

ELEC-H-473

Logic Circuits

Dragomir Milojevic
Université libre de Bruxelles

2023

Microprocessors & computers: top-down view

- **Distributed system** – many different & independent sub-systems that process information concurrently (in parallel); various parts of the system will produce some information based on inputs, and exchange this information with other sub-systems
- **Combinatorial & sequential digital logic circuits** are used to materialize Boolean algebra according to *Shannon switching theory* and to practically implement the above sub-systems
- Complementary Metal Oxide Semiconductor (**CMOS**) process technology is used to implement logic circuits and **Application Specific Integrated Circuit (ASIC)** (transistors used as switches)
- **Very Large Scale of Integration (VLSI)** is enabled with scaled CMOS technologies; today CMOS logic functional density allows to implement millions of logic gates per cm^2 of IC area
- **System-on-Chip (SoC)** paradigm: ASICs are seen as assembly of many functional blocks to cope with complexity of today's CPUs

Challenges today

- In the past, individual ingredients: system architecture, design (HW) and CMOS processing technology (top-down) were kept well isolated one from the other in “divide & conquer” approach
- Today academia & industry are unifying these views: **to better optimize systems we can't isolate different ingredients any more**
- Check for following concepts currently in vogue in the field:
 - ▷ **Device-Technology Co-Optimization (DTCO)** – How to concurrently optimize: a) the device, i.e. basic transistor architecture & b) processing technology used to effectively manufacture this device
 - ▷ **System-Technology Co-Optimization (STCO)** – How to concurrently optimize system architecture (i.e. computer organization) & processing technology (how do you effectively manufacture device & the complete circuit)
 - ▷ **Advanced IC packaging & heterogeneous system integration** – how to enable and design IC packages with **multiple** dies using optimized CMOS process for each functionality

Our focus

- Tomorrow computing systems will be designed **holistically**, i.e., by **taking all ingredients simultaneously** into account: from CMOS processing technology, device (transistor) & logic components design (gates), functional block micro-architecture, SoC assembly & implementation, up to SW design, so algorithm, program development in some language and generation of the executable
- **This is extremely ambitious ... but there is no other way out!**
- Main driver of this industry in the past, CMOS technology, is today reaching its limits (that good old Moore's law)
- To still enable system performance improvements, we need to chase sub-optimality at all levels of system design
- To enable holistic design of computer systems, we require in-depth view of all layers above: **and this is what we will try to do here!**

Today

1. Boolean logic
2. Combinatorial circuits
3. Sequential circuits
4. Typical logic circuits in microprocessors
5. Modeling and implementation of logic circuits today

1. Boolean logic

Formal definition

- B – is a set of 2 elements, TRUE & FALSE or $\{0,1\}$
- $'$ – or, prime symbol (used in this notes) denotes complement operator (changes 0 to 1 and inversely); other notation include a horizontal bar over the element of B , so $\bar{0} = 1$ and $\bar{1} = 0$
- \cdot – dot denotes AND operator
- $+$ – plus denotes OR operator
- Boolean algebra is then a quadruple $\{B, ', \cdot, +\}$
- Important to keep in mind that $+$ and \cdot are not arithmetic operators, but Boolean operators AND and OR
- Variable x is Boolean variable if it takes only values from B , so it can be either 0 or 1
 - ▷ Note that in the above “or” is exclusive, a Boolean variable can't be 0 and 1 at the same time – law of excluded middle – either this proposition or its negation is true, but not both

Boolean function

- Boolean function f is a function whose k arguments are Boolean variables
- Result of function f evaluation belongs to set B , noted as:

$$f : \{0,1\}^k \rightarrow 0,1$$

- The number of arguments k in the above is a non-negative integer called arity of the logic function; for small k it is possible to enumerate all possible operators
 - ▷ 0-ary Nullary – a Boolean constant that can be 0 or 1
 - ▷ 1-ary Unary – complement or logical NOT operator
 - ▷ 2-ary Binary – AND, OR that are not the only binary operators;
 - Can you explain why we can enumerate all operators?
 - How many different operators you could define for each case?
 - How many more operators we get for $k+1$ compared to k ?

Boolean operators

- Boolean operators can be defined using **truth tables** in which function is evaluated for all possible combinations of arguments: thus the truth table has k columns, and 2^k lines
- For AND, OR and XOR table has 2 columns and 4 lines:

		AND	OR	XOR
x	y	$x \cdot y$	$x + y$	$x \oplus y$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- Note that OR operator is **inclusive**, the result is also true when both variables are true; in **exclusive or (XOR)** the result is true only if **exclusively** one of the variables is true

Axioms and single argument theorems

- Using definitions of three Boolean operators, we define **4 axioms of Boolean algebra**; index a is used for \cdot (AND) & b for $+$ (OR):

A1a	$0 \cdot 0 = 0$	A1b	$1 + 1 = 1$
A2a	$1 \cdot 1 = 1$	A2b	$0 + 0 = 0$
A3a	$0 \cdot 1 = 1 \cdot 0 = 0$	A3b	$1 + 0 = 0 + 1 = 1$
A4a	$0' = 1$	A4b	$1' = 0$

- Following theorems hold for one Boolean variable x :

T1a	$x \cdot 0 = 0$	T1b	$x + 1 = 1$	Null element
T2a	$x \cdot 1 = 1 \cdot x = x$	T2b	$x + 0 = 0 + x = x$	Identity
T3a	$x \cdot x = x$	T3b	$x + x = x$	Idempotent
T4a	$(x')' = x$	T4b	NA	Double inv
T5a	$x \cdot x' = 0$	T5b	$x + x' = 1$	Inverse

- Above theorems can be proven by simply substituting the values of the Boolean variable x in different expressions

Theorems with 2 and 3 variables

- Following theorems hold for 2 & 3 variables x, y, z

T6a	$x \cdot y = y \cdot x$	T6b	$x + y = y + x$	Commutative
T7a	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	T7b	$(x + y) + z = x + (y + z)$	Associative
T8a	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	T8b	$x + (y \cdot z) = (x + y) \cdot (x + z)$	Distributive
T9a	$x \cdot (x + y) = x$	T9b	$x + (x \cdot y) = x$	Absorption
T10a	$(x \cdot y) + (x \cdot y') = x$	T10b	$(x + y) \cdot (x + y') = x$	Combining
T11a	$(x \cdot y)' = x' + y'$	T11b	$(x + y)' = x' \cdot y'$	DeMorgan's

- Above theorems can be proven using Truth Tables, axioms or already proven theorems; there could be multiple & valid ways to prove a given theorem, although some may be simpler than other
- Example:

$$\begin{aligned} x + (x \cdot y) &= (x \cdot 1) + (x \cdot y) && Th2a \\ &= x \cdot (1 + y) && Th8a \\ &= x \cdot (1) && Th1b \\ &= x && Th2a \end{aligned}$$

Duality Principle

- Dual of a logic expression is obtained by changing all + operators with · operators, and by changing all 0 with 1 (and vice versa, so we replace · with +, and 1 with 0)
- For example:

$$(x \cdot y' \cdot z) + (x \cdot y \cdot z') + (y \cdot z) + 0$$

has a dual:

$$(x + y' + z) \cdot (x + y + z') \cdot (y + z) \cdot 1$$

- **Attention !!!** The above does not state that a Boolean expression is equivalent to its dual; example:
 - ▷ $F_1 = x \cdot 0 = 0$ is true
 - ▷ So dual of F_1 , the $F_2 = x + 1 = 1$ is true too
 - ▷ But $F_1 \neq F_2$ since $F_1 = 0$ and $F_2 = 1$ and $0 \neq 1$

Functional completeness

- One can enumerate all Boolean operators; if we do this for 2 arguments we get 16 possible operators (**why 16?**)
- A **Functionally Complete Set (FCS)** of Boolean operators can express all possible truth tables (Boolean functions) **only by combining its members**; examples of FCS:
 - ▷ FCS1(AND, NOT) – 3rd operator OR can be calculated as a function of these two
 - ▷ Even more interesting – **singleton FCS** – composed of a single element; known singleton FCSs: FCS2(NAND) & FCS3(NOR)
- FCS are great in practical digital electronics: build few gates (and even only one in case of singletons!), and all other gates and or logic functions could be built using only these; this can seriously simplify digital circuit design

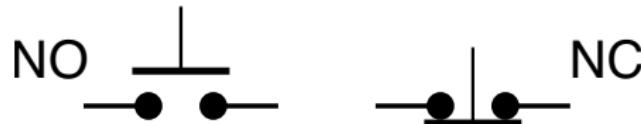
Logic circuits with singleton FCS

- Following table indicates how it is possible to build basic logic operators NOT, AND and OR from only binary NAND or NOR gates

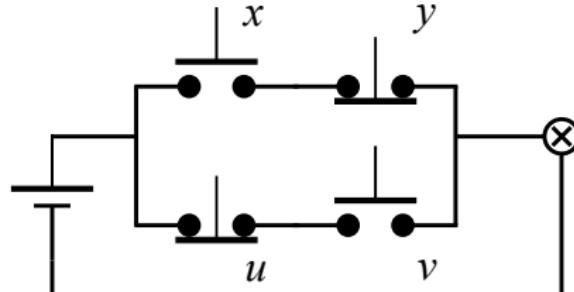
Source op.	Dest op.	Wiring
NOT	NAND	$x' = (xx)' = (x1)'$
NOT	NOR	$x' = (x+x)' = (x+0)'$
AND	NAND	$xy = ((xy)')'$
AND	NOR	$xy = ((xy)')' = (x'+y')'$
OR	NAND	$x+y = ((x+y)')' = (x'y')'$
OR	NOR	$x+you = ((x+y)')'$

Boolean logic & logic circuits

- Two valued Boolean algebra describe **switching circuits** operation
- Switch can be open (no current flow) or closed (current flows); ideal switches change from open to close & inversely instantly; 2 types of switches: **Normally Open (NO)** & **Normally Closed (NC)**



- NO switch is associated with Boolean variable x , & NC with inverted Boolean x' ; switches are assembled in a circuit to reflect Boolean equation; **what is the Boolean equation of the circuit?**



2. Combinatorial circuits

Definition of combinatorial systems

- Outputs O_i are Boolean functions of inputs I_j **only**:

$$O_i = f_i(I_j) \quad i \in 0, 1, \dots, n-1 \quad j \in 0, 1, \dots, m-1$$

- Inputs are random variables, they can change at any moment; however if two or more inputs need to change at the same time, they will do so instantly, and at the same time (for now)
- Multiple (hopefully different!) logic functions are **independent** one from the other, even if they use the same arguments (inputs)
- When implemented with electronic circuits, f_i logic functions will be **concurrent**, i.e., they will run in parallel
- Main property of combinatorial circuits:

Same input combination will always produce same output

Combinatorial systems are simplest form of automata in theory of automata

Representation of combinatorial systems

- Most of the time: Truth Tables, Logic Functions, & Schematics
- We call Minterm a combination of all input variables for which $O = 1$; Maxterm is the combination of all inputs for which $O = 0$
 - ▷ For truth table of function $F(a, b)$:

I_{10}	a	b	F
0	0	0	1
1	0	1	1
2	1	0	0
3	1	1	0

- ▷ $a'b'$, $a'b$ are Minterms; ab' , ab are Maxterms; 1st column is the decimal equivalent (I_{10} base 10) of an input combination, a is Most Significant Bit (MSB), & b Least Significant Bit (LSB)
- ▷ The choice is arbitrary but must be respected once fixed
- Logic function is then represented as:

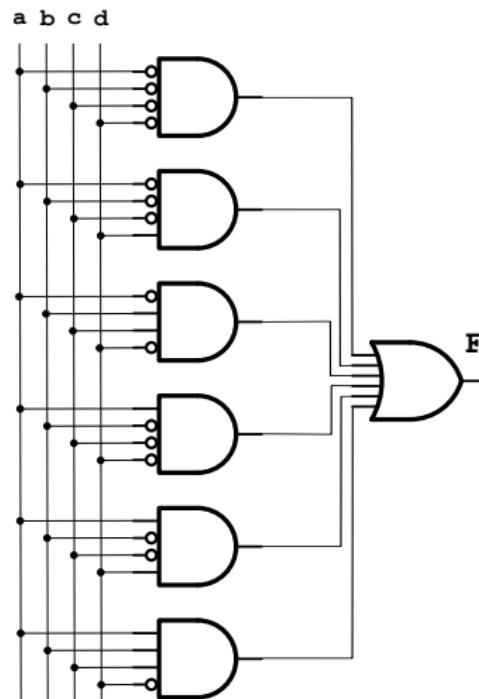
$$\begin{array}{l} \text{Sum of Minterms} \\ F = a'b' + a'b \end{array}$$

or

$$\begin{array}{l} \text{Product of Maxterms} \\ F = (a' + b)(a' + b') \end{array}$$

