Intro for Computer Architecture

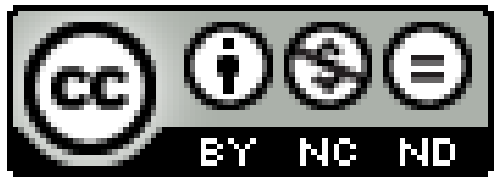
Tarek ElDeeb

SilMinds, LLC

SILMINDS
SILICON FOR GREEN COMPUTING

February 24, 2013

License
(Full License Legal Terms)



Attribution-NonCommercial-NoDerivs [BY-NC-ND]

This license only allows users to download works and share them with others as long as you credit SilMinds work, but users can’t change them in any way or use them commercially.

What is Computer Architecture?

According to the 1913 Webster, architecture is:

- the art or science of building;. . . or
- construction, in a more general sense.

Recent Dictionaries

. . .

4: (computer science) the structure and organization of a computer's hardware or system software; "the architecture of a computer's system software" [syn: computer architecture]

What is Computer Architecture?

According to the 1913 Webster, architecture is:

- the art or science of building;. . . or
- construction, in a more general sense.

Recent Dictionaries

. . .

4: (computer science) the structure and organization of a computer's hardware or system software; "the architecture of a computer's system software" [syn: computer architecture]

What is Computer Architecture?

According to the 1913 Webster, architecture is:

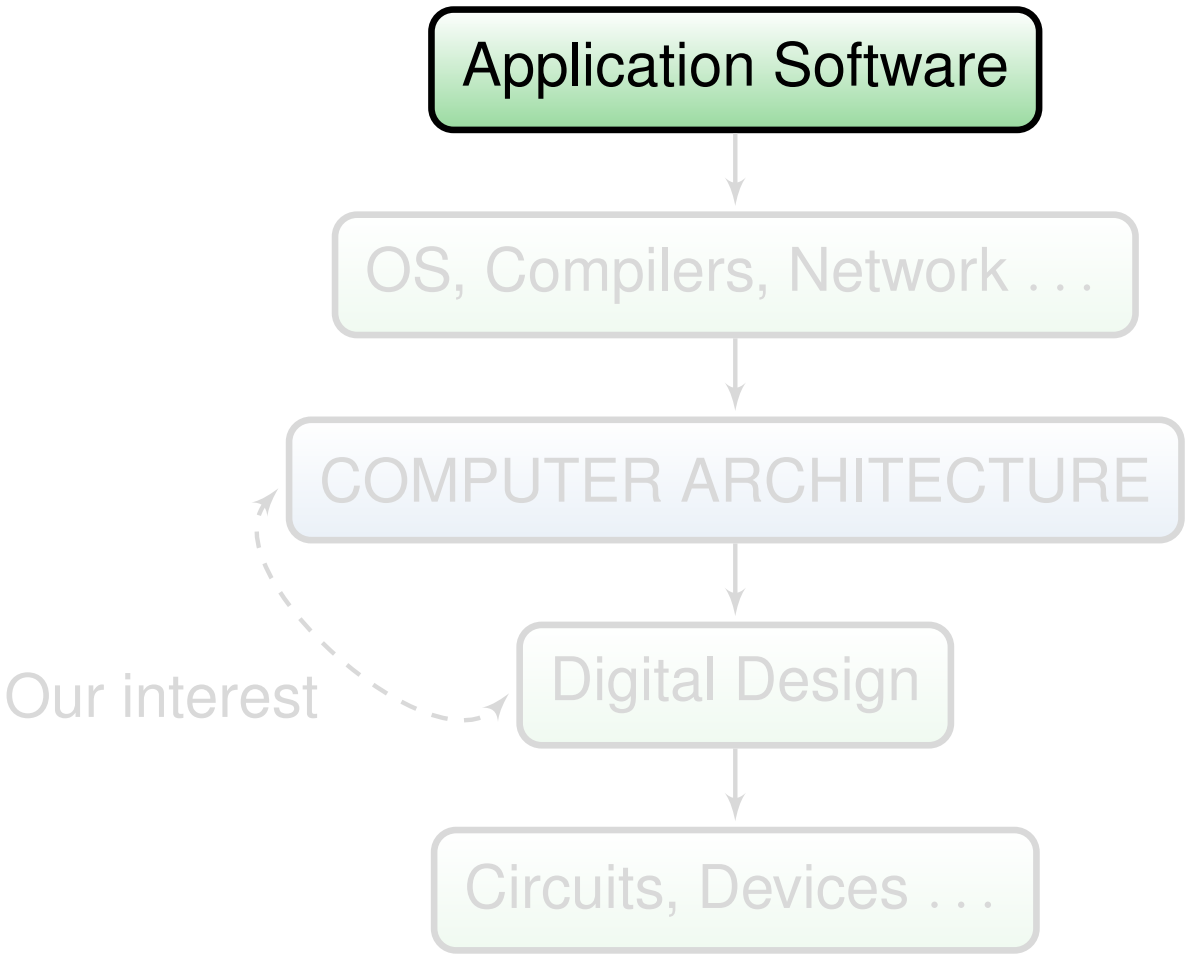
- the art or science of building;. . . or
- construction, in a more general sense.

Recent Dictionaries

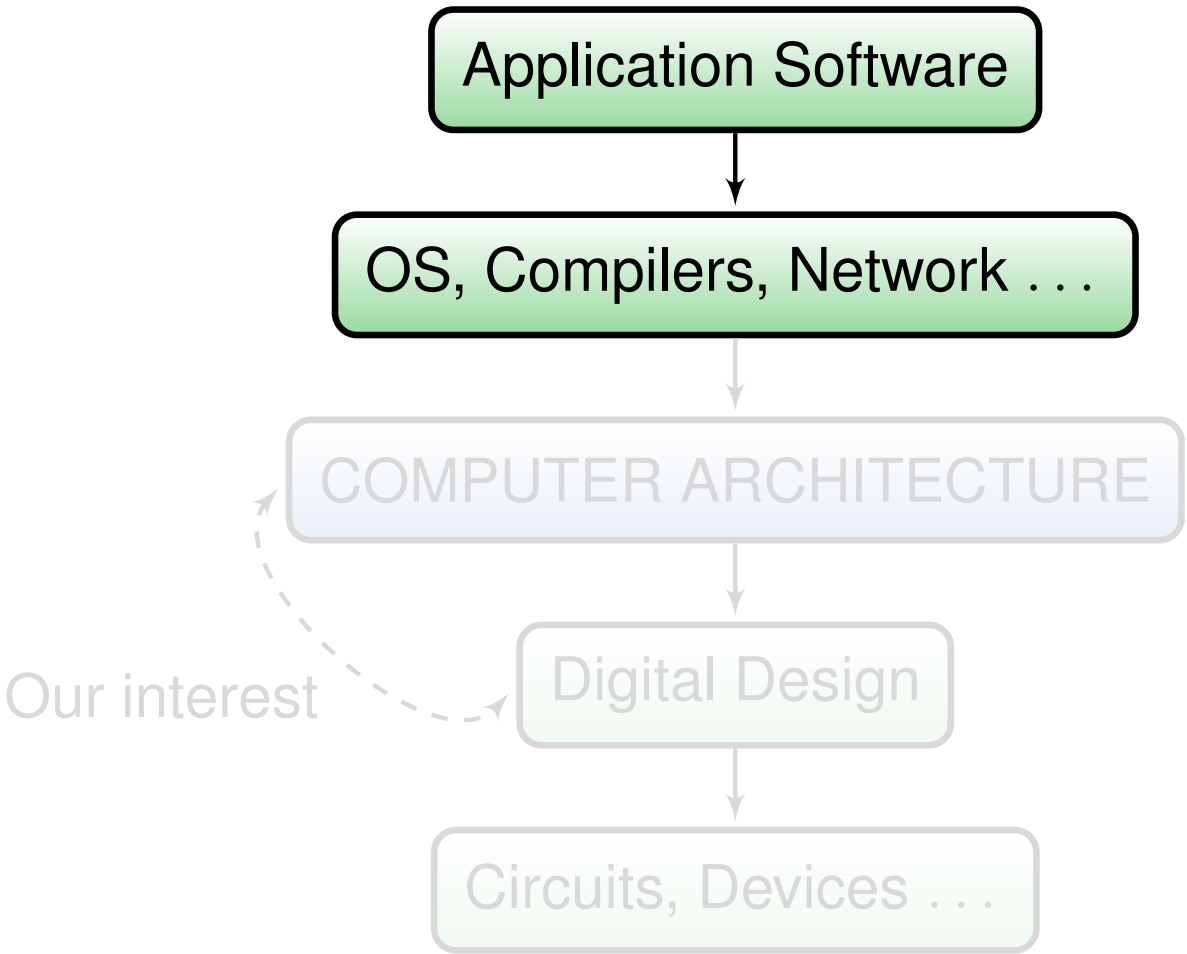
. . .

4: (computer science) the structure and organization of a computer's hardware or system software; "the architecture of a computer's system software" [syn: computer architecture]

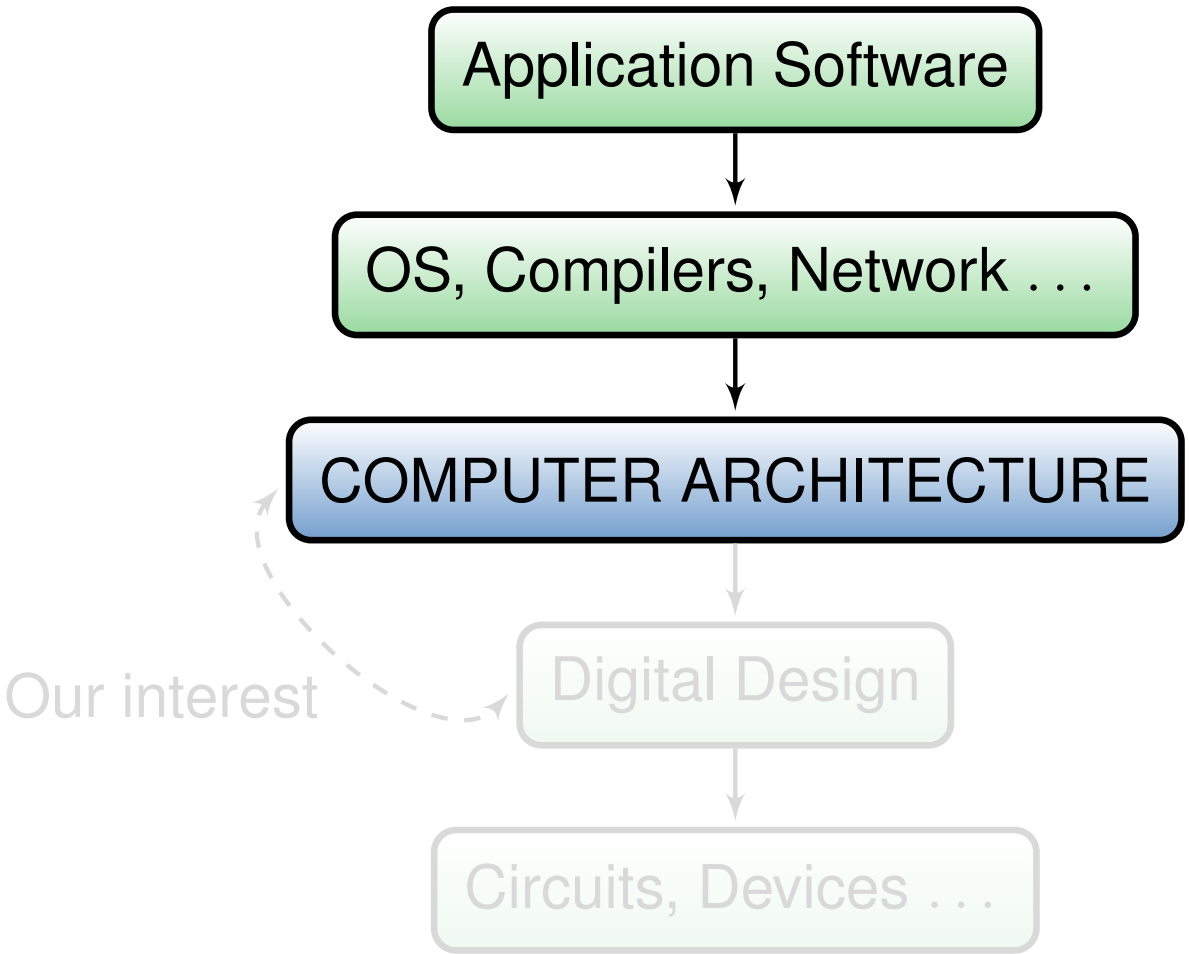
Where does computer architecture fit?



