Advance Computer Architecture and x86 ISA

University of Science and Technology of Hanoi

MS. LE Nhu Chu Hiep

Personal Introduction

Personal Introduction: Basic Information

Contact:

- Name: LE Nhu Chu Hiep
- Email: <le-nhu-chu.hiep@usth.edu.vn>

Education:

- Master for ICT at USTH (2023)
    - Speciality: Data Mining for IoT
- Bachelor for ICT at USTH (2020)

Personal Introduction: Teaching Career

Teaching:

- Lecturer of ICTLab USTH from 2024 (I’m young !!!)
- Major: Cyber Security
- Teaching courses:
  - Distributed System (ICT3)
  - Advance Computer Architecture and x86 ISA (CS2)

Personal Introduction: Research Career

Research:

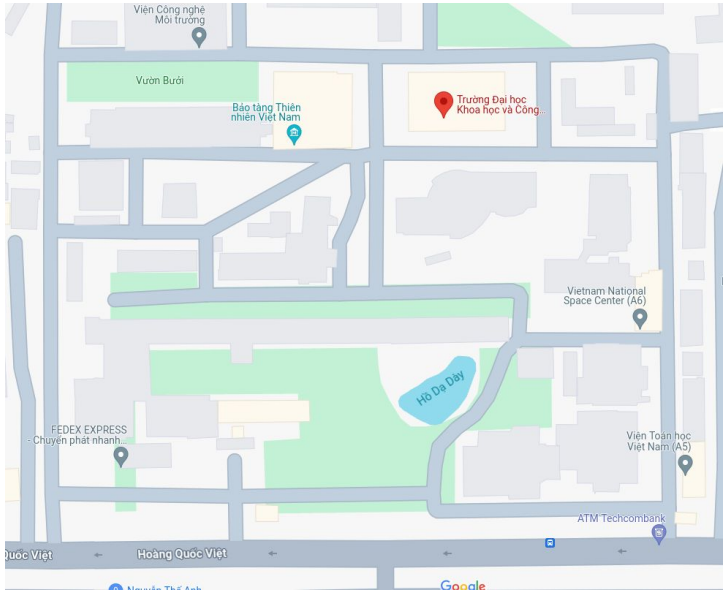
- Interested Domain:
  - Operating System
  - Networking
  - And Distributed System
- Research Topics:
  - Ulake: Microservice based data lake framework retrieving, storing, and querying scientific data
  - Pswap: Migration framework to move stateful container from VM to Host and vice versa

Personal Introduction: Working at ICTLab USTH

ICT Department:

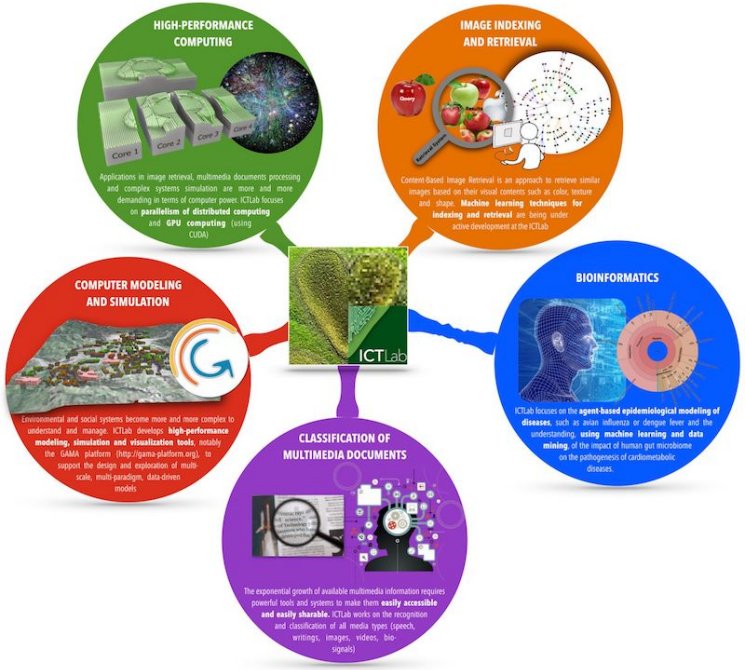
- Role: Researcher and Lecturer
- Website: https://ictlab.usth.edu.vn (**OLD** !!!)
- Location: 408, A21, 18 Hoang Quoc Viet,

  Nghia Do, Cau Giay, Hanoi



Personal Introduction: Working at ICTLab USTH (cont.)

- **Machine Learning, Deep Learning and Data Mining**
- Image and Speech Processing
- Modeling and Simulation
- **Sensor Networks and Embedded Systems**
- **High Performance Computing**
- Health Informatics and BioInformatics

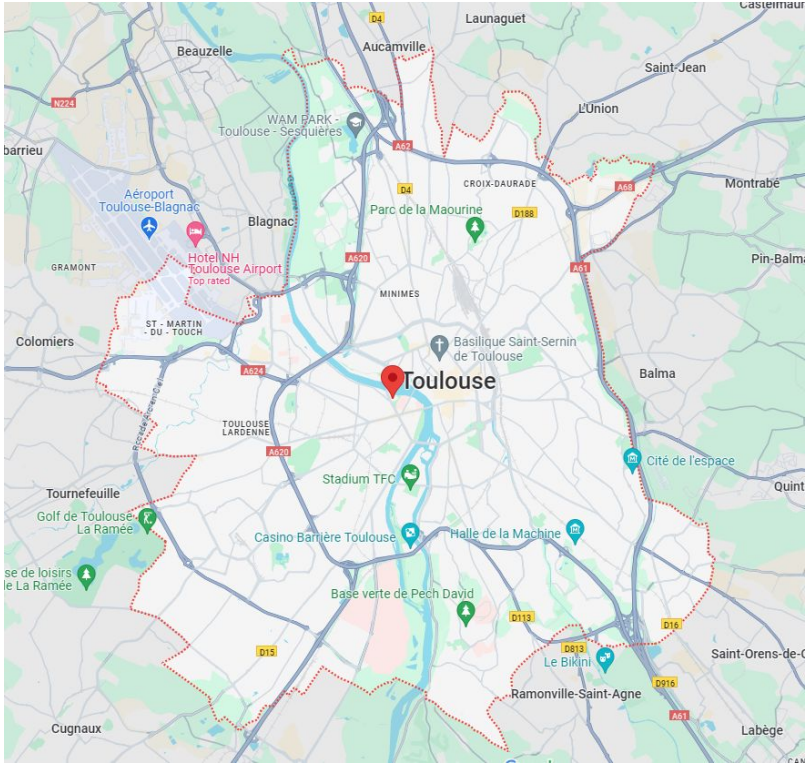


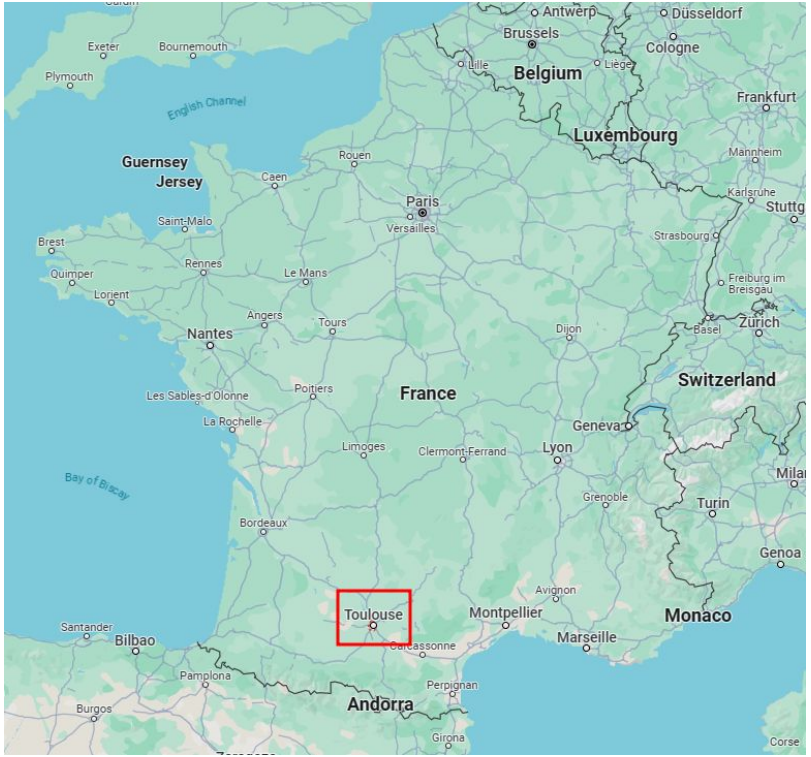
Personal Introduction: Working at INP ENSEEIHT

- National Polytechnic Institute of Toulouse [1]
- Role: M2 Internship ~ 6 months
- Ranking: **#943** in Best GLobal Universities [2]
- Website: https://www.enseeiht.fr/fr/index.html
  (Not **OLD**, but **FRENCH**)
- Location: 2 Rue Charles Camichel, 31000 Toulouse, France

1. C. (n.d.). *Ingénieur N7, Créateur du monde de demain*. Ametys V3. https://www.enseeiht.fr/fr/index.html
2. See where Institut National Polytechnique de Toulouse ranks among the world's Best Universities. (n.d.-a). https://www.usnews.com/education/best-global-universities/institut-national-polytechnique-de-toulouse-505784

Personal Introduction: Working at INP ENSEEIHT (cont.)

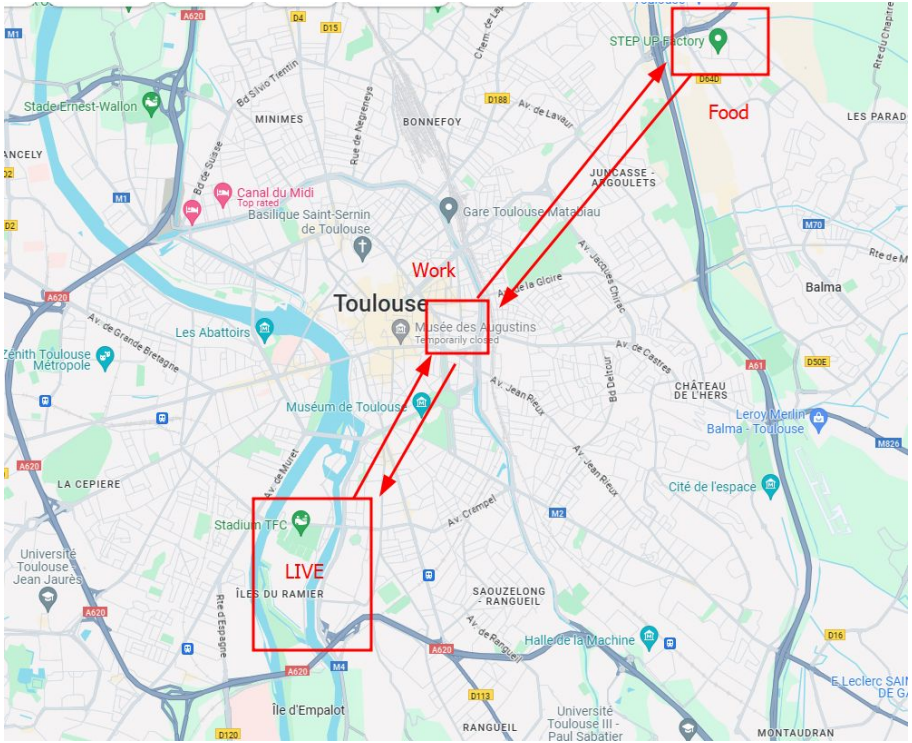




Personal Introduction: Working at INP ENSEEIHT (cont.)

- Top ranking universities (**INP**)
- Air France headquarter
- Old city with beautiful buildings
- Cheap cost of living
- Peaceful (**Not like Paris !!!**)
- Connect to USTH

Personal Introduction: Working at INP ENSEEIHT (cont.)



Course Introduction

Course Introduction

- 4 ETCS (40 hours)
  - 24h Lecturers - 16h Practical
  - Lecturers: 8 classes, 3 hours / class
  - Practicals: 4 classes, 4 hours / class
- Prerequisites:
  - Basic Programming
  - Computer Architecture
- Assessment:
  - Attendance: 10%
  - Assignment: 10%
  - Midterm: 30% (Project)
  - Final: 50% (Moodle Exam / Writing Exam)

Course Introduction: Objectives

- Understand basic concept of computer architecture (CA)
- Understand CA advance techniques to improve instruction execution
- Learn basic concept of x86 ISA
- Apply learned knowledge to improve instruction execution

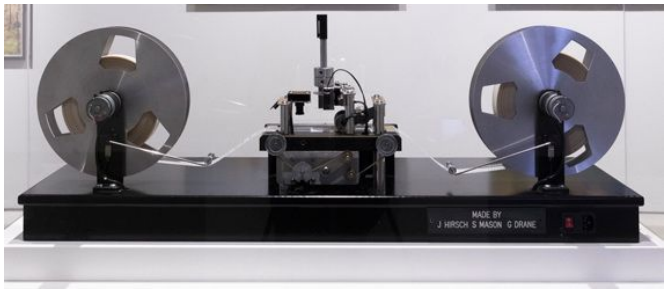
Course Introduction: Textbooks

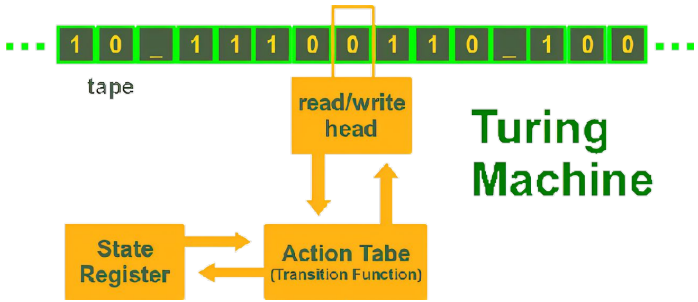
- **John L. Hennessy & David A. Patterson, Computer architecture : a quantitative approach, 5th Edition, Morgan Kaufmann, 2011**
- Andrew Tanenbaum & Todd Austin, Structured Computer Organization, 6th Edition, Pearson, 2012
- William Stallings, Computer organization and architecture, 10th Edition, 2015
- Barry Wilkinson & Michael Allen, Parallel programming, techniques and applications using networked workstations and parallel computers,Pearson, 2004

Computer Architecture

Computer

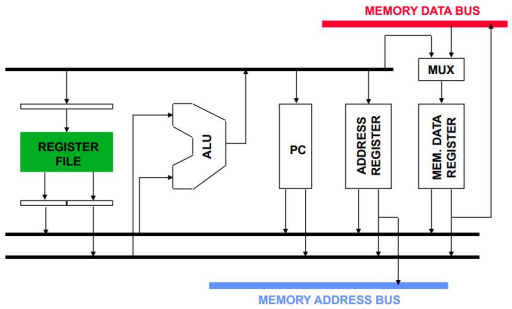
- A **digital system**
  - The finite state machine
  - The combination and sequential logic
  - The register and memory
- Purpose
  - Do whatever the **software** tell it to do
  - **Software** is a series of instructions

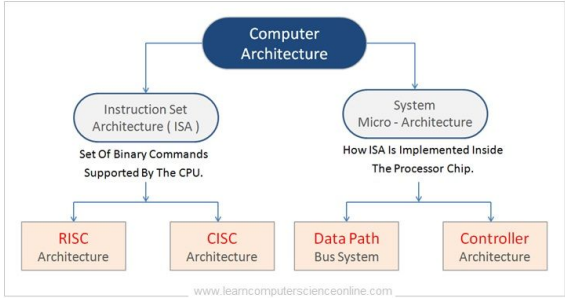


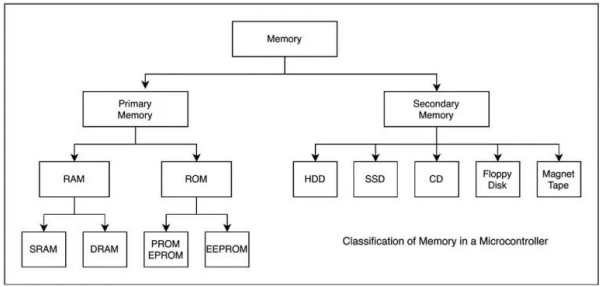


Computer Architecture

- Conceptual structure and functional behavior of computer
- Defines
  - Hardware **components** and their **organization**
  - Interaction among components to execute the program instructions
- Includes
  - **Instruction set architecture (ISA)**
  - **Memory organization**
  - Data paths and control mechanism
  - IO systems



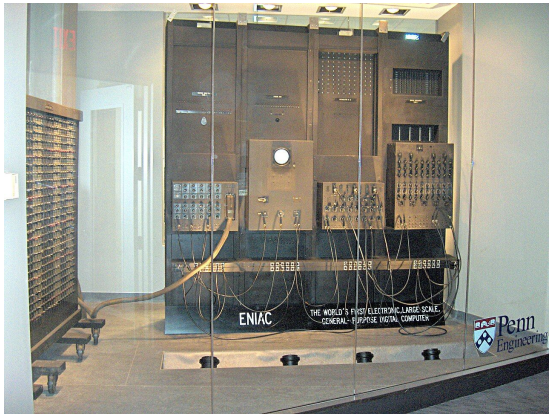




Computer History

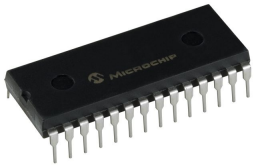
Computer History: Early Computer

- **1940s - 1950s**: First general purpose computers using vacuum tubes and punched cards
- **ENIAC**: chiefly served the military like calculating artillery firing table
- **UNIVAC**: meant for scientific and business purpose
- **Authors**: **Dr. Presper Eckert** and **Dr. John Mauchly**
- **Interface**: IBM card reader and card punch



Computer History: Commercial Computer

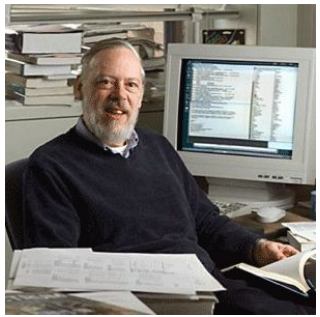
- **1950s - 1960s**: Introduction of transistors, leading to smaller, faster, and more reliable computers
- **1960s - 1970s**: Emergence of integrated circuits (ICs) and microprocessors
- **1970s - 1980s**: Rise of personal computers (PCs)

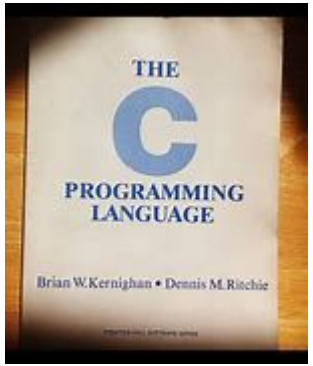


Computer History: Commercial Computer (cont.)

- **In 1969, the creation of UNIX**
  - Operating System developed at AT&T Bell Labs
  - Invented by Dennis Ritchie and Ken Thompson
  - Lower the cost and risk of bringing out a new architecture

- **In 1972, the creation of C programming language**
  - First general purpose programming language
  - Invented by Dennis Ritchie
  - Reduce the need for object-code compatibility





Computer History: RISC vs CISC

- **1980s-1990s:** RISC and CISC

- **RISC:** Reduced Instruction Set Computing is a small, highly optimized set of instruction that are executed in a single clock cycle

- **CISC:** Complex Instruction Set Computing is a larger set of more complex instructions that can performance multiple operations

- **Performance:** RISC is highly suitable to **parallelize instructions**

Computer History: Modern computer

- **1990s - Present: Shift toward parallel computing and multicore processors to improve performance**

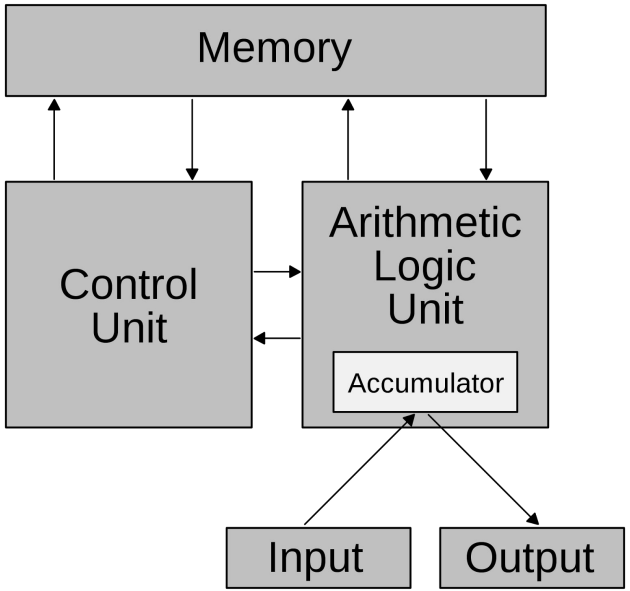
- **21st Century:** Power-efficient design and specialized architectures for tasks like graphics processing and AI

- **Future Trends:** Exploration of quantum computing for exponential processing

Computer Components

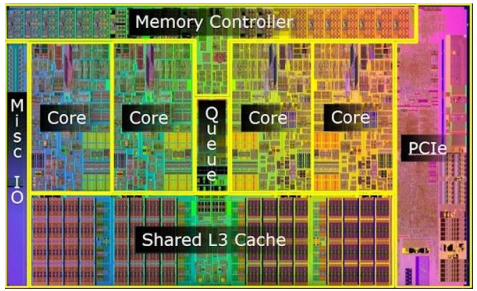
Computer Components: John von Neumann

- **CPU:** Central Processing Unit
  - **Control Unit:** coordinating and controlling the operation flow of computer
  - **Arithmetic Logic Unit (ALU):** performing a wide range of arithmetic, logical, and data manipulation operations
- **Memory:** primary storage for both instructions and data
- **IO Unit:** Input vs Output play essential roles in data exchange between the computer and the external world

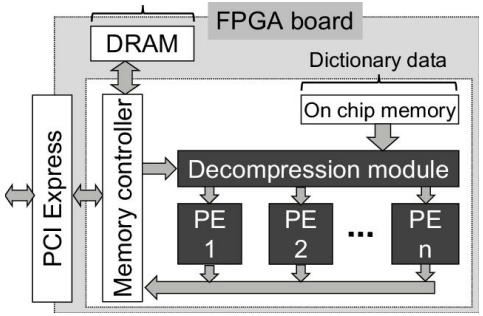


Computer Components (cont.)

- CPU: General-Purpose Processors
  - Designed for wide range of computing tasks
  - Examples: Intel Core series, AMD Ryzen series



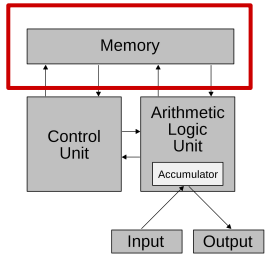
- FPGA: Field-Programmable Gate Arrays
  - Programmable logic devices that can be configured to perform specific tasks
  - Examples: Xilinx UltraScale series, Intel (formerly Altera) Stratix series



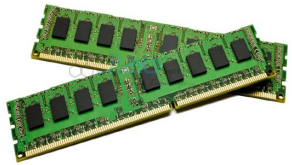
Computer Components (cont.)

- GPU: graphics Processing Units
  - Specialized for rendering graphics and accelerating parallel computations
  - Examples: NVIDIA GeForce, AMD Radeon

Computer Components: Memory



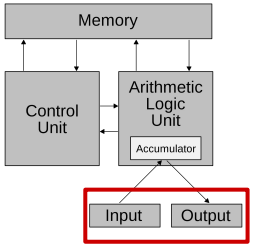
- RAM: Random Access Memory
  - Primary storage medium for data and instructions feeding to CPU during program execution
  - High transfer rate, volatile memory, small, expensive
  - Example: SRAM, DRAM, and DDR



- ROM: Read-Only Memory
  - Storing firmware and essential system instructions that is not frequently changed such as BIOS or embedded system firmware
  - Fast access specifically, difficult to write, non-volatile memory
  - Example: PROM, EPROM, and EEPROM

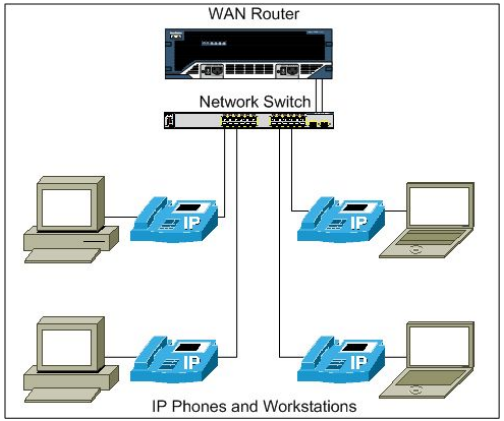
- Secondary Storage
  - Persisting data and program instruction in long time even after turning off power supply
  - Slow transfer rate, non-volatile memory, large, cheaper
  - Example: HDDs, SSDs, CD-ROM, and DVD

Computer Components: IO Devices



- Input Devices: Capture data and commands from user or other devices
  - **Examples:** Keyboards, mice, scanners, sensors, etc.

- Output Devices: Present information, results, or feedback to users or other devices
  - **Examples:** Monitors, printers, speakers, etc.

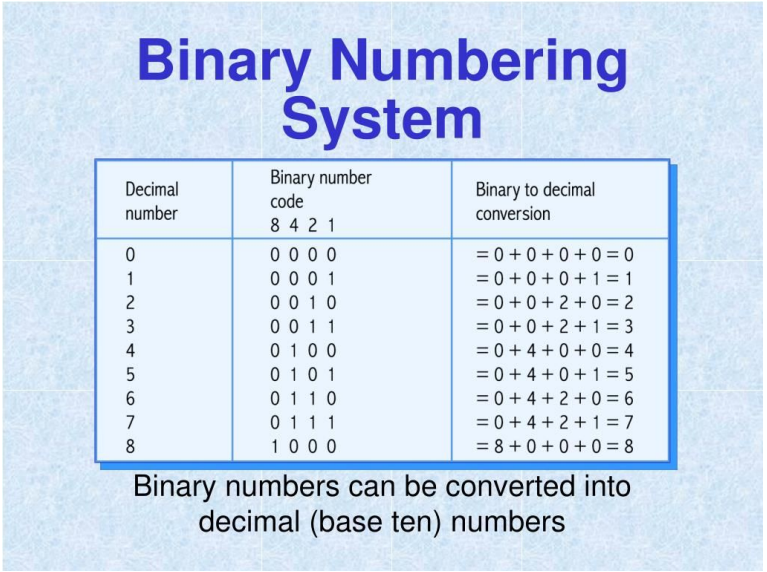


- Communication Devices: Communication between the computer and other devices or networks
  - **Examples:** Ethernet, Wi-fi, Bluetooth adapters, etc.