Four input truth table, logic expression & circuit

Dec.	a	b	c	d	F
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	0



$$F = a'b'c'd' + a'b'c'd + a'bcd' + ab'c'd' + ab'c'd + abcd'$$

Logic function transformation

- Expressing logic functionality using sum of Minterms (or product of Maxterms) **is not optimal**
- Expression on the previous slide can be simplified using axioms & theorems from previous section (mostly Inverse and Null element):

$$F_1 = a'b'c'd' + a'b'c'd + \textcolor{red}{a'bcd'} + ab'c'd' + ab'c'd + \textcolor{red}{abcd'}$$

$$F_2 = a'b'c'(d' + d) + (\textcolor{red}{a' + a})bcd' + ab'c'(d' + d)$$

$$F_3 = a'b'c' + bcd' + ab'c'$$

$$F_4 = (a' + 1)b'c' + bcd'$$

$$F_5 = b'c' + bcd'$$

- Final expression uses **significantly less logic gates!**, thus saving significant **circuit resources** → 2, instead of 6 AND gates, plus one 2-inputs, instead of one 6-inputs OR gate

Logic optimization & forms

- Expressions F_1 and F_5 are logically equivalent, so :

$$a'b'c'd' + a'b'c'd + a'bcd' + ab'c'd' + ab'c'd + abcd' = b'c' + bcd'$$

- Sum-of-Minterms (F_1) is called **Canonical Disjunctive Normal Form (CDNF)**; CDNF is unique expression for a given truth table
- Transformed expression F_5 called **Sum-of-Products (SoP)**, or **Product-of-Sums (PoS)**, is still a sum, but minterms are replaced with simple Boolean terms using arbitrary number of variables; different SoPs could be derived from a given Truth Table
 - ▷ Products & sums in the above are Boolean operators AND & OR, and not standard arithmetic operators; don't be confused
- Optimized SoPs, have better **Performance, Power, Area and Cost (PPAC)** parameters, key goal in digital electronics
- Logic optimization is crucial for practical circuit implementation

Practical logic optimization methods

- Application of axioms & theorems may be applicable to simple expressions, for more inputs this could be error prone & time consuming → we need better methods
- Karnaugh maps – graphical method, efficient up to 5 variables

		cd	
		00	01
ab		00	1
00	01	0	0
01	00	0	1
11	01	0	1
10	11	1	0

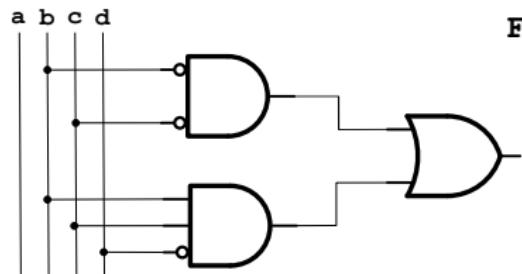
E.g. using K-Maps:

$$F = b'c' + bcd'$$

- Exhaustive methods such as Quinne-McCluskey, Espresso, etc. have been proposed to allow logic optimization automation

Two-level logic 1/3

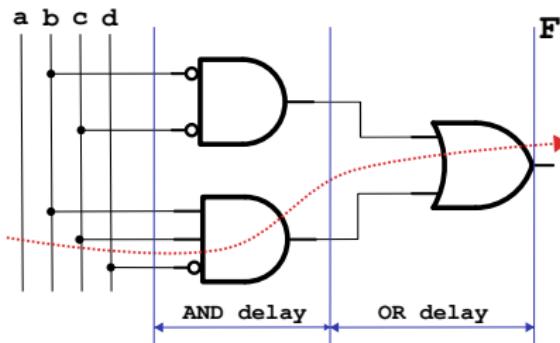
- We know that any SoP can be implemented using a set of AND and one OR logic gate (inverse for an PoS)
- Each AND gate correspond to 1 SoP term; if there is no logic redundancy, only 1 AND gate is active at a time for a given input combination; note that in optimized circuits multiple input combinations could affect the same AND gate: $b'c'$ gate will set output to 1 for $a = 1$ and $a = 0$ as long as $b = c = 0$



- Input signals will “travel” through one AND and one OR gate; because of this, SoP (PoS) circuits are called **two-level logic**, “levels” referring to number of gates that signal has to cross

Delay in two-level logic circuits 2/3

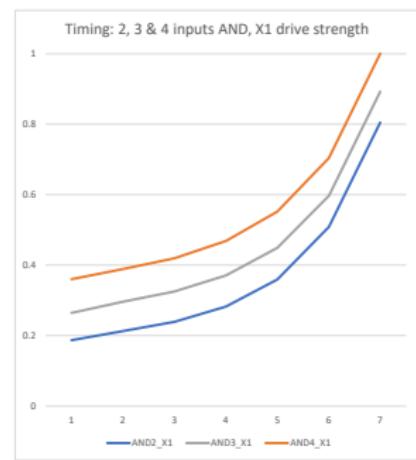
- If we consider practical circuit, implemented using actual electronic components that have non negligible RLC values for transistors & wires, there will be some **delay from input to output**



- If we neglect wire delays & keep only logic gate delays, when input changes, the new value for the output will appear 2 gate delays later
- Important remark** – no matter the logic function complexity (i.e. the number of inputs), the SoP performance should be the same: 2 gate delays → **this would be GREAT news ... if only it could be true** (you know that in life and math “*there is no free lunch*”!)

Two-level logic pitfalls and solution 3/3

- Where is the problem? Logic gate delay, power & area are non-linear function of number of inputs (many other parameters too, out of scope for now); illustration using 45nm CMOS tech
- Comparing AND gates with 2, 3 & 4 inputs, 4 being maximum number of inputs for a gate in this library
- Timing for single load & different input transition times normalized to max load → **limited number of inputs controls the increase in gate delay**
- Area of 3, 4 inputs is respectively 1.6 & 1.9 bigger than 2-inputs AND
- Consequence: **SoPs can be efficiently implemented only for VERY simple logic circuits with maximum 4 – 5 inputs**

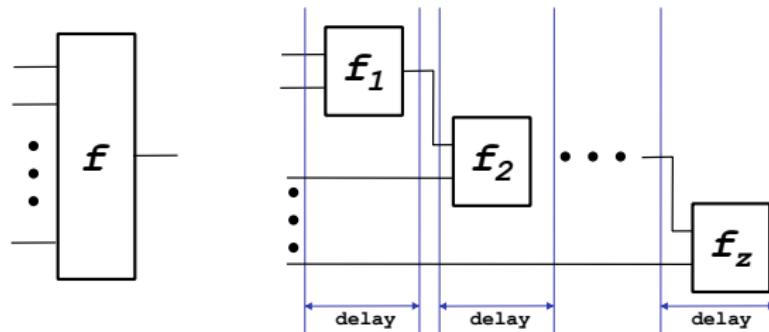


Multi-level logic – logic factoring

- What to do then with more complex logic expressions? Rather than using big, power hungry & slow gates, we decompose logic function into smaller functions that use gates with limited number of inputs (say < 5) and connect these logic functions in series

$$O = f(I_j) = f_z(z, \dots, f_2(d, (f_1(c, f_0(a, b)))))$$

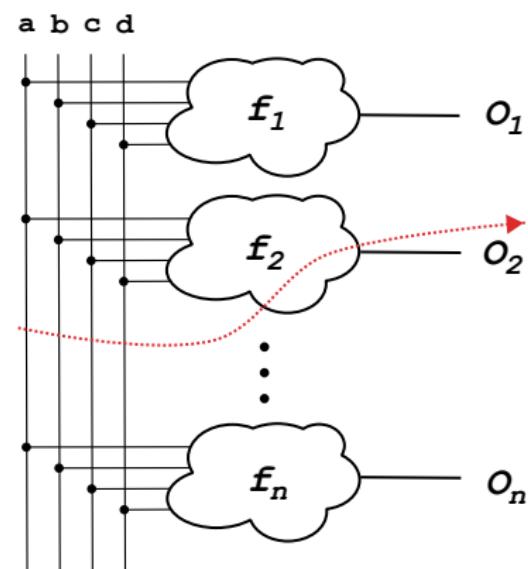
- Logic computation now has **multiple levels** since input signal will have to travel through more than 2 logic gates, hence the name



- Delay in logic circuit now increases linearly with the number of levels: computation is “stretched in time” with smaller individual delays

Complete combinatorial logic system view

- Each output correspond to a specific Boolean logic function; different logic functions work in parallel (concurrently)
- Individual logic functions are optimized, and may contain multi-level logic, the number of logic levels depend on function complexity on optimization targets
- In real circuits **the output of the system as a whole is valid only after the logic function with longest delay computes the output;** observing the output **before** will give us previous (old) output value



3. Sequential circuits

Sequential logic circuits

- **Outputs** (O_i) depend on **inputs** (I_j) and **system state** (S_k);
multiple Boolean values/expressions
- Future system states (S_k^+) are calculated using logic functions of
inputs AND previous states (S_k^-) represented using internal (or
state) variables; hence two sets of Boolean equations:

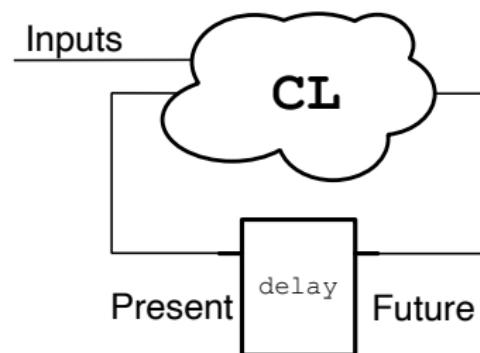
$$O_i = f_1(i, S_k^-)$$

$$S_k^+ = f_2(k, I_j, S_k^-)$$

$$i \in 0, 1, \dots, n - 1$$

$$j \in 0, 1, \dots, m - 1$$

$$k \in 0, 1, \dots, o - 1$$



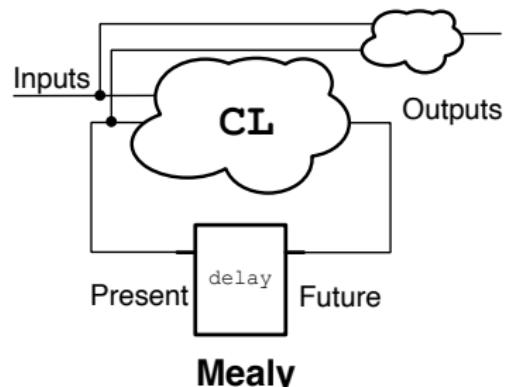
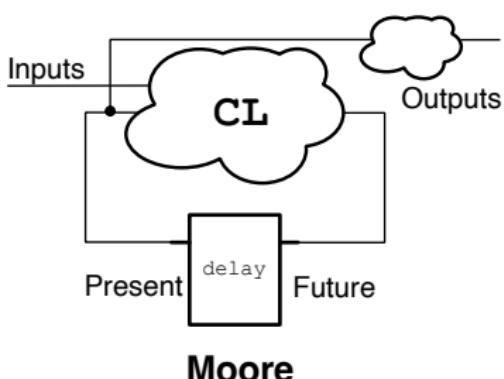
- Sequential logic circuit can be seen as a Combinatorial Logic (CL)
circuit with one (or more) **feedback(s)** for state variables, **all**
sequential systems have this feedback!

Properties

- In sequential logic systems the outputs can be different for same input combination → they can express more complex behavior
- This is possible thanks to the notion of state “stored” in the feedback wire that exhibits some kind of memorizing behavior
- While all possible input values can be enumerated for a fixed number of inputs (combinatorial logic), we can't do the same with sequential systems, even for a small number of inputs; example:
 - ▷ Design a system that will identify a given sequence of bits, serialized on a single-bit input line
 - ▷ The number of states is at least the number of bits in the sequence (plus eventually one state to indicate that sequence is wrong)
 - ▷ For large bit sequences, the system will have many states
- Number of states **must be bounded**, you should know why?
- Sequential logic circuits are often abstracted using **Finite State Machine (FSM)** model of computation, next in line in complexity in theory of automata (so, after combinatorial systems)

System outputs

- Figure on slide p. 29 omitted to represent outputs on purpose
- This is because we distinguish 2 sequential logic circuits classes:
 - ▷ **Moore machines** – the output is calculated as function of state variables only (present); we thus need to only decode the state
 - ▷ **Mealy machines** – the output is calculated as function of state variables **and** the inputs; **same state can have different outputs**
- Mealy machines potentially enable better state optimization, resulting in simpler circuits; not given, since system dependent



Sequential logic circuits representation 1/3

- **State tables** – define future state as function of current state (first column in the table) and new input combination
- State tables differentiate **stable states** (in bold), & **transitions**; the system remains in stable state as long as input doesn't change; transitions occur when system is in stable state & input changes

Moore

S_k^+	ab				
S_k^-	00	01	11	10	Z
1	1	1	3	2	0
2	2	1	4	2	1
3	1	3	3	4	1
4	2	3	4	4	0

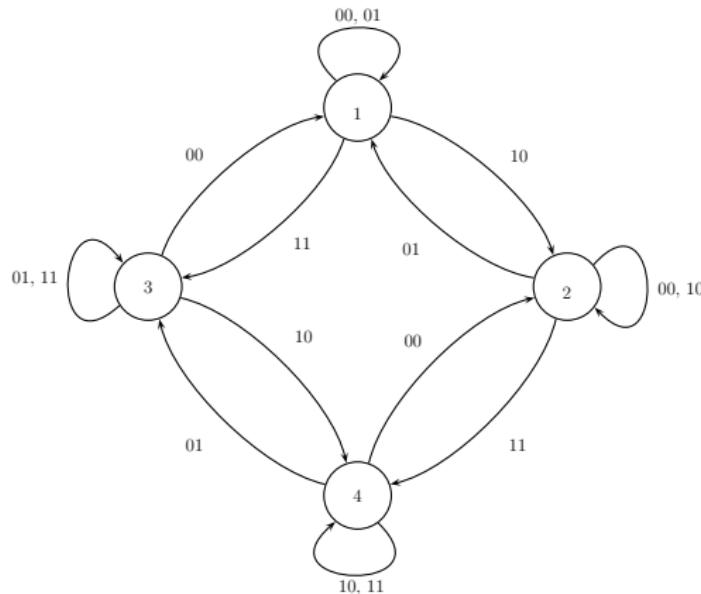
Mealy

S_k^+	ab				
S_k^-	00	01	11	10	
1	1/1	1/0	3	2	
2	2/0	1	4	2/1	
3	1	3/1	3/0	4	
4	2	3	4/1	4/0	