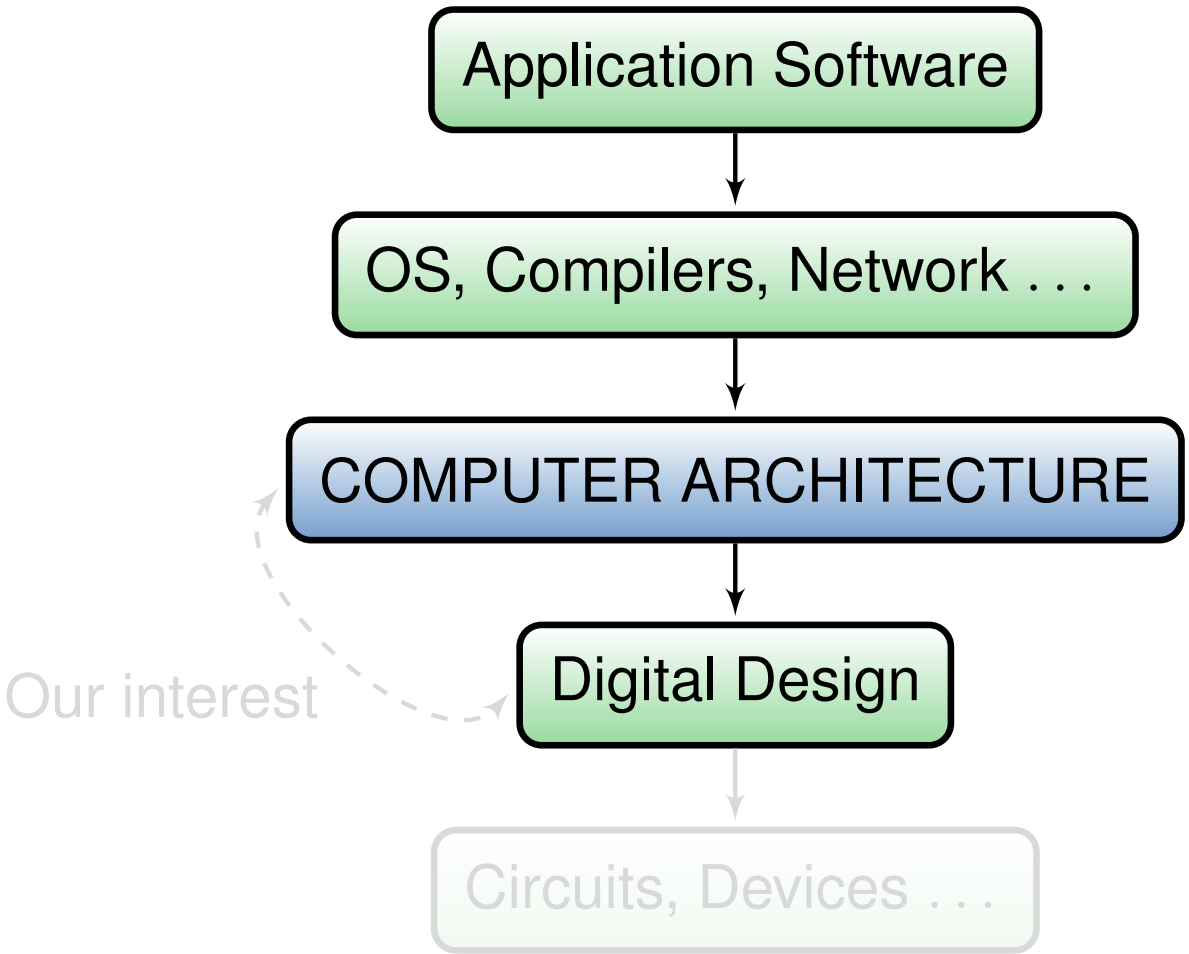
Where does computer architecture fit?



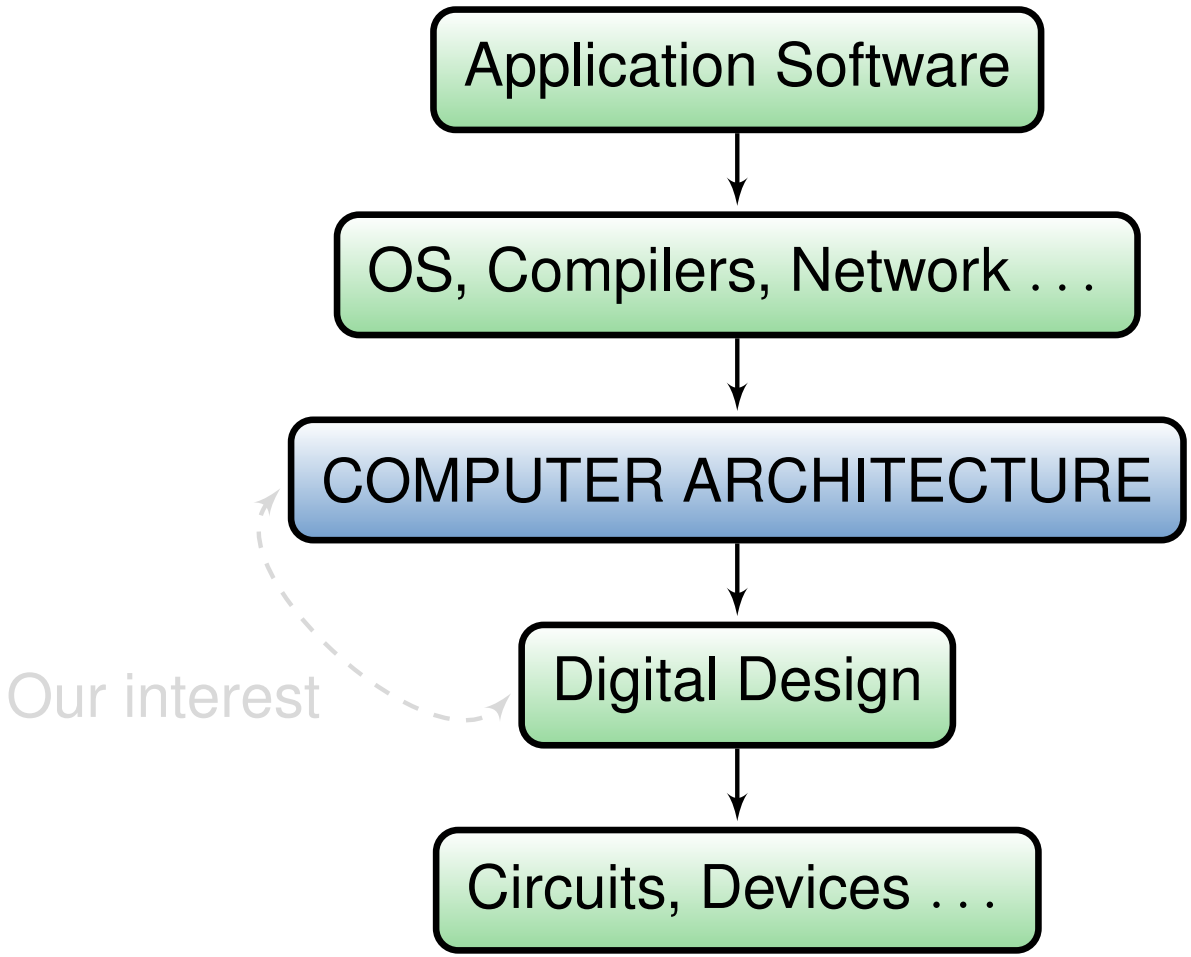
Where does computer architecture fit?



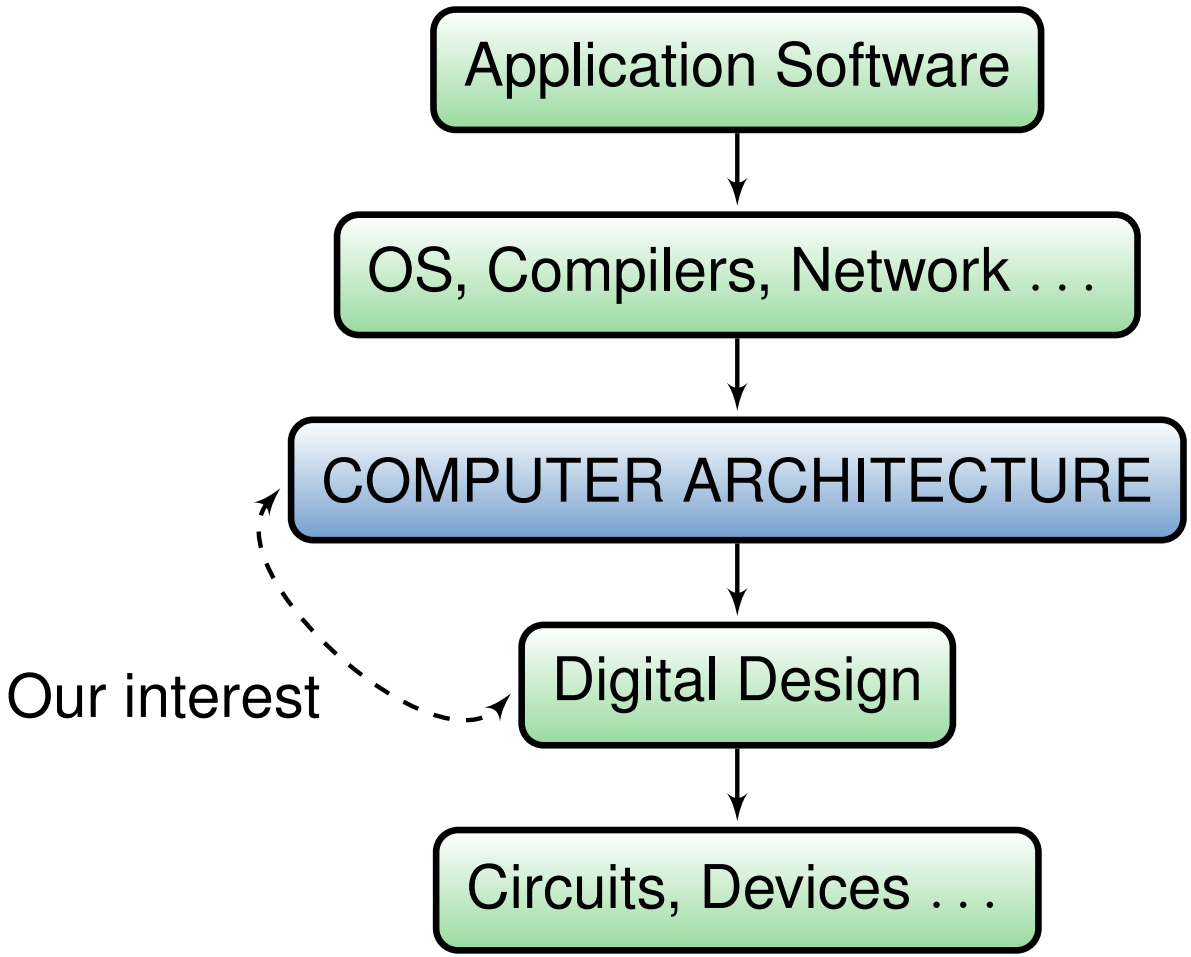
Where does computer architecture fit?