Binary number system & Digital logic

Binary number system

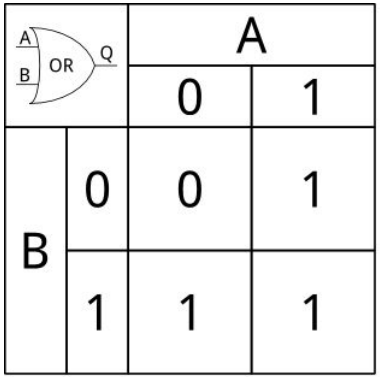
- Base-2 numeral system
- Represents using only 2 digitals: 0 and 1
- Each digit in a binary number is called a bit
- A bit have one of 2 possible values: 0 or 1



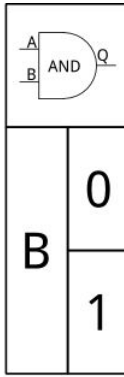
Digital Logic

- Boolean algebra:
  - Mathematical system for expressing and manipulating logic expressions that take into account 2 values of false (0) and true (1)
  - Defines logical operations on binary values: AND, OR, NOT, XOR, NAND, etc.
- Logic Gates:
  - Electronic circuits performing logical operations on input signals to produce output signals
  - Building blocks of digital circuits, enabling more complex logic functions
- Combinational Logic:
  - Outputs depend only on the current inputs, with no memory or feedback
- Sequential Logic:
  - Outputs depend on both current inputs and the state of memory elements

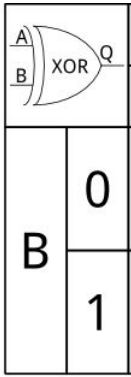
Digital Logic (cont.)



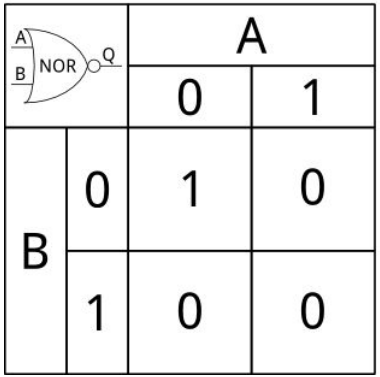
		A	
		0	1
B	0	0	1
	1	1	1



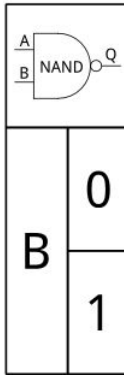
		A	
		0	1
B	0	0	0
	1	0	1



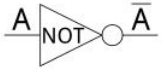
		A	
		0	1
B	0	0	1
	1	1	0



		A	
		0	1
B	0	1	0
	1	0	0



		A	
		0	1
B	0	1	1
	1	1	0

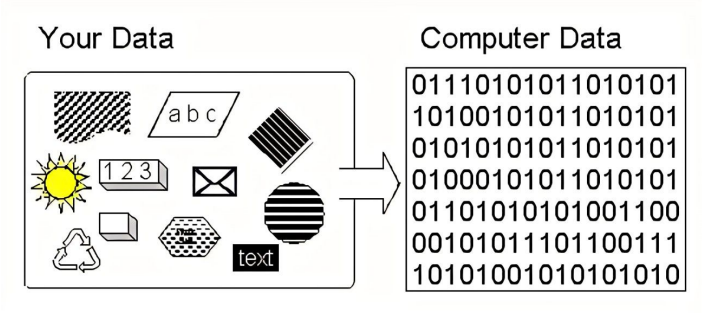


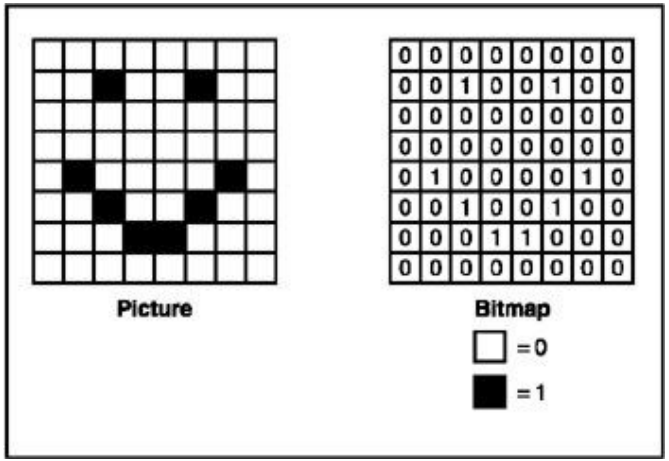
A	
0	1
1	0

Data Representation

Data Representation

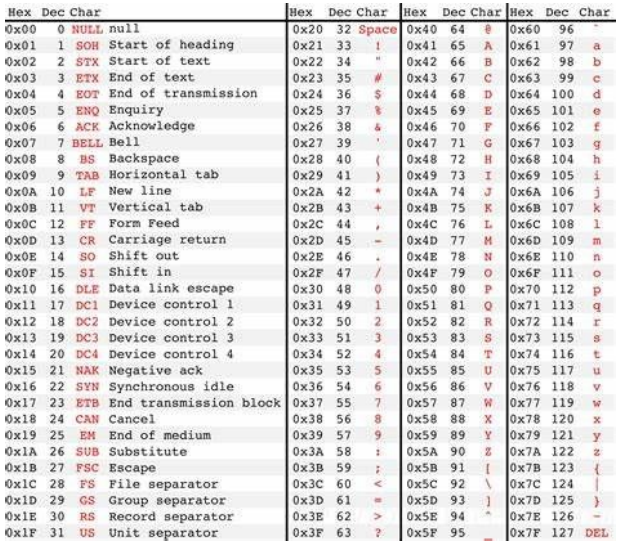
- Computer understands only binary data
- Human works with different data types: text, number, image, etc.
- So, data representation: the process of encoding data in a format suitable for computer
- Binary representation: encode data using only 2 symbols 0 and 1





Data Representation: Primary Data

- **Integer:** converting directly to binary bases format (holded by 4 bytes in C)
- **Floating:** using a sign bit, mantissa, and exponent with wide range of values with varying precision (holded by 8 bytes in C)
- **Character:** applying ASCII to map characters to 7-bit binary codes, providing a standard encoding for text characters (holded by 1 byte in C)



Hex	Dec	Char		Hex	Dec	Char		Hex	Dec	Char		Hex	Dec	Char	
0x00	0	NULL	null	0x20	32	Space		0x40	64	@		0x60	96	`	
0x01	1	SOH	Start of heading	0x21	33	!		0x41	65	A		0x61	97	a	
0x02	2	STX	Start of text	0x22	34	"		0x42	66	B		0x62	98	b	
0x03	3	ETX	End of text	0x23	35	#		0x43	67	C		0x63	99	c	
0x04	4	EOT	End of transmission	0x24	36	$		0x44	68	D		0x64	100	d	
0x05	5	ENQ	Enquiry	0x25	37	%		0x45	69	E		0x65	101	e	
0x06	6	ACK	Acknowledge	0x26	38	&		0x46	70	F		0x66	102	f	
0x07	7	BELL	Bell	0x27	39	'		0x47	71	G		0x67	103	g	
0x08	8	BS	Backspace	0x28	40	(		0x48	72	H		0x68	104	h	
0x09	9	TAB	Horizontal tab	0x29	41	)		0x49	73	I		0x69	105	i	
0x0A	10	LF	New line	0x2A	42	*		0x4A	74	J		0x6A	106	j	
0x0B	11	VT	Vertical tab	0x2B	43	+		0x4B	75	K		0x6B	107	k	
0x0C	12	FF	Form Feed	0x2C	44	,		0x4C	76	L		0x6C	108	l	
0x0D	13	CR	Carriage return	0x2D	45	-		0x4D	77	M		0x6D	109	m	
0x0E	14	SO	Shift out	0x2E	46	.		0x4E	78	N		0x6E	110	n	
0x0F	15	SI	Shift in	0x2F	47	/		0x4F	79	O		0x6F	111	o	
0x10	16	DLE	Data link escape	0x30	48	0		0x50	80	P		0x70	112	p	
0x11	17	DC1	Device control 1	0x31	49	1		0x51	81	Q		0x71	113	q	
0x12	18	DC2	Device control 2	0x32	50	2		0x52	82	R		0x72	114	r	
0x13	19	DC3	Device control 3	0x33	51	3		0x53	83	S		0x73	115	s	
0x14	20	DC4	Device control 4	0x34	52	4		0x54	84	T		0x74	116	t	
0x15	21	NAK	Negative ack	0x35	53	5		0x55	85	U		0x75	117	u	
0x16	22	SYN	Synchronous idle	0x36	54	6		0x56	86	V		0x76	118	v	
0x17	23	ETB	End transmission block	0x37	55	7		0x57	87	W		0x77	119	w	
0x18	24	CAN	Cancel	0x38	56	8		0x58	88	X		0x78	120	x	
0x19	25	EM	End of medium	0x39	57	9		0x59	89	Y		0x79	121	y	
0x1A	26	SUB	Substitute	0x3A	58	:		0x5A	90	Z		0x7A	122	z	
0x1B	27	FSC	Escape	0x3B	59	;		0x5B	91	[		0x7B	123	{	
0x1C	28	FS	File separator	0x3C	60	<		0x5C	92	\		0x7C	124	|	
0x1D	29	GS	Group separator	0x3D	61	=		0x5D	93	]		0x7D	125	}	
0x1E	30	RS	Record separator	0x3E	62	>		0x5E	94	^		0x7E	126	~	
0x1F	31	US	Unit separator	0x3F	63	?		0x5F	95	_		0x7F	127	DEL	

Data Representation: Complex Data

- **Image:** represented as grids of pixels, with each pixel encoded to represent color or grayscale using binary values
- **Audio:** a sequence of discrete samples using binary values encoded by pulse-code modulation (PCM)
- **Video:** Encodes moving images using binary data employing compression techniques (MPEG)

Data Representation: Stored format

- **File Formats**:
  - The structure and organization of data within files, facilitating interoperability and data exchange between different systems and applications
  - Examples: text format (plain text, CSV, etc.), binary formats (JPEG, MP3, etc.)
- **Endianness**:
  - Order in which bytes are stored in memory
  - **Little-Endian**: Least significant byte stored first
  - **Big-Endian**: Most significant byte stored first

Binary (Decimal: 149)	1	0	0	1	0	1	0	**1**
Bit weight for given bit position n (2^n)	2^7	2^6	2^5	2^4	2^3	2^2	2^1	**2^0**
Bit position label	MSb							**LSb**

Thank you for you listening

Advance Computer Architecture and x86 ISA

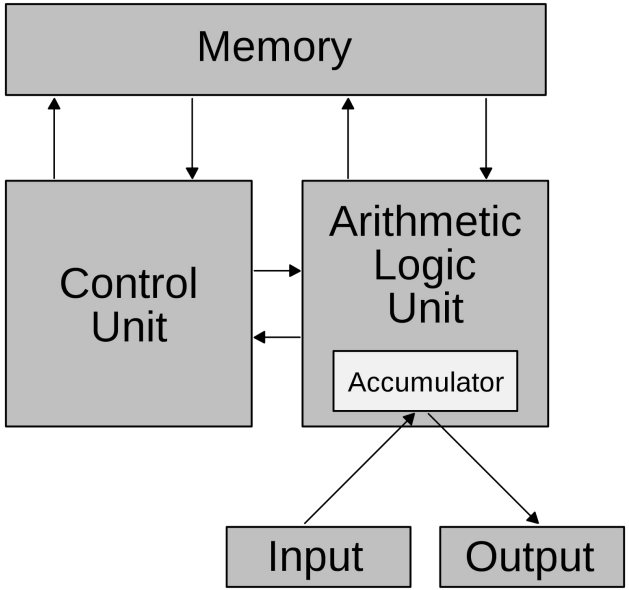
University of Science and Technology of Hanoi

MS. LE Nhu Chu Hiep

Von Neumann Processors

Remind ?

- Components:
  - Arithmetic Logic Unit
  - Control Unit
  - Memory
  - Input / Output
- Architecture:
  - **Accumulator based**



Instruction executions

- Instruction execution
  - Read instruction from memory
  - Decode instruction to execute
  - Execute the instruction
    - Control of the arithmetic logic unit to execute the instruction
  - Store the result
- Von neumann / Instruction cycle:
  - The time required by the CPU to execute one single instruction
  - In one clock cycle, the instruction could be executed in 4 stages
    - Fetch, decode, execute, and writeback

Instruction executions: read and decode

- Read the instruction from the memory
  - Access to the memory and load an instruction in the instruction register
  - The current address is stored in a specific register, called program counter (PC)
  - The program counter is incremented if no branch/jump is required, in the case of jump, the program counter is loaded with another value
- Decode instruction
  - After having read the instruction, the processor must know what to execute
  - Instruction decomposition
    - Operational code
    - Operands to use

Instruction executions: execute and store

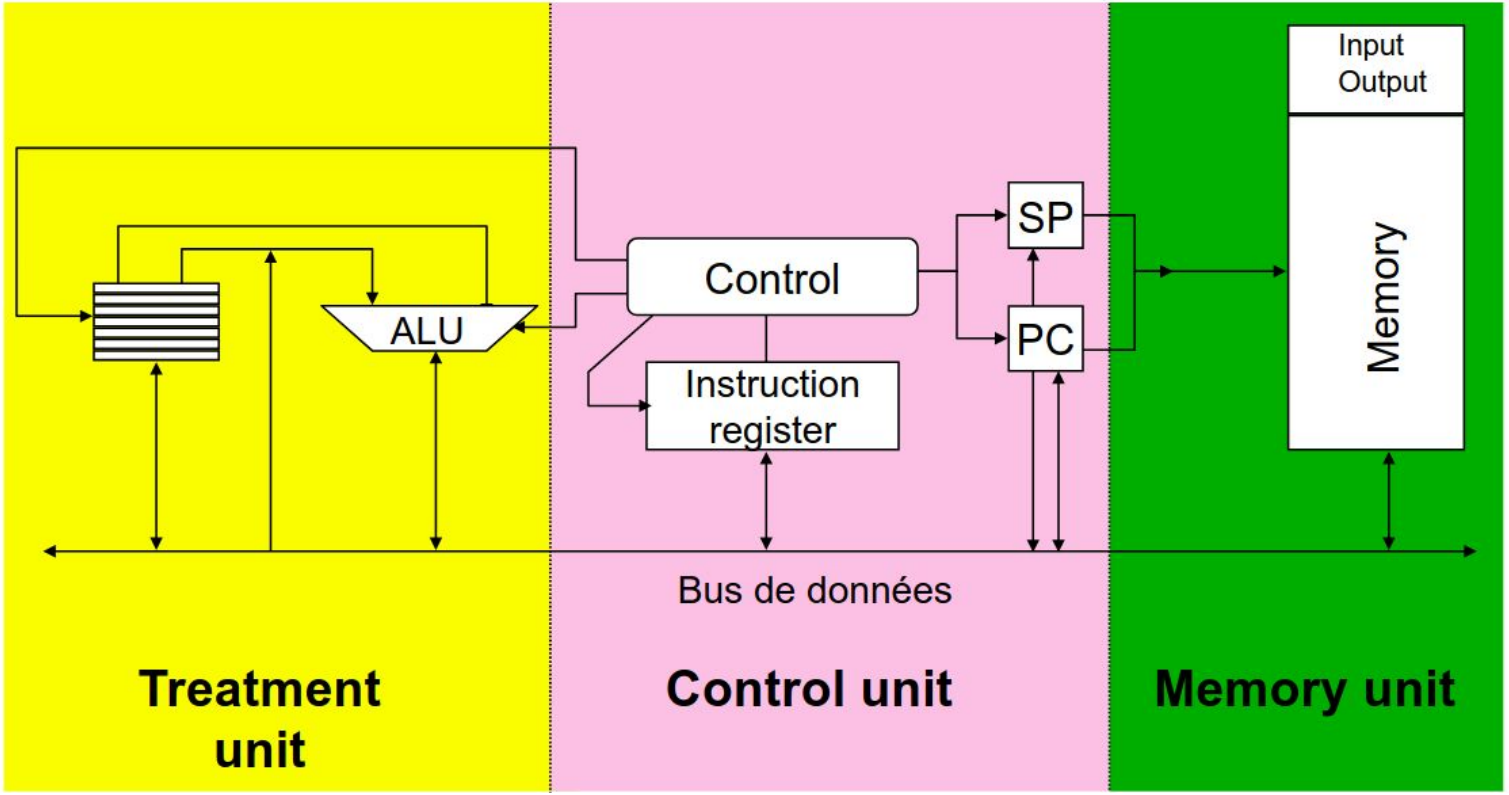
- Instruction execution
  - Decompose the instruction in a set of micro-instructions (set of signal to control) to ensure the correct execution of in the ALU
    - For example : imagine instruction ADD R1, R2, R3
    - This instruction needs control on R2 and R3, then control for ALU, then control for write register R1
  - Control the signals for the ALU
- Store the result
  - Store in memory or registers
  - Transfer data from register to memory
  - Store the stage of processor
    - Store the flags

Instruction executions: more detail

1. Load the next instruction from the memory to the instruction register
2. Modify the program counter (PC) for the next instruction to read
    a. PC = PC + 1
3. Decode the instruction
4. Find the operand necessary for the instruction
5. Load the operand in internal registers
6. Execute the instruction
7. Store the result in the memory
8. Go to the next instruction, return in step 1

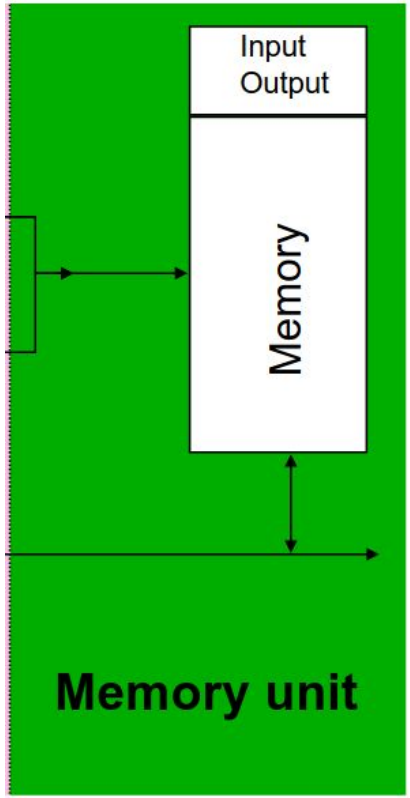
Von Neumann Designed Units

Von Neumann designed units



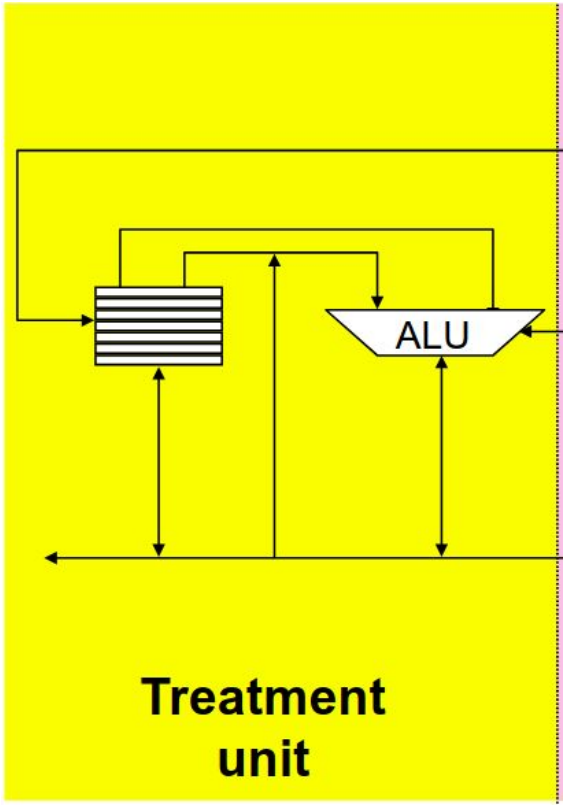
Memory unit

- Store the information which concern
  - The program, the application
  - The data to transform
- There are two types of memory model
  - Von Neumann model:
    - Instruction and data stored in single memory unit
    - Single access bus
  - Harvard model:
    - Instruction and data stored in different memory unit
    - Separate bus to access instruction and data



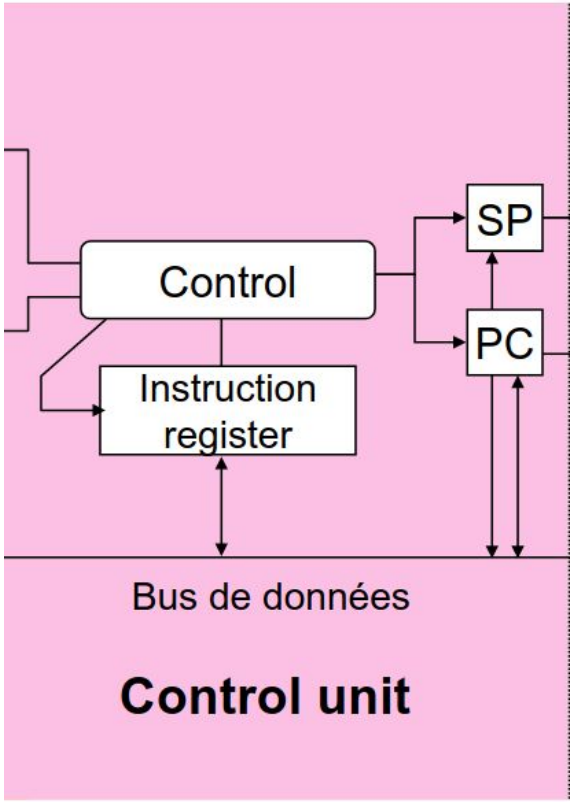
Treatment unit

- Composed of the ALU and a file of registers
- The registers store the data which are currently transformed by the ALU
  - Registers for data
  - Registers to store the state of the processor



Control unit

- The main part of the processor
- Ensure the execution of the instructions, of the program
  - Decode the instruction stored in the instruction register RI
  - Address the memory from the program counter
- Control execution signal
  - Interrupt signal
  - Instruct ALU, Memory, and IO device
- Hold instruction execution states
  - Stack Pointer (SP)
  - Program counter (PC)



Execution of the instructions

Three types of instructions

- Control instruction
  - Branch, jump, procedure call
- Treatment instruction
  - Arithmetic and logic
- Transfer instruction
  - Moving data
  - Between registers
  - Between memory and registers

Control / Branch Instruction

Control instructions

- Objective: modification of Program Counter (PC) depending on events
  - Internal events: results of some computation, state of processor
  - External events: I/O pads, interrupt
- The program counter is modified, and this change the sequence of addresses
- 3 types of jump/branch
  - **Conditional or unconditional**
  - **Explicit or implicit**
  - **With or without context saving**

Control instructions: conditional or unconditional branch

- Conditional
  - Result produces by a test, for example test of the flag values which indicates the state of the machine

- Unconditional
  - Jump/branch without any condition about the state of the processor

Control instructions: explicit or implicit branch

- Explicit
  - Case for a jump, or a conditional branch
- Implicit
  - Case for interrupt ⇒ External signal
  - Case for exceptions ⇒ For example, instruction which tries to divide by 0

Control instructions: jump/branch with context saving

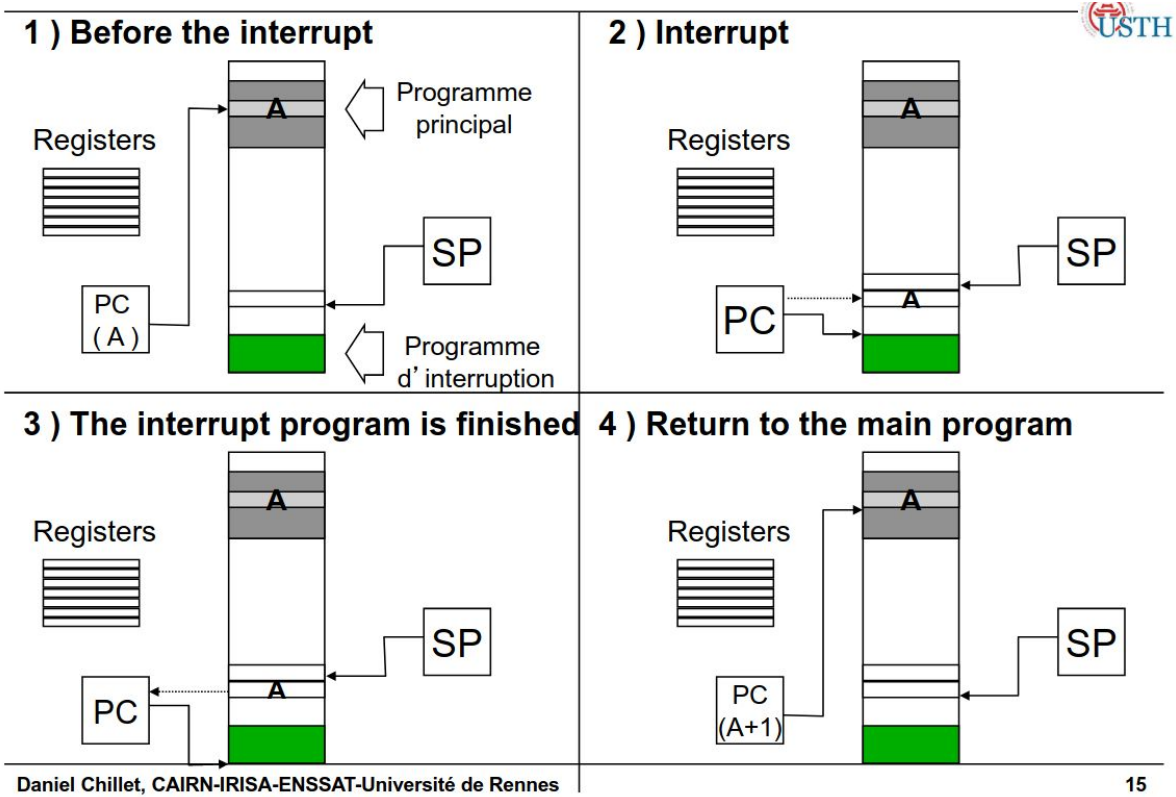
- The modification of PC due to an interruption or execution

  where the the current context of execution is stored

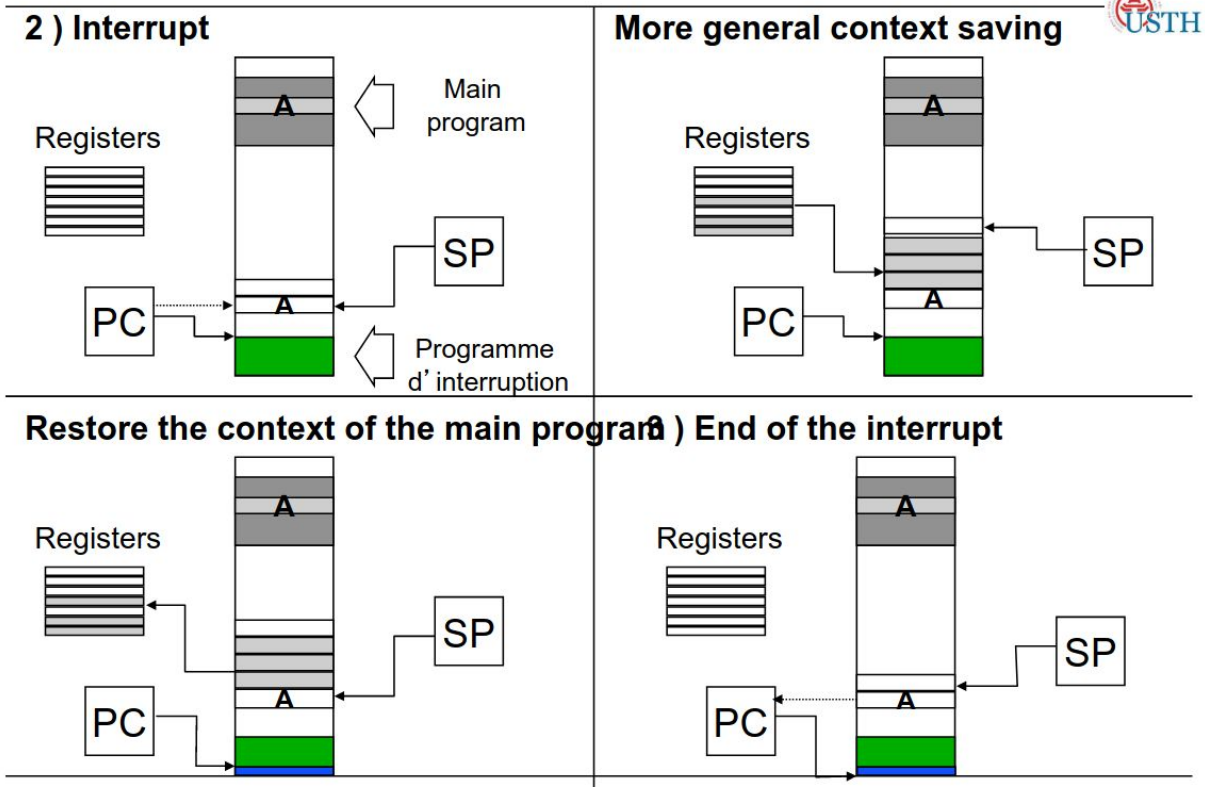
- The minimal context is composed of
  - The address of the current instruction
  - The content of the registers if needed ⇒ the software developer must

    explicitly specify them

Context saving: Memory State

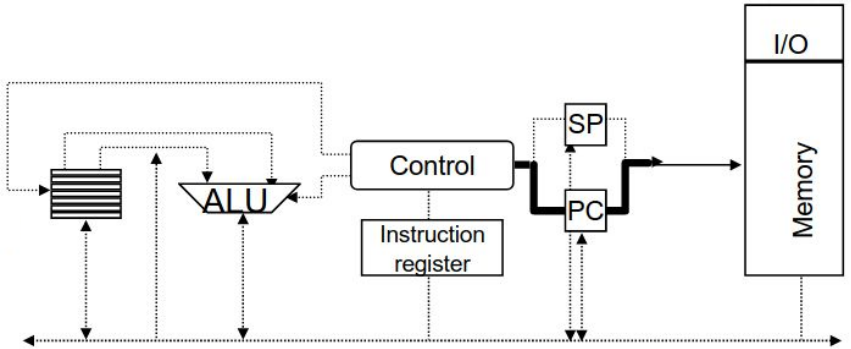


Context saving: Memory State (cont.)