- Important hypothesis: **transitions are instantaneous**; in real systems this can't be true; however same behavior can be achieved if the system forbids **input variables to change during transitions**

Sequential logic circuits representation 2/3

- **State diagrams** – derived from graph theory; stable states are represented using graph vertices; edges represent transitions with logical conditions on inputs marked as edge weights; used rarely outside textbooks, since cumbersome for complex systems



Sequential logic circuits representation 3/3

- Boolean logic equations – derived from the state table that uses an arbitrary alphabet, for example $S_k^+, S_k^- \in \Sigma \{a, b, c, d, \dots\}$
- Because we want to implement a sequential system with Boolean logic, we need to map states into binary equivalents – state encoding where unique binary code is assigned to each state
 - ▷ We need at least $\log_2 m$ bits to encode m states; each bit will correspond to one state (Boolean) variable
- Designer picks one out of four possible (standardized) memory elements (D,T,SR,JK flip-flops) and then derives logic equations that control these memories; in the example below a system with two state variables, JK memory & output logic function

$$J1 = ab,$$

$$K1 = a'b'$$

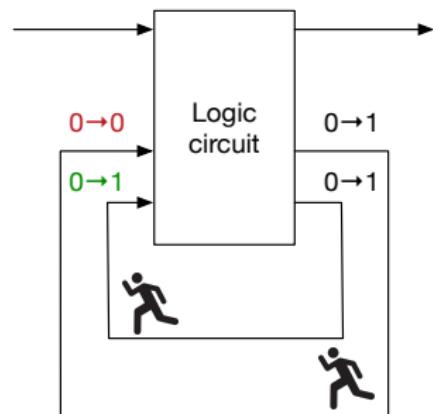
$$J2 = y_1 a' + y_1' a$$

$$K2 = y_1 b' + y_1' b$$

$$Z = b'y_1y_2 + ay_1'y_2$$

Race conditions

- In systems with two state variables or more, it is possible that more than one state variable change **at the same time**
- In real circuits, propagation delays in gates & wires will cause different arrival times for state variables at the input of the Logic Circuit, on the left side of the circuit → **race conditions**
- If one state variable arrives before the other, LC may “interpret” input wrongly, & end in wrong destination state
- Example: state transition from $00 \rightarrow 11$ results in 01 , and not 11 as it should
- Race conditions are major problem in systems with feedbacks since they will cause malfunction, they **must be avoided at any price!** in practical systems

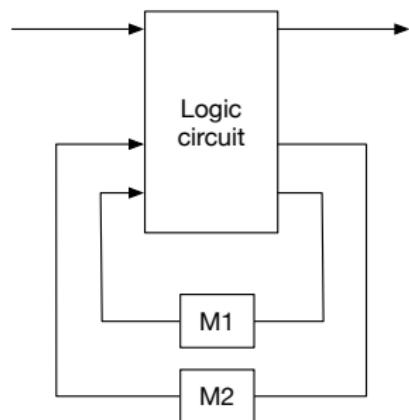


Asynchronous circuits

- One way of dealing with race conditions on state variable feedbacks is to build **asynchronous logic circuits**
- In such circuits no state transition between two stable states will ever involve **more than one state variable to change at a time**
- This can be done in different ways: using different state encoding, state table modifications (but by still preserving the machine behavior) and with adding supplementary state variables
- In the above, first two are not always possible and the third adds extra resources; all this constraints the system design; worse, asynchronous systems scale extremely poorly with the number of inputs and the number of states
- This is the reason why majority of digital systems **are not** implemented in this way → we need another solution to problem of race conditions in state variables – **synchronous systems**

Synchronous logic circuits

- Synchronous circuits solve race conditions by inserting **1 memory bit** per state variable (feedback loop) to split present from future
- Present state is updated only at a given moment allowing transients in wires & Logic Circuit to settle down
- In other words races will be “filtered out” at memory input
- Consequence: fastest **paths** will have to wait for the slower paths; i.e. we **downgrade system performance to the worst in class** – something that does not sound very optimal from the start
- If differences between delays are not significant, sacrifice is not huge (**make sure you can explain this**) but benefits are BIG since arbitrary number of state variables can now change simultaneously



Bistables, Latches & Flip-Flops 1/2

How to build memory elements for use in sequential logic circuits?

- **Bistable** – generic electronic circuit with two states, generally associated to Boolean values of 0 and 1
- **Latches** – bistables with a control signal C, whatever is found on input D is copied to the output Q as long as C=1
 - ▷ Latches are sensitive to level of C, hence they are “open” as long as C=1; any variation on D will be seen on the output
 - ▷ This is why they are also often called **transparent latches**
 - ▷ Typically banned from usage, unless in some very specific cases ...
- **Flip-Flops (FFs)** – bistables with a control signal C: input D is copied to output Q **only during transition of C from 0 to 1 rising edge** of C (or, falling edge when transition of C is from 1 to 0)
 - ▷ In principle control signal could take any form, but if it is periodic, we will speak of **clock (Clk)**; the period *P* and frequency *F* will have to be decided for each circuit depending on the needs

Bistables, Latches & Flip-Flops 2/2

- In digital logic we speak of different FFs: SR, JK, T & D
- They differ in the way how their **control inputs** are used to set the FF state; first two have two control inputs, and last two only one

S^+	SR			
S^-	00	01	11	10
0	0	0	-	1
1	1	0	-	1

S^+	JK			
S^-	00	01	11	10
0	0	0	1	1
1	1	0	0	1

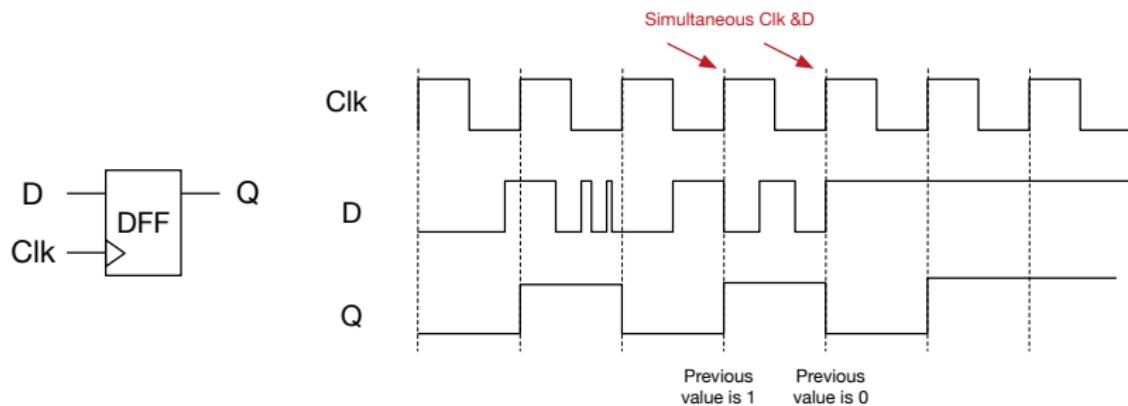
S^+	T	
S^-	0	1
0	0	1
1	1	0

S^+	D	
S^-	0	1
0	0	1
1	0	1

- Note: different FFs can be derived from a single D-FF, using extra combinatorial logic, D-FF is used as a basic building block

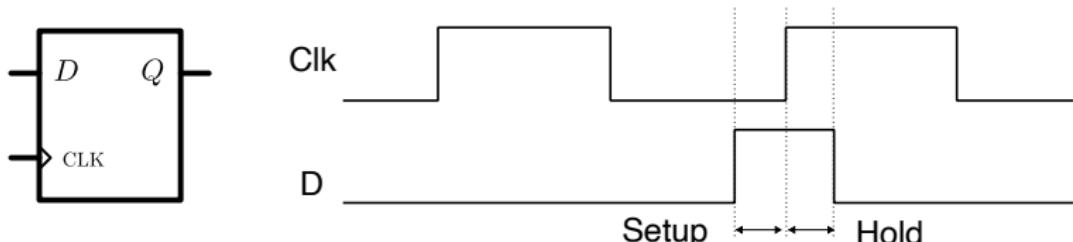
D-FF: mother of all Flip-Flops

- Input D is sampled & stored on the rising edge of a Clk; any input variation on D outside the rising edge is ignored → this is why D-FF is sometimes also called **sample & hold** FF
- Output value Q indicates the last D stored
- In case of simultaneous variations of D & Clk, **Q will take the previous value**, the one just before the Clk rising edge (this is very important hypothesis, see next slide)



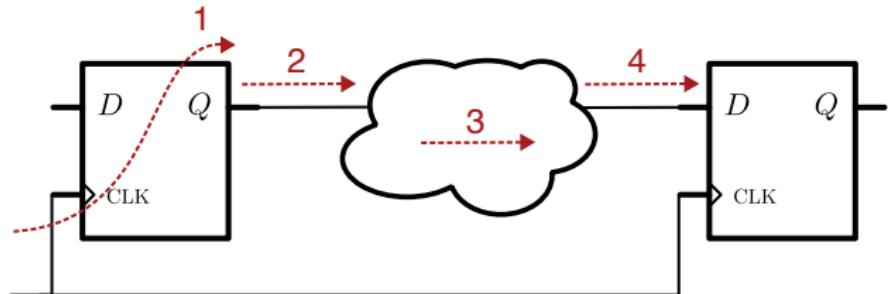
D-FF: the real thing

- Physical reality imposes following things:
 - ▷ Inputs are random variables and they could occur at any time, including simultaneous changes with the rising edge of the Clk
 - ▷ FF circuitry will have some *RLC* values preventing instantaneous switching, and some transition times on both inputs & outputs
 - ▷ Clock signal is generated with some physical circuit & will have jitter
- All of the above contribute to **metastability** – situations in which FFs can't guarantee that the right value is stored; **we need to avoid this!**
- For D-FF, the spec says that if we have simultaneous change of D & Clk the device uses the value before; how to achieve this?
 - ▷ Data must arrive **before** & should **remain** valid for some time with respect to the clock edge → **set-up** & **hold times**



D-FF in the real circuit

- Typically input data will come from a FF, so at first rising (**launch**) edge of the Clk, data in the source FF (FF1) will appear on Q output after some small delay – **clock to output delay** t_1



- Signal will have to travel through some wires – t_2 , some combinatorial logic (and wires) – t_3 and some wires t_4
- Destination FF (FF2) can safely store the data at the next (**capturing**) edge of the Clk, only if it **arrives on time**:

$$\sum_i^{i=4} t_i < \frac{1}{F_{Clk}} - t_{setup}$$

Impact on circuits & microprocessors

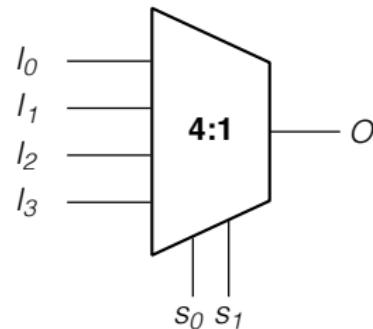
- Microprocessors are sequential logic circuits – in fact a collection of many independent sequential circuits that exchange processed information (CPU is a highly distributed system as said in the beginning)
- To make all these sequential circuits operational, we need to avoid race conditions, and use FFs to synchronize different microprocessor functional blocks and state machines
- Thus to improve the performance of a computing system, we need to improve the performance of the logic circuit by reducing the period, i.e. allow increase of the C_{lk} operating frequency F
- Assuming same logic circuit, to increase F we need improved electrical properties of components (transistors, wires etc.) used to implement a circuit; in ICs this is possible thanks to CMOS scaling
- Need for synchronization in sequential logic circuits and CMOS scaling are two key concepts that are behind the Moore's law, one of the most important paradigms in microprocessor design & implementation

4. Typical logic circuits in microprocessors

Multiplexers/Demultiplexers

- Multiplexers select one input signal I among n signals (typically n is power of 2) based on input word S of $\log_2 n$ bits
- Truth table of 4:1 MUX is given below; note that a given select combination makes the corresponding input bit to “pass” to the output; other bits are ignored as long as S is active

S_1	S_0	O
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

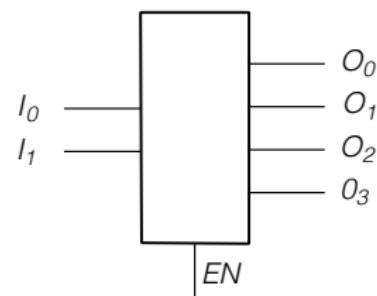


- Multiple multiplexers/demultiplexers circuits could be put in parallel – **crossbars** – to work with words: each word bit will have its own MUX; this is heavily used for communication in CPUs; however MUX/DEMUX circuitry PPAC scale quite poorly with n