Where does computer architecture fit?



Where does computer architecture fit?



Computer architecture: Structure

- **Within the processor** : Registers, Operational Units (Integer, Floating Point, special purpose, . . .)
- **Outside the processor**: Memory, I/O, . . .
- **Examples** : Sun SPARC, MIPS, Intel x86 (IA32), IBM S/390.
- **Defines** : data (types, endianness storage, and addressing modes), instruction (operation code) set, and instruction formats.

Computer architecture: Structure

- **Within the processor** : Registers, Operational Units (Integer, Floating Point, special purpose, . . .)
- **Outside the processor**: Memory, I/O, . . .
- **Examples** : Sun SPARC, MIPS, Intel x86 (IA32), IBM S/390.
- **Defines** : data (types, endianness storage, and addressing modes), instruction (operation code) set, and instruction formats.

Computer architecture: Structure

- **Within the processor** : Registers, Operational Units (Integer, Floating Point, special purpose, . . .)
- **Outside the processor**: Memory, I/O, . . .
- **Examples** : Sun SPARC, MIPS, Intel x86 (IA32), IBM S/390.
- **Defines** : data (types, endianness storage, and addressing modes), instruction (operation code) set, and instruction formats.

Computer architecture: Structure

- **Within the processor** : Registers, Operational Units (Integer, Floating Point, special purpose, . . .)
- **Outside the processor**: Memory, I/O, . . .
- **Examples** : Sun SPARC, MIPS, Intel x86 (IA32), IBM S/390.
- **Defines** : data (types, endianness storage, and addressing modes), instruction (operation code) set, and instruction formats.

Computer architecture: Organization

- **Within the processor** : Pipeline(s), Control Unit, Instruction Cache, Data Cache, Branch Prediction, . . .
- **Outside the processor**: Secondary Caches, Memory Interleaving, Redundant Disk Arrays, Multi-Processors, . . .

- From a programmer point of view, should I know about the organization?
- Which implementation is better? How do you define 'better'?

Computer architecture: Organization

- **Within the processor** : Pipeline(s), Control Unit, Instruction Cache, Data Cache, Branch Prediction, . . .
- **Outside the processor**: Secondary Caches, Memory Interleaving, Redundant Disk Arrays, Multi-Processors, . . .

- From a programmer point of view, should I know about the organization?
- Which implementation is better? How do you define 'better'?

Computer architecture: Organization

- **Within the processor** : Pipeline(s), Control Unit, Instruction Cache, Data Cache, Branch Prediction, . . .
- **Outside the processor**: Secondary Caches, Memory Interleaving, Redundant Disk Arrays, Multi-Processors, . . .

- From a programmer point of view, should I know about the organization?
- Which implementation is better? How do you define 'better'?

Computer architecture: Organization

- **Within the processor** : Pipeline(s), Control Unit, Instruction Cache, Data Cache, Branch Prediction, . . .
- **Outside the processor**: Secondary Caches, Memory Interleaving, Redundant Disk Arrays, Multi-Processors, . . .

- From a programmer point of view, should I know about the organization?
- Which implementation is better? How do you define 'better'?

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Instruction Set Architecture

- Common instructions
  - Arithmetic and Logic
  - Data transfer
  - Control
- Optional instructions
  - system
  - floating-point
  - graphics
- Some Control instructions
  - un/conditional branches,
  - function calls, and
  - returns

Comparing ISAs

Archi	Bits	Date	Op	Type	Design	Regs	Encoding	Endianness
Alpha	64	1992	3	Reg-Reg	RISC	32	Fixed	Bi
ARM	32	1983	3	Reg-Reg	RISC	16	Thumb-2: Variable (16/32 bit)	Bi
MIPS	64 (32→64)	1981	3	Reg-Reg	RISC	32	Fixed (32-bit)	Bi
PowerPC	32/64 (32→64)	1991	3	Reg-Reg	RISC	32	Fixed, Variable	Big/Bi
SPARC	64 (32→64)	1985	3	Reg-Reg	RISC	32	Fixed	Big → Bi
z/Archi	64 (32→64)	1964	?	Reg-Mem/ Mem-Mem	CISC	16	Fixed	Big
VAX	32	1977	6	Mem-Mem	CISC	16	Variable	Little
x86	32 (16→32)	1978	2	Reg-Mem	CISC	8	Variable	Little
x86-64	64	2003	2	Reg-Mem	CISC	16	Variable	Little

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Different ISAs

- CISC vs RISC. CPI?
- Memory Access
  - Direct: mem[1204];
  - Register indirect: mem[R4];
  - Displacement: mem[R1+constant];
  - Relative to PC: mem[PC+constant];
- Instruction Format
  - Fixed Length
  - Variable Length
  - Hybrid (common in em bedded systems)

Design goals

- **Functional** Should be correct! What functions should it support?
- **Reliable** A spacecraft is different from a PC. Is it really?
- **Performance** It is not just the frequency but the speed of real tasks. You cannot please everyone all the time.
- **Low cost** design cost (how big are the teams? How long do they take?), manufacturing cost, testing cost, . . .
- **Energy efficiency** this is the “running cost”. Energy is drawn from various sources. The cooling is a big issue.

Design goals

- **Functional** Should be correct! What functions should it support?
- **Reliable** A spacecraft is different from a PC. Is it really?
- **Performance** It is not just the frequency but the speed of real tasks. You cannot please everyone all the time.
- **Low cost** design cost (how big are the teams? How long do they take?), manufacturing cost, testing cost, . . .
- **Energy efficiency** this is the “running cost”. Energy is drawn from various sources. The cooling is a big issue.

Design goals

- **Functional** Should be correct! What functions should it support?
- **Reliable** A spacecraft is different from a PC. Is it really?
- **Performance** It is not just the frequency but the speed of real tasks. You cannot please everyone all the time.
- **Low cost** design cost (how big are the teams? How long do they take?), manufacturing cost, testing cost, . . .
- **Energy efficiency** this is the “running cost”. Energy is drawn from various sources. The cooling is a big issue.

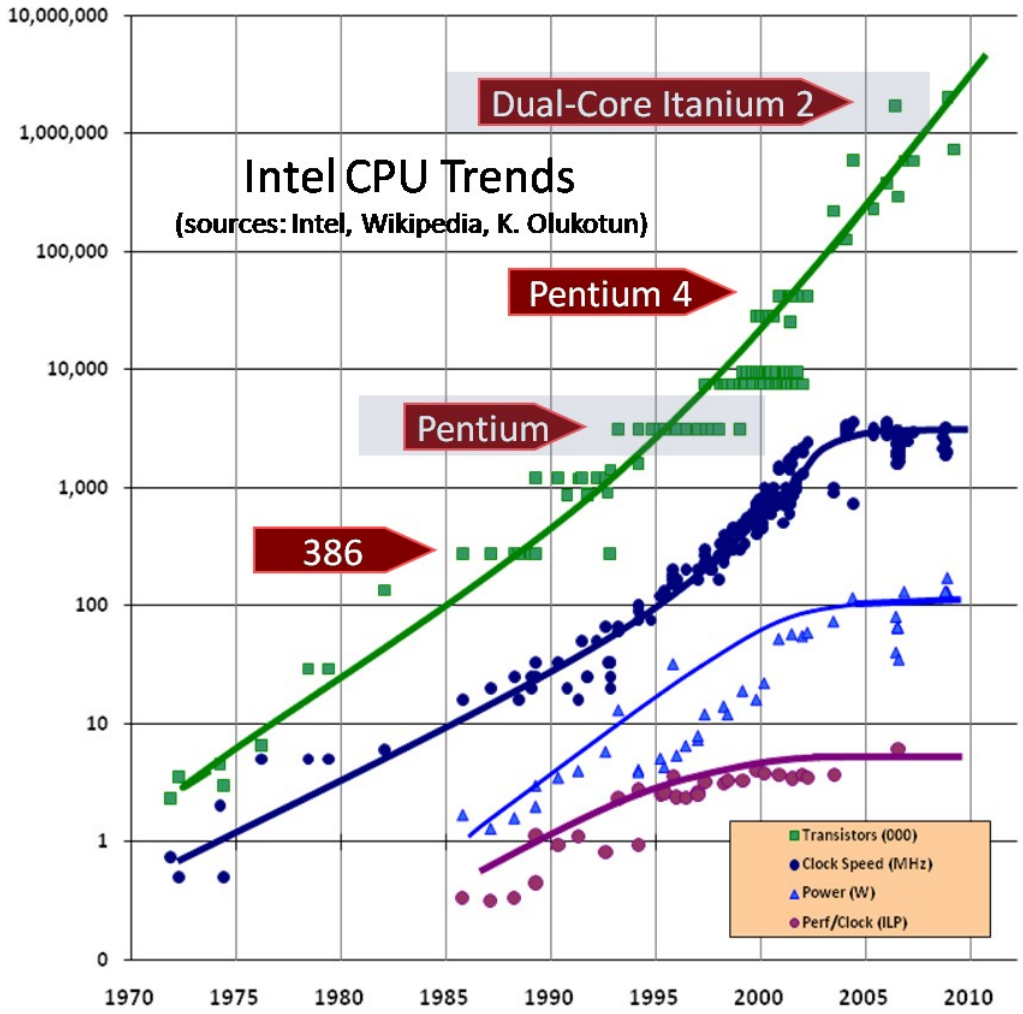
Design goals

- **Functional** Should be correct! What functions should it support?
- **Reliable** A spacecraft is different from a PC. Is it really?
- **Performance** It is not just the frequency but the speed of real tasks. You cannot please everyone all the time.
- **Low cost** design cost (how big are the teams? How long do they take?), manufacturing cost, testing cost, . . .
- **Energy efficiency** this is the “running cost”. Energy is drawn from various sources. The cooling is a big issue.

Design goals

- **Functional** Should be correct! What functions should it support?
- **Reliable** A spacecraft is different from a PC. Is it really?
- **Performance** It is not just the frequency but the speed of real tasks. You cannot please everyone all the time.
- **Low cost** design cost (how big are the teams? How long do they take?), manufacturing cost, testing cost, . . .
- **Energy efficiency** this is the “running cost”. Energy is drawn from various sources. The cooling is a big issue.

How do design goals change?



Performance?

- 'Latency' or 'Thoughput'?
- How do we measure time?
  - Real application. Portability?
  - Kernel. Real Complexity?
  - Selected Set of Application
- Benchmarks: SPEC, TPC, . . .
- CPU time: $T_1 =$ Dynamic instruction count $\times$ average CPI $\times$ Clock cycle time
- Speed-up: $S_p = \frac{T_1}{T_p} = \frac{T_1}{0.25T_1 + 0.75T1/P}$. Try $P = 3$ and $P = \infty$

Performance?

- 'Latency' or 'Thoughput'?
- How do we measure time?
  - Real application. Portability?
  - Kernel. Real Complexity?
  - Selected Set of Application
- Benchmarks: SPEC, TPC, . . .
- CPU time: $T_1 =$ Dynamic instruction count $\times$ average CPI $\times$ Clock cycle time
- Speed-up: $S_p = \frac{T_1}{T_p} = \frac{T_1}{0.25T_1 + 0.75T1/P}$. Try $P = 3$ and $P = \infty$

Performance?

- 'Latency' or 'Thoughput'?
- How do we measure time?
  - Real application. Portability?
  - Kernel. Real Complexity?
  - Selected Set of Application
- Benchmarks: SPEC, TPC, . . .
- CPU time: $T_1 =$ Dynamic instruction count $\times$ average CPI $\times$ Clock cycle time
- Speed-up: $S_p = \frac{T_1}{T_p} = \frac{T_1}{0.25T_1 + 0.75T1/P}$. Try $P = 3$ and $P = \infty$

Performance?