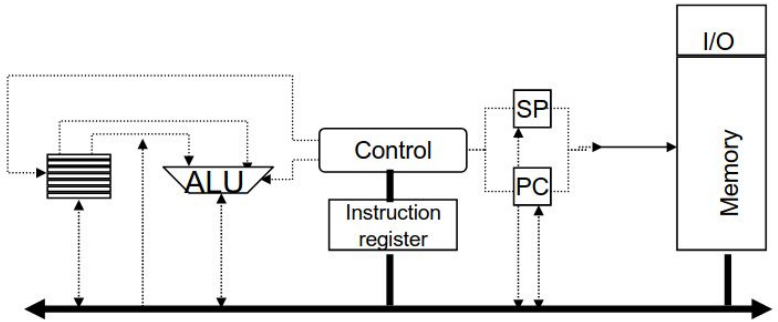


Control instruction: Organization

- Load instruction from memory

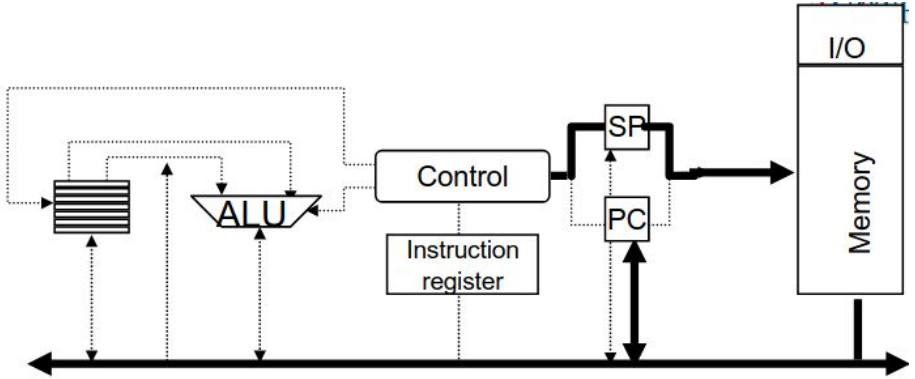


- Decode Instruction

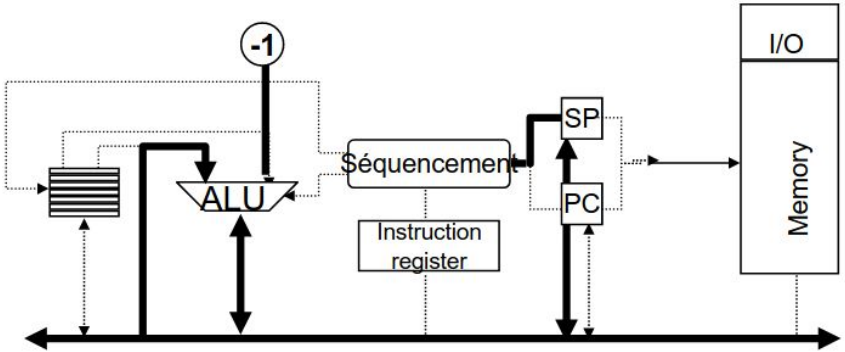


Control instruction: Organization (cont.)

- Context saving

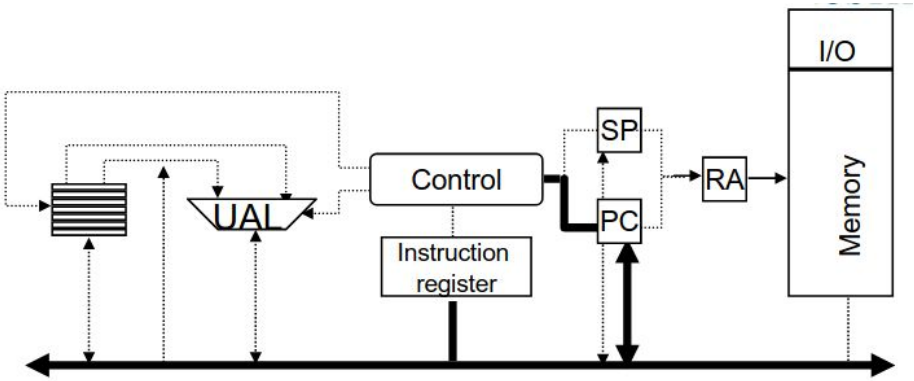


- Stack pointer modification



Control instruction: Organization (cont.)

- Extract the branch address



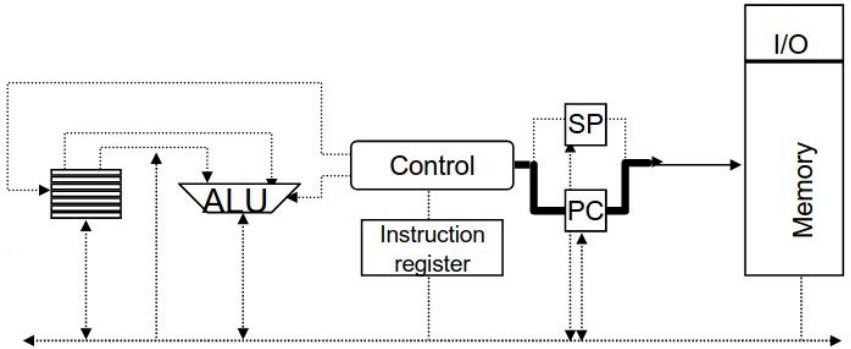
Treatment / Computation Instruction

Data treatment

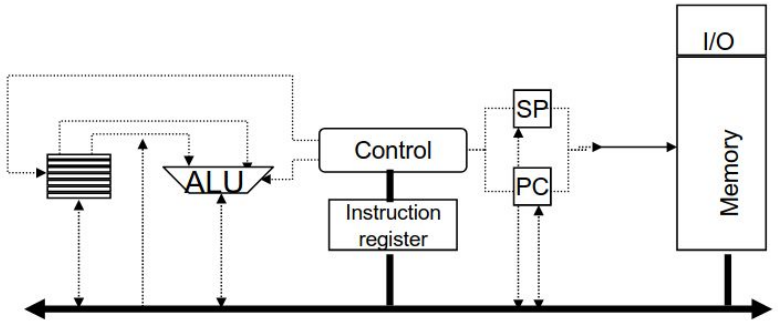
- Computation operations
  - Arithmetic
  - Logic
- Modification of the flags depending on the results produced by ALU
  - Result is equal to 0
  - Result is negative
  - Result is impossible to code with the classical number of bits, carry

Treatment instruction: Organization

- Load instruction from memory

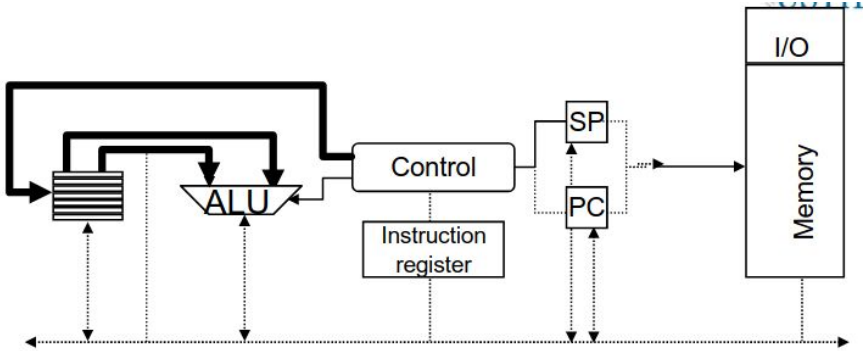


- Decode Instruction

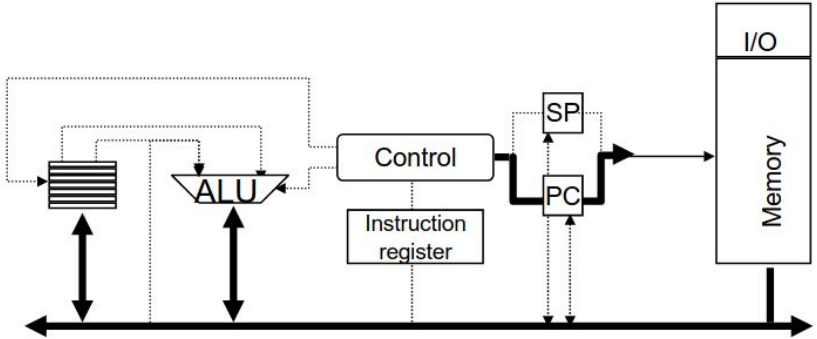


Treatment instruction: Organization (cont.)

- Instruction execution

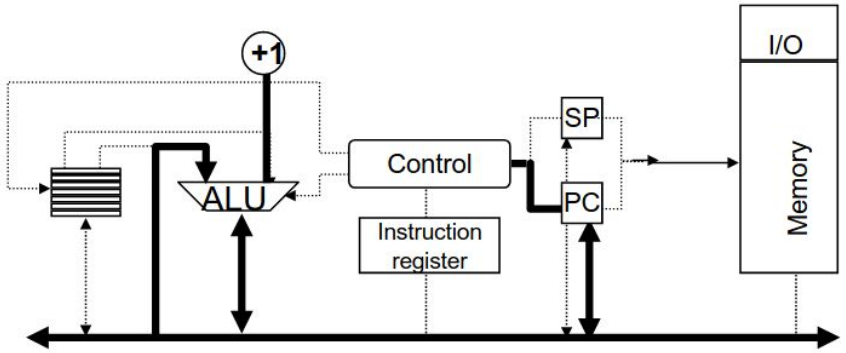


- Result storage



Treatment instruction: Organization (cont.)

- Go the next address, Increment the counter program



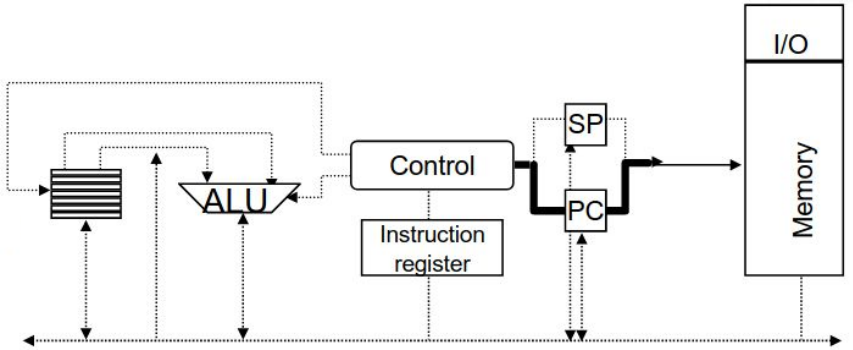
Transfer Instruction

Transfer Instructions

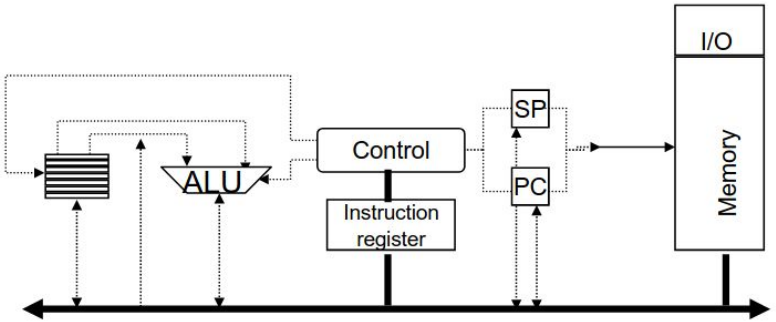
- Move data between different placements
- There are three types of transfers
  - **Between registers**
  - **Between register and memory**
  - Between two differents cells of memory
- To address a data, different techniques
  - Immediate addressing
  - Direct addressing
  - Indirect addressing
  - Indexed addressing
  - etc.

Transfer instruction: Organization

- Load instruction from memory

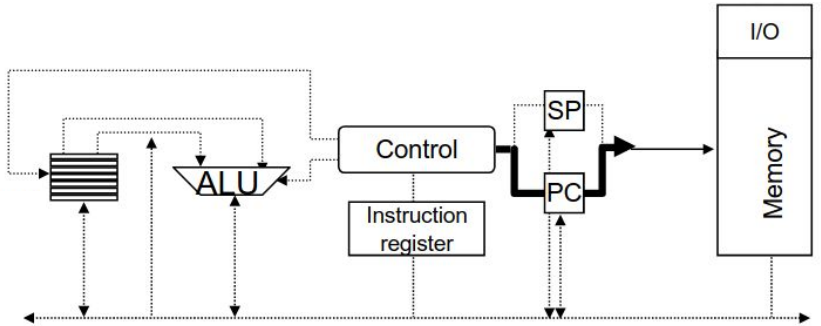


- Decode Instruction

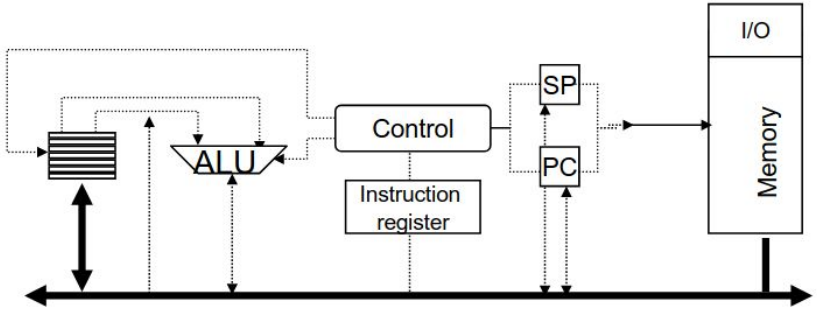


Transfer instruction: Organization (cont.)

- Looking for the data to transfert



- Transfer effective



Memory Addressing

Memory addressing

- Data is located at different locations
  - Registers
  - Memory
  - Instruction itself
  - ⇒ Need method to specify which data to process
- Effective Address (EA)
  - Final address of the operand (data) in the instruction
  - Reveals the location of operand
- Address Mode
  - Specifies how to calculate the Effective Address of the operand(s) from the instruction
  - Syntax: depending on the programming language

Memory addressing (cont.)

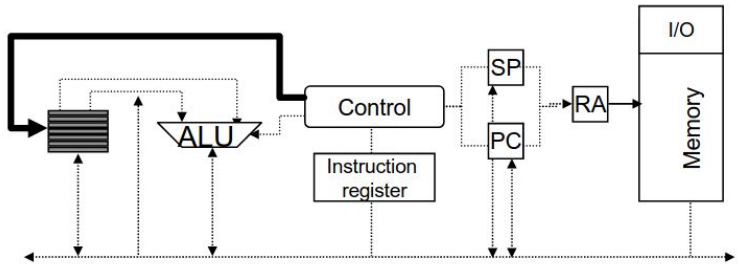
- **Register addressing - Transfer data inside register file**
- **Immediate addressing - Data is located inside instruction itself**
- **Direct addressing - Operand is a pointer point to data location**
- **Indirect addressing - Operand is a pointer of pointer to data location**
- **Indexed addressing - Calculate data location from (RI + based index)**
- Post / Pre increment indirect addressing - Same like (i++ / i–) in C

Register addressing

- Instruction coding

- Instruction examples:
  - add R1, R2, R3 ; R1 = R2 + R3
  - move R1, R2 ; R1 = R2

- Step of the instruction
  - Load instruction from memory
  - Decode instruction
  - Transfer effective

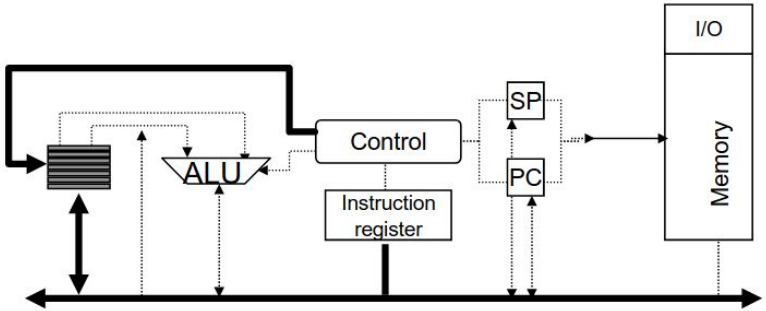


Immediate addressing

- Instruction coding

- Instruction examples:
  - add R1, R2, #3 ; R1 = R2 + 3
  - load R1, #20 ; R1 = 20

- Step of the instruction
  - Load instruction from memory
  - Decode instruction
  - Transfer effective



Direct addressing

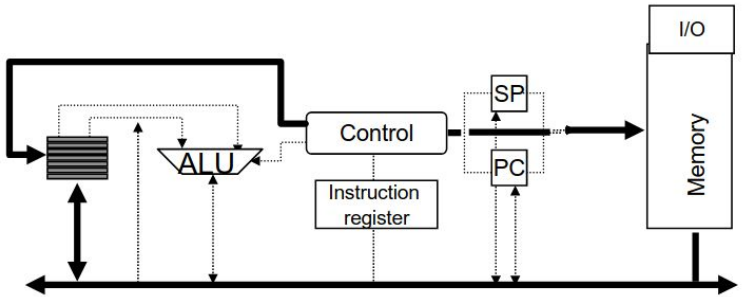
- Instruction coding

code	Destination register	Word 1
Part 1 of the address		Word 2
Part 2 of the address		Word 3

- Instruction examples:
  - add R1, R2, (1000) ; R1 = R2 + Memory[1000]
  - li R4, (2000) ; R4 = Memory[2000]

- Type of data: variable, constant

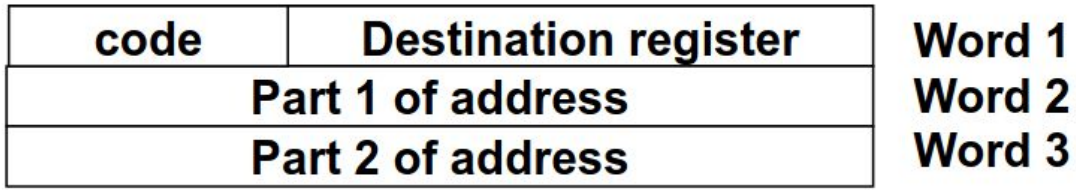
- Step of the instruction
  - Load instruction from memory
  - Decode instruction
  - Looking for the data to transfert
  - Transfer effective



Indirect addressing

- Instruction coding: indirect through register

- Instruction coding: indirect through memory



- Instruction examples:
  - Add R1, R2, @(R3) ; R1 = R2 + Memory[Memory[R3]]
  - Li R4, @(R5) ; R4 = Memory[Memory[R5]]

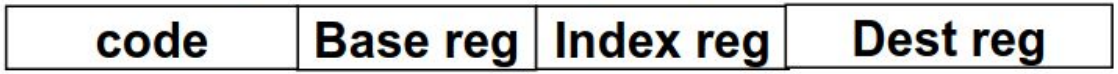
Indirect addressing

- Type of data: variable, array, pointer, constant
- Step of execution
  - **Indirect register:**
    - Load instruction from memory (A)
    - Decode instruction (B)
    - Looking for the data to transfert (C)
    - Transfer effective (D)
  - **Indirect memory:**
    - (A, B) * 3, (C, D) * 3

Indexed addressing

- Instruction coding: immediate indexing

- Instruction coding: Indexed through register



- Instruction example:
  - Add R1, R2, R3(R4) ; R1 = R2 + Memory[R3+R4]
- Types: variable, data structure

Indexed addressing (cont.)

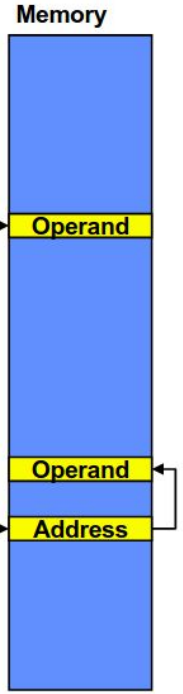
- Step of execution
  - **Indexation through register:**
    - Load instruction from memory (A)
    - Decode instruction (B)
    - Looking for the data to transfert (C)
    - Transfer effective (D)
    - Addition of base and index
  - **Immediate indexation:**
    - (A, B) * 2, C, D, addition of index and base

To resume

- Immediat

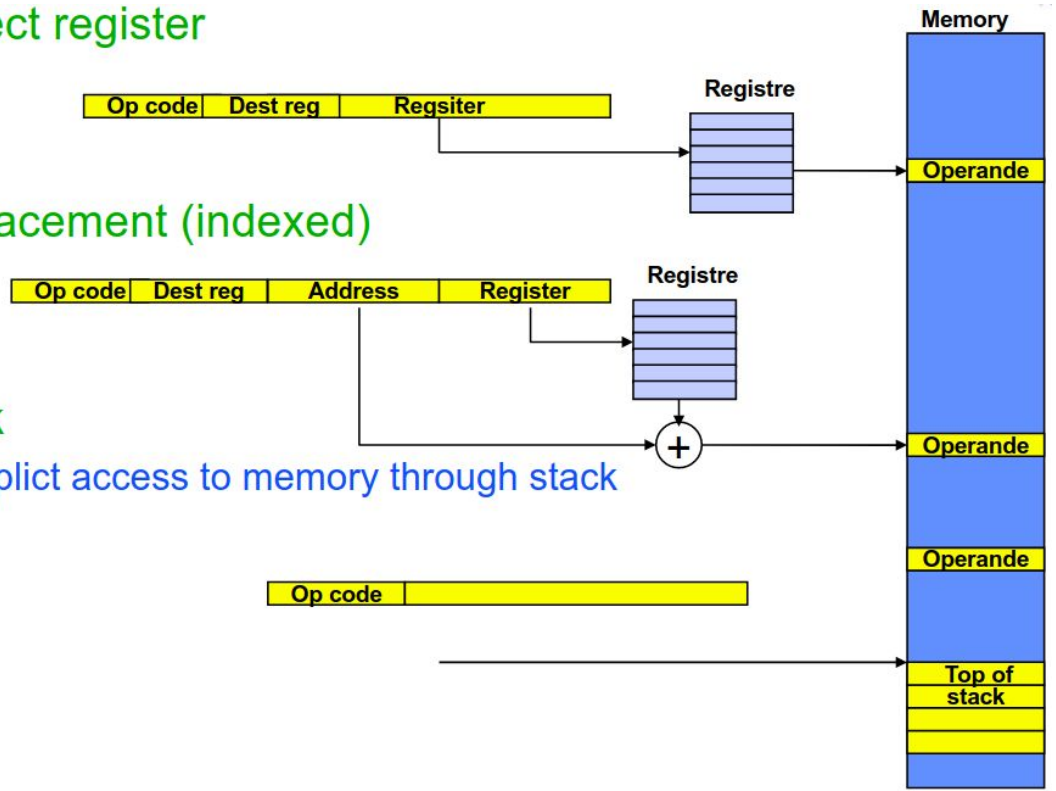
- Direct

- Indirect through memory



To resume (cont.)

- Indirect register
- Displacement (indexed)
- Stack
  - Implict access to memory through stack



ISA Extra Features

Size of instruction set

- **CISC processor:**
  - The number of words to code the instructions is variable

- **RISC processor:**
  - Each instruction is coded with fix size of word

- **Discussion: RISC make more simpler controller**

Data addressing methods

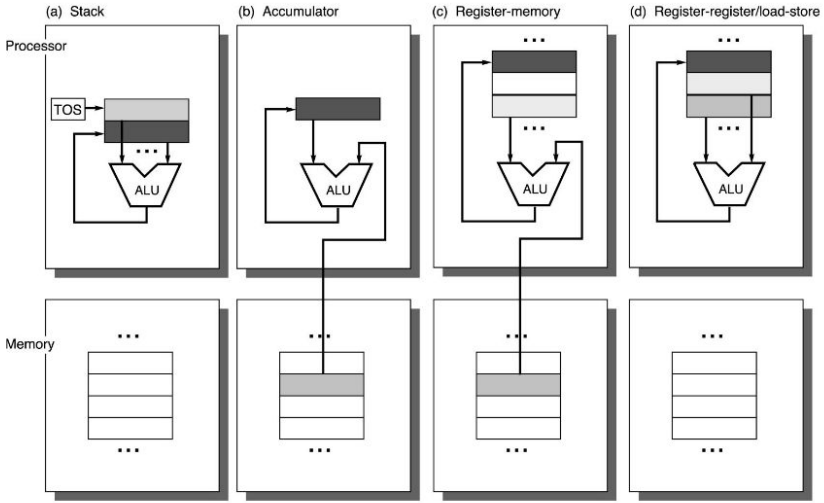
- **Memory based:** no register available for user, all the operations must be done with a memory cell
- **Accumulator based:** all the operations are executed with a specific register (the accumulator)
- **Register based:** all the operations are executed with the register file
- **Stack based:** the operations are stored in a stack

Data addressing methods (cont.)

Stack	Accumulator	Register-Memory	Load-store

Push A
Push B
Add
Push A
Mul
Pop C
	Load A
	Add B
	Mul A
	Store C
		Load R1,A
		Add R1,B
		Mul R1,A
		Store C,R1
			Load R1,A
			Load R2,B
			Add R3,R1,R2
			Mul R3,R3,R1
			Store C,R3

6 instr. 4 mem. op.	4 instr. 4 mem. op.	4 instr. 4 mem. op.	5 instr. 3 mem. op.