Decoder/Encoders

- **Decoders** – assert one out of O_n output lines depending on the value of I_m input, that is m -bit binary; m -to- n decoder that has m input, n output lines, with $n = 2^m$
- Circuit can be controlled using **enable** signal EN: if EN=0 all output lines are 0 (no matter values of I_1 & I_0); when EN=1 the output line whose index is equal to the value of the input binary data is asserted (truth table below uses **don't care** for I_1 , I_0 and EN=0)

EN	I_1	I_0	O_3	O_2	O_1	O_0
0	–	–	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

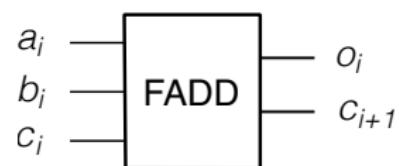


- **Encoders** do the opposite: from 2^m input lines they build n -bit output word ($n = \log_2 m$)

Adders

- Adders are possibly the most used arithmetic circuits in any IC, including microprocessors (and not only those used in ALUs)
- Assuming two operands a, b of arbitrary bit-length the sum $o = a + b$ will be calculated as follows:

$$o_i = a_i \oplus b_i \oplus c_i, \quad i \in 0, n-1$$
$$c_{i+1} = a_i \cdot b_i + c_i(a_i + b_i)$$



where o_i is the sum bit and c_i is the carry bit; note that the result word has one bit more than argument, because of the sum

- ▷ Expressions above are implemented in **Full-ADDder circuit (FADD)**
- We see that to compute the last bit of the sum $i = n - 1$ we need c_i to be propagated all the way from c_0
 - ▷ **Circuit delay is proportional to n , so for big n (64 or 128 bits, not uncommon these days) performance of the circuit is low**

Advanced adders 1/2

- Carry-lookahead (CLA) tend to improve adder PPAC, by explicitly calculating the cary bit as a function of inputs only (we stretch computation in space, opposite to computation serialization)
- Looking at $c_{i+1} = x_i y_i + c_i(x_i + y_i)$, we say:

$$g_i = x_i y_i$$

$$p_i = x_i + y_i$$

- Then $c_{i+1} = g_i + p_i c_i$ can be re-written as:

$$c_3 = g_2 + p_2 c_2$$

$$c_1 = g_0 + p_0 c_0 = g_2 + p_2(g_1 + p_1 g_0 + p_1 p_0 c_0)$$

$$c_2 = g_1 + p_1 c_1 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

$$= g_1 + p_1(g_0 + p_0 c_0) \quad c_4 = g_3 + p_3 c_3$$

$$= g_1 + p_1 g_0 + p_1 p_0 c_0 = g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0)$$

$$= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

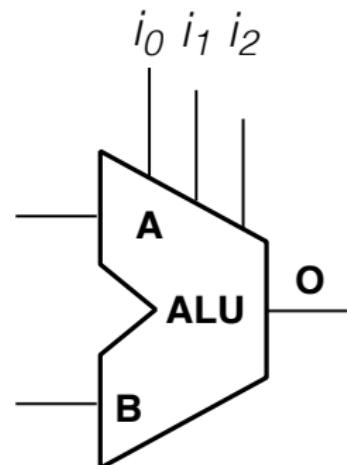
Advanced adders 2/2

- Note that in the above expressions p_i and g_i depend on inputs x_i and y_i only, there is no computational serialization
- Efficiency of a circuit could be better, however for larger x, y words (>8 bits so for typical operand sizes these days) we may experience quickly **combinatorial explosion**: expressions will become more complex, and thus less performant
- **Parallel prefix adders** – improve computation of c_{i+1} by introducing parallelization; depending on the approach adopted trade-off is made for key circuit parameters: area, speed, routing resources; many different variants, some key implementations:
 - ▷ **Brent-Kung** – minimum area (low-power), but poor performance
 - ▷ **Kogge-Stone** – more area, but better performance
 - ▷ **Sklansky** – minimal depth (best performance)

ALUs

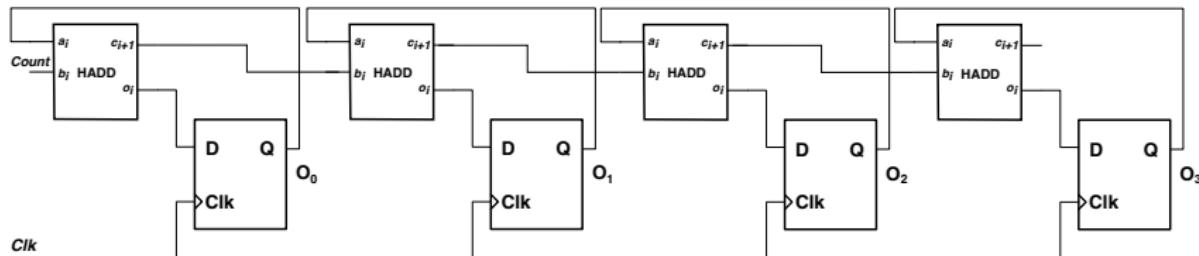
- ALUs are main components in microprocessors: they perform arithmetic (addition, subtraction) & logic (AND, OR, NOT)
- Note the absence of multiplications or divisions, generally performed using dedicated logic circuit for performance (HW)
- Assuming 3-bit opcodes, operands A & B table below indicates possible encoding of operations (choice is arbitrary):

i_2	i_1	i_0	Name	Operation
0	0	0	Pass A	$O \leftarrow A$
0	0	1	A AND B	$O \leftarrow A \& B$
0	1	0	A OR B	$O \leftarrow A B$
0	1	1	A'	$O \leftarrow \text{NOT}(A)$
1	0	0	$A+B$	$O \leftarrow A+B$
1	0	1	$A-B$	$O \leftarrow A-B$
1	1	0	$A+1$	$O \leftarrow A+1$
1	1	1	$A-1$	$O \leftarrow A-1$



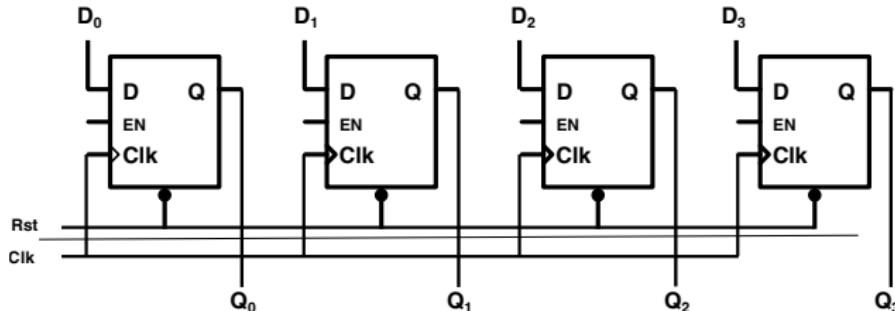
Counters

- Circuit examples we have seen so far are combinatorial, thus with limited power of expressivity; counters are sequential circuits, and like adders are very common in many different logic circuits including CPUs
- Many different implementations exist, depending on functional requirements: max count value, count direction, encoding for output values, etc.
- Generic approach uses half-adder & D-FFs; the use of half adder/subtractor circuit will enable up/down counter



Registers

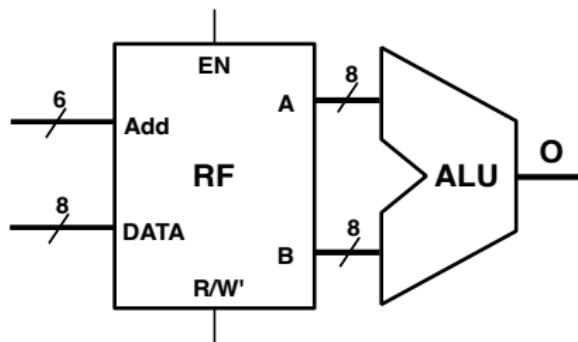
- Multiple D-FFs can be grouped together to form words of data: n -bit word can be stored using n D-FFs that share some inputs
- Various D-FFs are independent circuits, rising clock edge will “push” data to the output in parallel, complete word appears on the data bus
- Other than D there could be other control inputs (common for all D-FFs): obviously Clk, but also EN, RST etc.
- Example below shows 4-bit register with EN and RST:



- From above, different register variants may be implemented: shift registers (serial IO), serial-parallel IO for data (de)serialization, etc.

Register files

- Multiple registers can be bundled together for storage of arbitrary number of data *words* in an array – Register Files (RF)
- Since D-FFs are quite big circuits, best PPAC for RFs is obtained for small word count (few dozens at most); thus, the number of address bits is quite low (other technologies will be used for memories with high capacity)
- RFs are typically used to provide operands to ALU; figure below indicates a top-level schematic block of 8×8 -bit words RF:



- ▷ EN – activates the RF (Clk – not shown)
- ▷ R/W' – decides on RF access direction (exclusive R or W)
- ▷ Upper 3 bits of Add address output A, & lower output B

5. Modeling and implementation of logic circuits today

Motivation and solution for complex IC modeling

- Logic circuits synthesis using standard manual methods **can't be applied** for implementation of complex logic circuits composed of millions of gates enabled by CMOS technology in recent years ICs
- We need to enable logic circuit **abstraction** using **behavioral models** and use **design automation** for implementation process; design automation relies on highly parallel computers (with loads of memory) to enable reasonable run-time
- Design modeling & implementation use **Hardware Description Languages (HDLs)** and **Electronic Design Automation (EDA)** SW tools
- HDLs enable **formal specification** of logic circuits & **automated transformation** of these models into descriptions that can be used for practical circuit implementation (e.g. ASIC or FPGA)
- *Formal specifications* imply that the HDL model is complete, precise and without any ambiguities – this is crucial since in this step we translate verbal specifications (likely to be inaccurate since given in human language) to logic circuit models (that must be accurate)

Practical HDLs

- Most commonly used HDLs are VHDL, Verilog, System Verilog (the extension of the previous); here we focus on VHDL (maybe you have followed ELEC-H409)
- VHDL stands for **VHSIC HDL** where **VHSIC** = Very High Speed Integrated Circuits and it has been developed by the Department of Defense of USA in early '80 to improve digital circuit design
- HDLs enable **behavioral synthesis** as opposed to traditional manual logic synthesis that is lengthy, error prone & limited to circuits of a small size (dozens of variables max)
- Designer describes the circuit behavior, rather than how it should be implemented (gates), and the software will do the rest
- Do not misunderstand this: designer specifications need to be good to get a good circuit out of it → **in HW there is no magic**; components that are key for performance (e.g. arithmetic ops) will be carefully designed by “hand” even if synthesis is automated

Practical EDA

- High-level behavioral models are pushed through a series of software tools that transform initial system specification into physically implementable integrated circuit
- EDA tools implement sophisticated algorithms to create a “good” circuit; a good circuit provides desired trade-off among many parameters that are often condensed in **Performance, Power, Area & Cost (PPAC)** metrics
- Note that the use of “good”, and not “optimal” circuit, reflects a very important fact: these days optimization of one objective will necessarily compromise the other, so multi-criteria & multi-objective optimization paradigm are in game
- Typical example: you can't aim for performance & low-power at the same time; these two objectives are orthogonal
- Circuit implementation, or **design flow** is typically split into two phases: **synthesis** and **Place & Route**

Circuit synthesis and functional simulation

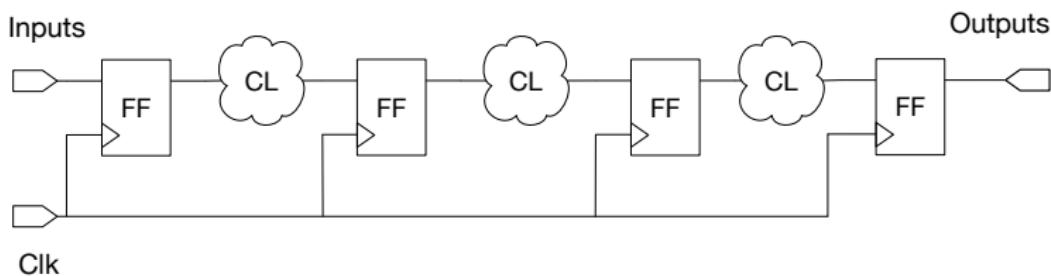
- **Synthesis** transforms abstracted, high-level, behavioral HDL models into logic, described in a **netlist** – a text file that contains all primitive Boolean logic operators (gates) & their connections
- Main goal of the synthesis is to obtain correct HDL specification of the circuit, that will behave as expected
- To confirm logic correctness and validate the HDL model designers use **functional simulation** to calculate outputs for a given set of input vectors (this is of course automated using computers)
- During synthesis some electrical parameters will be taken into account, but in an approximated way, since the circuit is still at higher abstraction level
- Initial PPA parameters are assessed with respect to initial constraints to evaluate the circuit feasibility; if at this stage the circuit doesn't meet constraints, there is no point in continuing; later implementation steps generally make things worse