- 'Latency' or 'Thoughput'?
- How do we measure time?
  - Real application. Portability?
  - Kernel. Real Complexity?
  - Selected Set of Application
- Benchmarks: SPEC, TPC, . . .
- CPU time: $T_1 =$ Dynamic instruction count $\times$ average CPI $\times$ Clock cycle time
- Speed-up: $S_p = \frac{T_1}{T_p} = \frac{T_1}{0.25T_1 + 0.75T1/P}$. Try $P = 3$ and $P = \infty$

Performance?

- 'Latency' or 'Thoughput'?
- How do we measure time?
  - Real application. Portability?
  - Kernel. Real Complexity?
  - Selected Set of Application
- Benchmarks: SPEC, TPC, . . .
- CPU time: $T_1 =$ Dynamic instruction count $\times$ average CPI $\times$ Clock cycle time
- Speed-up: $S_p = \frac{T_1}{T_p} = \frac{T_1}{0.25T_1 + 0.75T1/P}$. Try $P = 3$ and $P = \infty$

Performance?

- 'Latency' or 'Thoughput'?
- How do we measure time?
  - Real application. Portability?
  - Kernel. Real Complexity?
  - Selected Set of Application
- Benchmarks: SPEC, TPC, . . .
- CPU time: $T_1 =$ Dynamic instruction count $\times$ average CPI $\times$ Clock cycle time
- Speed-up: $S_p = \frac{T_1}{T_p} = \frac{T_1}{0.25T_1 + 0.75T1/P}$. Try $P = 3$ and $P = \infty$

Performance?

- 'Latency' or 'Thoughput'?
- How do we measure time?
  - Real application. Portability?
  - Kernel. Real Complexity?
  - Selected Set of Application
- Benchmarks: SPEC, TPC, . . .
- CPU time: $T_1 =$ Dynamic instruction count $\times$ average CPI $\times$ Clock cycle time
- Speed-up: $S_p = \frac{T_1}{T_p} = \frac{T_1}{0.25T_1 + 0.75T1/P}$. Try $P = 3$ and $P = \infty$

Performance?

- 'Latency' or 'Thoughput'?
- How do we measure time?
  - Real application. Portability?
  - Kernel. Real Complexity?
  - Selected Set of Application
- Benchmarks: SPEC, TPC, . . .
- CPU time: $T_1 =$ Dynamic instruction count $\times$ average CPI $\times$ Clock cycle time
- Speed-up: $S_p = \frac{T_1}{T_p} = \frac{T_1}{0.25T_1 + 0.75T1/P}$. Try $P = 3$ and $P = \infty$

The Upshot!

- Make the common case fast but remember that the uncommon case eventually sets the limit.
- You must have a balanced system where the resources are distributed according to where time is spent.
- Your system's performance must be above the required average! The peak will be reduced by dependencies and memory stalls.

The Upshot!

- Make the common case fast but remember that the uncommon case eventually sets the limit.
- You must have a balanced system where the resources are distributed according to where time is spent.
- Your system's performance must be above the required average! The peak will be reduced by dependencies and memory stalls.

The Upshot!

- Make the common case fast but remember that the uncommon case eventually sets the limit.
- You must have a balanced system where the resources are distributed according to where time is spent.
- Your system's performance must be above the required average! The peak will be reduced by dependencies and memory stalls.

How does the information move?

- By the rule of law. Each unit gets the inputs at a prescribed time and should deliver the output before a prescribed time. **Synchronous, with clocks.**
- By consensus. Tell me when you finish your part. **Asynchronous, with handshaking.**
- By its natural flow. Gates within a unit have a delay. Once the first level of gates ends its function, those gates start on new data while the second level of gates is processing the first data without extra signaling. **Wavepipelines**, must set a barrier somewhere!

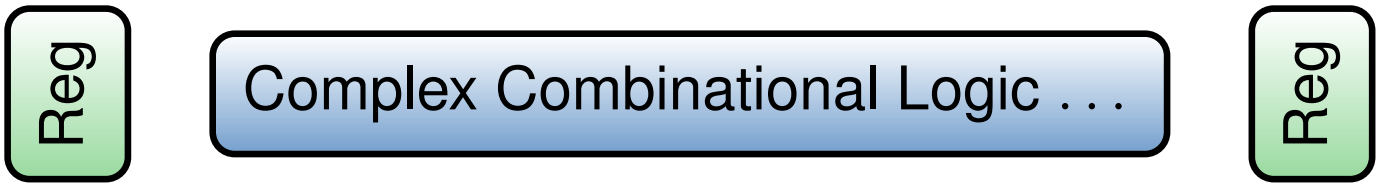
How does the information move?

- By the rule of law. Each unit gets the inputs at a prescribed time and should deliver the output before a prescribed time. **Synchronous, with clocks.**
- By consensus. Tell me when you finish your part. **Asynchronous, with handshaking.**
- By its natural flow. Gates within a unit have a delay. Once the first level of gates ends its function, those gates start on new data while the second level of gates is processing the first data without extra signaling. **Wavepipelines**, must set a barrier somewhere!

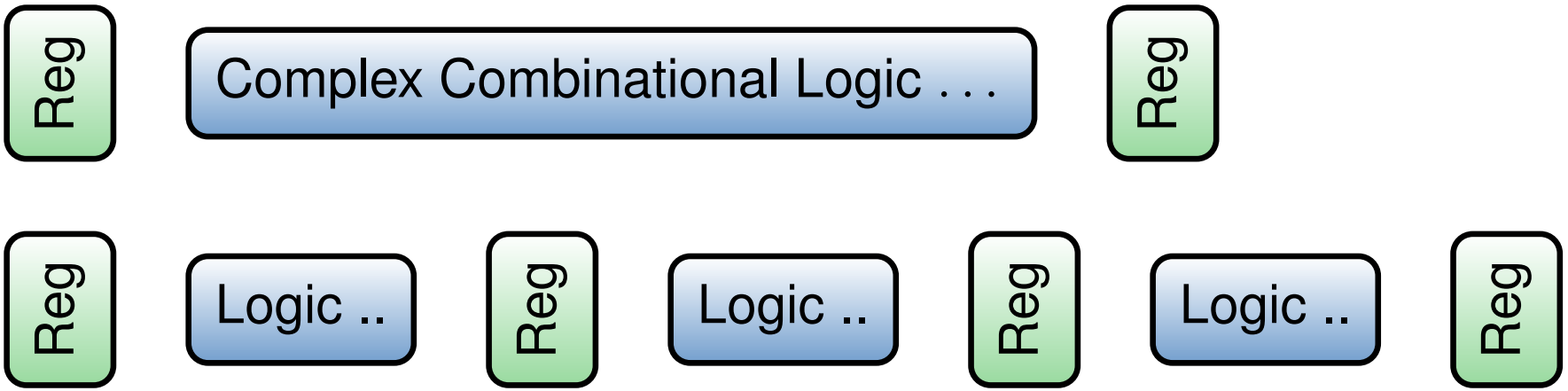
How does the information move?

- By the rule of law. Each unit gets the inputs at a prescribed time and should deliver the output before a prescribed time. **Synchronous, with clocks.**
- By consensus. Tell me when you finish your part. **Asynchronous, with handshaking.**
- By its natural flow. Gates within a unit have a delay. Once the first level of gates ends its function, those gates start on new data while the second level of gates is processing the first data without extra signaling. **Wavepipelines**, must set a barrier somewhere!

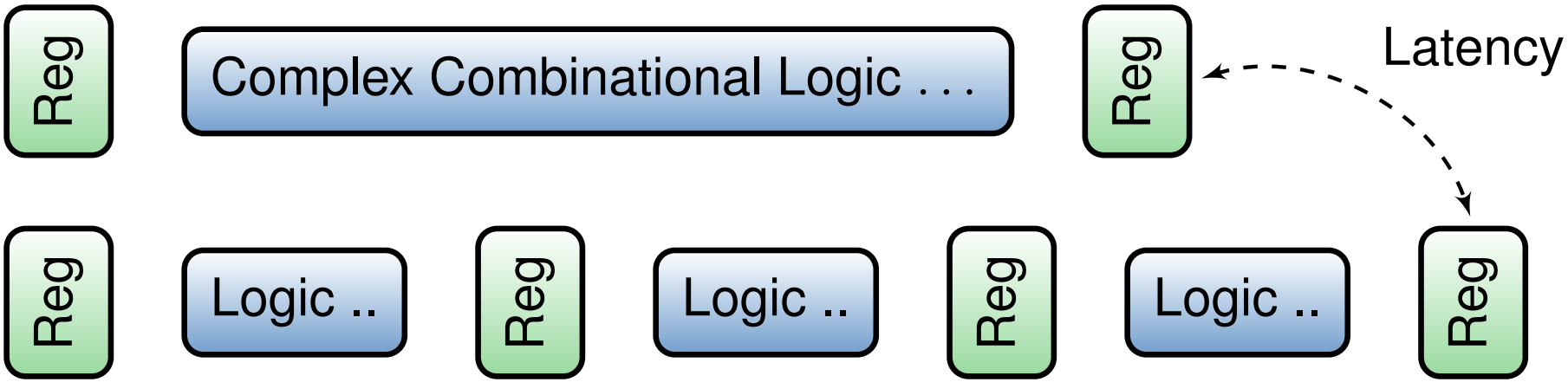
What is pipelining?



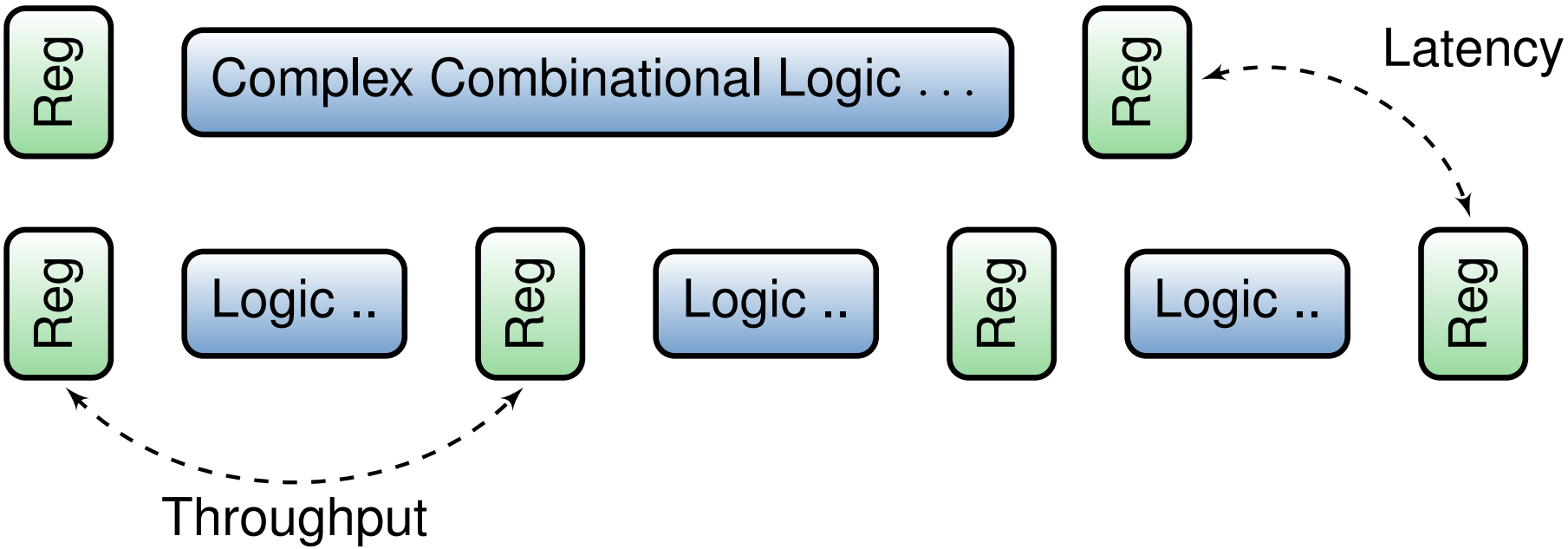
What is pipelining?