Orthogonal instructions and number of operands

- A computer instruction set architecture that:
  - No dependencies between operation code and addressing modes
  - All operations can used all the addressing modes
  - In general, these processors have small number of instructions
  - These processor are simple to program
  - Compiler easier to develop
- General number of operands in an instruction
  - 0 operand (NOP)
  - 1 operands
  - 2 operands (instructions modify one operand)
  - 3 operands

Thank you for you listening

Advance Computer Architecture and x86 ISA

University of Science and Technology of Hanoi

MS. LE Nhu Chu Hiep

Pipeline

How reduce the time for computing on data?

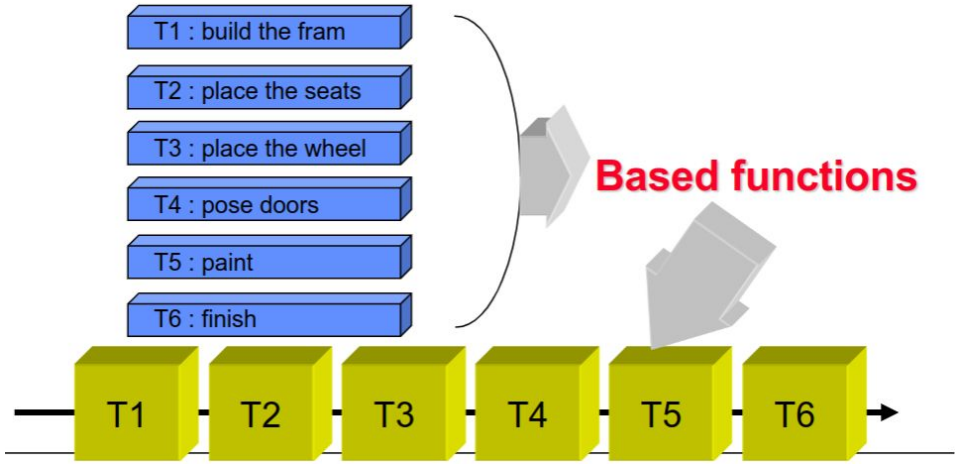
- Increase the clock frequency
- Parallel execution on several data

- Working on the chain

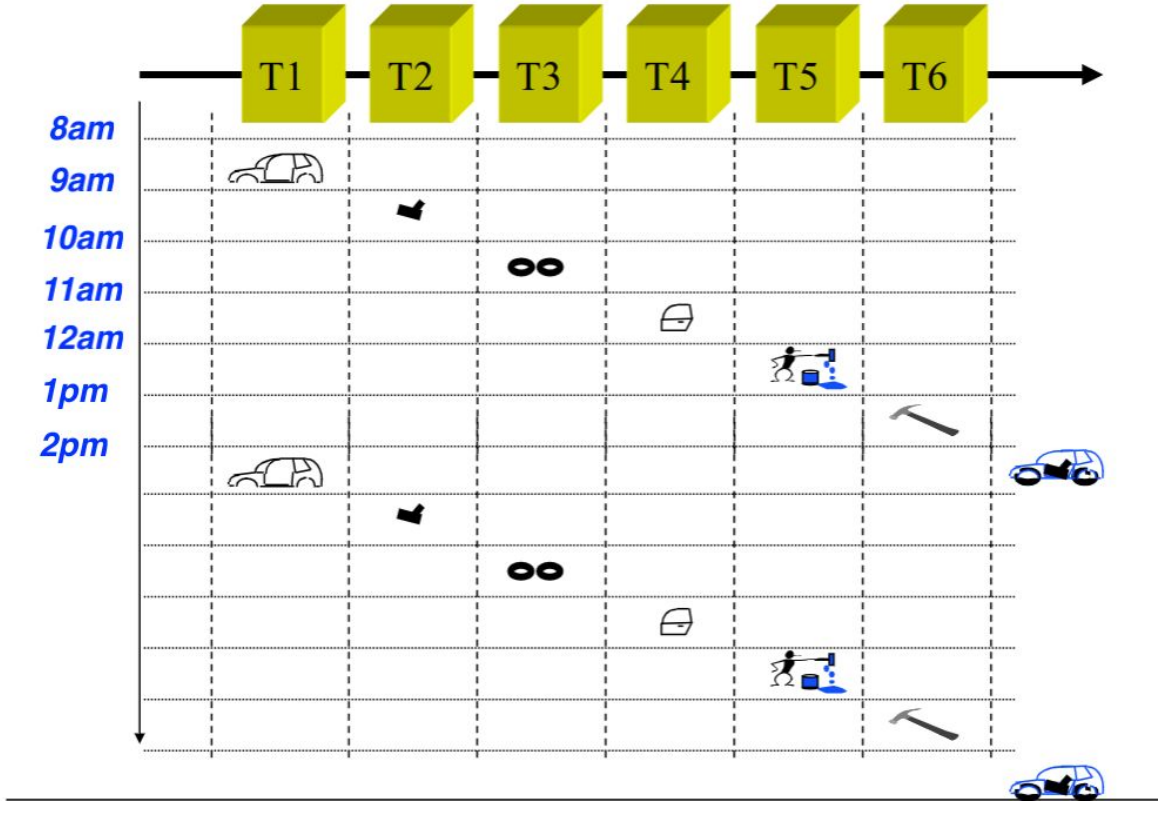
Assembly Line

Assembly line

- Task: building a car
- Composed of several sub-tasks
  - **T1:** Build the chase
  - **T2:** Place the seats
  - **T3:** Place the wheel
  - **T4:** Pose doors
  - **T5:** Paint
  - **T6:** Finish

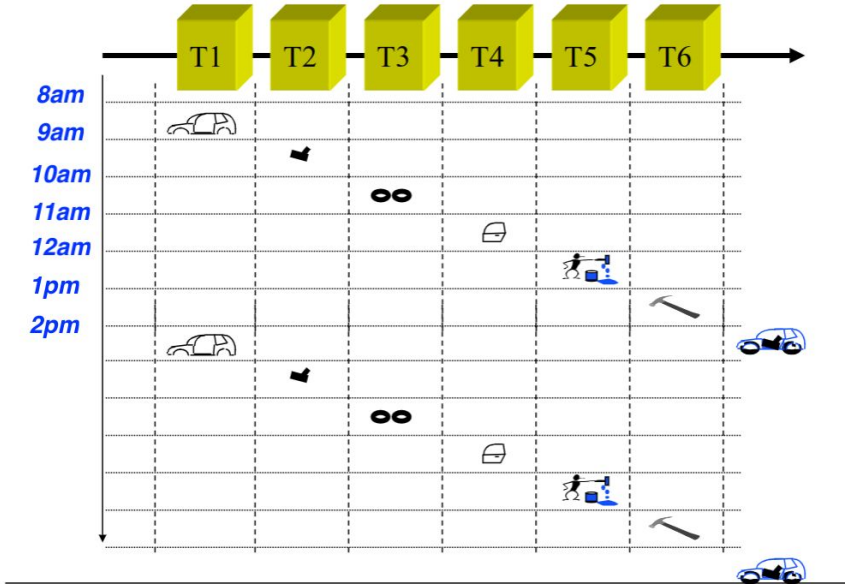


Building chain



Issues

- **Cost for this production technique**
  - 1 worker: Non specialist, no optimally, able to do everything
- **Time per car producing**
  - 6 UT (six unit time)
- **Production Cadence**
  - 1 new car each 6 time units

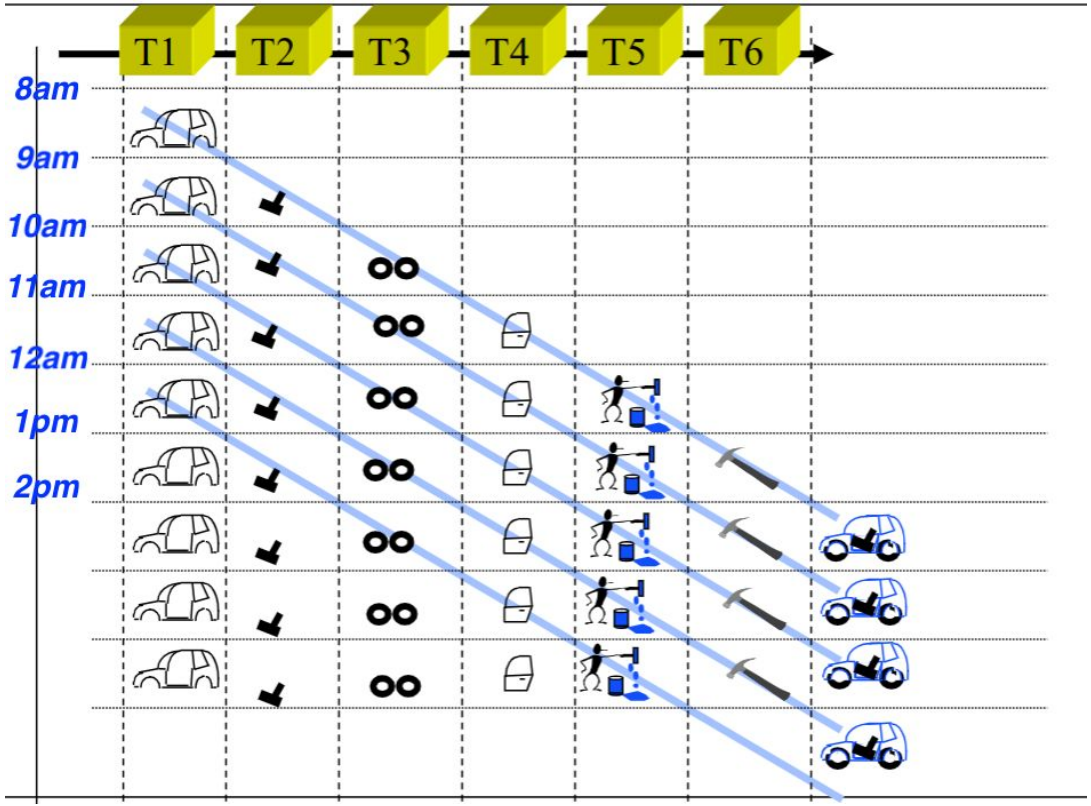


How to increase the production ?

How to increase the production ?

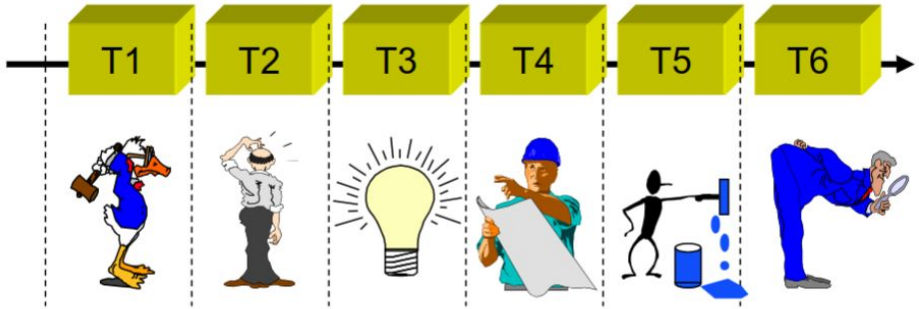
- Placing one worker for each elementary task
- Specialized worker, which is able to do one elementary thing
- Optimal work

Assembly line pipeline



Assembly line workers

- **Cost for this production technique**
  - 6 workers



- **Time per car producing**
  - 6 UT (six unit time)

- **Production Cadence**
  - 1 new car for each time unit

Instruction Pipeline

Definition

- An implementation technique
  - Multiple instructions are overlapped in execution
  - Takes advantage of parallelism that exists among the actions needed to execute an instruction
- Pipe stage / Pipe segment
  - Different steps are completing different parts of different instructions in parallel
- Processor cycle
  - Time required between moving an instruction one step down the pipeline

Definition

- The pipeline designer’s goal
  - Balance the length of each pipeline stage
  - If the stages are perfectly balanced, then the time per instruction on the pipelined processor

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

- Pipelining yields a reduction in
  - The number of clock cycles per instruction (CPI)
  - It is not visible to the programmer

RISC Instruction Set: Remind ?

- What is the basics instruction types of a RISC ?

- What is the implementation of a RISC ?

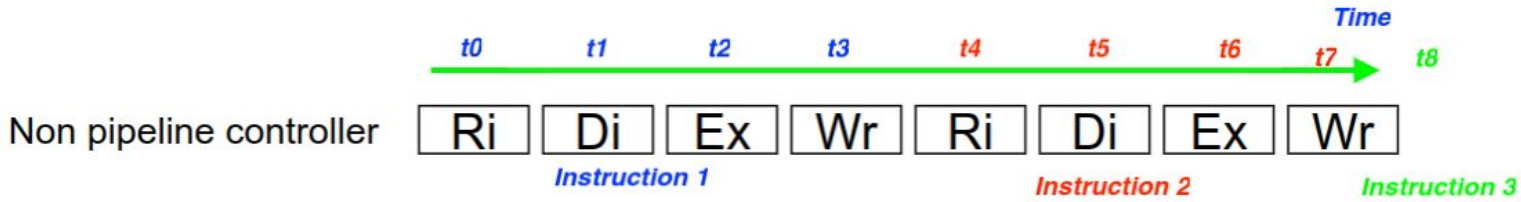
RISC Instruction Set: Remind ?

- What is the basics instruction types of a RISC ?
  - Control Instructions
  - Treatment Instructions
  - Transfer Instructions
- What is the implementation of a RISC ?
  - Instruction read cycle (RI)
  - Instruction decode/register fetch cycle (DI)
  - Execution/effective address cycle (EX)
  - Write-back cycle (WB)

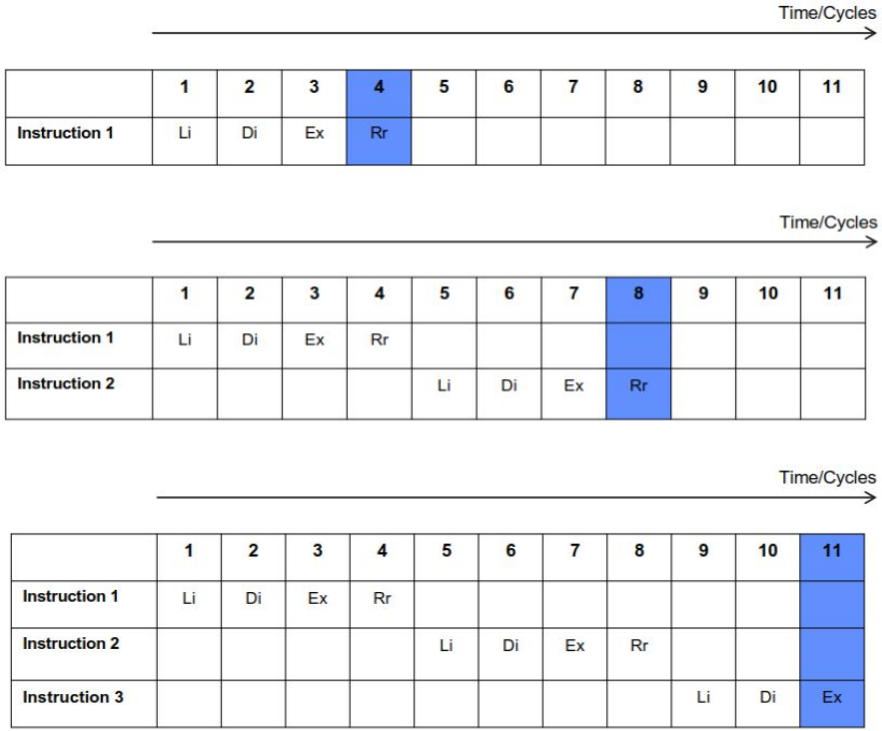
RISC is well designed to fit to the implementation stages per each cycle

Pipeline example

- Instruction read cycle (RI)
- Instruction decode/register fetch cycle (DI)
- Execution/effective address cycle (EX)
- Write-back cycle (WB)

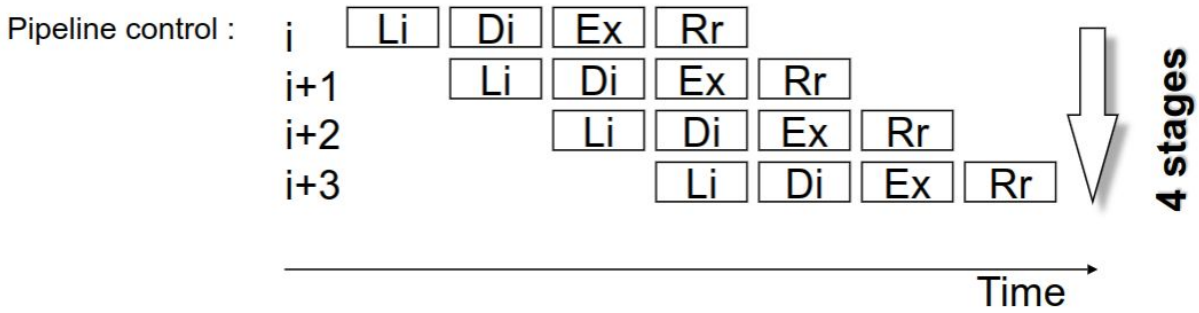


Pipeline example: non-pipeline

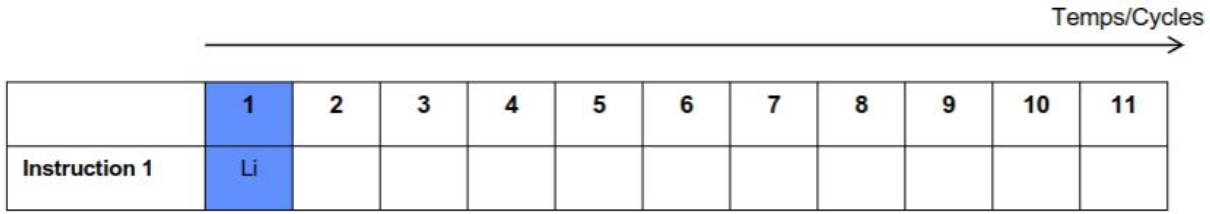


Pipeline example: with pipeline

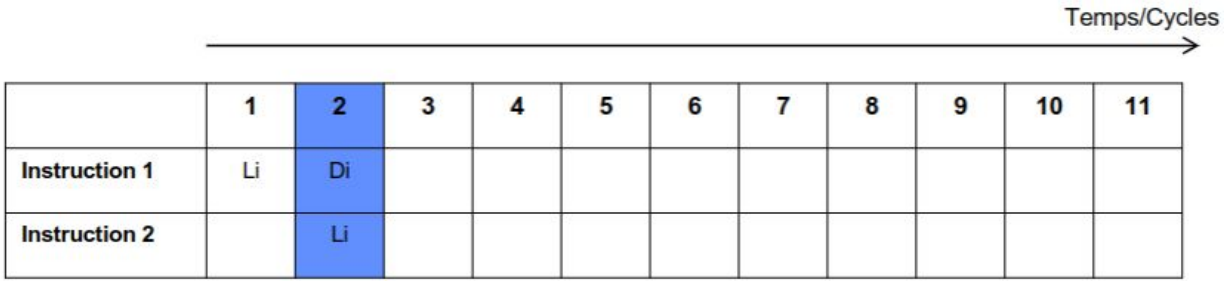
- 1 instruction per cycle
- Clock frequency increases by factor 4
- Speedup = *4



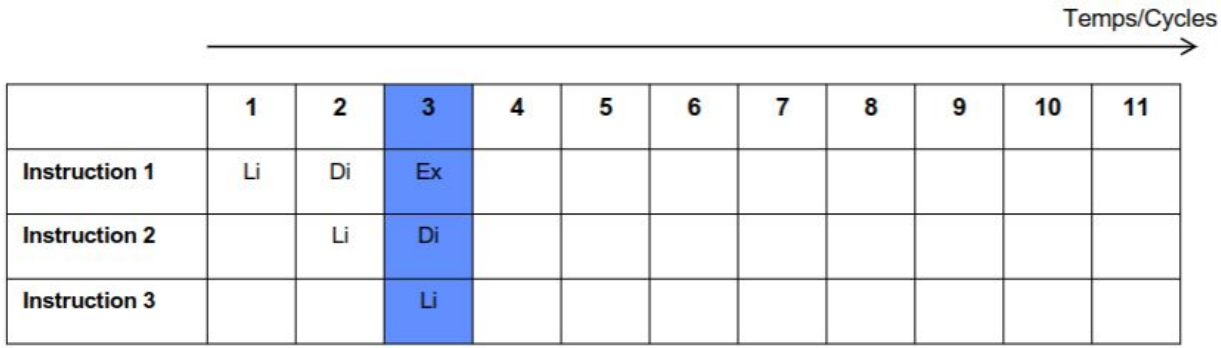
Pipeline example: with pipeline (1)



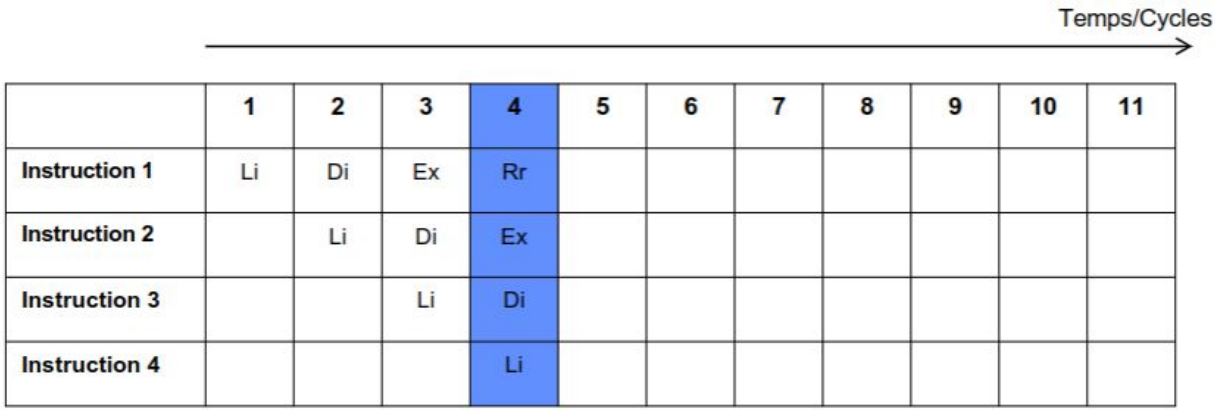
Pipeline example: with pipeline (2)



Pipeline example: with pipeline (3)

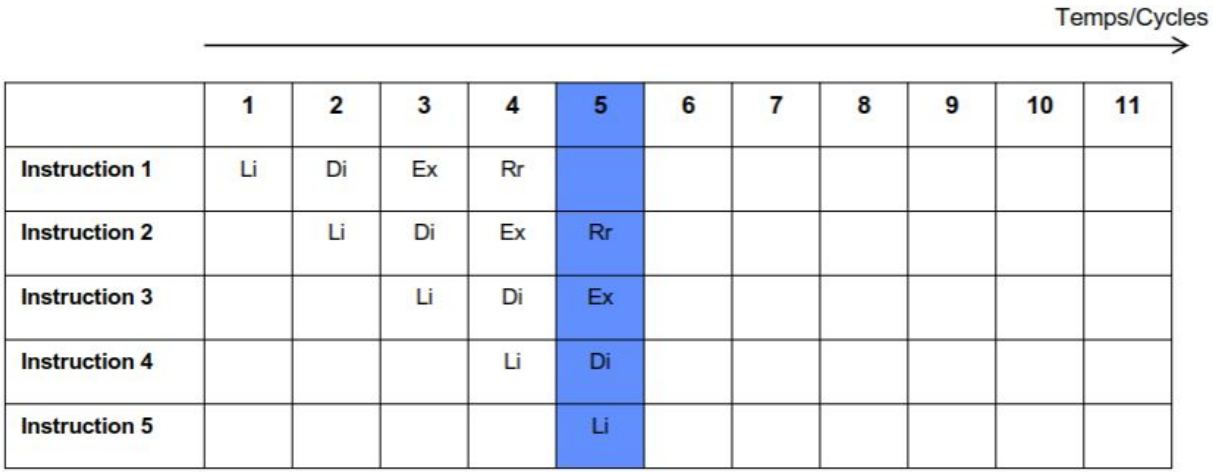


Pipeline example: with pipeline (4)



	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	Li	Di	Ex	Rr							
Instruction 2		Li	Di	Ex							
Instruction 3			Li	Di							
Instruction 4				Li							

Pipeline example: with pipeline (5)



	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	Li	Di	Ex	Rr							
Instruction 2		Li	Di	Ex	Rr						
Instruction 3			Li	Di	Ex						
Instruction 4				Li	Di						
Instruction 5					Li						

Pipeline example: with pipeline (6)

Temps/Cycles →

	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	Li	Di	Ex	Rr							
Instruction 2		Li	Di	Ex	Rr						
Instruction 3			Li	Di	Ex	Rr					
Instruction 4				Li	Di	Ex					
Instruction 5					Li	Di					
Instruction 6						Li					

Pipeline example: with pipeline (7)

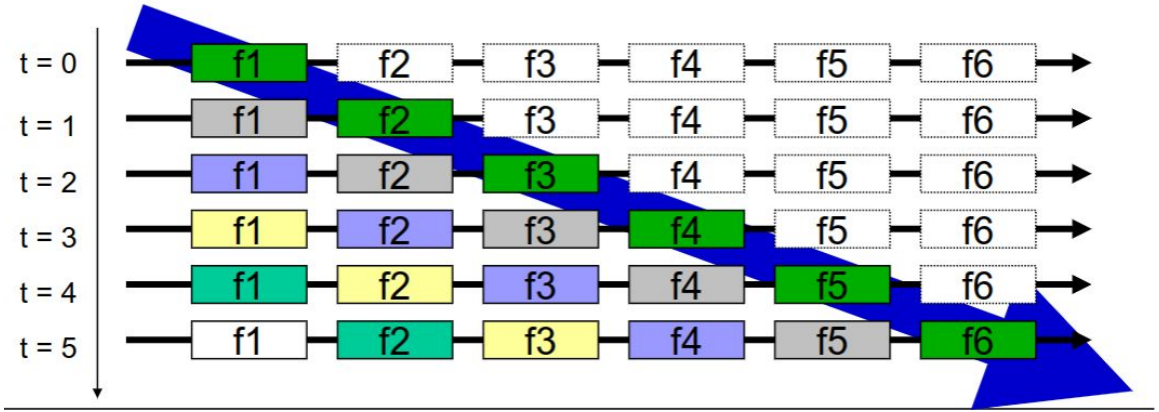
Temps/Cycles →

	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	Li	Di	Ex	Rr							
Instruction 2		Li	Di	Ex	Rr						
Instruction 3			Li	Di	Ex	Rr					
Instruction 4				Li	Di	Ex	Rr				
Instruction 5					Li	Di	Ex				
Instruction 6						Li	Di				
Instruction 7							Li				

Data Computation Time

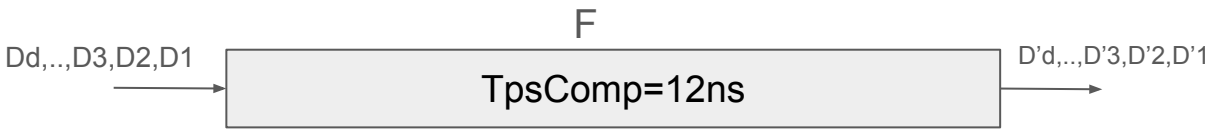
Data Computation Time

- Let F the computation to realized
- Let F = fn o fn-1 o ..... o f2 o f1 the decomposition of F
- Let tn, tn-1, ..., t2 and t1 the times for the computation of each elementary part
- The time to compute one data is $Tps = \sum ti$
- The cadence of computation is : Cadence = max (ti)



