many efficient smaller processors.

If one out of N processors fails, still N-1 processors are functioning.

Parallelism and Concurrency – what is the difference?

Concurrency is about **handling** many things at the same time. Concurrency may be viewed from the **software** viewpoint.

Parallelism is about **doing (executing)** many things at the same time. Parallelism may be viewed from the **hardware** viewpoint.

		Software	
		Sequential	**Concurrent**
Hardware	**Serial**	Example: matrix multiplication on a unicore processor.	Example: A Linux OS running on a unicore processor .
	Parallel	Example: matrix multiplication on a multicore processor.	Example: A Linux OS running on a multicore processor .

Note: As always, everybody does not agree on the definitions of concurrency and parallelism. The matrix is from H&P 2014 and the informal definitions above are similar to what was said in a talk by Rob Pike.

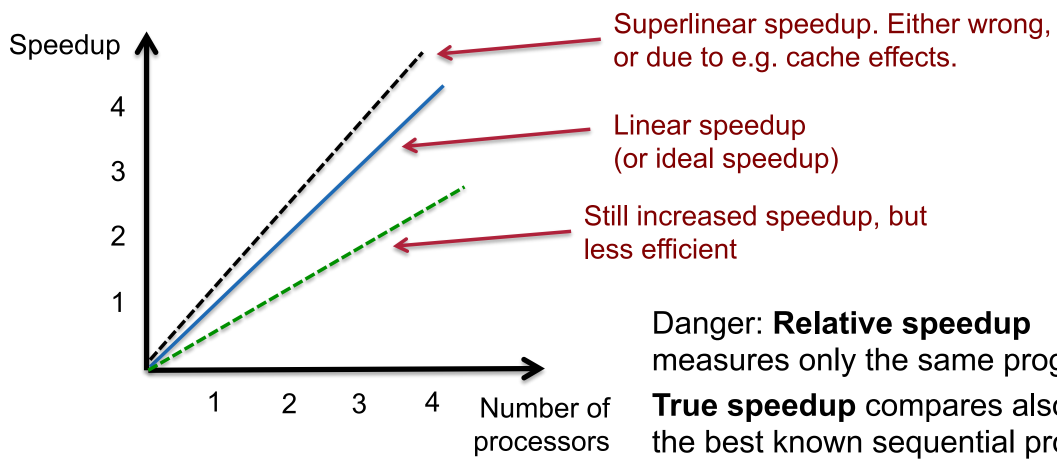
Speedup

How much can we improve the performance using parallelization?

$$\textit{Speedup} = \frac{T_{\text{before}}}{T_{\text{after}}}$$

T_{before}: Execution time of one program **before** improvement

T_{after}: Execution time **after** improvement



Danger: **Relative speedup** measures only the same program

True speedup compares also with the best known sequential program,

Amdahl's Law (1/4)

Can we achieve linear speedup?

Divide execution time before improvement into two parts.

Time affected by the improvement of parallelization

Time unaffected of improvement (sequential part)

$$\boxed{T = T_{affected} + T_{unaffected}}$$

Execution time after improvement

$$\boxed{T_{after} = \frac{T_{affected}}{N} + T_{unaffected}}$$

Amount of improvement (N times improvement)

$$\boxed{Speedup = \frac{T_{before}}{T_{after}} = \frac{T_{before}}{\frac{T_{affected}}{N} + T_{unaffected}}}$$

This is sometimes referred to as **Amdahl's law**

E

Amdahl's Law (2/4)

$$\boxed{Speedup = \frac{T_{before}}{T_{after}} = \frac{T_{before}}{\frac{T_{affected}}{N} + T_{unaffected}}}$$

Exercise: Assume a program consists of an image analysis task, sequentially followed by a statistical computation task. Only the image analysis task can be parallelized. How much do we need to improve the image analysis task to be able to achieve 4 times speedup?

Assume that the program takes 80ms in total and that the image analysis task takes 60ms out of this time.

Solution:

4 = 80 / (60 / N + 80 – 60)

60/N + 20 = 20

60/N = 0

It is impossible to achieve this speedup!