Circuit implementation

- **Placement** decides on (x, y) position of each gate in the design, assuming rectangular (often squared) Silicon real-estate
- **Clock tree synthesis** connects Clk pins of all FFs in the design; the idea is to provide clock signal with minimum skew, jitter and interconnect length; not simple if you assume realistic FF count
- **Routing** implement wires that connect all gates in the design
- Throughout different steps initial netlist will be transformed into implementable design database with detailed views that include:
 - ▷ **Electrical views** – used to checking timing (combinatorial, wire, set-up/hold delays, etc.), currents (fan-out, density, max currents etc.), power dissipation and delivery, temperature (average & hotspots), mechanical etc. etc. etc.
 - ▷ **Geometrical views** – all shapes need to be conform for the CMOS manufacturing process as indicated by the IC foundry
- These models are exhaustively analyzed for final system validation, before the design can be shipped for IC manufacturing (**sign-off**)

Digital system seen as RTL

- In general a circuit is composed of combinatorial logic (even in sequential circuits) and Flip-Flops (even in combinatorial circuits)
- Register Transfer Level or Logic (RTL) – is a system view in which we consider **data flow** of signals going through an arbitrary number of combinatorial circuits separated with Flip-Flops (FFs) sharing the same (but sometimes different) clock sources



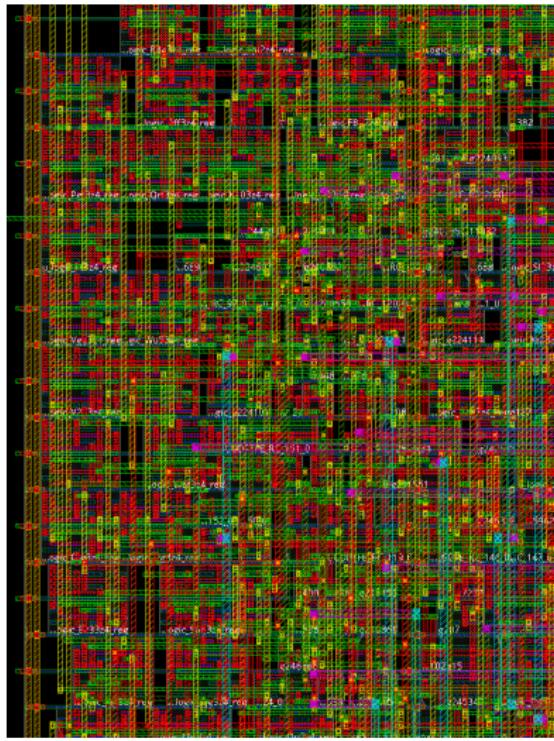
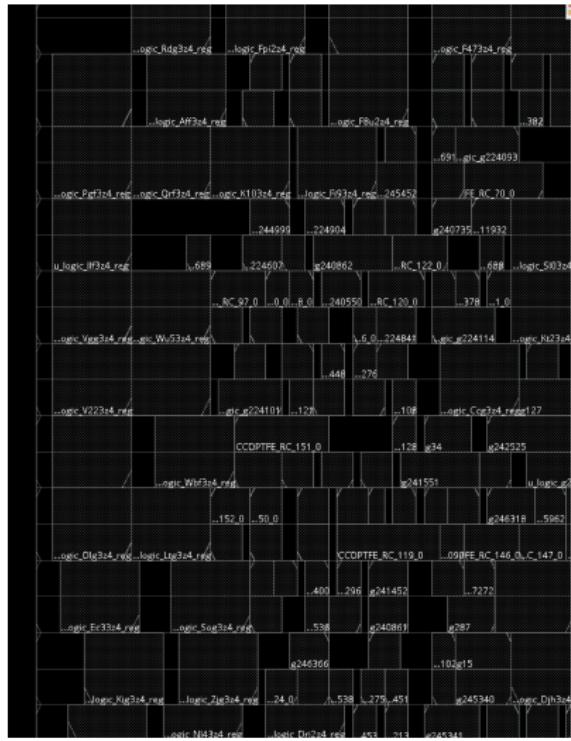
- Each “sub-system” appears as independent circuit, and all circuits work concurrently (in parallel) → **important for microprocessors**
 - ▷ What needs to be fulfilled to maximize the system performance?

Illustration: from RTL to netlist

```
ram_resp[initvar] = _GEN_1[1:0];
`endif
ifdef RANDOMIZE_REG_INIT
    _GEN_2 = {1{random}};
    _T_41 = _GEN_2[0:0];
`endif
ifdef RANDOMIZE_REG_INIT
    _GEN_3 = {1{random}};
    _T_43 = _GEN_3[0:0];
`endif
ifdef RANDOMIZE_REG_INIT
    _GEN_4 = {1{random}};
    maybe_full = _GEN_4[0:0];
`endif
end
`endif
always @ (posedge clock) begin
    if (ram_id__T_52_en & ram_id__T_52_mask) begin
        ram_id[ram_id__T_52_addr] <= ram_id__T_52_data;
    end
    if (ram_resp__T_52_en & ram_resp__T_52_mask) begin
        ram_resp[ram_resp__T_52_addr] <= ram_resp__T_52_data;
    end
    if (reset) begin
        _T_41 <= 1'h0;
    end else begin
        if (do_enq) begin
            _T_41 <= _T_59;
        end
    end
    if (reset) begin
        _T_43 <= 1'h0;
    end else begin
        if (do_deq) begin
            _T_43 <= _T_64;
        end
    end
    if (reset) begin
        maybe_full <= 1'h0;
    end else begin
        if (_T_55) begin
            maybe_full <= do_enq;
        end
    end
`endif
endmodule
module Queue_11(
    input  clock,
    input  reset,
    output io_enq_ready,
    input  io_enq_valid,
    input  [3:0] io_enq_bits_id,
    input  [63:0] io_enq_bits_data,
    input  [1:0] io_enq_bits_resp,
```

```
FA1STKD1 g802651(.A (n_6841), .B (r246), .CI (r187), .CO (UNCONNECTED3563), .S (n_67106));
FA1STKD1 g802652(.A (r726), .B (r689), .CI (n_6764), .CO (UNCONNECTED3564), .S (n_67107));
FA1STKD1 g802653(.A (r477), .B (r410), .CI (n_6825), .CO (UNCONNECTED3565), .S (n_67108));
FA1STKD1 g802654(.A (r330), .B (r228), .CI (n_6629), .CO (UNCONNECTED3566), .S (n_67109));
FA1STKD1 g802655(.A (r1910), .B (r1898), .CI (n_6566), .CO (UNCONNECTED3567), .S (n_67110));
A021D1 g802656(.A1 (n_67111), .A2 (n_41783), .B (n_43033), .Z (n_67112));
INR2D0 g802657(.A1 (n_32181), .B1 (n_50676), .ZN (n_67111));
A021D1 g802658(.A1 (n_67113), .A2 (n_63733), .B (n_34644), .Z (n_67114));
INR2D0 g802659(.A1 (n_32183), .B1 (n_48554), .ZN (n_67113));
A021D1 g802660(.A1 (n_67115), .A2 (n_41900), .B (n_41124), .Z (n_67116));
INR2D0 g802661(.A1 (n_31794), .B1 (n_48551), .ZN (n_67115));
A021D1 g802662(.A1 (n_67117), .A2 (n_41497), .B (n_38424), .Z (n_67118));
INR2D0 g802663(.A1 (n_31798), .B1 (n_46304), .ZN (n_67117));
A021D1 g802664(.A1 (n_67119), .A2 (n_41880), .B (n_38427), .Z (n_67120));
INR2D0 g802665(.A1 (n_31796), .B1 (n_46301), .ZN (n_67119));
A021D1 g802666(.A1 (n_67121), .A2 (n_41766), .B (n_38284), .Z (n_67122));
INR2D0 g802667(.A1 (n_31792), .B1 (n_46298), .ZN (n_67121));
A021D1 g802668(.A1 (n_67123), .A2 (n_63733), .B (n_38286), .Z (n_67124));
INR2D0 g802669(.A1 (n_31645), .B1 (n_46295), .ZN (n_67123));
A021D1 g802670(.A1 (n_67125), .A2 (n_40308), .B (n_34632), .Z (n_67126));
INR2D0 g802671(.A1 (n_31800), .B1 (n_46307), .ZN (n_67125));
A021D1 g802672(.A1 (n_67127), .A2 (n_41900), .B (n_34732), .Z (n_67128));
INR2D0 g802673(.A1 (n_31647), .B1 (n_44240), .ZN (n_67127);
XNR2D0 g802674(.A1 (n_67129), .B (n_9822), .ZN (n_67130));
X0R2D1 g802675(.A (n_1465), .A2 (r1101), .Z (n_67129));
FA1STKD1 g802676(.A (n_2508), .B (n_6519), .CI (n_25738), .CO (UNCONNECTED3568), .S (n_67131));
FA1STKD1 g802677(.A (n_17082), .B (n_4392), .CI (n_1196), .CO (UNCONNECTED3569), .S (n_67132));
FA1STKD1 g802678(.A (n_829), .B (n_4396), .CI (n_14984), .CO (UNCONNECTED3570), .S (n_67133));
FA1STKD1 g802680(.A (n_20329), .B (n_3475), .CI (n_161), .CO (UNCONNECTED3571), .S (n_67135));
FA1STKD1 g802681(.A (n_3524), .B (n_3405), .CI (n_18483), .CO (UNCONNECTED3572), .S (n_67136));
FA1STKD1 g802682(.A (r982), .B (r904), .CI (n_14487), .CO (UNCONNECTED3573), .S (n_67137));
FA1STKD1 g802683(.A (n_14683), .B (n_1494), .CI (n_2133), .CO (UNCONNECTED3574), .S (n_67138));
FA1STKD1 g802684(.A (n_11689), .B (n_1923), .CI (n_501), .CO (UNCONNECTED3575), .S (n_67139));
FA1STKD1 g802685(.A (n_17497), .B (r1299), .CI (r1266), .CO (UNCONNECTED3576), .S (n_67140));
```

Illustration: placed & routed design



Microprocessors design as logic circuit

- Today all microprocessors, from tiny micro-controllers to super powerful CPUs are designed using RTL and HDLs
- Complex software tools are used to transform initial HDL models into implantable design databases that can be shipped to a manufacturer who fabricates the **Application Specific Integrated Circuit (ASICs)** using CMOS technology (we will see this)
- Huge teams of engineers work on a very specific task in the whole process for significant amount of time; typical IC system design cycle is \sim year, for approximately \sim 20% of new functionality, this means that \sim 80% of system specification is legacy HDL (RTL re-use from previous generation products)
- All this is quite expensive, hence requires large markets (sale volumes) to compensate for investments; at the end of the day the activity could generate significant profits (for some!)
- For us, circuit logic circuit design & implantation will have a strong impact on processor & computer systems architecture

ELEC-H-473 – Microprocessor architecture

Th02: C and assembly

Dragomir Milojevic
Université libre de Bruxelles

2023

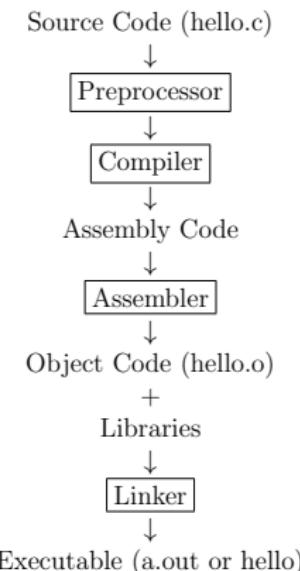
Today

1. C language – quick start
2. Basic data types, variables and operators
3. Control Structures
4. Functions
5. Pointers
6. Arrays, memory and data structures
7. File access
8. Examples to analyze
9. Assembly primer

1. C language – quick start

Life of a C program

- C code is written as **plain text** in a **source file**; learn how to use a good text editor, it can save you a lot of time!
 - ▷ There are plenty of good editors, my personal favorite is VIM: complex learning curve but great productivity, for many the best editor ever
- During compilation, the source file first goes through a **pre-processor** who deals with **pre-processor directives** (more on these later)
- Source code is then **compiled** into **assembly** code (machine language) and **assembler** is used to produce the **object file** out of assembly code
- Multiple object files are **linked** together into one **executable file** stored on hard drive & loaded into the system memory for execution
- There might be (or not) some external (previously written) libraries that define other functionalities (**re**)used in your source code



Your first C program: “Hello world”

- Minimal program that will compile and do something:

```
1 #include<stdio.h>
2 int main(void) {
3     printf("Hello World\n");
4     return 0;
5 }
```

- The above is written into a source file say HelloWorld.c
- First line is a preprocessor macro, indicated with # symbol before include statement; this command will tell preprocessor to look for a header file that use .h file extension
- Header files typically contain function prototypes, that define function interface: arguments and return value
- In the example above function printf is defined in stdio.h header file; compiler knows arguments that we need to use, here a string "Hello World"; note the last character \n indicating that IO device (terminal) should go to the next line (carriage return)

C and functions

- In C language a basic building block of any program is a **function**
- Functions can take some **arguments** (inputs) ... or not; and functions can produce some **output** ... or not
- Reserved keyword **void** is used to tell that a function does not take anything at input; void can be also used for output, if a function does not need to return anything
- Any C program has to contain at least **one function** & this function must be called **main**; when the program executes, main will be called first (other functions may be called from main)
- In the “*Hello World*” example main doesn’t take any input parameters, hence we pass **void**, but it does return something: a constant, 0 that could mean: “*Everything went well!*”

```
1 #include<stdio.h>
2 int main(void) {
3     printf("Hello World\n");
4     return 0;
5 }
```

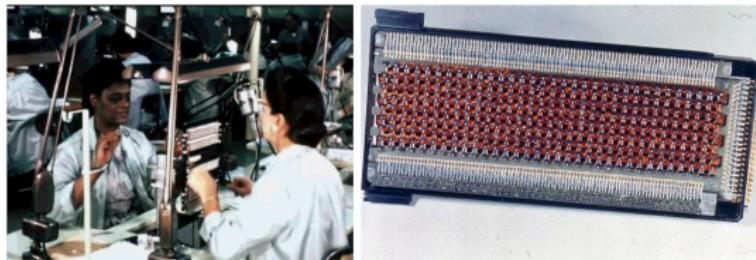
2. Basic data types, variables and operators

Data types: why bother?