Generalization

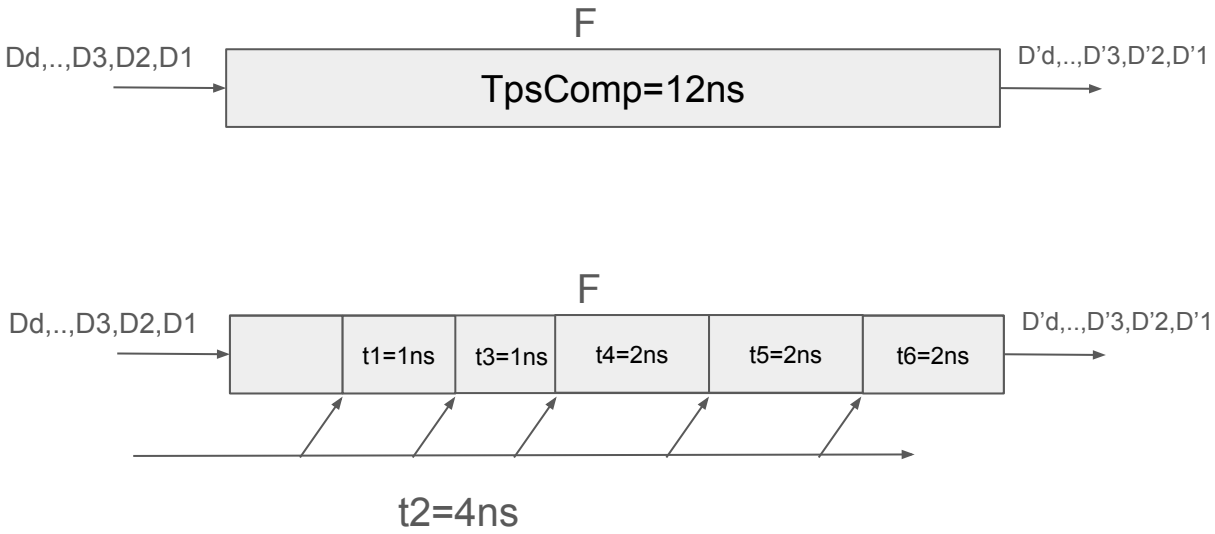
- Let TpsComp the computation time for the global treatment
- Let N the number of pipeline stages
- Let Tcycle the cycle time (cadence of computing)
    - Tcycle = TpsComp / N + Treg



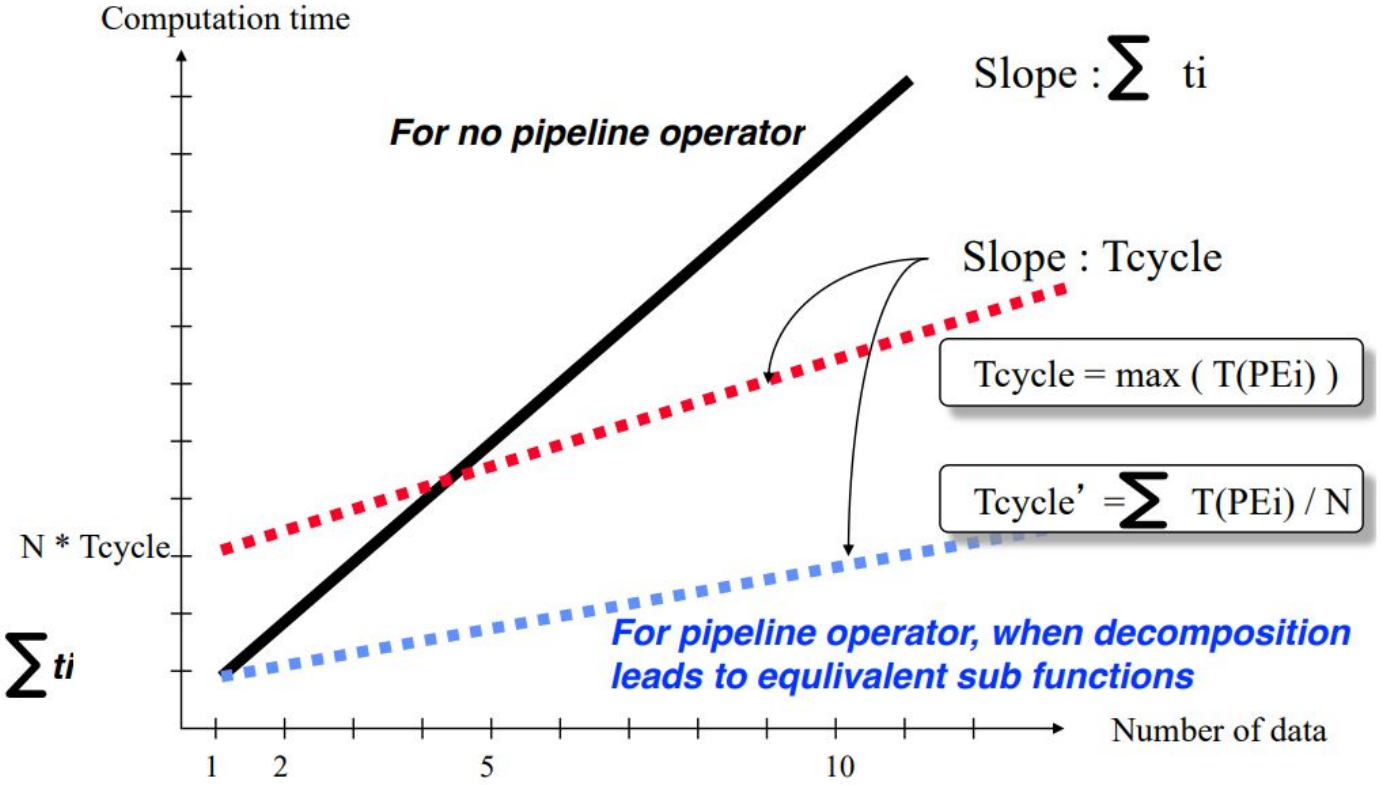
Generalization (cont.)

- Let D the number of data to transform
- Let T(D) the time to transform D data
  - T(D) = (D + N - 1) * Tcycle
- The time to transform one data if given by
  - T(D) / D = (D + N - 1) * (TpsComp / N + Treg) / D
- If D go to infinity and D >> N then the transformation time per data is
  - T(D) / D = TpsComp / N + Treg
- If N go to infinity and D << N then the transformation time per data is
  - T(D) / D = N / D * Treg

Generalization Example



Computation time performance



Thank you for you listening

Advance Computer Architecture and x86 ISA

University of Science and Technology of Hanoi

MS. LE Nhu Chu Hiep

Pipeline Hazard

Pipeline Hazard

- A situations that prevent the next instruction in the instruction stream from executing during its designed clock cycle
- Hazards in pipeline can make it necessary to stall the pipeline
  - Some instructions are allowed to proceed while others are delayed
  - **Cycle loss**
- Reducing the performance of the ideal speedup gained by pipelining
- There are three classes of hazards
  - Structural hazard
  - Data hazard
  - Control hazard

Hazard types

- **Structural hazards**
  - Caused from the resource conflicts when the hardware can not support all instructions simultaneously
- **Data hazards**
  - An instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
- **Control hazards**
  - The pipelining of branches and other instructions that change the PC

Pipeline performance with Stalls

- Speedup equation from pipelining

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}} \\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

- Speedup equation encounting the stall

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

Structural Hazards

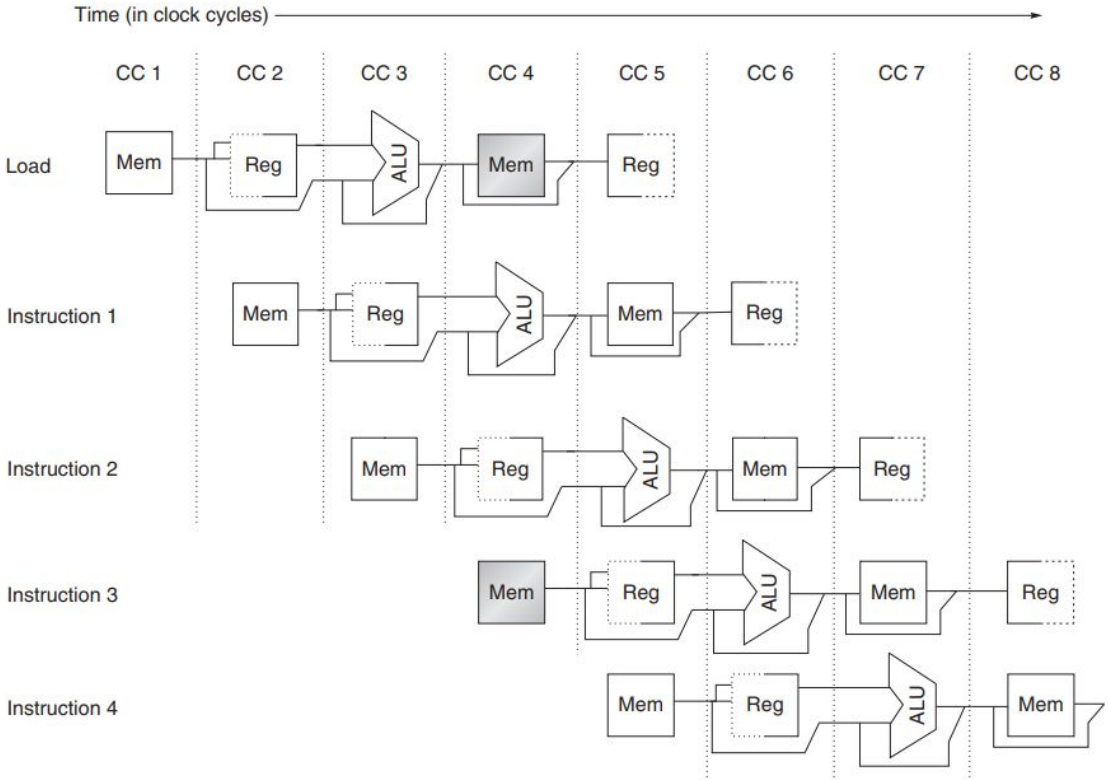
Structural Hazards

- A structural hazard: pipeline cannot be accommodated because of resource conflicts
  - Not fully pipelined functional unit => sequence of instructions cannot proceed at the rate of one per clock cycle
  - Not enough duplicated resources => not allow all combinations of instructions in the pipeline to execute
- Examples:
  - A processor have only one register-file write port but the pipeline need two writes in a clock cycle
  - The pipeline will stall one of the instructions until the required unit is available
  - The stalls increase the CPI from its ideal value of 1

Structural hazard example

- Pipelined processors
  - Shared a single-memory pipeline for data and instructions
  - When an instruction contains a data memory reference
  - And the processor try to fetch a later instruction
  - There are at least two memory access requested simultaneously => conflict !!!
- Resolve hazard
  - When the data memory access occurs
  - Stall the pipeline for 1 clock cycle
  - A pipeline **bubble** - taking space but carrying no useful work

Structural hazard example (cont.)



Structural hazard example (cont.)

	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				Stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Structural hazard (Quest.)

- How much might the load structural hazard cost ?
  - The data references constitute 40% of the mix
  - The ideal CPI of the pipelined processor is 1
  - The clock rate with structural hazard = 1.05 * clock rate without the hazard

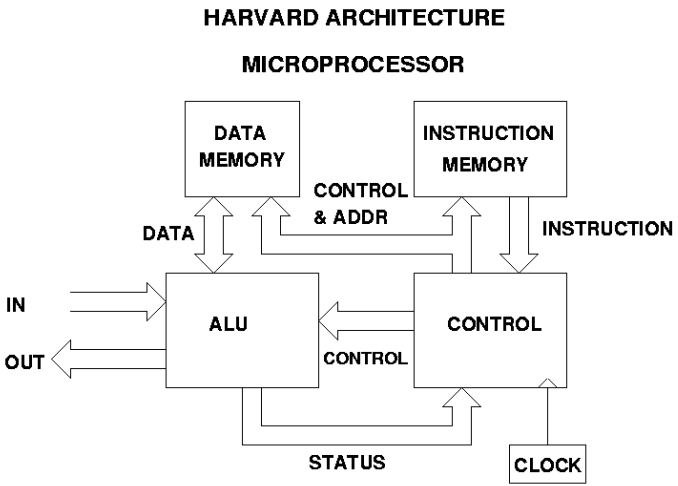
Structural hazard (Answer.)

- How much might the load structural hazard cost ?
  - The data references constitute 40% of the mix
  - The ideal CPI of the pipelined processor is 1
  - The clock rate with structural hazard = 1.05 * clock rate without the hazard
- The simplest method is to compare average instruction time with and without hazard:

$$\begin{aligned}\text{Average instruction time} &= \text{CPI} \times \text{Clock cycle time} \\ &= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05} \\ &= 1.3 \times \text{Clock cycle time}_{\text{ideal}}\end{aligned}$$

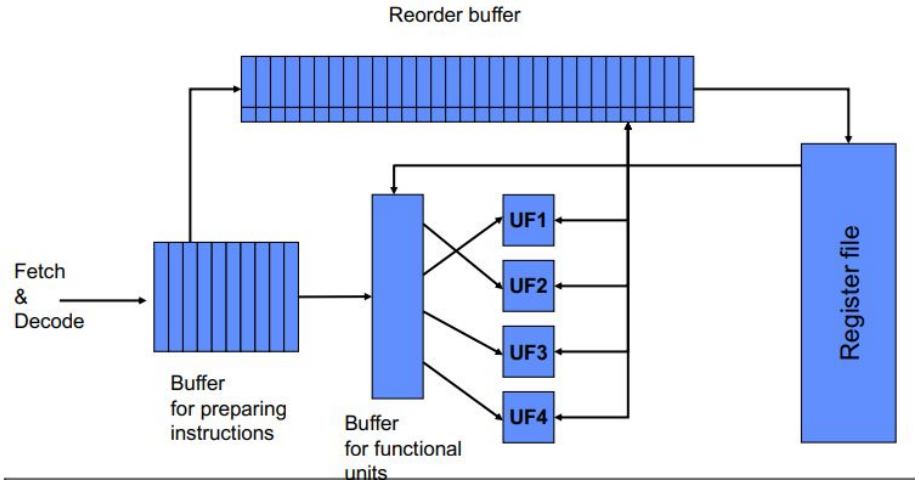
Structural hazard alternative solution

- Harvard model
  - A memory access for data
  - A memory access for instruction
  - No conflict between data reference instruction and next instruction fetching



Structural hazard alternative solution

- Applying set of buffers
  - Reorder buffer
  - **Instruction buffer**
  - Functional buffer

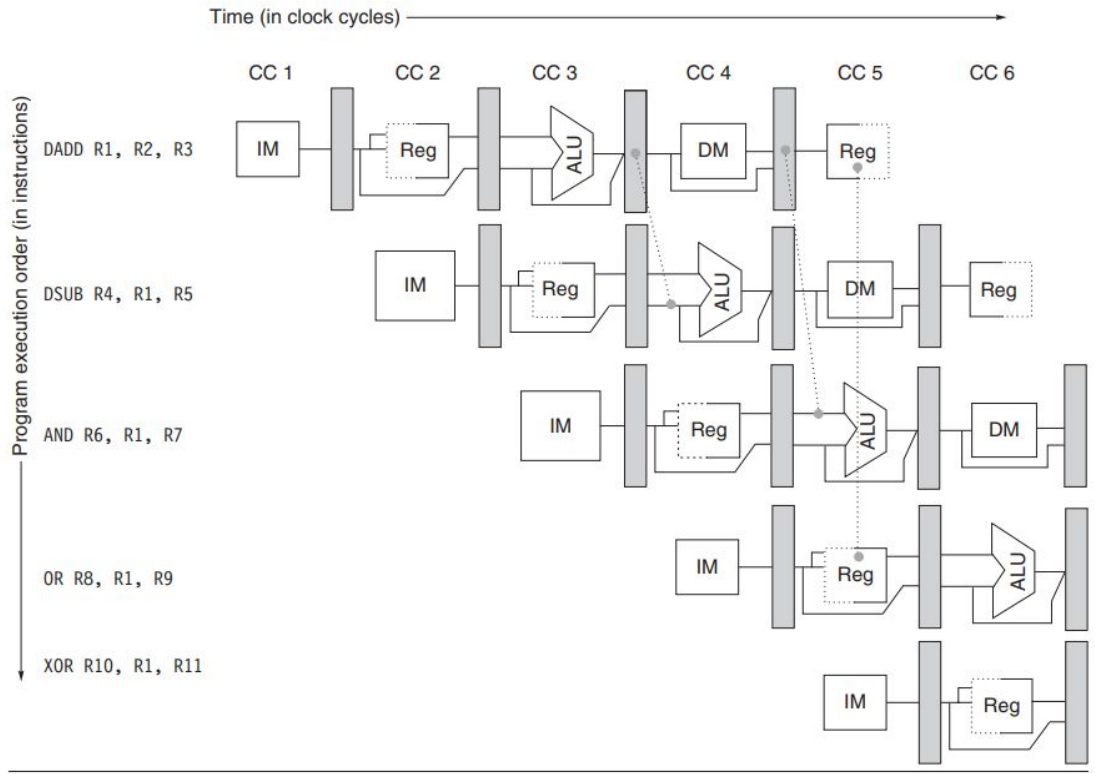


Data Hazards

Data hazard

- Changing the order of read/write access to operands so that order differs from the order seen by sequentially executing
  - The operand values are different between pipelined and non-pipelined processing
  - The stall needed to avoid old operands loaded
- Example
  - DADD R1,R2,R3
  - DSUB R4,R1,R5
  - AND R6,R1,R7
  - OR R8,R1,R9
  - XOR R10,R1,R11

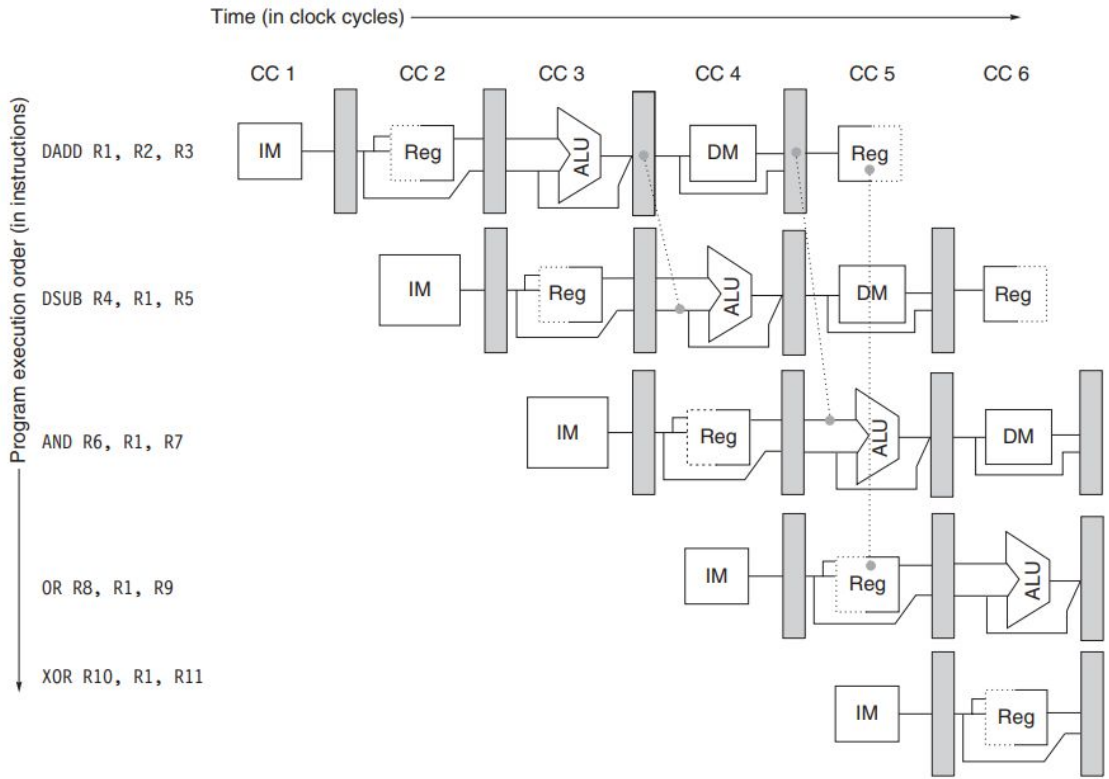
Data hazard (cont.)



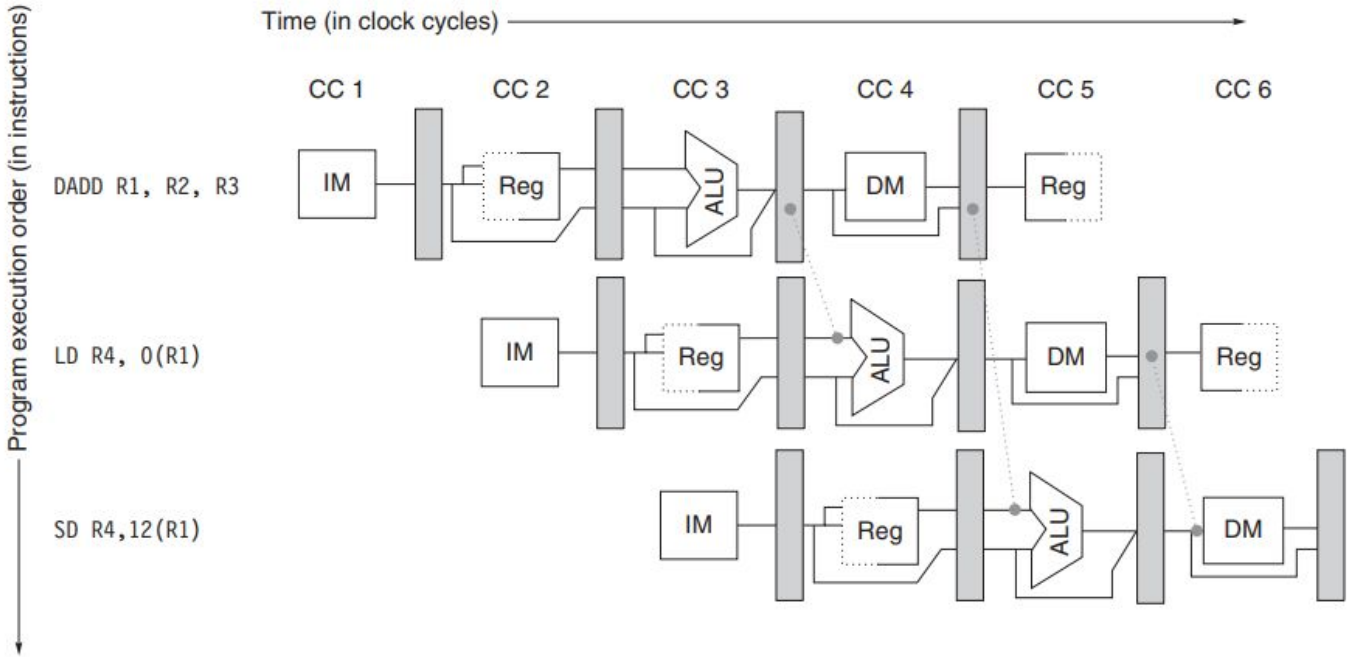
Minimizing data hazard stalls by bypassing

- Bypassing / short-circuiting
  - The result is passing directly to the functional unit that requires it
  - A result is forwarded from the pipeline register corresponding to the output of one unit to the input of same / another unit
  - Then the stall could be avoid

Minimizing data hazard stalls by bypassing



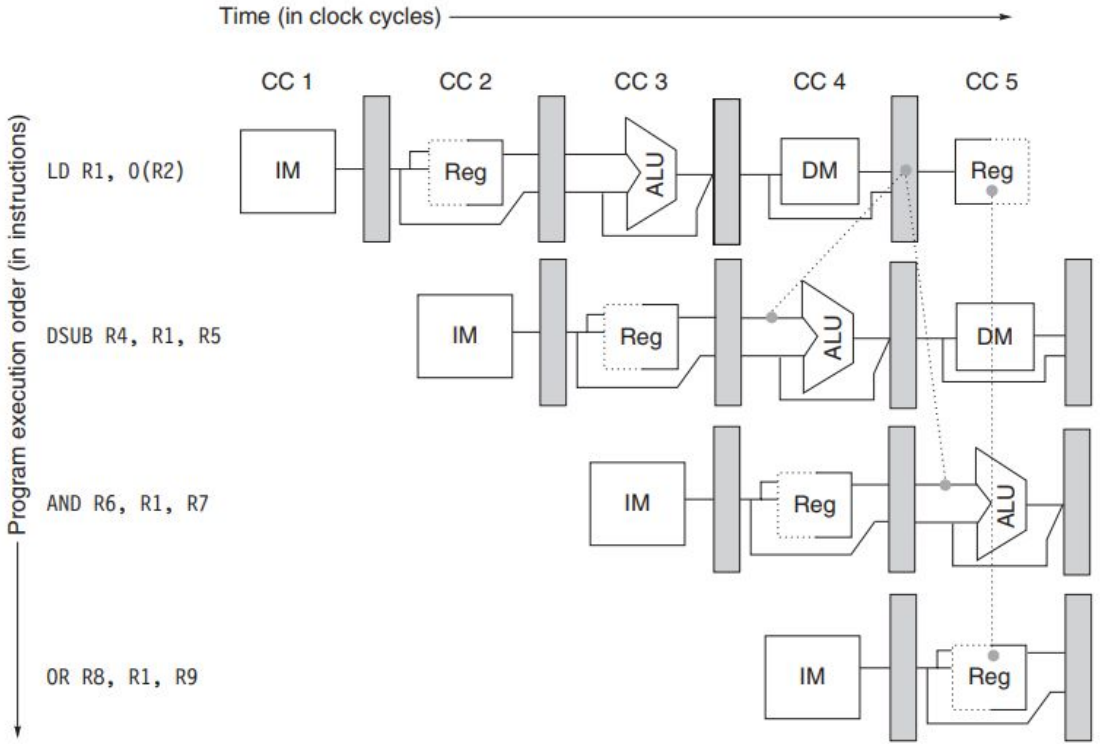
Minimizing data hazard stalls by bypassing (cont.)



Data hazards requiring Stalls

- Not all data hazards can be handled by bypassing
- Example
  - LD R1,0(R2)
  - DSUB R4,R1,R5
  - AND R6,R1,R7
  - OR R8,R1,R9

Data hazards requiring Stalls (cont.)



Data hazards requiring Stalls solution

- The hazard need to be detected
- The stall need to happen in hardware level to delay some cycle
- Invented of hardware **interlock**
  - Detect a hazard
  - Stall / bubble the pipeline until the hazard is cleared
  - CPI for the stalled instruction increases by the length of the stall

Data hazards requiring Stalls solution (cont.)