Amdahl's Law (3/4)

$$\textit{Speedup} = \frac{T_{\text{before}}}{T_{\text{after}}} = \frac{T_{\text{before}}}{\frac{T_{\text{affected}}}{N} + T_{\text{unaffected}}}$$

Assume that we perform 10 scalar integer additions, followed by one matrix addition, where matrices are 10x10. Assume additions take the same amount of time and that we can only parallelize the matrix addition.

Exercise A: What is the speedup with 10 processors?

Exercise B: What is the speedup with 40 processors?

Exercise C: What is the maximal speedup (the limit when N → infinity)

Solution A:
(10+10*10) / (10*10/10 + 10) = 5.5

Solution B:
(10+10*10) / (10*10/40 + 10) = 8.8

Solution C:
(10+10*10) / (10*10/N + 10) = 11 when N → infinity

Amdahl's Law (4/4)

Example continued. What if we change the size of the problem (make the matrices larger)?

Size of matrices	Number of processors: 10	Number of processors: 40
10x10	Speedup 5.5	Speedup 8.8
20x20	Speedup 8.2	Speedup 20.5

But was not the maximal speedup 11.1 when N → infinity?

Strong scaling = measuring speedup while keeping the problem size fixed.

Weak scaling = measuring speedup when the problem size grows proportionally to the increased number of processors.

Main Classes of Parallelisms

DLP

Data-Level Parallelism (DLP)
Many data items can be processed at the same time.

Example – Sheep shearing
Assume that sheep are data items and the task for the farmer is to do sheep shearing (remove the wool). Data-level parallelism would be the same as using several farm hands to do the shearing.

TLP

Task-Level Parallelism (TLP)
Different tasks of work that can work in independently and in parallel

Example – Many tasks at the farm
Assume that there are many different things that can be done on the farm (fix the barn, sheep shearing, feed the pigs etc.) Task-level parallelism would be to let the farm hands do the different tasks in parallel.

E

SISD, SIMD, and MIMD

An old (from the 1960s) but still very useful classification of processors uses the notion of instruction and data streams.

		Data Stream	
		Single	**Multiple**
Instruction Stream	**Single**	**SISD** E.g. Intel Pentium 4	**SIMD** E.g. SSE Instruction in x86
	Multiple	**MISD** No examples today	**MIMD** E.g. Intel Core i7

Data-level parallelism. Examples are multimedia extensions (e.g., SSE, streaming SIMD extension), vector processors.

Graphical Unit Processors (GPUs) are both SIMD and MIMD

Task-level parallelism. Examples are multicore and cluster computers

Physical Q/A
What is a modern Intel CPU, such as Core i7? Stand for MIMD, on the table for SIMD

Part II

Instruction-Level Parallelism

Acknowledgement: The structure and several of the good examples are derived from the book "Computer Organization and Design" (2014) by David A. Patterson and John L. Hennessy

What is Instruction-Level Parallelism?

Instruction-Level Parallelism (ILP) may increase performance without involvement of the programmer. It may be implemented in a SISD, SIMD, and MIMD computer.

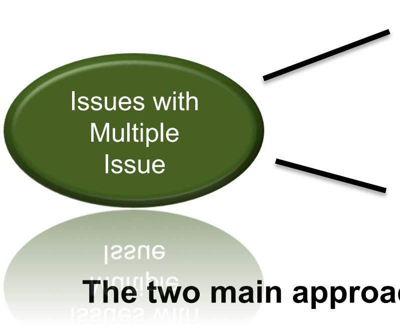
Two main approaches:

1. Deep pipelines with more pipeline stages
If the length of all pipeline stages are balanced, we may increase performance by increasing the clock speed.

2. Multiple issue
A technique where multiple instructions are issued in each in cycle.

ILP may decrease the CPI to lower than 1, or using the inverse metric *instructions per clock cycle (IPC)* increase it above 1.

Multiple Issue Approaches



Determining how many and which instructions to issue in each cycle. Problematic since instructions typically depend on each other.

How to deal with data and control hazards

The two main approaches of multiple issue are

1. Static Multiple Issue
Decisions on when and which instructions to issue at each clock cycle is determined by the compiler.

2. Dynamic Multiple Issue
Many of the decisions of issuing instructions are made by the processor, dynamically, during execution.