- **Apollo Guidance Computer** – helped to land the first man on the moon in 1969: it had only 4kB of memory & weighted 32kg!



- Part of the weight went into **core-rope memory**, where a single bit has been manually assembled (bits were literally visible ...)



- These days programs have been crafted not to waste computing & memory resources; data used only bits that were strictly needed!

Data types in C

- C language (invented in 1970) defines different **data types** to optimize memory usage
- Data type names are standardized, but the number of bytes allocated for each data type is system dependent
- Table below summarizes some data types, and ranges assuming certain word bit depth
 - ▷ You should be able to derive the range of a given data type knowing the word depth (at least for integers)

Type	Bytes	Range (min,max)
char	1	- , -
unsigned char	1	$0, 2^8 - 1$
short int	2	$-2^{15}, 2^{15} - 1$
unsigned short int	2	$0, 2^{16} - 1$
int	4	$-2^{31}, 2^{31} - 1$
float	4	$\pm 3.2 \times 10^{\pm 38}$
double	8	$\pm 1.7 \times 10^{\pm 308}$

Data types today

- These days memory density (bits/mm^2) increased significantly, & they are cheap, just think of DRAMs → **most general purpose computing programs don't care about data types any more!**
- There is a famous quote that people laugh on today:
"640K ought to be enough for anybody"

Bill Gates, to defend 640KB usable RAM limit (1981)

- Still, data types do have their importance today:
 - ▷ Different data types are processed with different ALUs in CPUs and have different performance/power cost implications (we will see this in details later)
 - ▷ Data sets could be huge, so memory could be a bottleneck even in high-performance systems with loads of memory
 - ▷ Memory limitations are crucial when programming **embedded systems** & applications (e.g., pillcams)

Variables 1/2

- In computer science vocabulary, **variable** (or scalar) is used to **reference** a value; variable **name** is also known as **identifier**
- **Variables need to be declared** – variable declaration will reserve and name area in the memory that will store a value of a particular **data type**
 - ▷ Why is it important to know the data type at this stage?
- Example:

```
1 int a;
```
- Variable name is a **symbolic** name of a memory address; this is way more convenient for human use, addresses are just numbers; above a could refer to a counter value, easier to manipulate than the address 0x56472384
- What is the size of the address space assuming the address above?

Variables 2/2

- Variables are **dynamically** assigned & changed at **run-time**, meaning during the program execution
- They are initially set to some random values; it is the program, during execution, that **must** set variables to something meaningful **before** they are to be used; C allows to create and initialize variables in a single line of code

```
1 int c=5;
```

- Variables names are case sensitive, so x, X are different variables, can contain digits & underscores (_), but can't begin with a digit
- Multiple variables (of the same type) can be declared on the same line of your text file using commas:

```
1 int c=5, a=0, k=123;
```

- **Variables can and should be initialized when defined, a good programming practice that you are (strongly!) advised to follow**

Variables – examples

```
1 int main(void) {
2     int          a, a1, a2; // uninitialized variables
3     long int    b, b1, b2;
4     short int   c=5;      // initialized variable
5     unsigned int d;
6     char        e;
7     float       f;
8     double      g;
9     // If you use uninitialized variables on
10    // right hand side
11    // you could have anything as a result
12    // So, this will work, but you will get bogus result
13    a = a1+a2;
14
15    //Typically, you should do first:
16    a1=1;
17    a2=3;
18
19    // with initialized variables we can then do something:
20    a = a1+a2;
21    return 0;
22 }
```

Calculating the size of a given data type

- Data type sizes shown on slide 9 are **not a standard**
- Actual size, i.e. the exact byte count of a given data type, will depend on compiler, OS, machine etc. (never count on consensus, satellites have fallen because of this)
- Good idea is to use a systematic check of the operand size using a built-in C-function `sizeof` that returns the size of a given variable (in fact of that variable data type) in bytes
 - ▷ This is a major concern when writing **portable code** – a source code that targets different computer architectures, compilers etc.

```
1 int main(void) {  
2     int i;  
3     char c;  
4     printf("Size of integer: %d", sizeof(i));  
5     printf("Size of character: %d", sizeof(c));  
6     return 0;  
7 }
```

Byte order

- Memory is addressed byte-by-byte: this is to enable addressing of a single byte of information such as `unsigned char`
- For an `int` we need to store 4 bytes; question is in what order we will store these bytes? one could chose from:
 - ▷ **Little-endian** – low byte first, high byte last
 - ▷ **Big-endian** – the other way around
 - ▷ Intel x86 CPU architectures use little-endian
- What is the 4 byte integer stored at address `0X12345678` assuming little endian order?

0X12345678	0x10
0X12345679	0xAB
0X1234567A	0x23
0X1234567B	0x75

Printing variables

- Function `printf` takes few parameters as arguments, we have seen one, a string "Hello World"
- But string could contain a parameter indicated with `%` that will decide the print **format** of a variable (or constant) that follows:

```
1 #include<stdio.h>
2 void main(void) {
3     int i=12345;
4     printf("The color: %s\n", "blue");
5     printf("First number: %d\n", i);
6     printf("Second number: %04d\n", 25);
7     printf("Third number: %i\n", 1234);
8     printf("Float number: %3.2f\n", 3.14159);
9     printf("Hexadecimal: %x\n", 255);
10    printf("Octal: %o\n", 255);
11    printf("Unsigned value: %u\n", 150);
12    printf("Just print the percentage sign %%\n", 10);
13 }
```

- Formatting is necessary to **decode** binary data in human readable content, machines don't see the difference between the number & a string, all this is just binary information

Variable scope

- Variables take place in memory, we know that the exact amount of **allocated** space will vary depending on the data type
- Not all variables have to be saved and kept in memory all the time
 - ▷ Think of all intermediate computations that will occur in any computer program (computer is a Turing machine after all!)
- To control the amount of variables kept in memory we introduce two very different types of variables:
 - ▷ **Local variables** – they last throughout the life of a function and they are not **accessible outside the function in which they are defined**, hence the name
 - ▷ **Global variables** – they last throughout the program life, and are thus accessible from any function called in the program (i.e. other functions called from main function)

**It is very important to understand
the difference between the two!**

Variable scope within the function

In the code below we consider local variables only

```
1 void main(void) {
2     int i=4;
3     int j=10;
4     i++;           // -----
5                 // |
6     if (j > 0) {   //
7         printf("i is %d\n",i);    // i is visible
8     }               // |
9                 // |
10    if (j > 0) {    //
11        // |
12        int i=100; // this i is local to this code block
13        printf("i is %d\n",i);    //
14    }             // the i of this block dies here
15                 // |
16        printf("i is %d\n",i);    //
17 // This will print i=5
18 // Where does this 5 comes from? -----
19 }
```

Variable scope outside the function

```
1 int i=4;           // global variable defined
2
3 void main(void) {
4     int j=7;       // local to this function
5     j++;
6     i++;          // global i is visible here
7     func();        // we now call a function
8 }
9
10 func() {
11     int j=10;     // j declared local to this function
12     j++;          // j incremented
13     i++;          // global i is visible and incremented
14                         // local j lost
15 }
```

What are the values of i, j
before and after func() has been called?

Variable scope outside the function: variants (1/2)

- You can use the same name for local and global variables!
- Local variable inside a function **will take preference**, i.e. the scope of the function will rule out the global variable
- **But using the same name for local and global variables is not advisable (code of good programmer conduct)** – unless you explicitly want to write unreadable code! which you might want to do, check out my personal favorite: <https://www.ioccc.org>

```
1 #include <stdio.h>
2 int g = 20; // global variable
3
4 int main (void) {
5     int g = 10; // local variable same name as global
6     printf ("value of g = %d\n", g);
7     return 0;
8     // This will print 10 and not 20
9 }
```

Variable scope outside the function: variants (2/2)

- Also, function arguments are treated as local variables, i.e. **they will rule over global variables**, so the code below:

```
1 #include <stdio.h>
2 int a = 20;
3 int main (void) {
4     int a = 10;           // Note that we re-define a
5     int b = 20;
6     int c = 0;
7     printf ("value of a in main() = %d\n", a);
8     c = sum( a, b );
9     printf ("value of c in main() = %d\n", c );
10    return 0;
11 }
12 int sum(int a, int b) {
13     printf ("value of a and b in sum() = %d, %d\n", a, b );
14     return a + b;
15 }
```

will produce:

```
1 value of a in main() = 10
2 value of a in sum() = 10
3 value of b in sum() = 20
4 value of c in main() = 30
```

Mathematical operators

- Basic arithmetic operators:
 - ▷ + Addition, - Subtraction, / Division, * Multiplication
 - ▷ Also % Modulo
 - ▷ Work on integer & floating point
- **Watch out** – division with two integers will do integer division; if either argument is a float, it does floating point division
- Other complex mathematical functions are part of **math library**:

Trigonometric functions	
<code>cos</code>	Compute cosine (function)
<code>sin</code>	Compute sine (function)
<code>tan</code>	Compute tangent (function)
<code>acos</code>	Compute arc cosine (function)
<code>asin</code>	Compute arc sine (function)
<code>atan</code>	Compute arc tangent (function)
<code>atan2</code>	Compute arc tangent with two parameters (function)

Hyperbolic functions	
<code>cosh</code>	Compute hyperbolic cosine (function)
<code>sinh</code>	Compute hyperbolic sine (function)
<code>tanh</code>	Compute hyperbolic tangent (function)
<code>acosh</code>	Compute arc hyperbolic cosine (function)
<code>asinh</code>	Compute arc hyperbolic sine (function)
<code>atanh</code>	Compute arc hyperbolic tangent (function)

Exponential and logarithmic functions	
<code>exp</code>	Compute exponential function (function)
<code>frexp</code>	Get significand and exponent (function)
<code>ldexp</code>	Generate value from significand and exponent (function)
<code>log</code>	Compute natural logarithm (function)
<code>log10</code>	Compute common logarithm (function)
<code>modf</code>	Break into fractional and integral parts (function)
<code>exp2</code>	Compute binary exponential function (function)
<code>expm1</code>	Compute exponential minus one (function)
<code>ilogb</code>	Integer binary logarithm (function)
<code>log1p</code>	Compute logarithm plus one (function)
<code>log2</code>	Compute binary logarithm (function)
<code>logb</code>	Compute floating-point base logarithm (function)
<code>scalbn</code>	Scale significand using floating-point base exponent (function)
<code>scalbln</code>	Scale significand using floating-point base exponent (long) (function)

Power functions	
<code>pow</code>	Raise to power (function)
<code>sqr</code>	Compute square root (function)
<code>cbrt</code>	Compute cubic root (function)
<code>hypot</code>	Compute hypotenuse (function)

Mathematical operators and variables (1/2)

- Most mathematical operations should take operands of the same data type (apples & apples and **not** apples & peaches)
- But sometimes mixing types is allowed, only if automatic **type promotion** could be applied
- Example: `char(1B)` and `int(2B)` can be combined in arithmetic expressions such as: `int (a) + char (b)`
- **Compiler promotes** the smaller type (`char`) to be the same size as the larger type (`int`) before combining the values
- Promotions are determined at compile time based purely on the types of values in expressions
- Promotions **do not lose information** (essential) – they always convert from a given to compatible, larger, type to avoid information loss; they are valid for integers

Mathematical operators and variables (2/2)

- What will happen in the code below?

```
1 #include <stdio.h>
2 void main(void) {
3     int sum = 17, count = 5;
4     double mean;
5     mean = sum / count;
6     printf("Value: %f\n", mean);
7 }
```

- Automatic promotion will not work!
- We need to do explicit type change, done using casting operator:

```
1 #include <stdio.h>
2 void main(void) {
3     int sum = 17, count = 5;
4     double mean;
5     mean = (double) sum / count;           //type cast done here
6     printf("Value: %f\n", mean);
7 }
```

Boolean variables and operators

- In C there is **no** Boolean data type that will use a single bit of addressable memory; minimum data size one can use is at least one byte wide
- However there are predefined Boolean values:
 - ▷ FALSE – any integer value set to 0 (valid for any data type), and
 - ▷ TRUE – anything different from 0; thus TRUE is not necessarily a binary 1 at LSB position!
- As for the operators, there are two different classes of Booleans:
 - ▷ **Boolean logic operators** – manipulate the above defined Boolean values of TRUE & FALSE as in standard Boolean algebra with traditional Boolean operators AND, OR, NOT
 - ▷ **Bitwise operators** – manipulate variables at bit-level (for the same weight) with traditional Boolean operators

Boolean logical operators

- Represented using double characters, so `&&`, `||` for AND and OR (remember that bit-wise operators are single character wide: `&`, `|`)
- Bitwise operators have higher precedence over booleans

```
1 #include <stdio.h>
2 void main(void) {
3     int a = 5;
4     int b = 20;
5     int c ;
6     if ( a && b ) printf("Line 1 - Condition is true\n");
7     if ( a || b ) printf("Line 2 - Condition is true\n");
8     // change the value of a and b
9     a = 0;
10    b = 10;
11    if ( a && b ) {printf("Line 3 - Condition is true\n");}
12        else {printf("Line 3 - Condition is not true\n");}
13
14    if ( !(a && b) ) {printf("Line 4 - Condition is true\n");}
15 }
```

- Line 1, 2 & 4 will be printed, but not Line 3. Can you show why?