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND	R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR	R8,R1,R9				stall	IF	ID	EX	MEM	WB

Control Hazards

Control hazards

- The pipelining of branches and other instructions that possibly change the PC
- Recall of branches
  - If the branch changes PC to its target address => taken branch
  - If it falls through => not taken
- The taken branch leads to current loaded instructions in pipeline to be useless and cause great performance loss

Branch hazard stall example

- The branch instruction could not be detected until ID stages
- When the branch is detected, the first IF cycle of branch successor is essentially a stall
- Then a new instruction needs to be fetched following new PC
- **However**, if the branch is untaken, then first IF cycle still has meaning and should not be ignored => **could take advantage**

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Reducing pipeline branch penalties

- Four compile time schemes to deal with branch hazard
  - **Stall the pipeline** until the branch destination is known
  - **Treat every branch as not taken**, allowing the hardware to continue until the branch outcome is definitely known
  - **Treat every branch as taken**, begin fetching and executing the target branch as soon as the branch is decoded and the target address is computed
  - **Delayed branch**

Treat branch as taken / not taken

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Delayed Branch

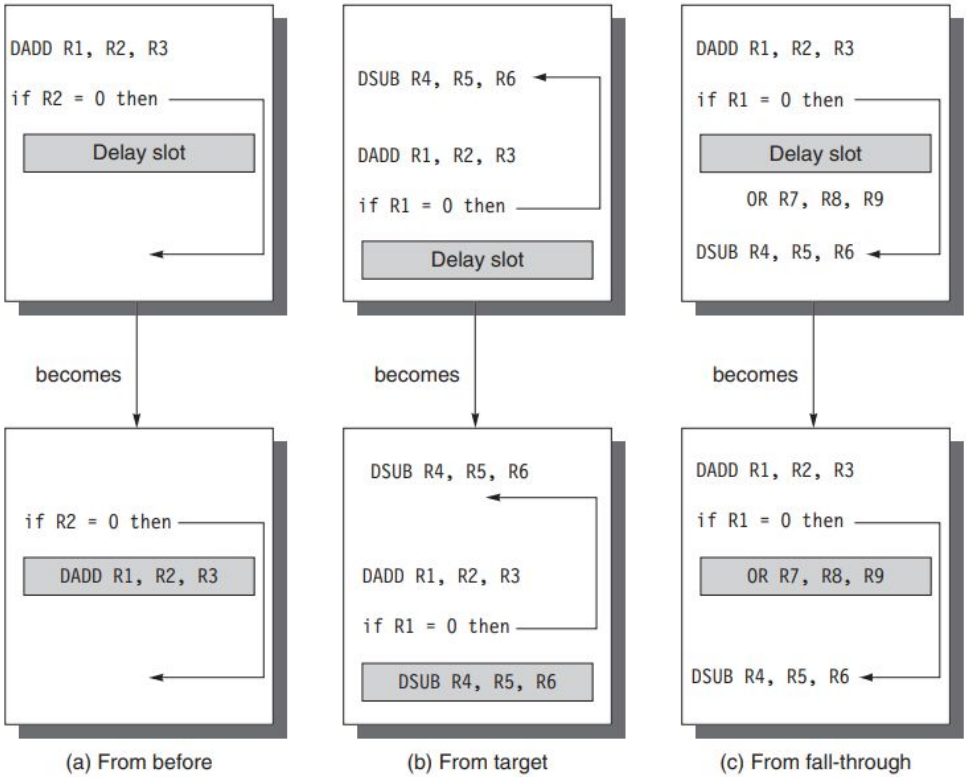
- A technique that intentionally execute instructions in a “branch delay slot” while waiting for the branch outcome is definitely known
- Example: the execution cycle with a branch delay of one is
  - Branch instruction
  - Sequential successor
  - Branch target if taken
- The sequential successor is in the branch delay slot
- This instruction is executed whether or not the branch is taken

Delayed Branch (cont.)

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Scheduling the branch delay slot

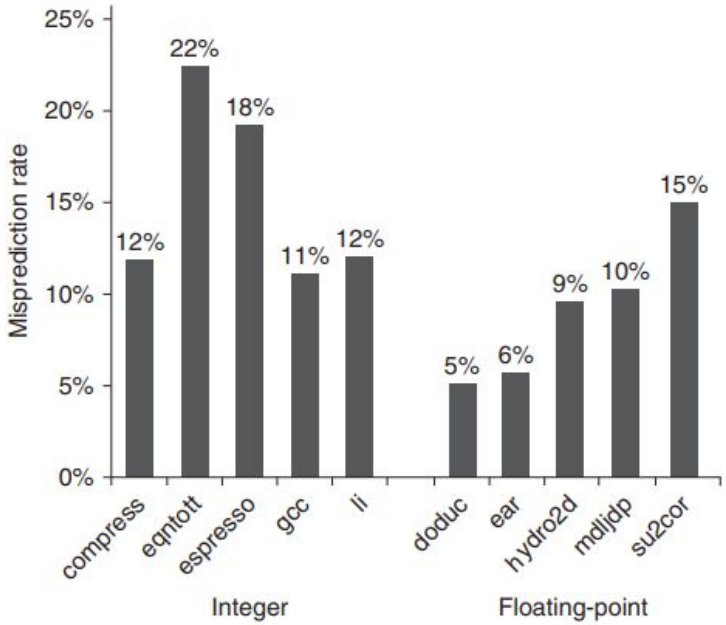


Reducing Cost of Branch through Prediction

Static branch prediction

- Rely on information available at compile time (profile information collected from earlier runs) to propose a branch prediction
- Based on key observation
  - The behavior of branches is often bimodally distributed
  - An individual branch is often highly biased toward taken or untaken
- Effectiveness of scheme depends both on
  - The accuracy of the scheme
  - The frequency of conditional branches

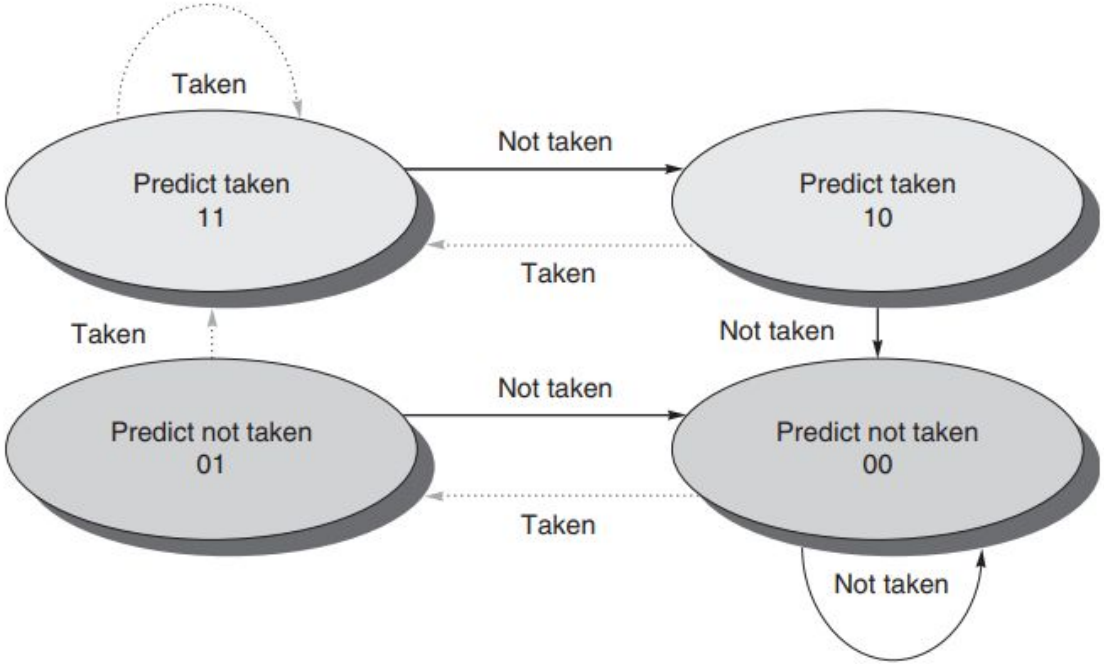
Static branch prediction (cont.)



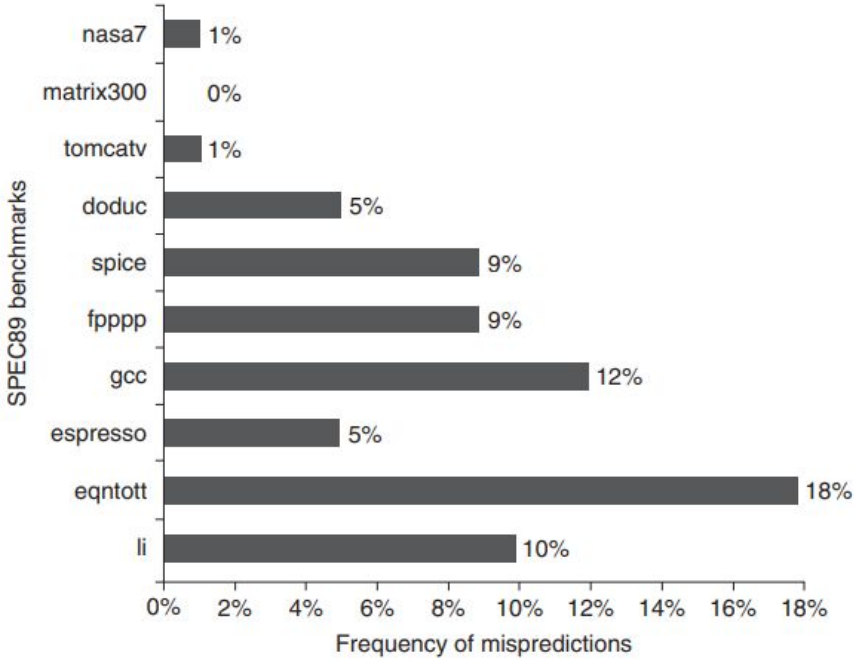
Dynamic branch prediction

- Branch-prediction buffer / Branch history table
  - A small memory indexed by the lower portion of the address of the branch instruction
  - Contains 1 bit that says whether the branch was recently taken or not
  - Simplest sort of buffer to be a hint for the branch prediction direction
- The performance of the buffer depends on
  - How often the prediction is for the branch of interest
  - How accurate the prediction is when it matches
- Improvement of branch-prediction buffer
  - Using 2-bit prediction schemes instead of 1-bit schemes
  - The prediction must miss twice before it is changed between taken and untaken
  - Based on a finite-state processor to provide prediction

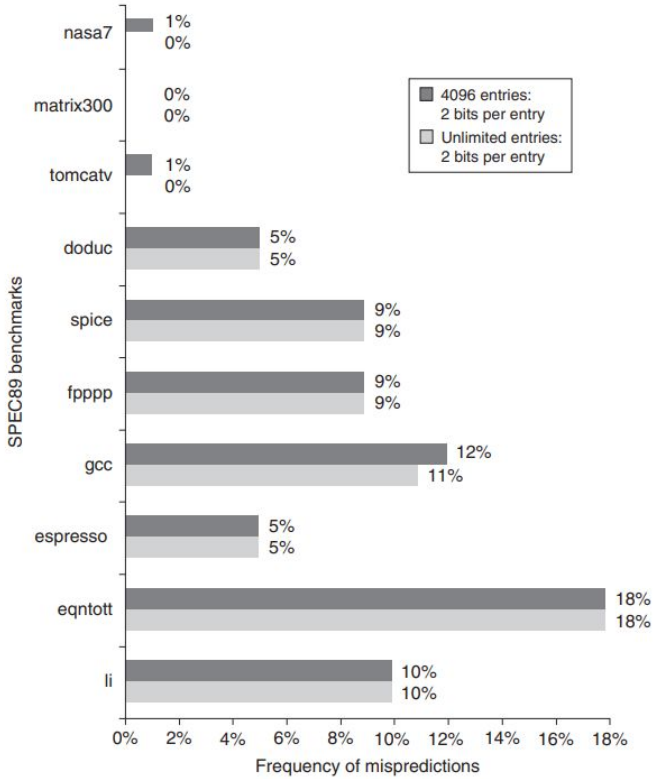
Dynamic branch prediction state machine



Dynamic branch prediction performance



Dynamic branch prediction performance (cont.)



Thank you for you listening

Advance Computer Architecture and x86 ISA

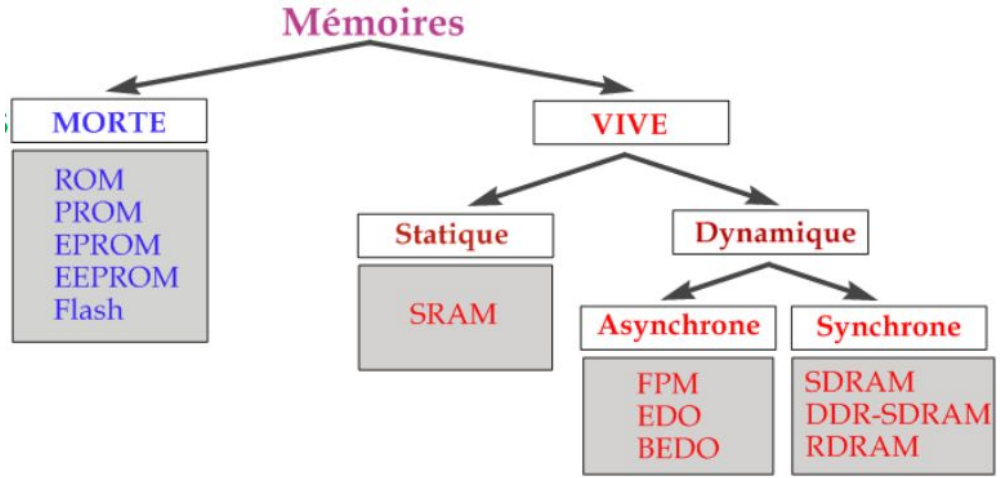
University of Science and Technology of Hanoi

MS. LE Nhu Chu Hiep

Introduction

Recall

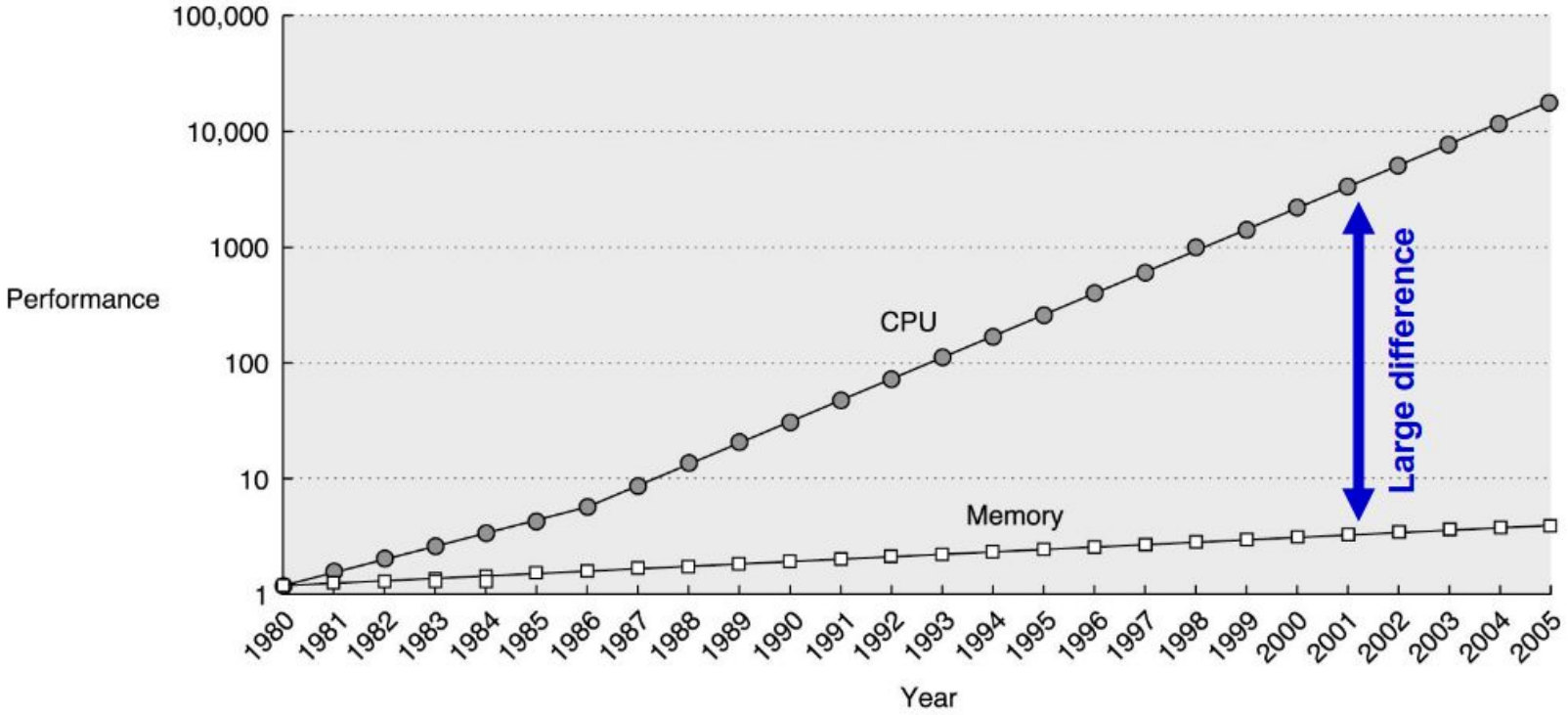
- The memories
  - Store program data and instructions
- Memory types
  - Permanent memory
  - Temporary memory
- Permanent memory
  - Nonvolatile, large, cheap, slow
- Temporary memory
  - Volatile, small, expensive, fast



Why defining a memory organization ?

- Application complexity increase
  - Data to store dramatically increases
  - Size of memories increases
- Evaluation of computer
  - 1980, computer has just few kilobytes of memory
  - Today, several gigabyte of memory are needed
- The memories must be
  - Very **large** in size
  - But also need to propose **small** access time
  - **Contradiction !!!**

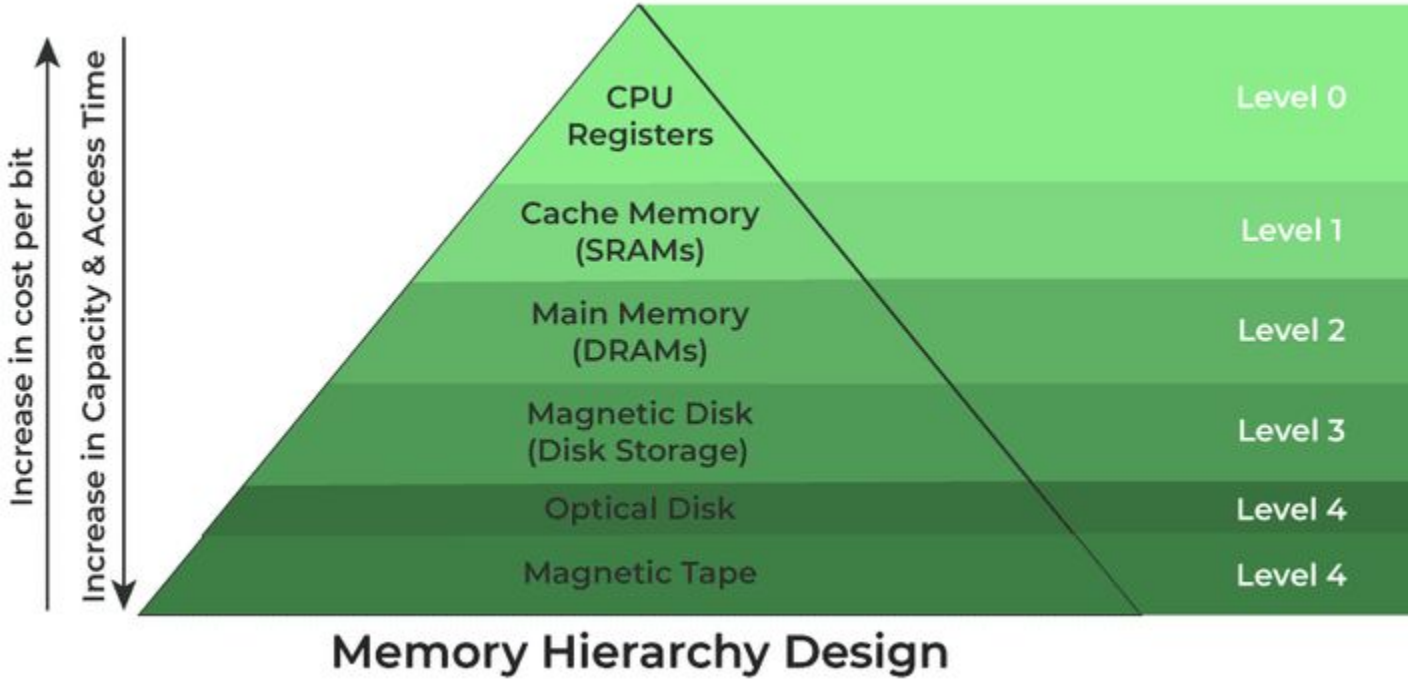
Evaluation of processors and memory performance



Example of access time

Technologies	Access time	Human time	Capacities	Price $ / Mo
Register	1 à 2 ns	1 s	64 * 64 bits	Include in the Processor
Internal cache	3 ns	3 s	32 ko	Include in the Processor
External cache	25 ns	25 s	4 Mo	40
Main Memory	200 ns	3 min	1 Go	2,5
Drive	12 ms	139 jours	10 Go	0,010
Tape	10 à 20 s	315 ans et plus	50 Go	< 0,05

Level representation of memories

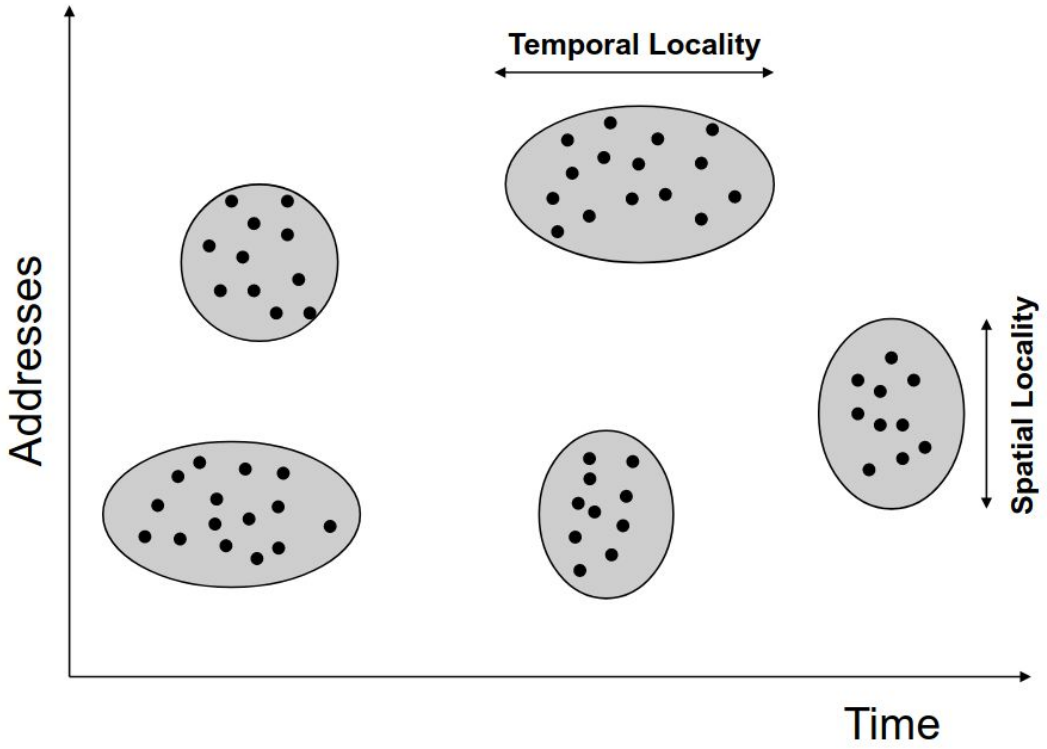


Localities

Principle of locality

- The tendency of a processor to access the same set of memory locations repetitively over a short period of time
- Spatial locality
  - If the processor execute instruction from address @i, it probably want to execute the instruction stored at address @i+1
- Temporal locality
  - If the processor accesses to data d at time t, he will probably reuse this data d in a short time

Localities of memory access



Example of localities

```
for (i = 0 ; i < N ; i++) {
    for (j = 0 j < N ; j++) {
        y[i] =y[i] + a[i][j] * x[j] ;
    }
}
```

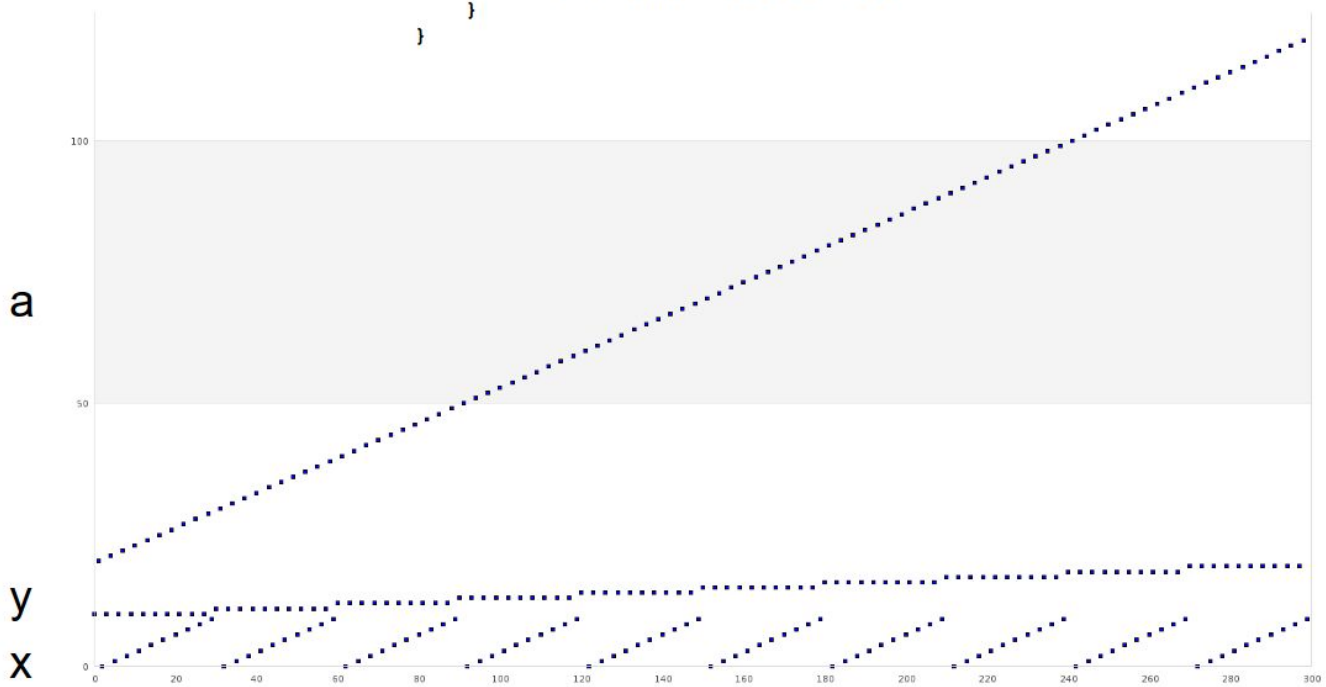
```
Boucle   LD      R0, R1
         ADD     R2, R3, R4
         MULT    R5, R2, R7
         SUB     R10, R10, #1
         BRnz    R10, Boucle
```

- y[i] : spatial and temporal localities
- a[i][j] : spatial locality
- x[j] : spatial and temporal localities