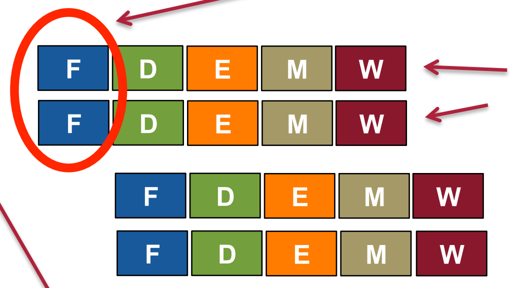
Static Multiple Issue (1/3) VLIW

Very Long Instruction Word (VLIW) processors issue several instructions in each cycle using **issue packages.**

An **issue package** may be seen as one large instruction with multiple operations.

```
add  $s0, $s1, $s2
add  $t0, $t0, $0

and  $t2, $t1, $s0
lw   $t0, 0($s0)
```



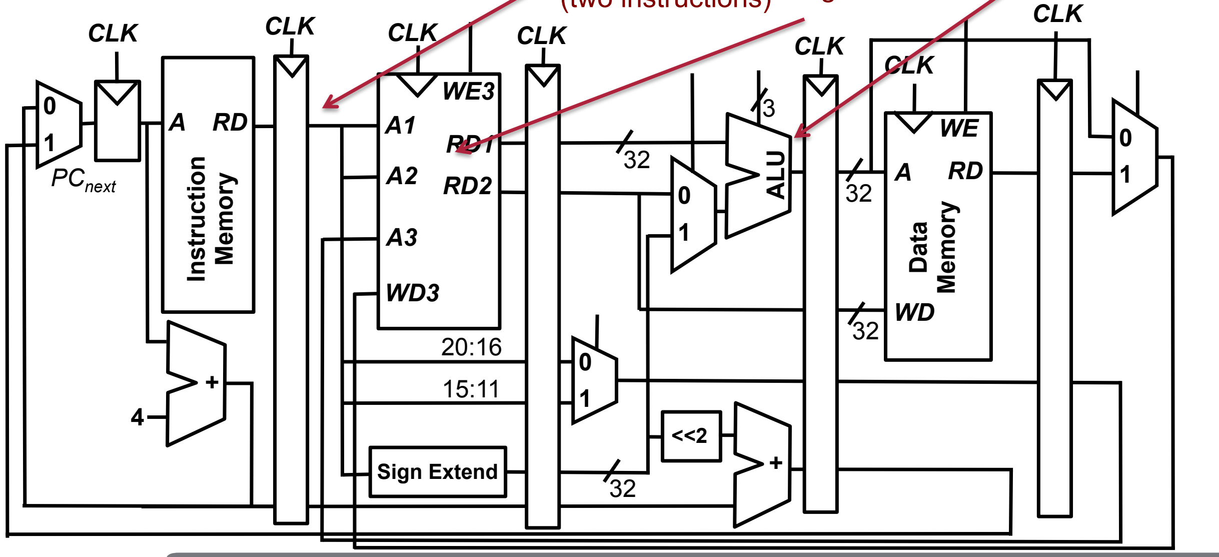
Each issue package has two **issue slots**.

The compiler may insert **no-ops** to avoid hazards.

How is VLIW affecting the hardware implementation?

Static Multiple Issue (2/3) Changing the hardware

To issue two instructions in each cycle, we need the following (not shown in picture):



Static Multiple Issue (3/3) VLIW

Exercise: Assume we have a VLIW processor that can issue two instructions in the same clock cycle. For the following code, schedule the instructions into issue slots and compute IPC. Assume that no hazards occurs.

```
    addi $s1, $0, 10
L1: lw   $t0, 0($s2)
    ori  $t1, $t0, 7
    sw   $t1, 0($s2)
    addi $s2, $s2, 4
    addi $s1, $s1, -1
    bne  $s1, $0, L1
```

Solution. IPC = (1 + 6*10) / (1 + 4*10) = 1.487… (max 2)