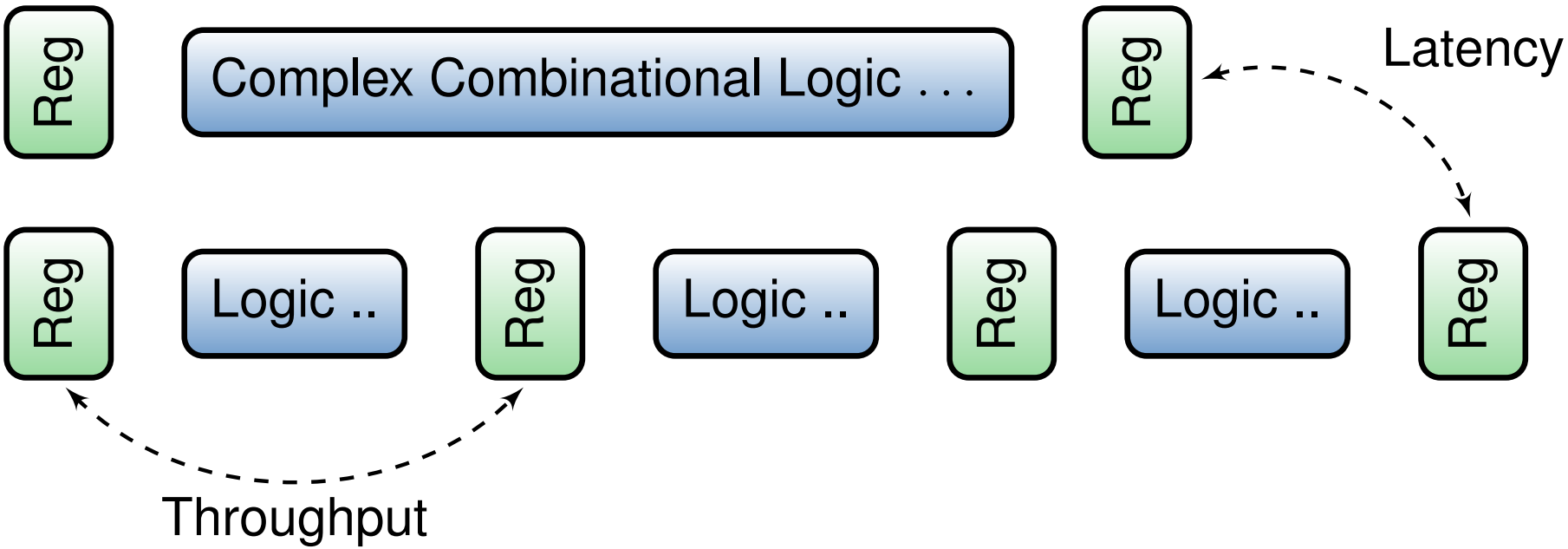
What is pipelining?



What is pipelining?

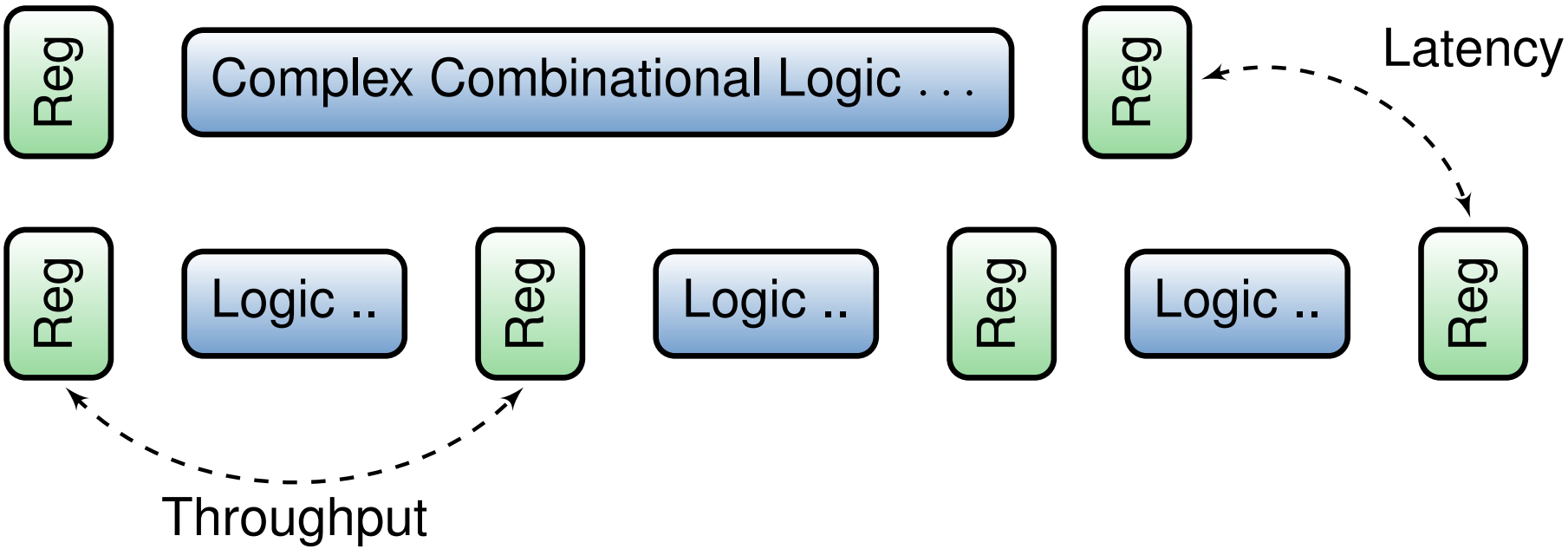


What is pipelining?



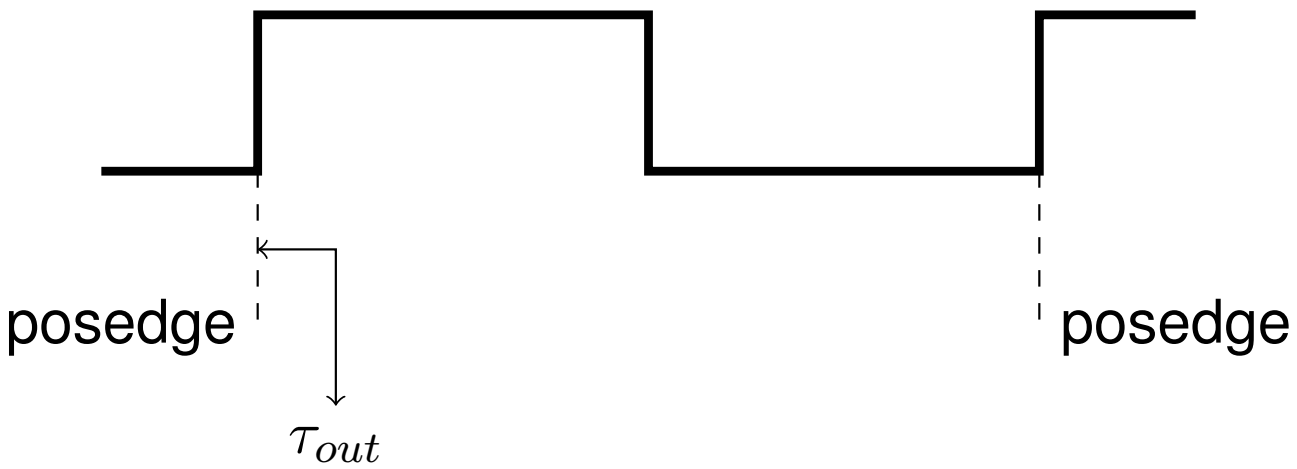
- Operational Frequency.

What is pipelining?

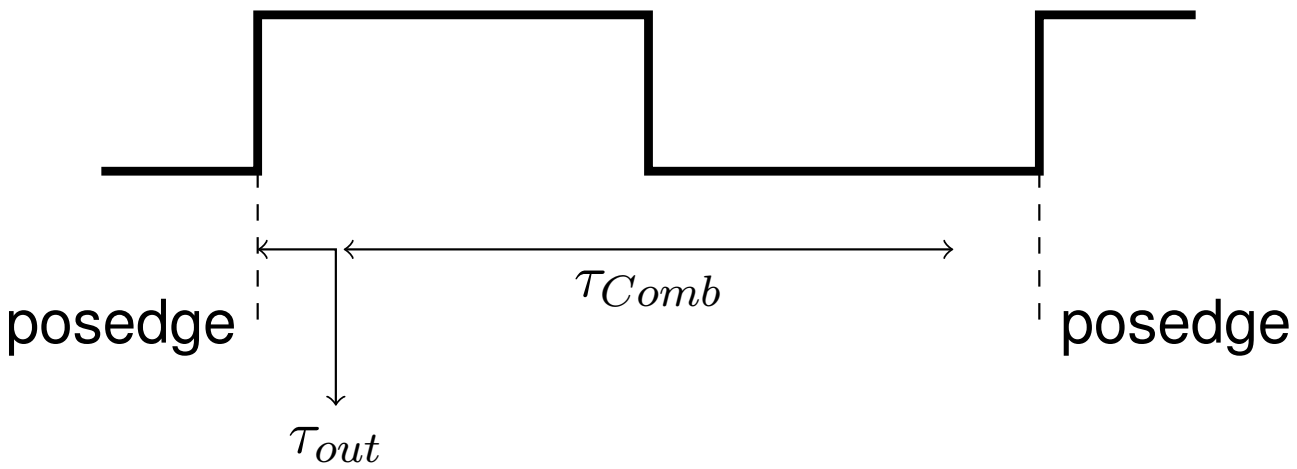


- Operational Frequency.
- Optimal number of stages?