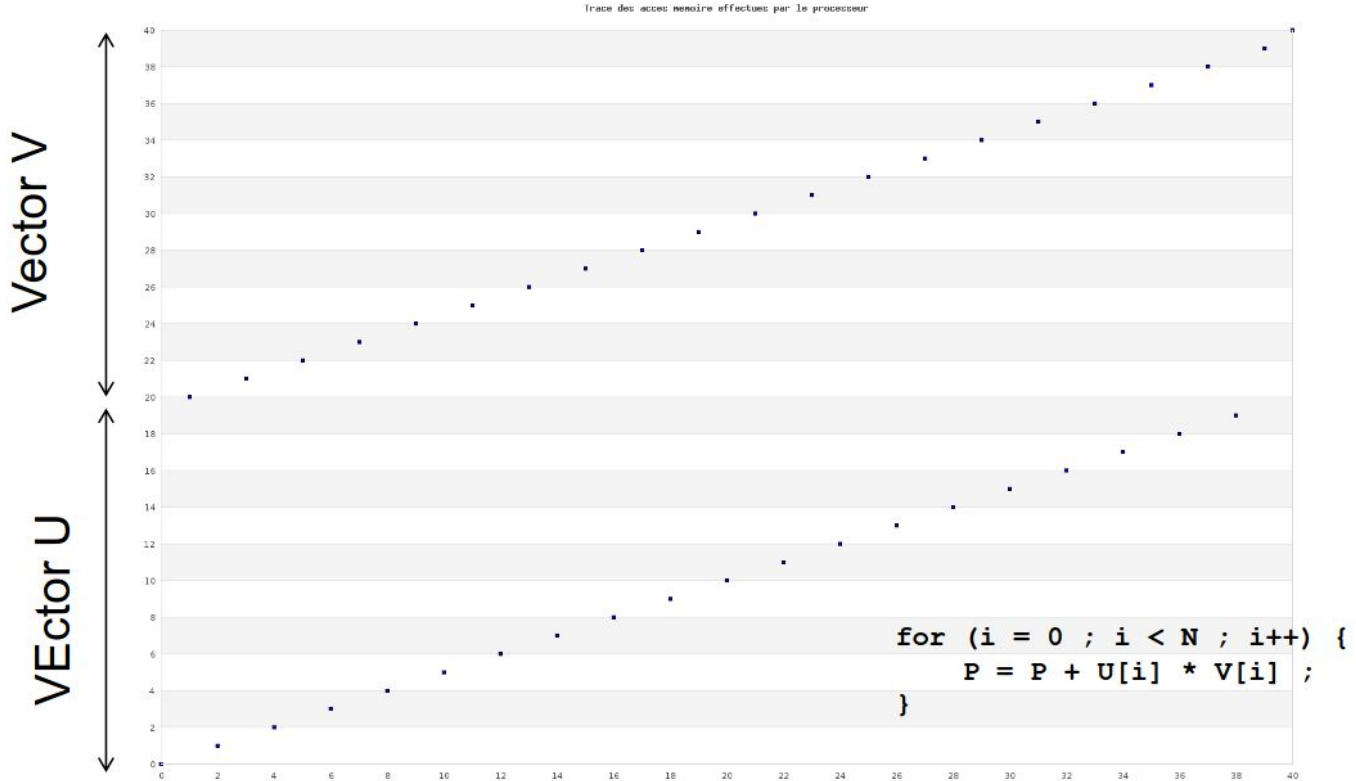
- In the loop, large reuse of instructions

Example of data access localities

```
for (i = 0 ; i < N ; i++) {
    for (j = 0 j < N ; j++) {
          y[i] =y[i] + a[i][j] * x[j] ;
    }
}
```



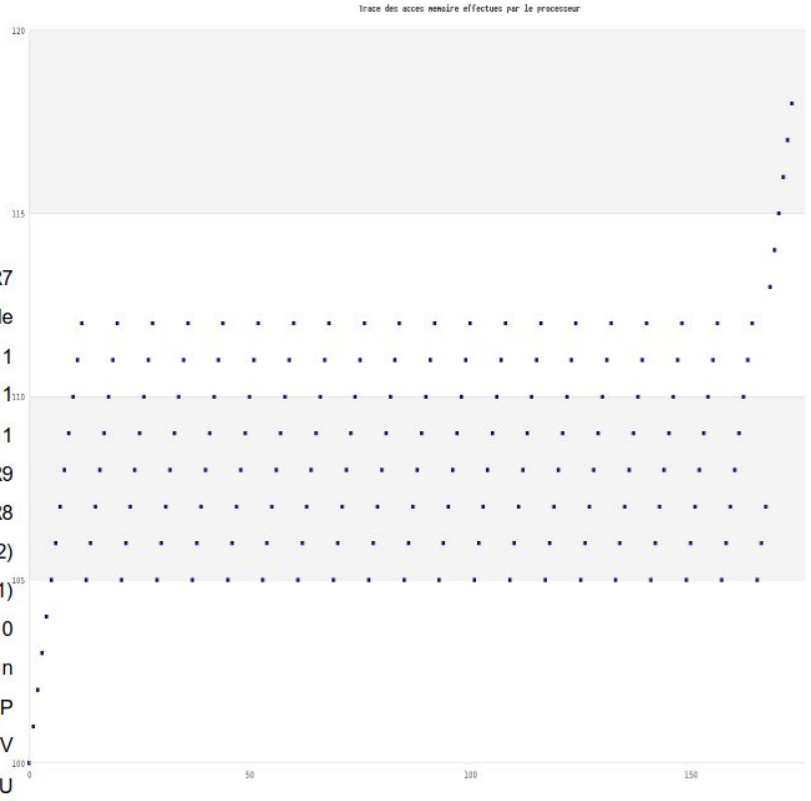
Example of data access localities for Scala product



Example of instruction access localities for Scala product

Load in memory

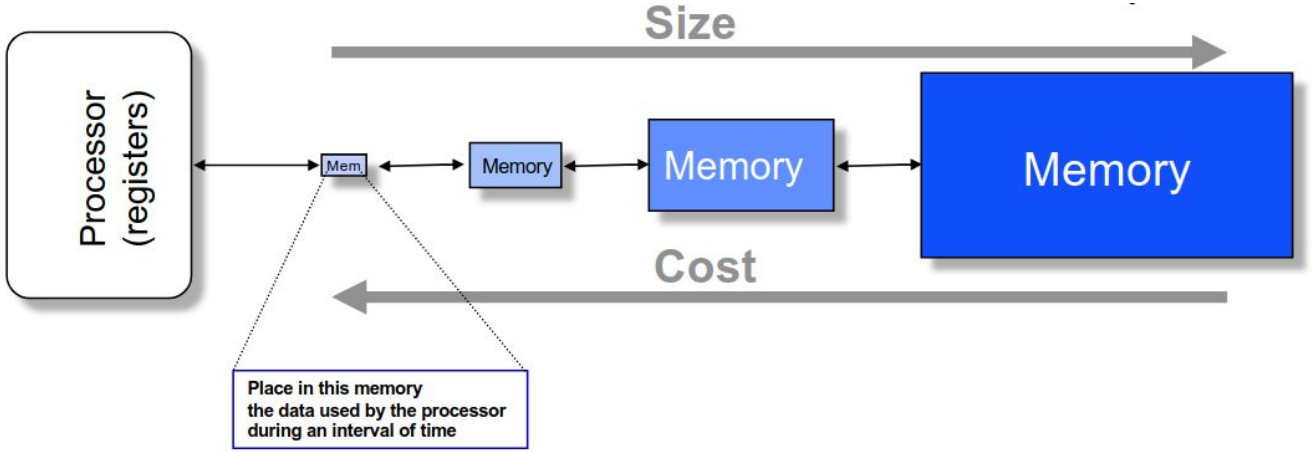
```
program 100
            li R1, U
            li R2, V
            li R3, P
            li R4, n
            li R7, 0
boucle
            li R8, (R1)
            li R9, (R2)
            mult R9, R9, R8
            add R7, R7, R9
            add R1, R1, 1
            add R2, R2, 1
            sub R4, R4, 1
            brnz R4, boucle
            si (R3), R7
            exit
end
```



Memory Organization

Idea

- Place a hierarchy of memory
  - Small memory near the processor, fast, but small
  - Large memory far the processor, large but slow



Performance of a memory hierarchy

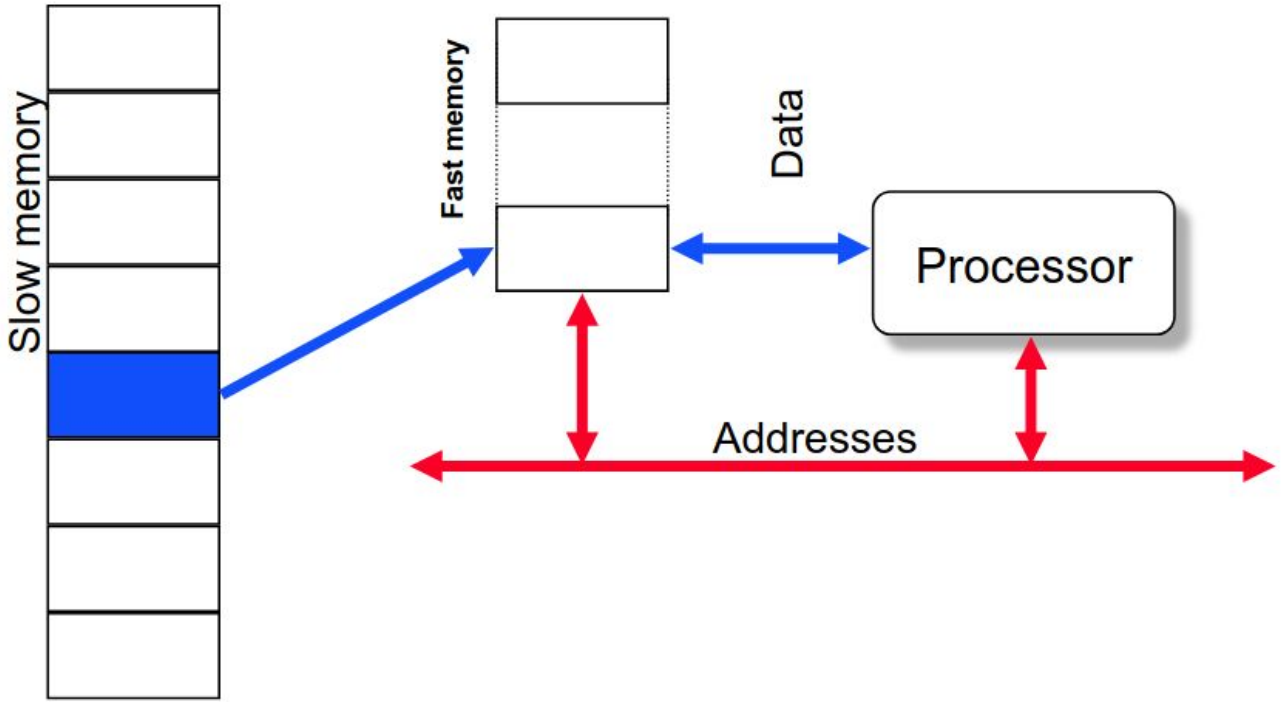
- Performance of a memory hierarchy

$$\text{Perf} = \frac{\text{Time with fast memory}}{\text{Time with the memory hierarchy}}$$

- Gain obtained with memory hierarchy

$$\text{Gain} = \frac{\text{Time with slow memory}}{\text{Time with the memory hierarchy}}$$

How it work ?



Performance and gain

Let N : the block size
P : the number of adat read or written in the block
Ts : access time for slow memory
Tf : access time for fast memory

$$\text{Perf} = \frac{P * Tf}{2 * N * Ts + P * Tf}$$

$$\text{Acc} = \frac{P * Ts}{2 * N * Ts + P * Tf}$$

Performance and gain (cont.)

- Condition to ensure for efficient memory hierarchy

$$2 * N * Ts + P * Tf < P * Ts$$

- Interest of memory hierarchy

$$P > \frac{2 * N * Ts}{Ts - Tf}$$

Interest of memory hierarchy

- The more the difference between memories is, the more the data must be reused
- If P is small, then the memory hierarchy is not so interesting
- Example:
  - Let N = 256, Ts = 2 * Tf

$$P > \frac{2 * 256 * 2 * Tf}{2 * Ts - Tf}$$

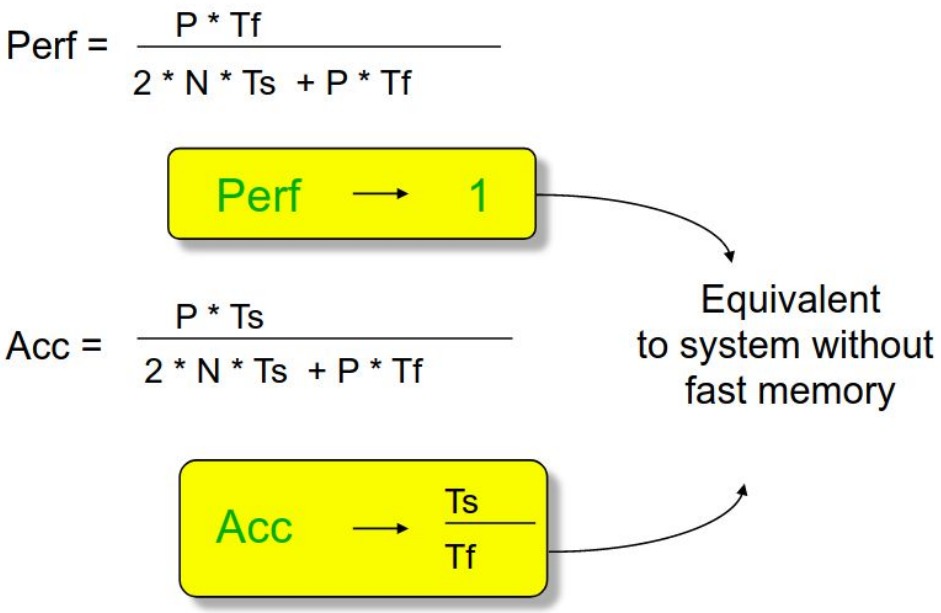
$$P > 2 * 256 * 2$$

$$P > 4 * 256$$

In average, each data must be read/write 4 times

Interest of memory hierarchy (cont.)

- If P -> infinity then:



Thank you for you listening

Advance Computer Architecture and x86 ISA

University of Science and Technology of Hanoi

MS. LE Nhu Chu Hiep

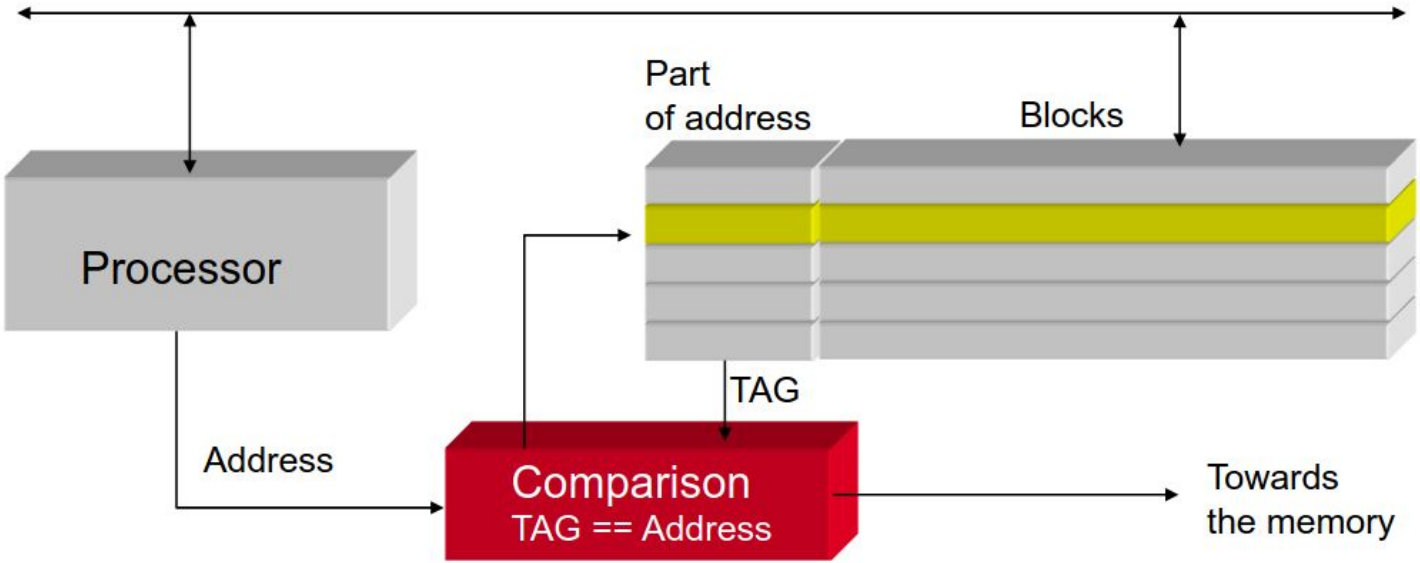
Cache

Cache Memory

- Fast memory
  - Access time and cycle time of processor are very closed
- Cache taxonomy
  - **Cache miss**: A cache default appears when the data required by the processor is not in the cache memory
  - **Cache hit**: A cache appears when the data required by the processor is in the cache memory
- The memory cache copies some small blocks of main memory

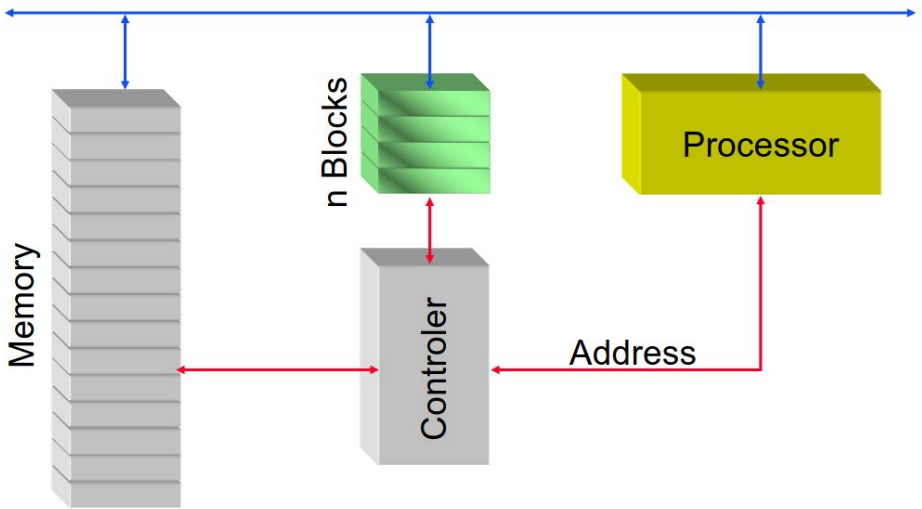
Cache blocks and memory blocks

- Data (or blocks of data)
- addresse (part of address)



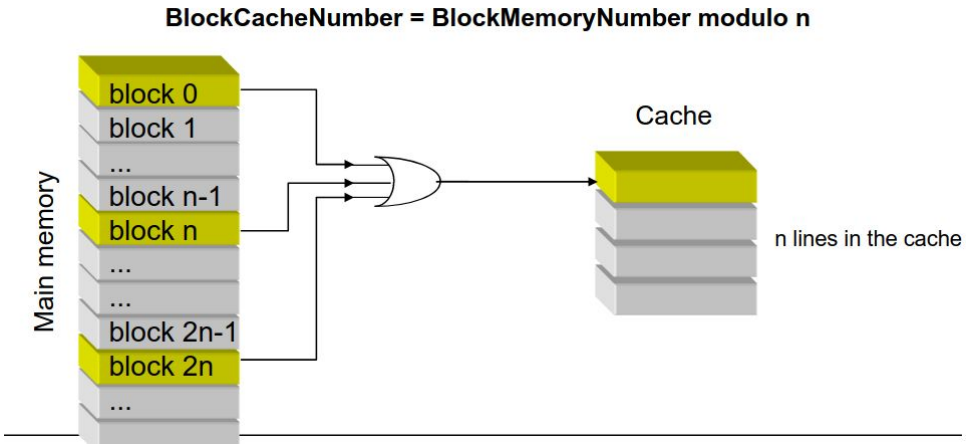
Cache blocks and memory blocks (cont.)

- The cache memory is divided into n blocks
- The main memory is divided into N blocks
- N >> n



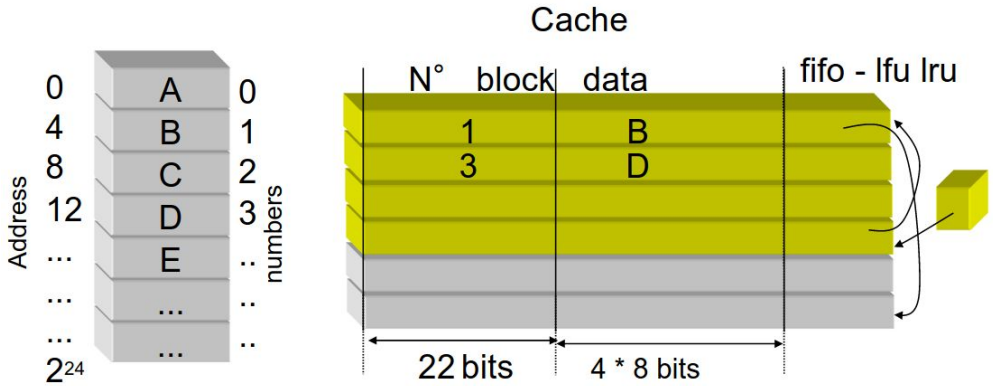
Cache block organization

- Cache memory with direct associativity
  - A bock have a fix location in the cache
  - Simple technique, management of storage needs small logic
  - No optimal technique to replace a block
  - The block i from memory is stored in the cache in the block



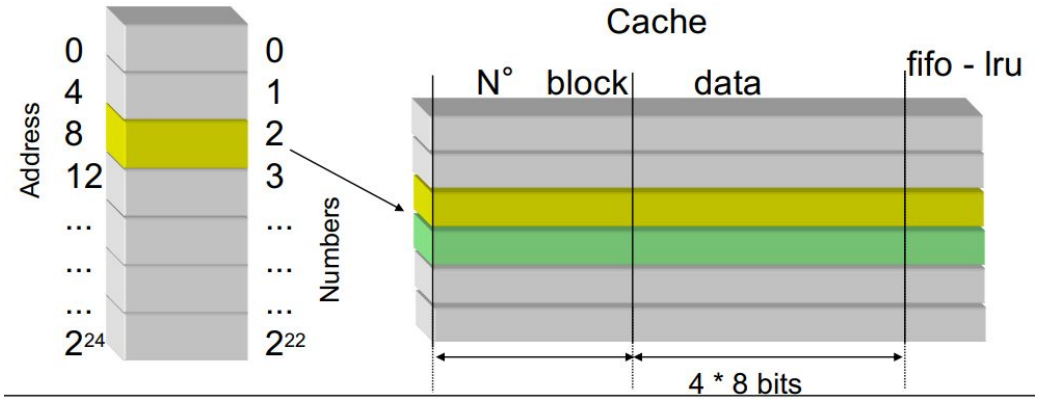
Cache block organization (Associative memory)

- A block can have different location in the cache, it depends on the state of cache when a new block must be stored
- Enable to implement a more adapted replacement policy
  - Random : Random replacement
  - Fifo : First in first out
  - Lru : The last recently used
  - Lfu : The least frequently used



Cache block organization (N-way Associative memory)

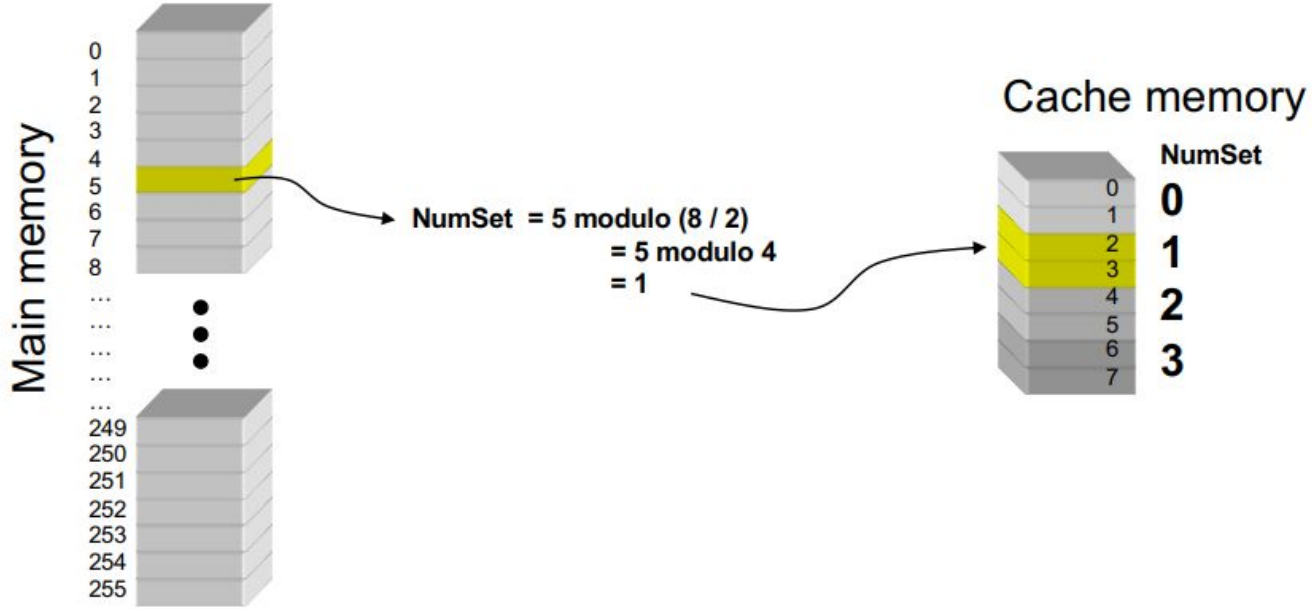
- A block can have N locations in the cache memories
- Simplification of control
- Generally sufficient to ensure good performance
- Policies to replace in set of clocks
  - Random : Random replacement
  - Fifo : First in first out
  - Lru : The last recently used
  - Lfu : The least frequently used



Example: find a new location in the cache

- If N if the number of the main memory block
- If L is the number of cache lines
- If V is the number of way for the associativity
- Then
- NumSet = N mod (L/V)
- The numbers of possible blocks are:
    - NumSet * V <= NumBlocksCache <= (NumSet + 1) * V -1

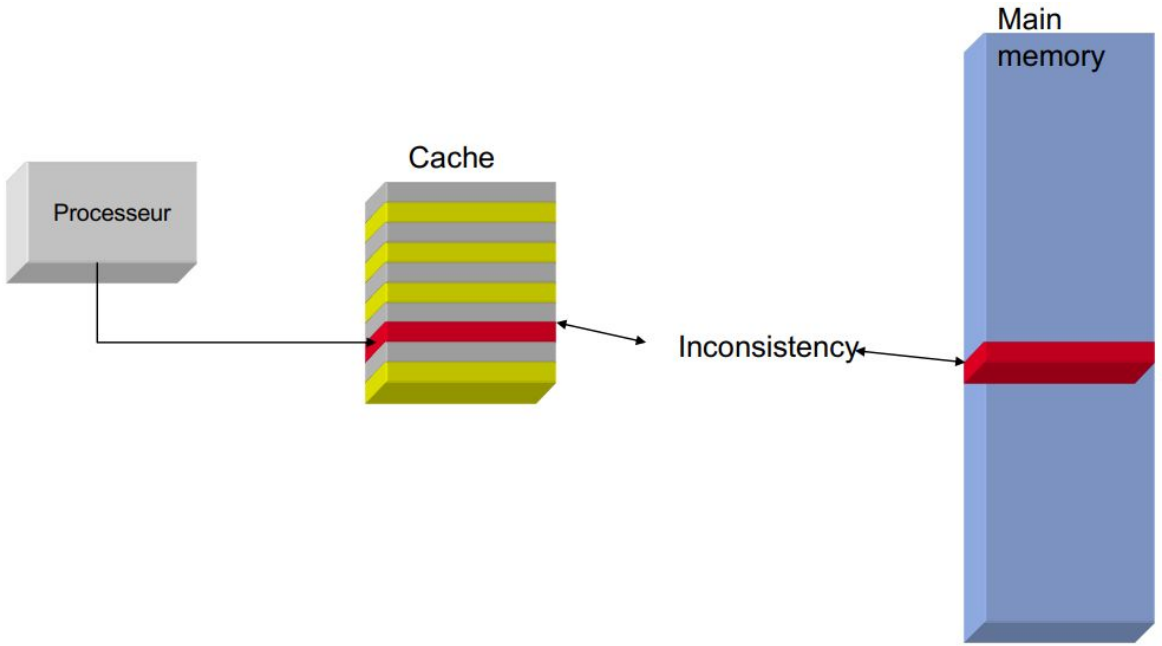
Example: find a new location in the cache (cont.)