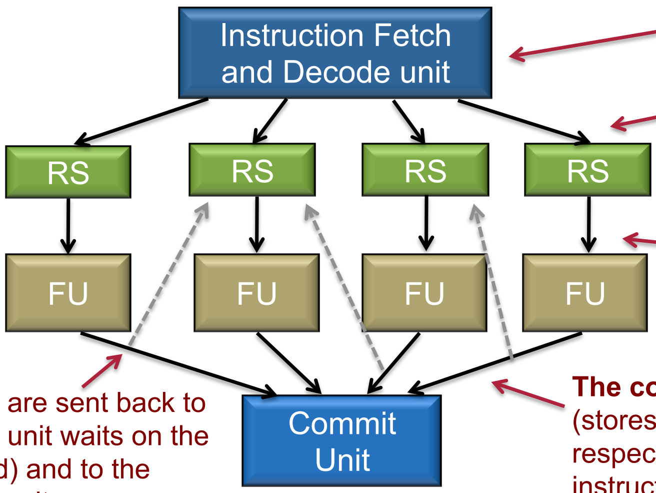
Reschedule to avoid stalls because of **lw**

	Slot 1	Slot 2	Cycle
	`addi $s1, $0, 10`		1
L1	`lw   $t0, 0($s2)`		2
	`addi $s2, $s2, 4`	`addi $s1, $s1, -1`	3
	`ori  $t1, $t0, 7`		4
	`bne  $s1, $0, L1`	`sw   $t1, -4($s2)`	5

Empty cells means no-ops

Dynamic Multiple-Issue Processors (1/2) Superscalar Processors

In many modern processors (e.g., Intel Core i7), instruction issuing is performed dynamically by the processor while executing the program. Such processor is called **superscalar**.



The first unit fetches and decodes several instruction **in-order**

Reservation Stations (RS) buffer operands to the FU before execution. Data dependencies are handled dynamically.

Functional Units (FU) execute the instruction. Examples are integer units, floating point units, and load/store units.

Results are sent back to RS (if a unit waits on the operand) and to the commit unit.

The commit unit commits instructions (stores in registers) concervatively, respecting the obervable order of instructions. Called **in-order commit**.

Dynamic Multiple-Issue Processors (2/2) Out-of-Order Execution, RAW, WAR

If the superscalar processor can reorder the instruction execution order, it has an **out-of-order execution** processor

```
lw   $t0, 0($s2)
addi $t1, $t0, 7
```

Example of a **Read After Write (RAW)** dependency (dependency on $t0). The superscalar pipeline must make sure that the data is available before read.

```
sub  $t0, $t1, $s0
addi $t1, $s0, 10
```

Example of a **Write After Read (WAR)** dependency (dependency on $t1). If the order is flipped due to out-of-order execution, we have a hazard.

```
addi $r1, $s0, 10
sub  $t0, $t1, $s0
```

WAR dependencies can be resolved using **register renaming**, where the processor writes to a nonarchitectural renaming register (here in the pseudo asm code called $r1, not accessible to the programmer)

Note that out-of-order processor is not rewriting the code, but keeps track of the renamed registers during execution.

Some observations on Multi-Issue Processors

Multi-Issue Processors

VLIW processors tend to be more **energy efficient** than superscalar out-of-order processors (less hardware, the compiler does the job)

Superscalar processors with dynamic scheduling can **hide some latencies** that are not statically predictable (e.g., cache misses, dynamic branch predictions).

Although modern processors issues 4 to 6 instructions per clock cycle, few applications results in IPC over 2. The reason is dependencies.

Intel Microprocessors, some examples

Processor	Year	Clock Rate	Pipeline Stages	Issue Width	Cores	Power
Intel 486	1989	25 MHz	5	1	1	5 W
Intel Pentium	1993	66 MHz	5	2	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	1	103W
Intel Core	2006	2930 MHz	14	4	2	75W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	1	87W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	8	77W

Clock rate increase stopped (the power wall) around 2006

Pipeline stages first increased and then decreased, but the number of cores increased after 2006.

The power consumption peaked with Pentium 4

Source: Patterson and Hennessey, 2014, page 344.

Bonus Part

Time-Aware Systems Design Research Challenges

David Broman @ KTH

(not part of the Examination in IS1200)

Time-Aware Systems - Examples

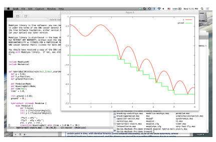
Cyber-Physical Systems (CPS)

Automotive

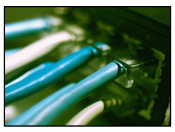
Process Industry and Industrial Automation

Aircraft

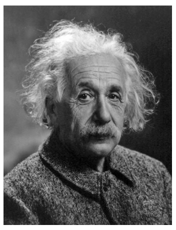
Time-Aware Simulation Systems


Physical simulations (Simulink, Modelica, etc.)