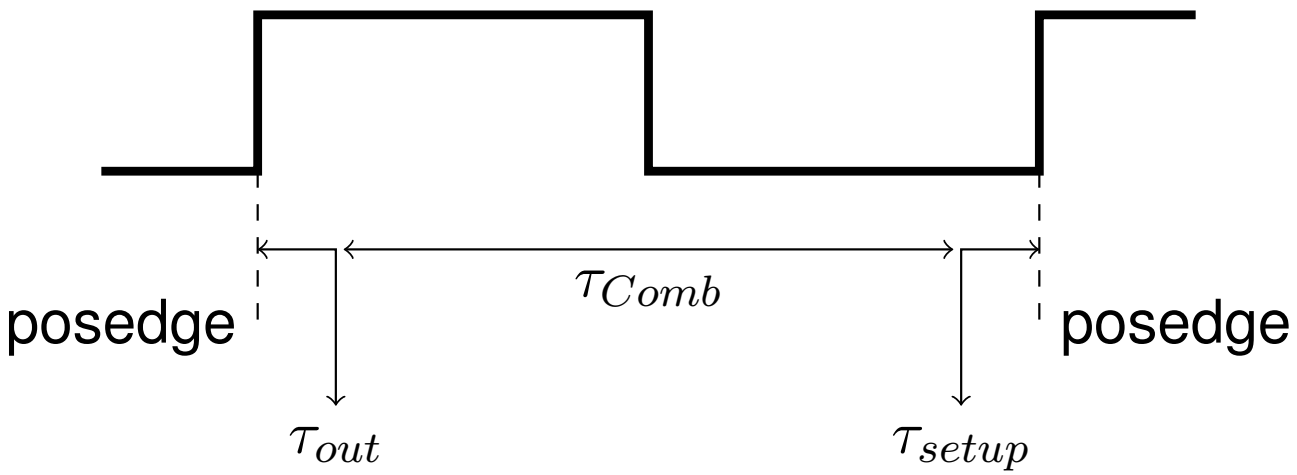
About clock edges



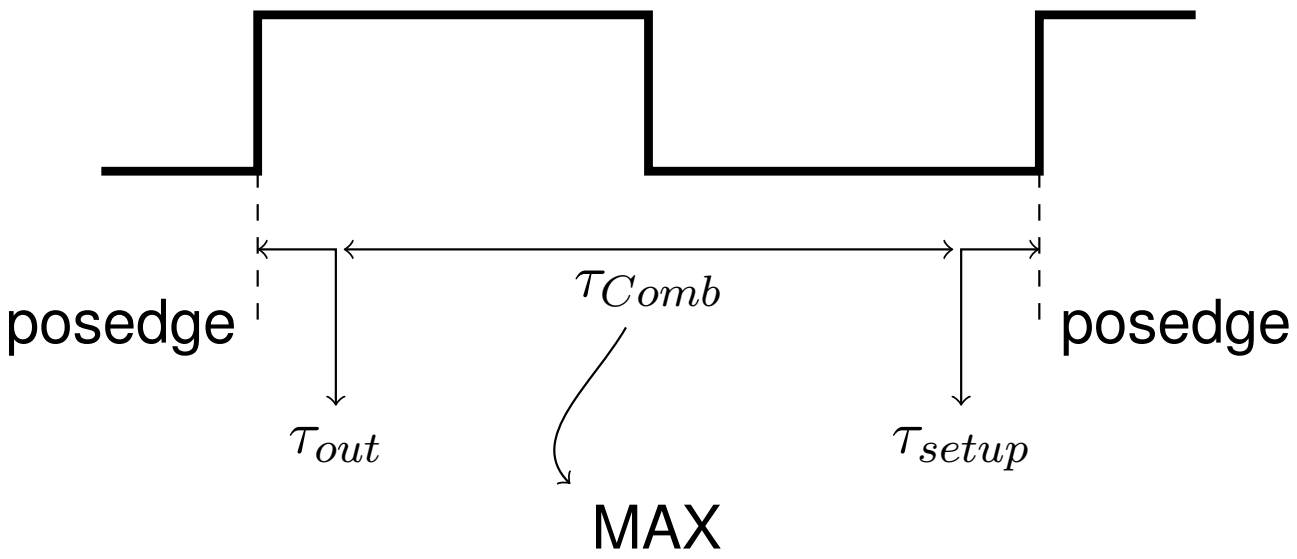
About clock edges



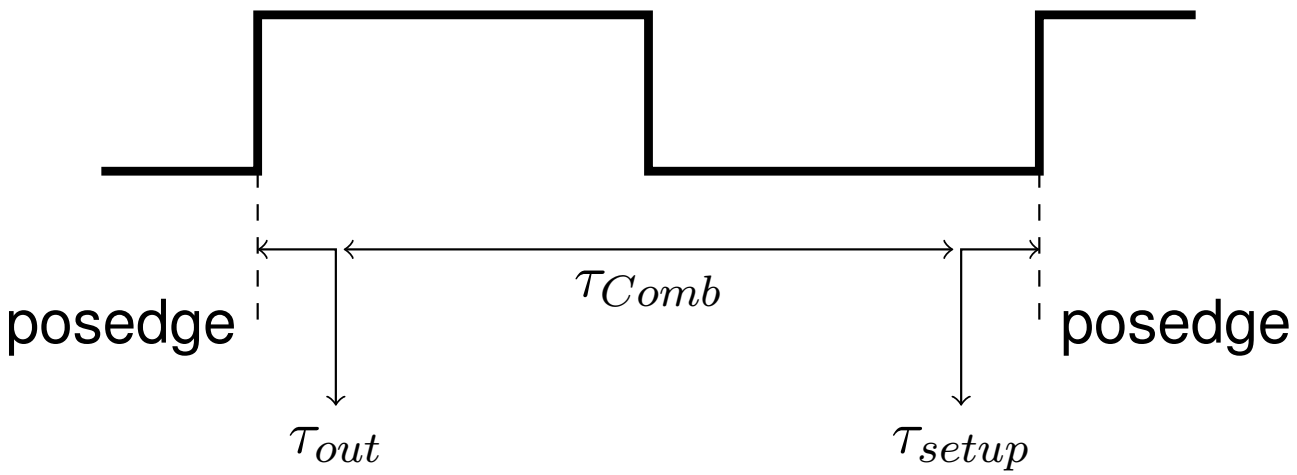
About clock edges



About clock edges

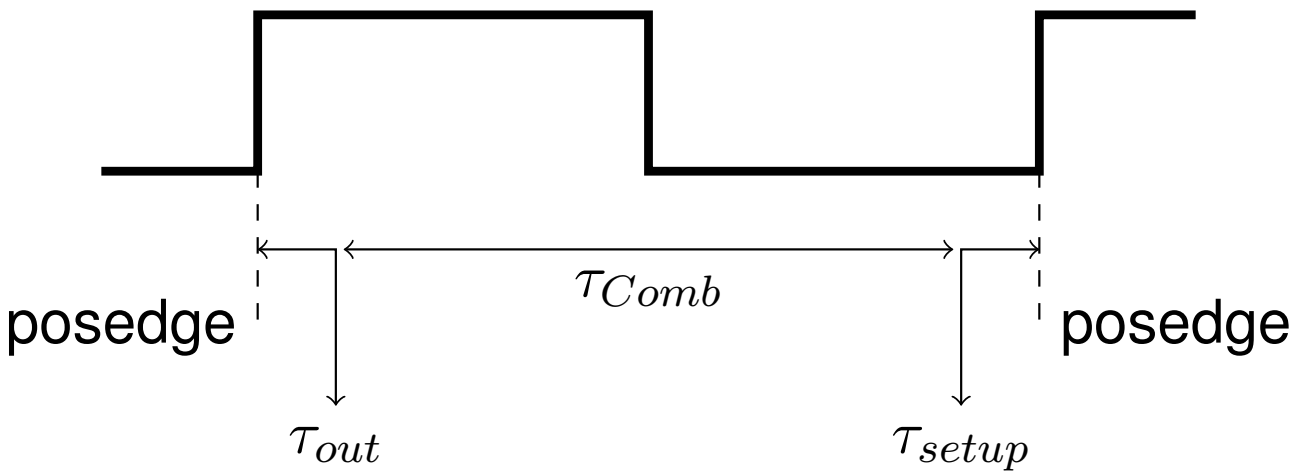


About clock edges



- τ_{Skew}

About clock edges



- τ_{Skew}
- If $\tau_{Comb-min} < \tau_{Skew}$?

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Pipelining a CPU

- Pipelines belong to the organization of the processor
- As we have seen, we need to analyze the instruction frequencies of the anticipated workload.
- The main stages of a processor are to fetch the instructions, execute them, and then to save the results. These may be divided into
  1. address generation for the instruction (IA),
  2. instruction fetch (IF),
  3. decode (D),
  4. address generation (AG),
  5. data fetch (DF),
  6. execution (EX), and
  7. put away (PA).
- Static, dynamic and multiple-issues pipelines

Effecting the CPI

- Ideal pipe operation ..

Cycle #	1	2	3	4	5
Ins # 1	IF	D	EX	PA	
Ins # 2		IF	D	EX	PA
Ins # 3			IF	D	EX

- A branch instruction ..

Ins # 1	IF	D	**EX**	PA	
Ins # 2		~~IF~~	~~D~~	~~EX~~	~~PA~~
Ins # 2'				IF	D

- Assuming the branch frequency is 15%, then the $CPI = 1 + 0.15 \times 2 = 1.3$
- Branch Prediction? 2-bit.

Effecting the CPI

- Ideal pipe operation ..

Cycle #	1	2	3	4	5
Ins # 1	IF	D	EX	PA	
Ins # 2		IF	D	EX	PA
Ins # 3			IF	D	EX

- A branch instruction ..

Ins # 1	IF	D	**EX**	PA	
Ins # 2		~~IF~~	~~D~~	~~EX~~	~~PA~~
Ins # 2'				IF	D

- Assuming the branch frequency is 15%, then the $CPI = 1 + 0.15 \times 2 = 1.3$
- Branch Prediction? 2-bit.

Effecting the CPI

- Ideal pipe operation ..

Cycle #	1	2	3	4	5
Ins # 1	IF	D	EX	PA	
Ins # 2		IF	D	EX	PA
Ins # 3			IF	D	EX

- A branch instruction ..

Ins # 1	IF	D	**EX**	PA	
Ins # 2		~~IF~~	~~D~~	~~EX~~	~~PA~~
Ins # 2'				IF	D

- Assuming the branch frequency is 15%, then the $CPI = 1 + 0.15 \times 2 = 1.3$
- Branch Prediction? 2-bit.

Effecting the CPI

- Ideal pipe operation ..

Cycle #	1	2	3	4	5
Ins # 1	IF	D	EX	PA	
Ins # 2		IF	D	EX	PA
Ins # 3			IF	D	EX

- A branch instruction ..

Ins # 1	IF	D	**EX**	PA	
Ins # 2		~~IF~~	~~D~~	~~EX~~	~~PA~~
Ins # 2'				IF	D

- Assuming the branch frequency is 15%, then the $CPI = 1 + 0.15 \times 2 = 1.3$
- Branch Prediction? 2-bit.

Data Hazards

- RAW, WAW and WAR. RAR?
- Forward

Cycle #	1	2	3	4	5
$Add\ \ R5 \leftarrow R2, R1$	IF	D	EX	PA	
$Add\ \ R4 \leftarrow R5, R3$		IF	D	EX	PA

- Stalls

Cycle #	1	2	3	4	5
$LW\ R5 \leftarrow ()$	IF	D	EX	PA	
$Add R4 \leftarrow R5, R3$		IF	D	–	EX
Ins # 3			IF	–	D
Ins # 4				–	IF

Data Hazards

- RAW, WAW and WAR. RAR?
- Forward

Cycle #	1	2	3	4	5
$Add\ \ R5 \leftarrow R2, R1$	IF	D	EX	PA	
$Add\ \ R4 \leftarrow R5, R3$		IF	D	EX	PA

- Stalls

Cycle #	1	2	3	4	5
$LW\ R5 \leftarrow ()$	IF	D	EX	PA	
$Add R4 \leftarrow R5, R3$		IF	D	–	EX
Ins # 3			IF	–	D
Ins # 4				–	IF

Data Hazards

- RAW, WAW and WAR. RAR?
- Forward

Cycle #	1	2	3	4	5
$Add\ \ R5 \leftarrow R2, R1$	IF	D	EX	PA	
$Add\ \ R4 \leftarrow R5, R3$		IF	D	EX	PA

- Stalls

Cycle #	1	2	3	4	5
$LW\ R5 \leftarrow ()$	IF	D	EX	PA	
$Add R4 \leftarrow R5, R3$		IF	D	–	EX
Ins # 3			IF	–	D
Ins # 4				–	IF

Data Hazards .. cnt'd

- Multi-cycle execution. In-order completion?

Cycle #	1	2	3	4	5	6	7
Ins # 1	IF	D	EX	EX	EX	PA	
Ins # 2		IF	D	EX	PA		
Ins # 3			IF	D	EX	EX	PA

- Register Renaming

```
Lw  R1
Div  R5  <- R1,R2
Add  R1 <- R3,R4
Mul  R0 <- R1,R7
```

  - Rename $R1$ in Instructions # 3,4 to $R6$

- Dynamic scheduling

Data Hazards .. cnt'd

- Multi-cycle execution. In-order completion?

Cycle #	1	2	3	4	5	6	7
Ins # 1	IF	D	EX	EX	EX	PA	
Ins # 2		IF	D	EX	PA		
Ins # 3			IF	D	EX	EX	PA

- Register Renaming

```
Lw  R1
Div  R5  <- R1,R2
Add  R1 <- R3,R4
Mul  R0 <- R1,R7
```

  - Rename $R1$ in Instructions # 3,4 to $R6$

- Dynamic scheduling

Data Hazards .. cnt'd

- Multi-cycle execution. In-order completion?

Cycle #	1	2	3	4	5	6	7
Ins # 1	IF	D	EX	EX	EX	PA	
Ins # 2		IF	D	EX	PA		
Ins # 3			IF	D	EX	EX	PA

- Register Renaming

```
Lw  R1
Div  R5  <- R1,R2
Add  R1 <- R3,R4
Mul  R0 <- R1,R7
```

  - Rename $R1$ in Instructions # 3,4 to $R6$

- Dynamic scheduling

Data Hazards .. cnt'd

- Multi-cycle execution. In-order completion?