Boolean bitwise operators

- Can be applied on different (integer) data types
- Bitwise operators – manipulate memory at the bit-level:
 - ▷ (~) **bitwise negation** (unary) – flip 0 to 1 and 1 to 0 throughout
 - ▷ (&) **AND** (|) **OR** and (^) **exclusive OR** (XOR)
 - ▷ >> / << Right/Left Shift of the source operand by 1 bit
(destination is the same as source)
 - The above is equivalent to multiply / divide by power 2 – **Why?**
- In the example:

```
1 unsigned char a=195;
2 unsigned char b=87;
3 unsigned char c;
4 c=a&b;
5
6 printf("Result c: %d\n", c);
```

printf will show c=01000011 in decimal

- ▷ **Can you show why?**

3. Control Structures

Control of the instruction flow

- Control statements allow **conditional** execution of instructions, we have already used execution of mutually exclusive sequences of instructions with `if ... then ... else`
- Different forms are given below, where `<expression>` evaluates to a Boolean value of TRUE xor FALSE as defined previously; **Can you explain “xor” in this sentence?**

```
1 if (<expression>) <statement> // Simple form with no {}'s or else clause
2
3 if (<expression>) {
4     <statements>
5 }
6
7 if (<expression>) {
8     <statements>
9 } else {
10     <statements>
11 }
```

- Other instructions that allow to alter the execution flow ...

Switch statements

- To allow more compact and readable code when a single variable could take many different values we should favor switch rather than a series of if:

```
1  switch (<expression>) {  
2      case <const-expression-1>:  
3          <statement>  
4          break;  
5      case <const-expression-2>:  
6          <statement>  
7          break;  
8      // exp.3, expr4 run same statements  
9      case <const-expression-3>:  
10     case <const-expression-4>:  
11         <statement>  
12         break;  
13     default:    // optional  
14         <statement>  
15 }
```

Loops – repeat set of statements

- `while` – evaluates the test expression before every iteration; it can execute **zero times** if the condition is initially FALSE
- `do while` – will always execute **at least once**
- `for` – initialization, continuation condition and action

```
1  while (<expression>) {           // This one could execute zero times
2      <statements>
3  }
4
5  do {                           // This one executes at least once
6      <statements>
7  } while (<expression>)
8
9  for (i = 0; i < 10; i++) {    // Will execute exactly 10 times
10     <statements>
11 }
12
13 for (i = 1; i <= 10; i++) {   // Will execute exactly 10 times
14     <statements>
15 }
```

Forced loop exit

- Any loop can be **broken**, i.e. we can force the exit from the loop using another condition and **before** the loop condition becomes FALSE; for example:

```
1 #include <stdio.h>
2
3 int main () {
4     /* local variable definition */
5     int a = 10;
6     /* while loop execution */
7     while( a < 20 ) {
8         printf("value of a: %d\n", a);
9         a++;
10
11     if( a > 15) {
12         /* This will terminate the loop */
13         break;
14     }
15 }
16 return 0;
17 }
```

4. Functions

Functions enable program structuring

- Allow us to implement complex things using **divide and conquer** approach: we cut big problems into bunch of smaller functionalities to be executed one after the other
- Functions need to implement **call** and **return** mechanism
- Functions use **arguments** to exchange data:

```
1 #include<stdio.h>
2 // main
3 int main(int argc, char* argv[]) {
4 }
```

Who's providing arguments to the program above?

Passing values to functions

```
1 #include<stdio.h>
2 // Two different functions defined
3 void F1(int sizex, int sizey);
4 void F2(unsigned char src, unsigned char dest);
5
6 // main
7 int main() {
8     int x=3,y=5;
9
10    F1(x,y);
11    F2(x,y);
12 }
13 // implementation of function F1
14 void F1(int sizex, int sizey) {
15     // some computations
16     ...
17 }
18 // implementation of function F2
19 void F2(unsigned char src, unsigned char dest) {
20     // some computations
21     ...
22 }
```

What seems to be the problem here?

Read only argument

- It is possible to prevent an argument of being modified within a function using keyword `const` before the variable type
- In the example below `x` can't be used on the left side of the assignment operator (`=`)

```
1 // Some function
2 void doit(const int x) {
3     x=5; // this would be illegal
4 }
5 // Main
6 int main()
7 {
8     int z=27;
9     doit(z);
10    printf("z is now %d\n", z);
11    return 0;
12 }
```

- Limited utility for type of variables we consider now, but could be useful for those we will see next → **pointers** ...

5. Pointers

Basic concepts

- Pointers are variables, just like normal variables used so far, but rather than data (so integers, floats) they store addresses
- Since they are variables they need to be declared and they **need to have a certain data type**
 - ▷ Why do we need to define a data type for a pointer?
- The size of a pointer depends on HW architecture but typically 32 or 64 bits these days (the size of the pointer will define the addressable space of a program, make sure you understand this)
 - ▷ What data type are the addresses?
- To differentiate from other variables, pointer names are preceded with a (*) when declared
- Same symbol (*) is also used to access the content of the address stored in the pointer later in the program
- So, pointers refer to two things:
 - ▷ Some address in the system memory
 - ▷ Data stored on that address – *value pointed by the pointer*

Example

```
1 #include <stdio.h>
2
3 int main () {
4     int *ptr;           // pointer variable declaration
5                         // good practice to suffix/prefix with ptr
6
7     ...                // some initialization code
8
9     // print the address stored in the pointer
10    // not the hex format
11    printf("Address stored in ip variable: %x\n", ptr);
12
13    // access the value using the pointer
14    printf("Value of *ptr variable: %d\n", *ptr);
15    // we print the value pointed by ptr
16    return 0;
17 }
```

- Pointer initialization is crucial!
- Read operation on the wrong address could be tolerated, **but certainly not write** → such things typically cause nasty crashes
- Explain the reasoning for the above statement

On variables & pointers

- Variables are stored in memory, and memory works with addresses
- So any variable will have an address – e.g. variable a stored at address 0x1034556 stores integer value 5
- We can access the address of any normal variable using & operator placed before the variable name; and we of course use a pointer to store that address

```
1 #include <stdio.h>
2 int main () {
3     int var1;           // variable
4     int *ptr;          // pointer
5     var1 = 20;          // store value in var1
6     ptr = &var1;         // pointer ptr stores the address of var1 now
7     printf("Variable var1: %d\n", var1);
8     printf("Address of var1: %x\n", &var1);
9     printf("Content of the address: %d\n", *ptr);
10    printf("Address stored in pointer: %x\n", ptr);
11    printf("Address of the pointer: %x\n", &ptr);
12 }
```

What do we see in the terminal after execution of the code above?

Pointers, variables & memory

```
short int var1=20
```

```
&var1=0x000000FF
```

```
short int *ptr
```

```
ptr=&var1
```

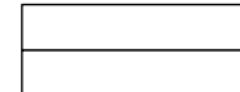
```
ptr= 0x000000FF
```

```
*ptr= 20
```

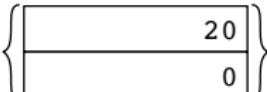
```
&ptr=0xFF000000
```

Memory
structured in 1Byte

Address space
32 bits



...



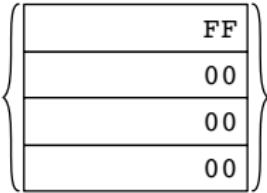
20

0

...



...



FF

00

00

00

2 Bytes reserved

0x000000FF

0x00000100

4 Bytes reserved

0xFF000000

0xFF000001

0xFF000002

0xFF000003

Pointers, functions calls and arguments 1/2

Look at the code below:

```
1 void ChangeI(int i);
2 main() {
3     int i=0;           // some local variable
4     ChangeI(i);       // that we pass to a function
5 }
6 void ChangeI(int i) {
7     i=10;             // Function modifies the argument
8                         // Is variable modified when returning?
9 }
```

- Despite the common sense saying that the above program is ok:
this will never work in practice!
- This is because the variable `i` in `ChangeI`, when passed as an argument, **is a copy** of `i`, not the variable itself
 - We say: **argument is passed by value**
- The copy of `i` used in `ChangeI` will be destroyed on exit, so in function `main`, the variable `i` will remain unchanged

Pointers, functions calls and arguments 2/2

And now:

```
1 void ChangeI(int *i);
2 main() {
3     int i=0;
4     ChangeI(&i);           // we now pass the address to the function
5 }
6 void ChangeI(int *i) {
7     *i=10; // we access the value of the address
8 }
```

- This one will work! – we pass the pointer, i.e. the address
We say: argument is passed by reference
- Since this is a reference, i.e. absolute memory location, it can be accessed anywhere in the program & we can do whatever we want with the content of it
- Another advantage: we can pass an arbitrary number of arguments to a function even if we don't know the number of variables to pass at compile-time; Can you explain?

Example: parsing command line arguments

Consider the code below:

```
1 #include <stdio.h>
2
3 int main (int argc, char *argv[]) {
4     int i=0;
5     printf("cmdline args count=%s \n", argc);
6
7     // First argument is executable name only
8     printf("Executable name=%s \n", argv[0]);
9
10    // Print all arguments
11    for (i=1; i< argc; i++) {
12        printf("Arg%d=%s \n", i, argv[i]);
13    }
14
15    printf("\n");
16    return 0;
17 }
```

Imagine a concrete example of the execution for the code above?

6. Arrays, memory and data structures

One dimensional arrays

- In C you can define an **array** of variables of the same data type, that will be stored in memory as **contiguous** list of elements
- Size of the area must be supplied to determine the amount of memory that needs to be reserved when variable is declared:

```
1 int myArray[100], myArray2[100];
2 myArray[0] = 17;           // set first element
3 myArray[99] = 47;         // set last element
4 myArray2[0] = 15;         // set first element
```

here myArray needs `sizeof(int) × 100 bytes`

- First element in the array is indexed with a 0, so in the above index 99 corresponds to the 100th (last) element in the array
- Write access to an element **outside of the array range could cause problems** since the program could erase some vital information
- What will happen in the above example if we add as line 5:

```
1 myArray[100]=59;
```

Manipulating 1D array

- Following program will do some simple arithmetic on two arrays:

```
1 int main(void) // Main
2 {
3     int A[100],B[100],C[100];
4
5     // option 1
6     for(i=0;i<100;i++)
7         C[i]=A[i]+B[i];
8     // option 2
9     for(i=1;i<=100;i++)
10        C[i-1]=A[i-1]+B[i-1];
11     return 0;
12 }
```

- Make sure you can:
 - ▷ Explain the above `for` loops in details
 - ▷ Discuss index in both cases
 - ▷ What do we miss here?

Multi dimensional arrays

- Multi-dimensional arrays could be defined as follows:

```
1 int myArray[10][10];
2 myArray[0][0] = 17;
3 myArray[5][5] = 47;
```

- Memory space is contiguous, so in reality multi-dimensional arrays are one 1D block of data in the memory, so previous variable declaration is equivalent to:

```
1 int myArray[100];
2 board[0] = 17;
3 board[5+5*10] = 47; // note how do we access element 5,5 in 1D array
```

- In general any element (i, j) of the array is accessed using:

$$i + j \times \text{sizeof(arrayX)}$$

where i, j are respectively row and column indexes of 2D array and arrayX is the “width” of the 2D array

More on memory

- We know that memory, seen by the executed program, is a **contiguous** 1D memory space with addresses ranging from, say 0x00000000 to 0xFFFFFFFF in increments of 1
 - ▷ What is the size of the memory above assuming 1 byte per address?
- Each memory location stores 1 byte, this is a HW choice, it could be something else, but 1B/address is the most common
 - ▷ You should be able by now to explain why.
- Everything is saved in the memory: the program itself, execution context of each function called and of course all the data!
- That is many things, so we need an efficient way to manage this
- User accessible memory is split into two VERY distinct parts:
 - ▷ Stack
 - ▷ Heap

Stack

- Stack is a memory region storing temporary variables created by each function, including main () function
- Stack is managed by the CPU in LIFO fashion (Last In, First Out)
 - ▷ When function declares new variable, it is pushed onto the stack
 - ▷ When we exit a function, all variables pushed onto stack by that function, are freed, i.e. deleted (stack variables are local)
 - ▷ Once a stack variable is freed, that region of memory becomes available for other stack variables
- Memory management is automated, so the programmer does not have to allocate or free stack memory explicitly
- CPU organizes stack efficiently & RD/WR ops. to stack are fast
- This approach is extremely handy since it enables memory management automation well suited for computations on smaller data sets with lots of functions

Static variables

- Not all variables are erased when the function is left → **static variables**; they preserve their value even if out of scope!
- In other words a static variable will preserve previous value from the previous scope; i.e., it will not be initialized again in the new scope (e.g. after two successive function calls); example:

```
1 #include<stdio.h>
2
3 int sf(void) {
4     static int count = 0;
5     count++;
6     return count;
7 }
8
9 int main(void) {
10    printf("%d, ", sf());
11    printf("%d, ", sf());
12    printf("%d ", sf());
13    return 0;
14 }
```

- Execution of the program above will produce: 1, 2, 3.