Time-Aware Distributed Systems


Time-stamped distributed systems (E.g. Google Spanner)

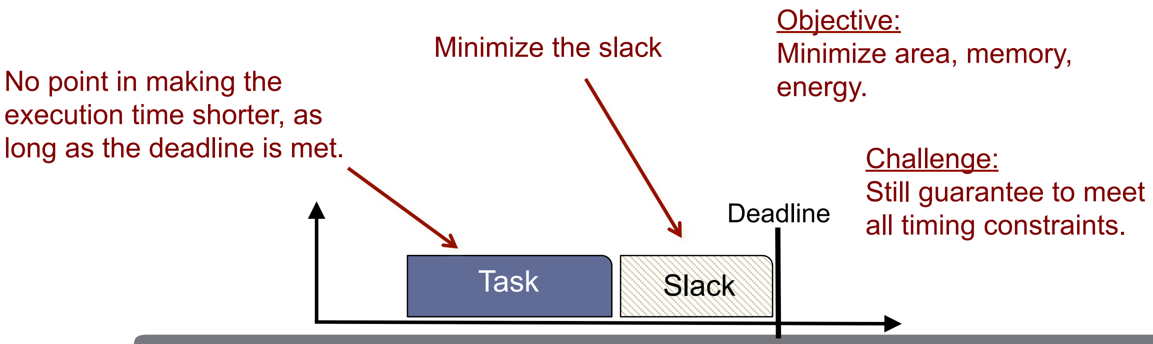
What is our goal?



"Everything should be made as simple as possible, but not simpler"

attributed to Albert Einstein

Execution time should be as short as possible, but not shorter



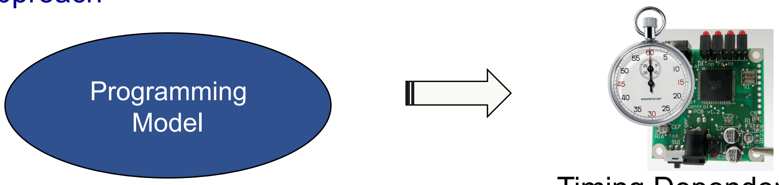
Objective:
Minimize area, memory, energy.
Minimize the slack
No point in making the execution time shorter, as long as the deadline is met.
Challenge:
Still guarantee to meet all timing constraints.
Deadline
Task
Slack

Programming Model and Time

Timing is not part of the software semantics

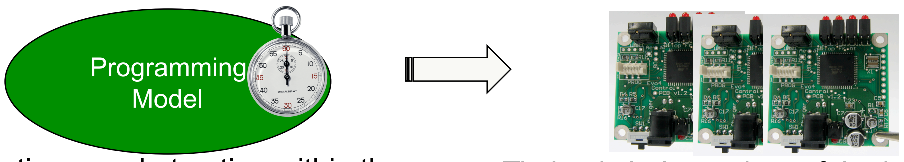
Correct execution of programs (e.g., in C, C++, C#, Java, Scala, Haskell, OCaml) has nothing to do with how long time things takes to execute.

Traditional Approach



Timing Dependent on the Hardware Platform

Our Objective



Make time an abstraction within the programming model

Timing is independent of the hardware platform (within certain constraints)

Are you interested to be challenged?

If you are interested in

- Programming language design, or
- Compilers, or
- Computer Architecture

You are

- ambitious and interested in learning new things

You want to

- do a real research project as part of you Bachelor's or Master's thesis project

Please send an email to dbro@kth.se, so that we can discuss some ideas.

Summary

Some key take away points:

- **Amdahl's law** can be used to estimate maximal speedup when introducing parallelism in parts of a program.
- **Instruction Level Parallelism (ILP)** has been very important for performance improvements over the years, but improvements have not been as significant lately.

Thanks for listening!

Reading for next lecture:
P&H Chapter 6 about parallel processors and the cloud