Cycle #	1	2	3	4	5	6	7
Ins # 1	IF	D	EX	EX	EX	PA	
Ins # 2		IF	D	EX	PA		
Ins # 3			IF	D	EX	EX	PA

- Register Renaming

```
Lw  R1
Div  R5  <- R1,R2
Add  R1 <- R3,R4
Mul  R0 <- R1,R7
```

  - Rename $R1$ in Instructions # 3,4 to $R6$

- Dynamic scheduling

ILP: Exploring around

- Scoreboard (control flow) and the Tomasulo (data flow)
- Super-scalar: N^2 Dependencies and buses. Branch Prediction?
- Alternatives: Compiler loop unrolling and renaming
- VLIW (More than a super-scalar)
- Schedule (order) and Issue (start)

	Schedule	Issue
Static	HW	HW
Dynamic (out-of-order)	HW	HW
In-Order Superscalar	SW	HW
Pure VLIW	SW	SW

ILP: Exploring around

- Scoreboard (control flow) and the Tomasulo (data flow)
- Super-scalar: N^2 Dependencies and buses. Branch Prediction?
- Alternatives: Compiler loop unrolling and renaming
- VLIW (More than a super-scalar)
- Schedule (order) and Issue (start)

	Schedule	Issue
Static	HW	HW
Dynamic (out-of-order)	HW	HW
In-Order Superscalar	SW	HW
Pure VLIW	SW	SW

ILP: Exploring around

- Scoreboard (control flow) and the Tomasulo (data flow)
- Super-scalar: N^2 Dependencies and buses. Branch Prediction?
- Alternatives: Compiler loop unrolling and renaming
- VLIW (More than a super-scalar)
- Schedule (order) and Issue (start)

	Schedule	Issue
Static	HW	HW
Dynamic (out-of-order)	HW	HW
In-Order Superscalar	SW	HW
Pure VLIW	SW	SW

ILP: Exploring around

- Scoreboard (control flow) and the Tomasulo (data flow)
- Super-scalar: N^2 Dependencies and buses. Branch Prediction?
- Alternatives: Compiler loop unrolling and renaming
- VLIW (More than a super-scalar)
- Schedule (order) and Issue (start)

	Schedule	Issue
Static	HW	HW
Dynamic (out-of-order)	HW	HW
In-Order Superscalar	SW	HW
Pure VLIW	SW	SW

ILP: Exploring around

- Scoreboard (control flow) and the Tomasulo (data flow)
- Super-scalar: N^2 Dependencies and buses. Branch Prediction?
- Alternatives: Compiler loop unrolling and renaming
- VLIW (More than a super-scalar)
- Schedule (order) and Issue (start)

	Schedule	**Issue**
Static	HW	HW
Dynamic (out-of-order)	HW	HW
In-Order Superscalar	SW	HW
Pure VLIW	SW	SW

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

More into VLIW

- Pros:
  - Simple HW, and
  - higher performance
- Cons:
  - Complex organization disposed to compilers,
  - Porting (Transmeta), and
  - Variables Cache effect
  - NOPs
- GPUs, Itanium ...

Vector Processors

- SIMD. MIMd = VLIW ?
- Performance
  - The amount of the program expressed in a vectorizable form
  - Vector startup costs. Length?
  - Chaining Support
  - Simultaneous Access to/from Memory
  - # of Vector registers
- Typical Speedup: $P_s \leq 4$ (Chaining boosts to $P_s \leq 7$)

Vector Processors

- SIMD. MIMd = VLIW ?
- Performance
  - The amount of the program expressed in a vectorizable form
  - Vector startup costs. Length?
  - Chaining Support
  - Simultaneous Access to/from Memory
  - # of Vector registers
- Typical Speedup: $P_s \leq 4$ (Chaining boosts to $P_s \leq 7$)

Vector Processors

- SIMD. MIMd = VLIW ?
- Performance
  - The amount of the program expressed in a vectorizable form
    - Vector startup costs. Length?
    - Chaining Support
    - Simultaneous Access to/from Memory
    - # of Vector registers
- Typical Speedup: $P_s \leq 4$ (Chaining boosts to $P_s \leq 7$)

Vector Processors

- SIMD. MIMd = VLIW ?
- Performance
  - The amount of the program expressed in a vectorizable form
  - Vector startup costs. Length?
  - Chaining Support
  - Simultaneous Access to/from Memory
  - # of Vector registers
- Typical Speedup: $P_s \leq 4$ (Chaining boosts to $P_s \leq 7$)

Vector Processors

- SIMD. MIMd = VLIW ?
- Performance
  - The amount of the program expressed in a vectorizable form
  - Vector startup costs. Length?
  - Chaining Support
  - Simultaneous Access to/from Memory
  - # of Vector registers
- Typical Speedup: $P_s \leq 4$ (Chaining boosts to $P_s \leq 7$)

Vector Processors

- SIMD. MIMd = VLIW ?
- Performance
  - The amount of the program expressed in a vectorizable form
  - Vector startup costs. Length?
  - Chaining Support
  - Simultaneous Access to/from Memory
  - # of Vector registers
- Typical Speedup: $P_s \leq 4$ (Chaining boosts to $P_s \leq 7$)

Vector Processors

- SIMD. MIMd = VLIW ?
- Performance
  - The amount of the program expressed in a vectorizable form
  - Vector startup costs. Length?
  - Chaining Support
  - Simultaneous Access to/from Memory
  - # of Vector registers
- Typical Speedup: $P_s \leq 4$ (Chaining boosts to $P_s \leq 7$)

Vector Processors

- SIMD. MIMd = VLIW ?
- Performance
  - The amount of the program expressed in a vectorizable form
  - Vector startup costs. Length?
  - Chaining Support
  - Simultaneous Access to/from Memory
  - # of Vector registers
- Typical Speedup: $P_s \leq 4$ (Chaining boosts to $P_s \leq 7$)

Vector Processors ... Performance

- Vector versus multiple issue (superscalar)

	Vector	**Multiple Issue**
Pros	good S_p on large scientific loads	good S_p on small problems general purpose
Cons	limited to regular data Vector Registers overhead requires a high memory BW	complex scheduling large D cache inefficient use of ALUs

Thread Level Parallelism

- ILP have stalled since the late-1990s
- TLP?
  - The Block Multi-threading
  - Interleaved Multi-threading. GPUs?
  - Multi-cores?
  - Simultaneous Multi-threading (with superscalars)
- Maximum typical threads

Thread Level Parallelism

- ILP have stalled since the late-1990s
- TLP?
  - The Block Multi-threading
  - Interleaved Multi-threading. GPUs?
    - Multi-cycles?
  - Simultaneous Multi-threading (with superscalars)
- Maximum typical threads

Thread Level Parallelism

- ILP have stalled since the late-1990s
- TLP?
  - The Block Multi-threading
  - Interleaved Multi-threading. GPUs?
    - Multi-cycles?
  - Simultaneous Multi-threading (with superscalars)
- Maximum typical threads

Thread Level Parallelism

- ILP have stalled since the late-1990s
- TLP?
  - The Block Multi-threading
  - Interleaved Multi-threading. GPUs?
    - Multi-cycles?
  - Simultaneous Multi-threading (with superscalars)
- Maximum typical threads

Thread Level Parallelism

- ILP have stalled since the late-1990s
- TLP?
  - The Block Multi-threading
  - Interleaved Multi-threading. GPUs?
    - Multi-cycles?
  - Simultaneous Multi-threading (with superscalars)
- Maximum typical threads

Thread Level Parallelism

- ILP have stalled since the late-1990s
- TLP?
  - The Block Multi-threading
  - Interleaved Multi-threading. GPUs?
    - Multi-cycles?
  - Simultaneous Multi-threading (with superscalars)
- Maximum typical threads

Thread Level Parallelism

- ILP have stalled since the late-1990s
- TLP?
  - The Block Multi-threading
  - Interleaved Multi-threading. GPUs?
    - Multi-cycles?
  - Simultaneous Multi-threading (with superscalars)
- Maximum typical threads

Why do we have multiple processors?

- The large problems exceed the capacity of the largest processors and using a few (or many!) in parallel could help.
- The chip area available is better used to support multiple cores than to just increase the cache size and levels!
- Some environments are inherently “parallel”. Search engines?
- Partitioning, scheduling and synchronization (cache coherency)

Why do we have multiple processors?

- The large problems exceed the capacity of the largest processors and using a few (or many!) in parallel could help.
- The chip area available is better used to support multiple cores than to just increase the cache size and levels!
- Some environments are inherently “parallel”. Search engines?
- Partitioning, scheduling and synchronization (cache coherency)

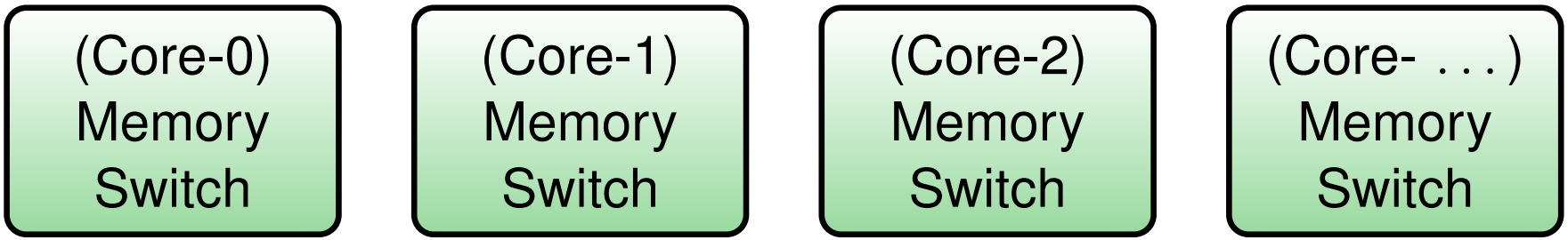
Why do we have multiple processors?

- The large problems exceed the capacity of the largest processors and using a few (or many!) in parallel could help.
- The chip area available is better used to support multiple cores than to just increase the cache size and levels!
- Some environments are inherently “parallel”. Search engines?
- Partitioning, scheduling and synchronization (cache coherency)