Cache Synchronize

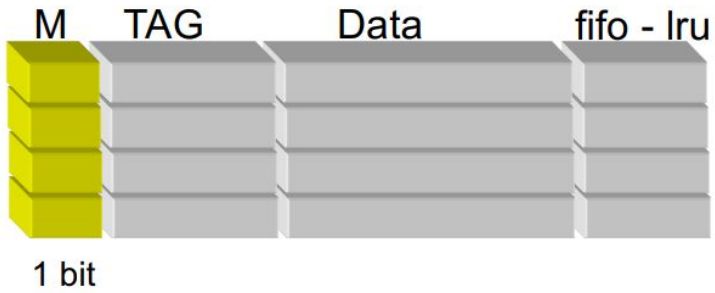
Policy for writing in the main memory

- When the processor modifies a value stored in a cache block



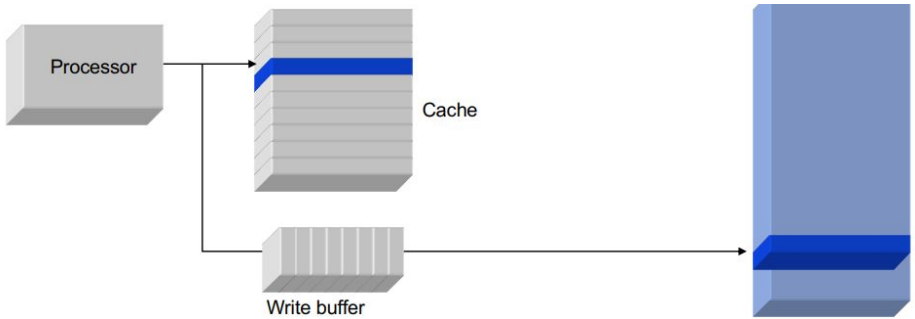
Policies to update the main memory (Write back)

- Write back
  - Update is done when block replacement
  - The memory and cache can stored inconsistency blocks
  - A cache miss on modified block is costly
    - Need a block copy to main memory, then a copy from memory of the new block
  - The cache stores the information about block modification: bit M



Policies to update the main memory (Write through)

- Write through
  - Update of main memory is done at each write operation in the cache
  - The main memory and the cache are always consistent
  - No need a bit to store inconsistency
  - The memory access bus is overload by all the write operation
    - These write operations are slow
    - Solution: a write buffer places between processor and main memory



Policies for writing a data which is not in cache

- Write through, write allocate
  - The block is placed in cache, the write operation is done in the cache
    - The penalty is the same as classical miss
    - The block must be copy from memory to cache, then vice versa
- Write through, no write allocate
  - The write operation is done in the main memory, not in cache
    - The next write operation in the same blocks provokes a new write operation in the main memory
- Write back
  - The write operation is done when the block is ejected from the cache memory

		Write in cache memory	
		Yes	No
Write in main memory	Yes	Write through Write allocate	Write through No write allocate
	No	Write Back	

Cache memory simulations

- Example: matrix product (2 matrices 32 * 32)
  - Unified cache memory
  - Total number of access: 269476
  - Cache memory size 1024 octets
  - Simulations for cache having
    - The following block sizes:
      - 4, 8, 16, 32, 64, 128
    - The following replacement policies:
      - Direct, LRU, FIFO, Random
    - The following associativities (between 1 to Nb Blocks, always power of 2)

```
for (i = 0 ; i < N ; i++) {
    for (j = 0 j < N ; j++) {
            c[i][j] = 0;
            for (k = 0 ; k < N ; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j] ;
            }
    }
}
```

Example: the program of matrix product

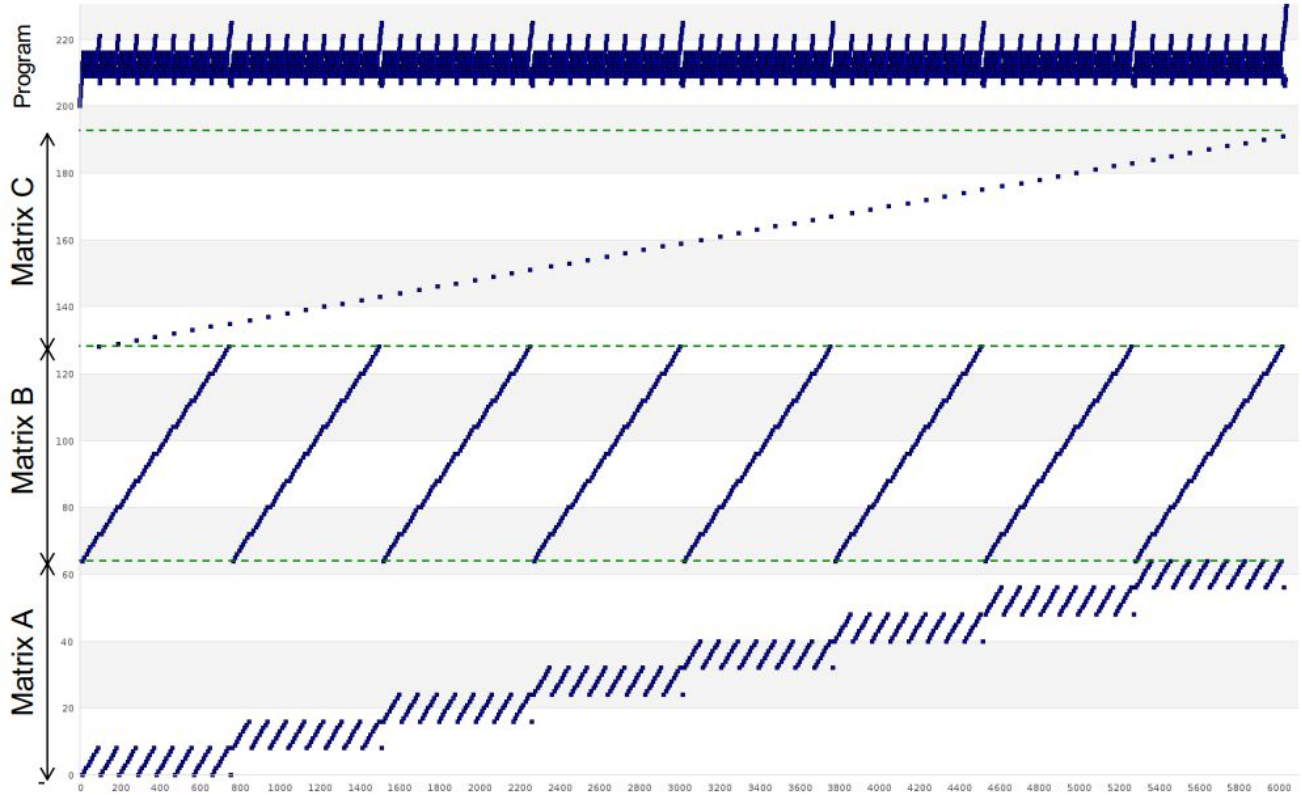
- The program

```
1600        MOVE R1, @a00
            MOVE R2, @b00
            MOVE R3, @c00
            MOVE R4, N
boucle1
            MOVE R5, N
boucle2
            MOVE R6, N
            MOVE R7, 0
boucle3
            MOVE R8, (R1)++
            MOVE R9, (R2)++
            MULT R8, R9
            ADD  R7, R8
            SUB  R6, 1
            BRnz boucle3
            MOVE (R3)++, R7
            SUB  R1, N
            SUB  R5, 1
            BRnz boucle2
            MOVE R2, @b00
            ADD  R1, N
            SUB  R4, 1
            BRnz boucle1
```

- Access list

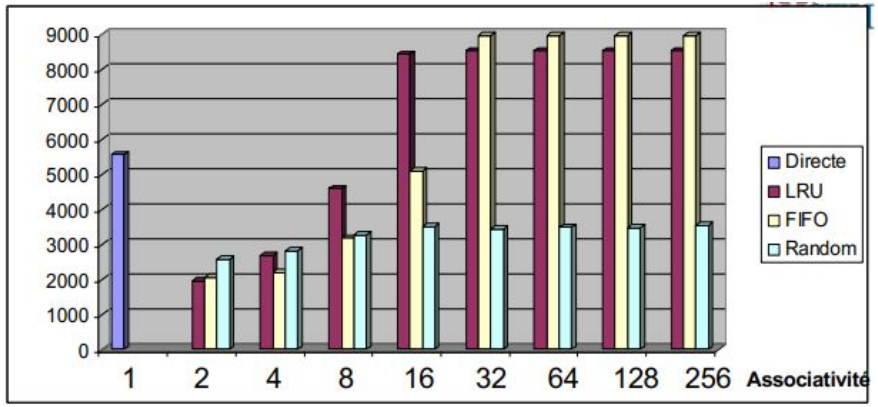
```
2 640 -- Acces a l'instruction MOVE R1,@a00
2 641 -- Acces a l'instruction MOVE R2,@b00
2 642 -- Acces a l'instruction MOVE R3,@c00
2 643 -- Acces a l'instruction MOVE R4,N
2 644 -- Acces a l'instruction MOVE R5,N
2 645 -- Acces a l'instruction MOVE R6,N
2 646 -- Acces a l'instruction MOVE R7,0
2 647 -- Acces a l'instruction MOVE R8,(R1)++
0 0 --- Acces A[0][0]
2 648 -- Acces a l'instruction MOVE R9,(R2)++
0 40 --- Acces B[0][0]
2 649 -- Acces a l'instruction MULT R8,R9
2 64a -- Acces a l'instruction ADD R7,R8
2 64b -- Acces a l'instruction SUB R6,1
2 64c -- Acces a l'instruction BRnz boucle3
2 647 -- Acces a l'instruction MOVE R8,(R1)++
0 1 --- Acces A[0][1]
2 648 -- Acces a l'instruction MOVE R9,(R2)++
0 41 --- Acces B[1][0]
2 649 -- Acces a l'instruction MULT R8,R9
2 64a -- Acces a l'instruction ADD R7,R8
...
...
2 649 -- Acces a l'instruction MULT R8,R9
2 648 -- Acces a l'instruction MOVE R9,(R2)++
0 47 --- Acces B[7][0]
2 649 -- Acces a l'instruction MULT R8,R9
2 64a -- Acces a l'instruction ADD R7,R8
2 64b -- Acces a l'instruction SUB R6,1
2 64c -- Acces a l'instruction BRnz boucle3
2 64d -- Acces a l'instruction MOVE (R3)++,R7
1 80 --- Acces C[0][0]
2 64e -- Acces a l'instruction SUB R1,N
...
...
2 646 -- Acces a l'instruction MOVE R7,0
2 647 -- Acces a l'instruction MOVE R8,(R1)++
0 0 --- Acces A[0][0]
2 648 -- Acces a l'instruction MOVE R9,(R2)++
0 48 --- Acces B[0][1]
2 649 -- Acces a l'instruction MULT R8,R9
```

Localities for 8*8 matrix product

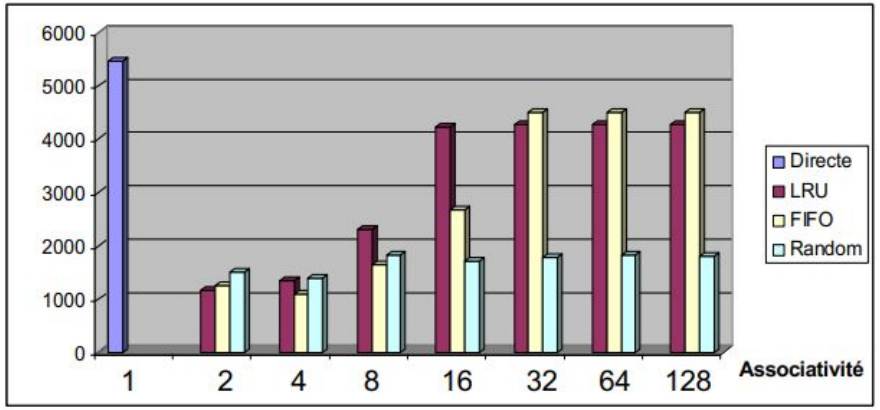


Simulation Evaluation

- Block size = 4

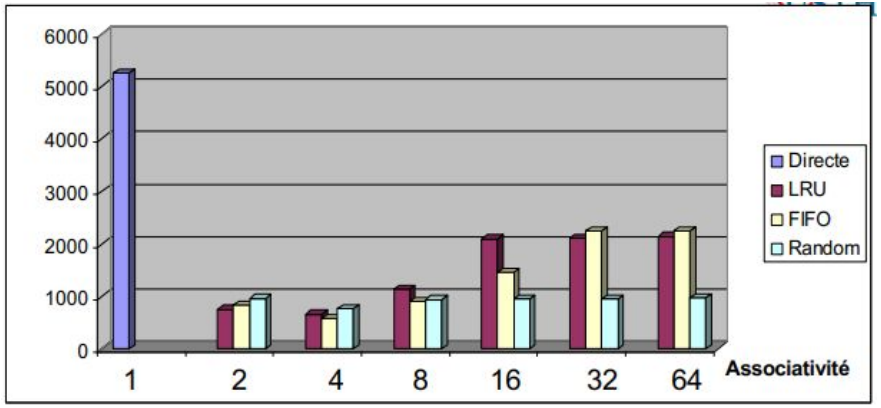


- Block size = 8

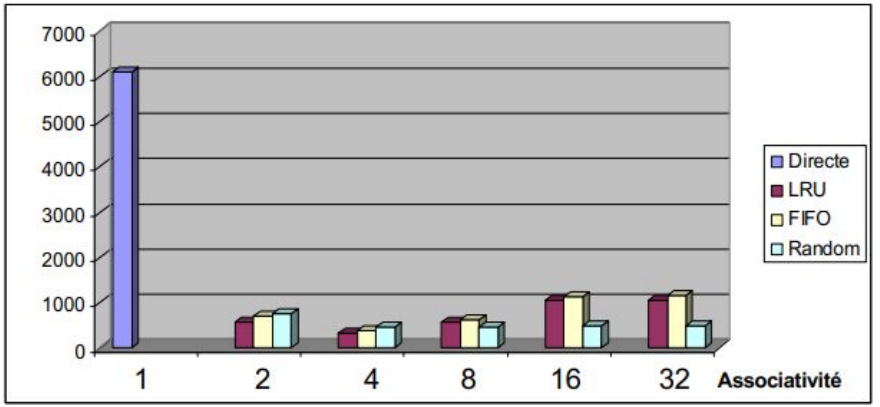


Simulation Evaluation (cont.)

- Block size = 16

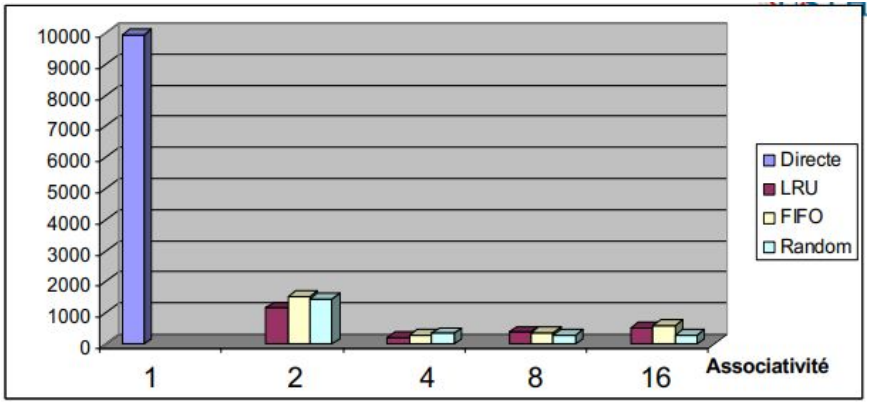


- Block size = 32

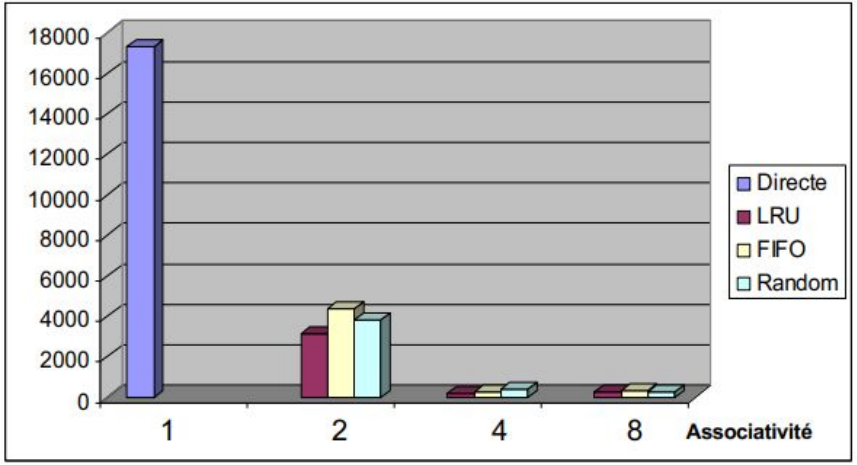


Simulation Evaluation (cont.)

- Block size = 64



- Block size = 128



Conclusions of these simulation

- A small size leads to large number of misses
  - Small size leads to more flexible cache, but large number of misses
- A large associativity does not improve the result
  - In general, the cache memory are associated with a way equal to 2, 4, and 8

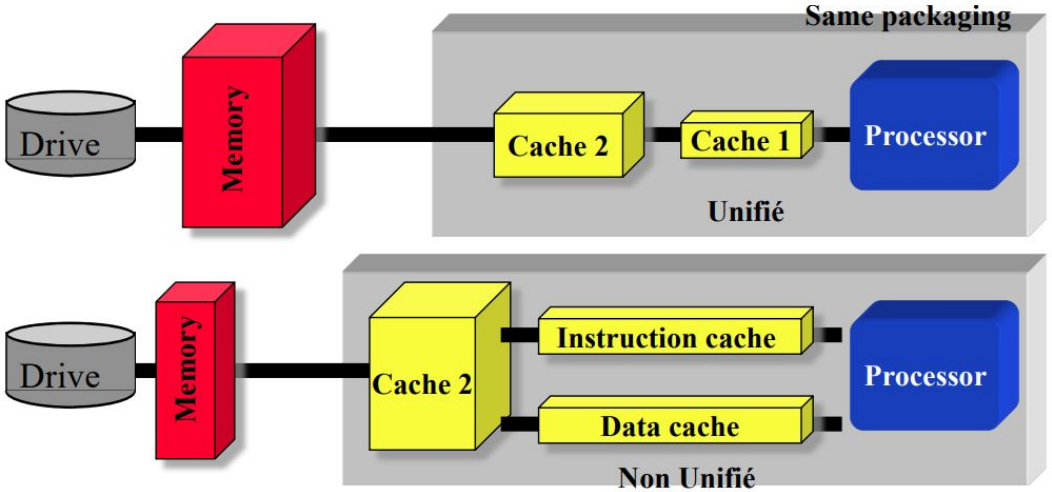
Cache Evolution

Evolutions

- The cache sizes increase
- The access times are reduced
- Caches are more and more often include in the chip
  - Enables to use access time equal to processor cycle
  - Enables to increase the bus between processor and cache
  - More instructions can be loaded at each cycle
- Number of levels increases
- The first level of cache is generally non unified
  - Instruction cache and data cache
- The second level of cache is generally unified

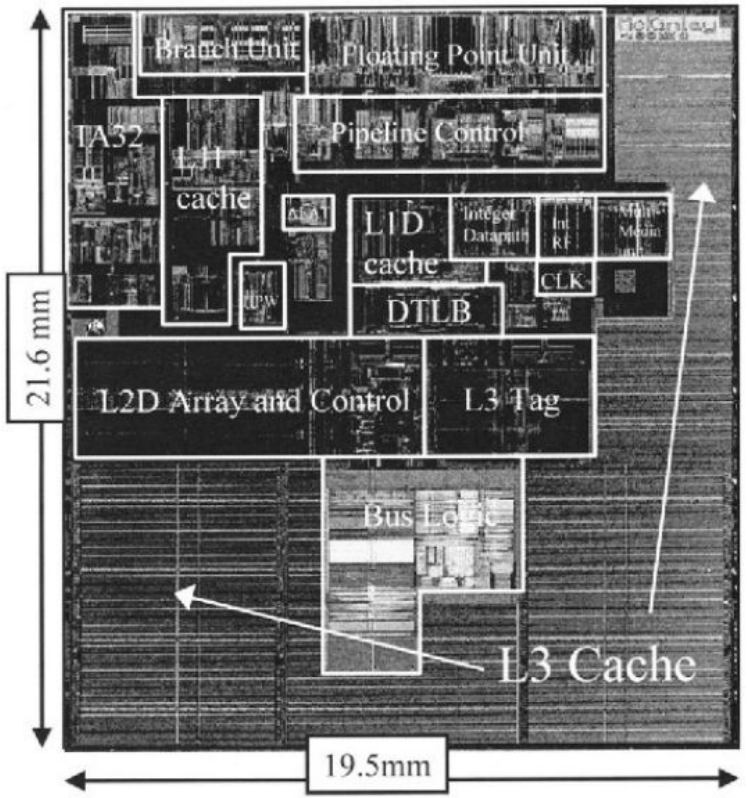
Evolutions (cont.)

- Enables to place different policies for the two different cache
- Supports no blocking cache: during a miss on data cache access, it is possible to continue with the other cache



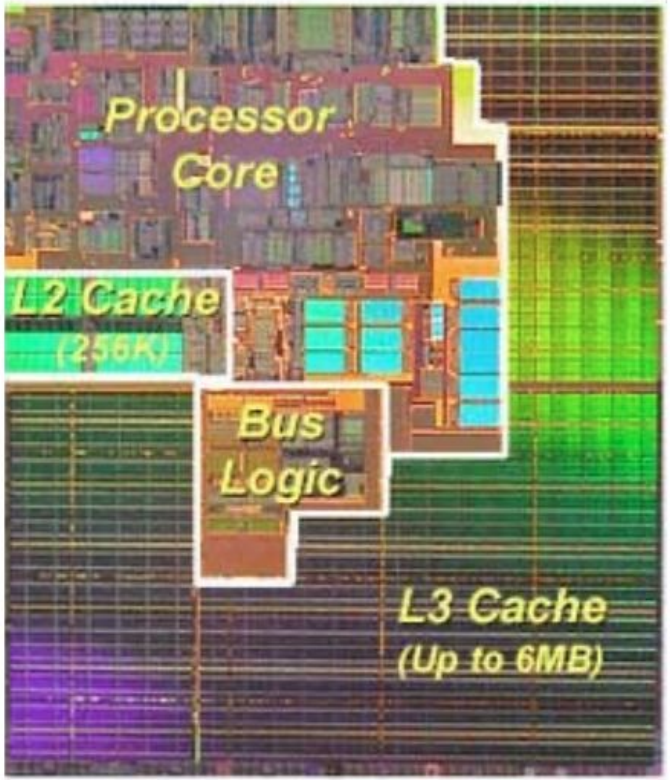
Third level of cache

- Solution developed for Itanium 2
  - 3MB for level 3



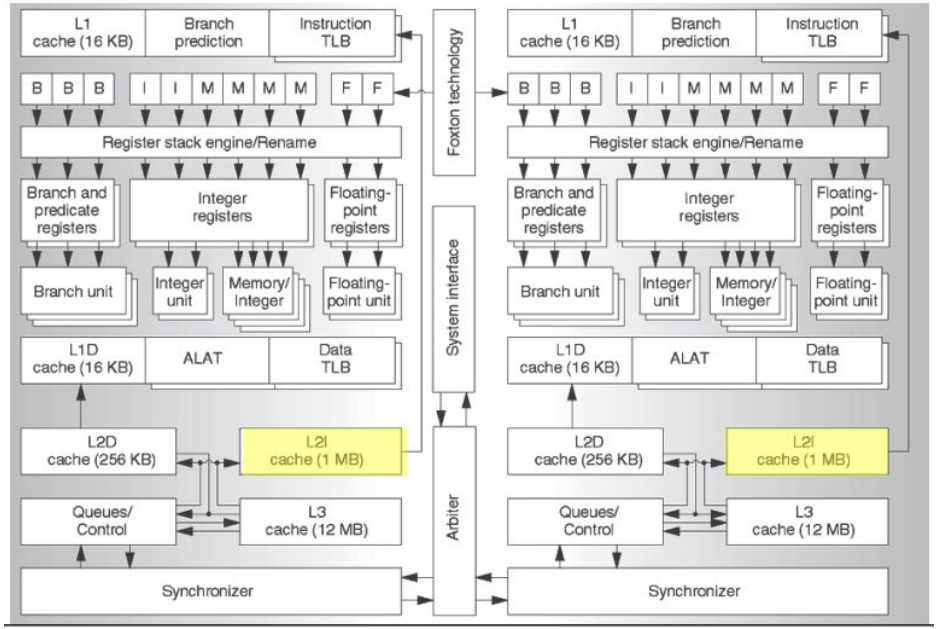
Third level of cache (cont.)

- Itanium 2, Madison core
  - 410 millions of transistors



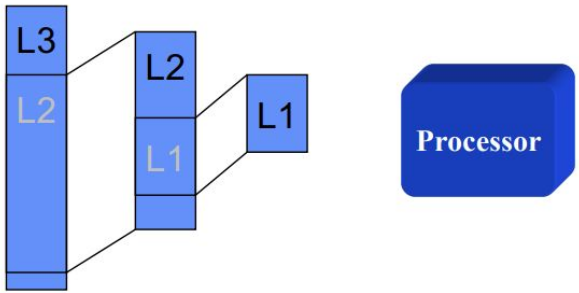
Third level of cache (cont.)

- Montecito: Dual Core Itanium
  - Non unified level 2 cache memory
  - Total cache: 27 Moctets



Inclusive and Exclusive caches

- Inclusive: L1 -> L2 -> … -> Memory
  - A block placed in L1 cache is also copied in the L2, L3, etc
- Exclusive
  - A block placed in L1 cache is not automatically stored in L2, L3, etc
  - When a block is ejected from cache Li, it is copied in cache Li + 1



Thank you for you listening