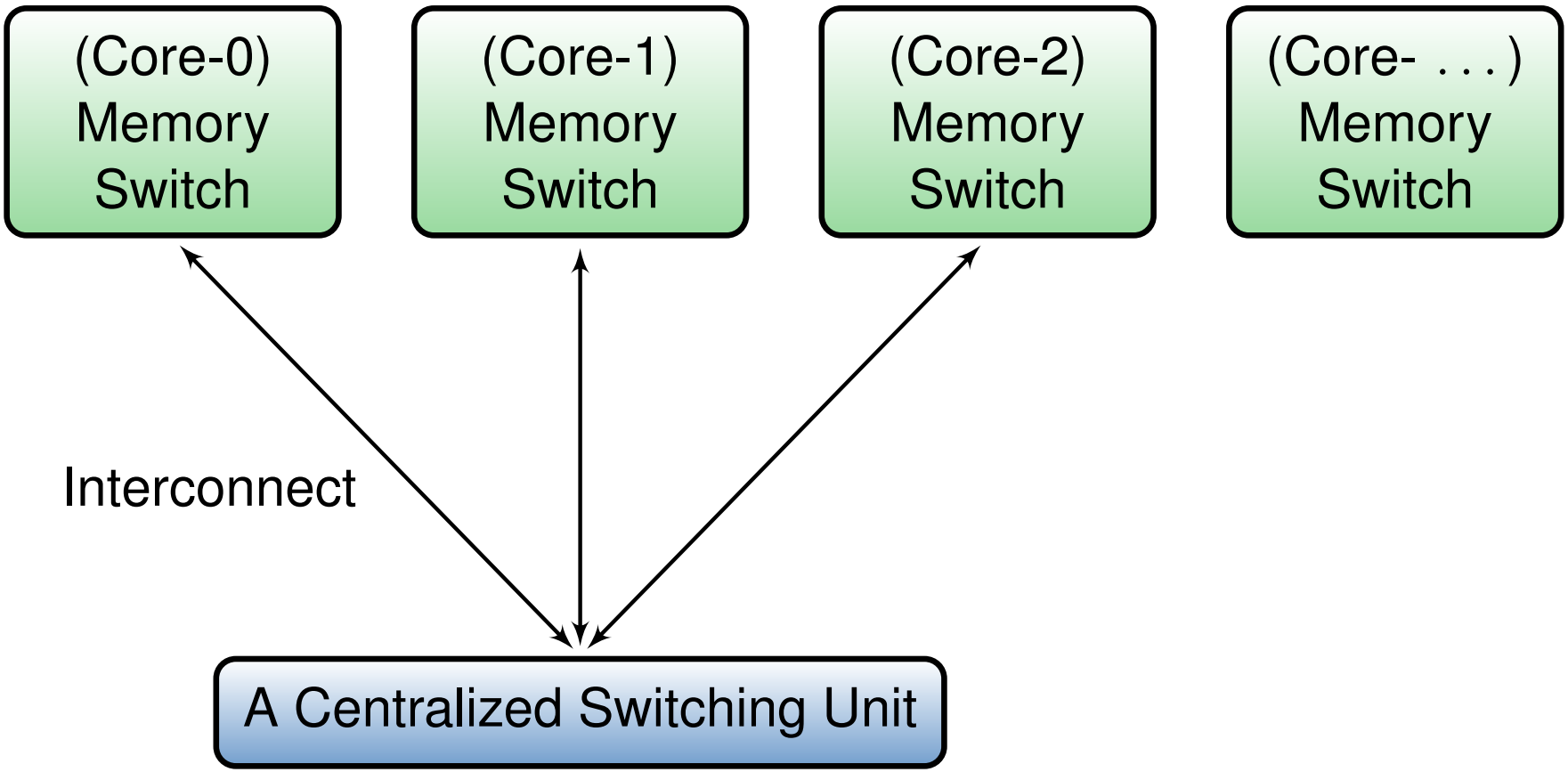
Why do we have multiple processors?

- The large problems exceed the capacity of the largest processors and using a few (or many!) in parallel could help.
- The chip area available is better used to support multiple cores than to just increase the cache size and levels!
- Some environments are inherently “parallel”. Search engines?
- Partitioning, scheduling and synchronization (cache coherency)

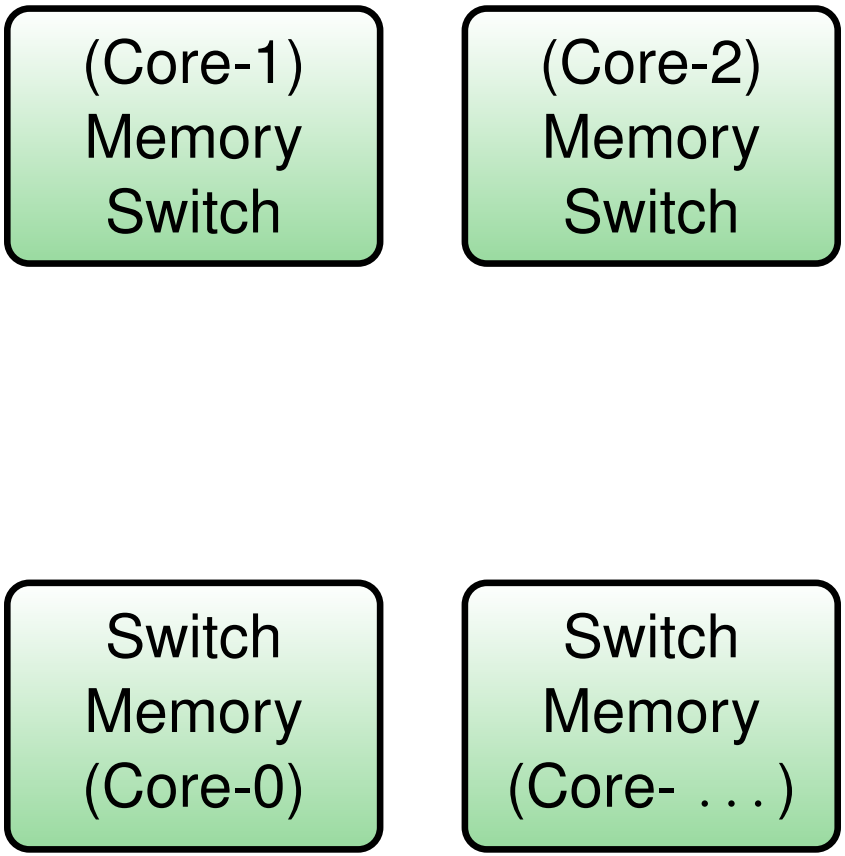
How to connect multiple processors?



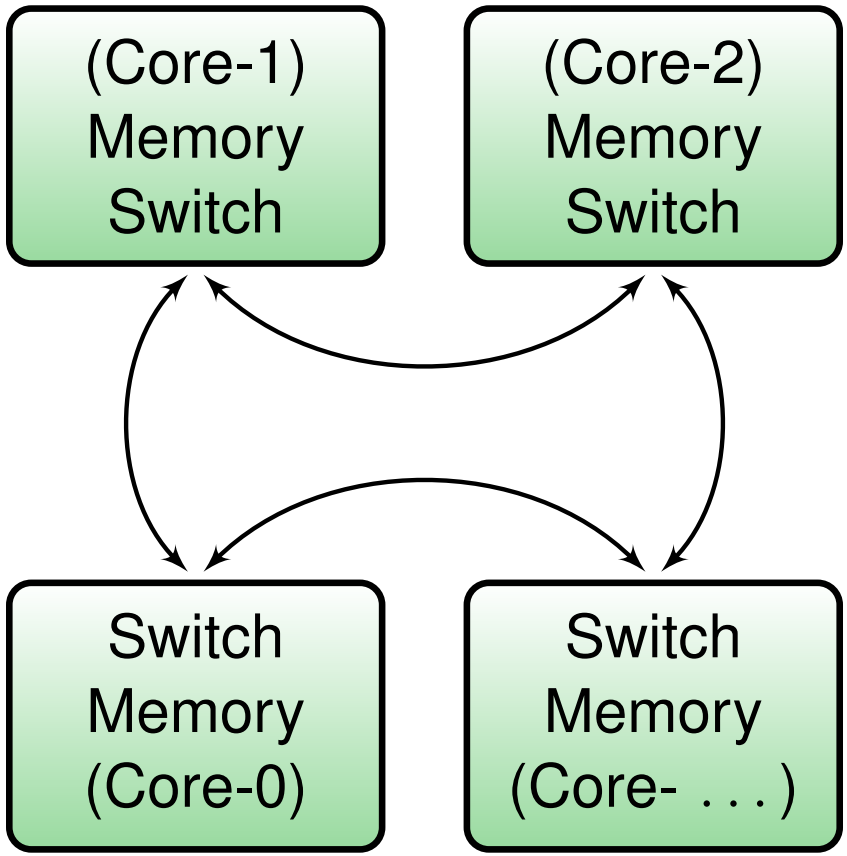
How to connect multiple processors?



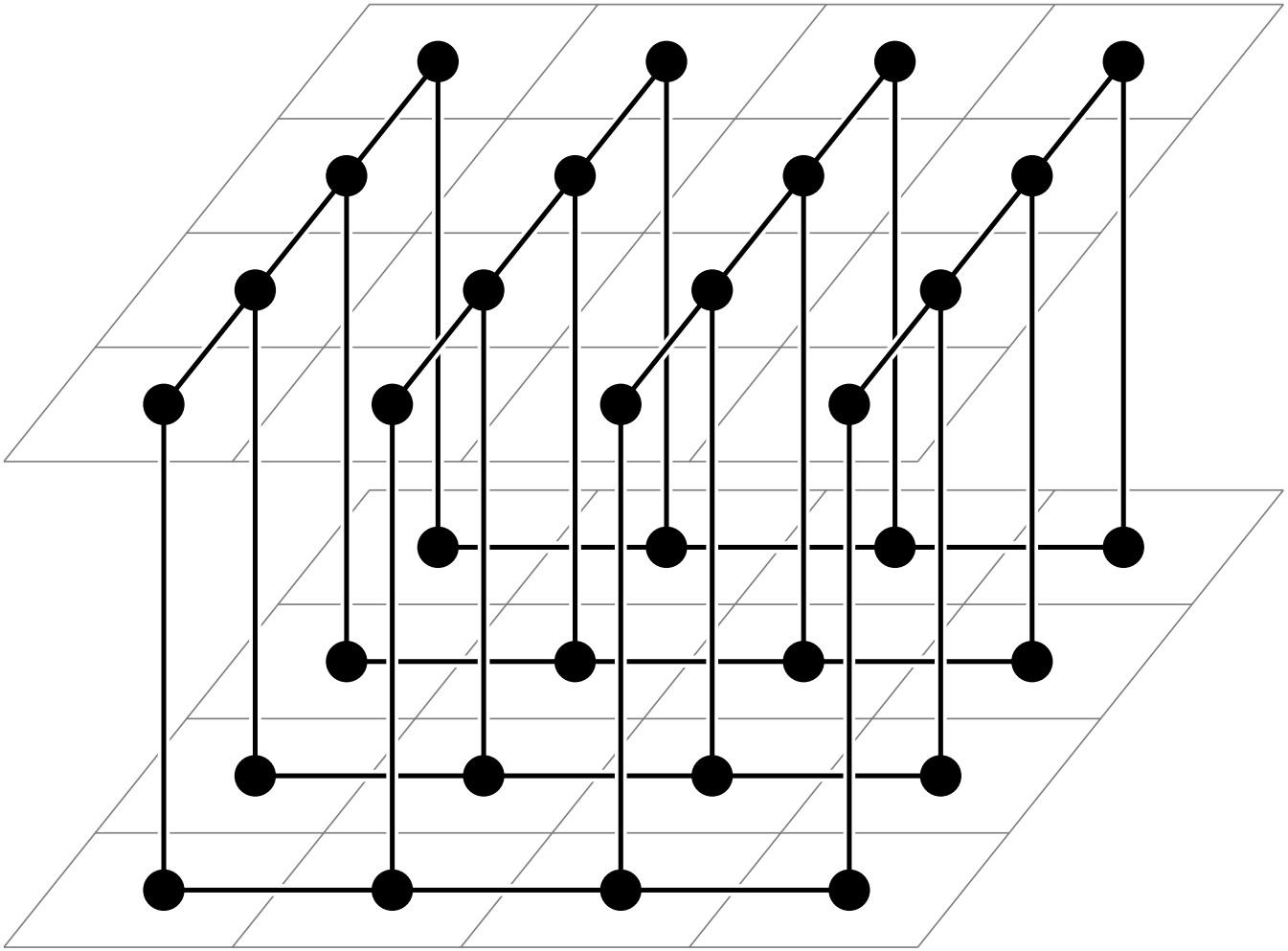
How to connect multiple processors?



How to connect multiple processors?



Scaling up with interconnects



Let's Sum Up

- Set your design goal; functionality, performance, power and price
- Structure and Organization
- Make the common case fast, and distribute the resources accordingly
- Instructions and threads level dependencies and parallelism

Let's Sum Up

- Set your design goal; functionality, performance, power and price
- Structure and Organization
- Make the common case fast, and distribute the resources accordingly
- Instructions and threads level dependencies and parallelism

Let's Sum Up

- Set your design goal; functionality, performance, power and price
- Structure and Organization
- Make the common case fast, and distribute the resources accordingly
- Instructions and threads level dependencies and parallelism

Let's Sum Up

- Set your design goal; functionality, performance, power and price
- Structure and Organization
- Make the common case fast, and distribute the resources accordingly
- Instructions and threads level dependencies and parallelism

☺uestions .. ?

Tarek.Eldeeb@silminds.com