Practical stack considerations

- Stack size is fixed for a given program during compile-time and can't be changed during program execution; in other words if you want the program to use bigger stacks it needs to be recompiled from scratch using another compiler setting
- Default stack size is typically few to few tens of kBytes of memory only, not more (check your compiler default stack size)
- Small stack size allows certain depth of successive (nested) function calls: function that calls other function that calls other function, and so on, including recursive functions (see further)
- Assume memory of 16GB and stack of 1MB, how many nested functions calls you will be able to perform?
- What will happen if we try to allocate more than what is made available by the stack definition?

Stack overflow

*In software, a **stack buffer overflow** or **stack buffer overrun** occurs when a program writes to a memory address on the program's call stack outside of the intended data structure, which is usually a fixed-length buffer. Stack buffer overflow bugs are caused when a program writes more data to a buffer located on the stack than what is actually allocated for that buffer. This almost always results in corruption of adjacent data on the stack, and in cases where the overflow was triggered by mistake, will often cause the program to crash or operate incorrectly. Stack buffer overflow is a type of the more general programming malfunction known as buffer overflow (or buffer overrun).*

Overfilling a buffer on the stack will derail program execution (and even OS, if OS is poor!), this is because stack contains return addresses for all active function calls!

Heap

- To handle data sets bigger than typical stack size, or if the size of the data set is not known during compile time we use **heap** memory that allows access to all addressable memory space
- Heap is accessed through **dynamic memory allocation** (meaning at run-time, during execution), using standardized high-level functions that interface your program with the OS:
 - ▷ For allocation: `malloc()` (memory not initialized) or `calloc()` (initializes allocated memory to 0)
 - ▷ For liberation: `free()` deallocates memory not needed any more
- **Memory management** is performed explicitly by the programmer; if memory is not freed, the program will have **memory leaks**
- Depending on amount of memory leaks, and the time your program runs, computer will sooner or later **run out of memory**
- **This will cause program to crash, or to exit gracefully, depending on how well you write your code**

Example: dynamic memory allocation

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main (void) {
5     int i,n;
6     char *buffer;           // pointer that will point to sting array
7                         // the length of a string is user defined
8     printf ("How long do you want the string? ");
9     scanf ("%d", &i);
10                // malloc returns a void pointer we need to cast
11     buffer = (char *) malloc ((i+1)*sizeof(char));
12                // graceful exit!
13                // in case we have memory leaks elsewhere
14     if (buffer==NULL) {printf ("No memory!"); exit (1);}
15
16                // fill array with random strings
17     for (n=0; n<i; n++)
18         buffer[n]=rand()%26+'a';
19                 // append end character
20     buffer[i]='\0';
21     printf ("Random string: %s\n",buffer);
22                 // we free unused memory in the end, i+1 in line 11
23     free (buffer);
24     return 0;
25 }
```

Example: memory, arrays & data types 1/3

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #define SIZE    300
5 int main() {
6     printf("Playing with arrays !\n");
7     unsigned char *ptUnsChar;
8     int          *ptInt;
9     long         *ptLongInt;
10    float        *ptFloat;
11    double       *ptDouble;
12    ptUnsChar = (unsigned char *)malloc(SIZE*sizeof(unsigned char));
13    ptInt      = (int *)malloc(SIZE*sizeof(int));
14    ptLongInt = (long *)malloc(SIZE*sizeof(long));
15    ptFloat    = (float *) malloc (SIZE*sizeof(float));
16    ptDouble   = (double *) malloc (SIZE*sizeof(double));
17    if (ptUnsChar == NULL || ptInt == NULL || ptLongInt ==NULL || ptFloat == NULL || ptDouble ==
18        NULL) {
19        printf("No memory !!!"); exit(0);
20    } else {printf("Memory successfully allocated !\n\n\n");}
21    for (long i=0 ; i < SIZE ; i++ ) {
22        ptUnsChar[i]=i;
23        ptInt [i]=i;
24        ptLongInt[i]=i;
25        ptFloat[i]=pow(1.0/3.0,-64);
26        ptDouble[i]=pow(8.0/3.0,256);
27    }
```

Contd. on the next slide

Example: memory, arrays & data types 2/3

```
1 printf("Size of unsigned char: %ld \n", sizeof(unsigned char));
2 printf("Size of int: %ld \n", sizeof(int));
3 printf("Size of long int: %ld \n", sizeof(long));
4 printf("Size of float: %ld \n", sizeof(float));
5 printf("Size of double: %ld \n\n", sizeof(double));
6 printf("Array of unsigned chars: \n");
7 printf("Address of 1st element:    %p; : %d \n",      &ptUnsChar[0], ptUnsChar[0]);
8 printf("Address of 25th element:   %p; : %d \n",      &ptUnsChar[24], ptUnsChar[24]);
9 printf("Address of last element:   %p; : %d \n",      &ptUnsChar[SIZE-1], ptUnsChar[SIZE-1]);
10 printf("Address of last + 1 element: %p; : %d \n\n",  &ptUnsChar[SIZE], ptUnsChar[SIZE]);
11 printf("Array of simple integers: \n");
12 printf("Address of 1st element:    %p; : %d \n",      &ptInt[0], ptInt[0]);
13 printf("Address of 25th element:   %p; : %d \n",      &ptInt[24], ptInt[24]);
14 printf("Address of last element:   %p; : %d \n",      &ptInt[SIZE-1], ptInt[SIZE-1]);
15 printf("Address of last + 1 element: %p; : %d \n\n",  &ptInt[SIZE], ptInt[SIZE]);
16 printf("Array of long integers: \n");
17 printf("Address of first element:  %p; : %ld \n",     &ptLongInt[0], ptLongInt[0]);
18 printf("Address of 25th element:   %p; : %ld \n",     &ptLongInt[24], ptLongInt[24]);
19 printf("Address of last element:   %p; : %ld \n",     &ptLongInt[SIZE-1], ptLongInt[SIZE-1]);
20 printf("Address of last + 1 element: %p; : %ld \n\n",  &ptLongInt[SIZE], ptLongInt[SIZE]);
21 printf("Array of floats: \n");
22 printf("Address of first element:  %p; : %f \n",      &ptFloat[0], ptFloat[0]);
23 printf("Address of 25th element:   %p; : %f \n",      &ptFloat[24], ptFloat[24]);
24 printf("Address of last element:   %p; : %f \n",      &ptFloat[SIZE-1], ptFloat[SIZE-1]);
25 printf("Address of last + 1 element: %p; : %f \n\n",  &ptFloat[SIZE], ptFloat[SIZE]);
26 printf("Array of doubles: \n");
27 printf("Address of first element:  %p; : %lf \n",     &ptDouble[0], ptDouble[0]);
28 printf("Address of 25th element:   %p; : %lf \n",     &ptDouble[24], ptDouble[24]);
29 printf("Address of last element:   %p; : %lf \n",     &ptDouble[SIZE-1], ptDouble[SIZE-1]);
30 printf("Address of last + 1 element: %p; : %lf \n\n",  &ptDouble[SIZE], ptDouble[SIZE]);
```

Contd. on the next slide

Example: memory, arrays & data types 3/3

```
1 // We free the memory  
2  
3     free(ptDouble);  
4     free(ptFloat);  
5     free(ptLongInt);  
6     free(ptInt);  
7     free(ptUnsChar);  
8     return 0;  
9 }
```

- Calculate manually and confirm with program execution the address of 11th element in each case?
- What will happen if you modify SIZE=300000000000?
- What would be the max size of the array?
- What will happen if *i* is int and SIZE=300000000000?
- Explain why last element in unsigned char is 43?

Complex data structures 1/3

- It is possible to group variables together to form more complex, composite data types → **structures**
- Thus a single variable addresses multiple (and different!) elements that have been defined in the structure
- Access to particular variable is enabled with a dot (.)

```
1 // User defined data structure
2 struct fraction {
3     int numerator;
4     int denominator;
5 };
6
7 // Don't forget the semicolon!
8
9 struct fraction f1, f2; // declare two variables for structures
10
11 f1.numerator    = 22;    // some assignments
12 f1.denominator = 7;
13
14 f2 = f1;        // this copies over the whole struct
```

Complex data structures 2/3

- You can copy two records of the same type using a single assignment statement, in the previous example:

```
1 // This will copy all the elements of f1 into f2  
2 f2 = f1;
```

- However **test will not work on structures** (e.g. equal using ==); we should implement a function (or a code block with {}) that will check for equality of all elements one by one
- Different elements of the structure are stored in the memory, in the order in which they are defined
- We can mix different data types in a single structure definition (this seems obvious, right?)
- Function `sizeof()` works on structures too (and is very useful)

Complex data structures 3/3

- Structure data types can be defined with pointers, to access different elements in the structure we use symbol ->

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct fraction {      // User defined data structure
4     int numerator;
5     int denominator; };
6 int main(void) {
7     struct fraction *f1=NULL, *f2=NULL; // declare 2 vars as pointers
8     f1 = (struct fraction*)malloc(1*sizeof(struct fraction));
9     f2 = (struct fraction*)malloc(1*sizeof(struct fraction));
10    if (f1 == NULL || f2 == NULL) {printf("No memory !!!"); exit(0);}
11    else {printf("Memory successfully allocated !\n\n\n");}
12    f1->numerator = 22;           // some assignments
13    f1->denominator = 7;
14    //f2 = f1;                  // THIS WILL BE A PROBLEM !!!
15    printf("N:%d D:%d \n",f1->numerator,f1->denominator);
16    printf("N:%d D:%d \n",f2->numerator,f2->denominator);
17    f2->numerator = 33;           // some assignments
18    f2->denominator = 3;
19    printf("N:%d D:%d \n",f1->numerator,f1->denominator);
20    printf("N:%d D:%d \n",f2->numerator,f2->denominator);
21    free(f1); free(f2);
22 }
```

Explain what will happen if we enable line 15?

7. File access

Open file for reading

- Using `fopen`, defined in `stdio.h`; header file `stdio.h` must be included, otherwise compiler doesn't know what `fopen` is
- Arguments: file name and the access type (`r`, `w`, `a`, etc.); returns pointer to the (data) **stream**, or `NULL` pointer if file not opened
- We use `fread` built-in function to copy `size` elements from file to source pointer

```
1 //Open some files and define a pointer to the file
2 FILE *fp = fopen("data.test", "r");
3 // check if file has been successfully opened, always do this
4 if (fp!=NULL) {
5     printf("File opened ... \n");
6     fread(source, sizeof(unsigned char), size, fp);
7     printf("File read! \n");
8     fclose(fp);
9 } else {
10     printf("Can't open specified file!\n"); // file not found, exit
11     exit(0);
12 }
```

What is missing & what happens during execution?

Open file for writing

- Change the argument: access type is now w
- We also need to provide information on what to copy:
 - ▷ from where in memory (destination)
 - ▷ and how many elements (size)
- Once we are finished we need to close file pointer(s), and this should be done for both read & write file operations

```
1 FILE *fp2 = fopen("res_normal.raw", "w");      // pointer to file
2 if (fp2!=NULL) {                                // test
3     printf("File opened for write ... \n");
4     fwrite(destination, sizeof(unsigned char), size, fp2);
5     printf("File written! \n");
6     fclose(fp2);
7 } else {
8     printf("Can't open specified file!\n");
9     exit(0);
10 }
```

Most of the tips in the above code: you should be able to figure them out by **reading documentation** on C-functions

Learn how to read documentation of C-functions

```
size_t fread(  
    void *buffer,  
    size_t size,  
    size_t count,  
    FILE *stream  
)
```

Parameters

buffer

Storage location for data.

size

Item size in bytes.

count

Maximum number of items to be read.

stream

Pointer to FILE structure.

Return Value

fread returns the number of full items actually read, which may be less than count if an error occurs or if the end of the file is encountered before reaching count. Use the feof or ferror function to distinguish a read error from an end-of-file condition. If size or count is 0, fread returns 0 and the buffer contents are unchanged. If stream or buffer is a null pointer, fread invokes the invalid parameter handler, as described in [Parameter Validation](#). If execution is allowed to continue, this function sets errno to EINVAL and returns 0.

See [_doserrno](#), [errno](#), [_sys_errlist](#), and [_sys_nerr](#) for more information on these, and other, error codes.

8. Examples to analyze

Complete example: merge two files

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     FILE *fs1, *fs2, *ft;
5     char ch, file1[20], file2[20], file3[20];
6
7     printf("Enter name of first file\n"); gets(file1);
8     printf("Enter name of second file\n"); gets(file2);
9     printf("Enter name of resulting file\n"); gets(file3);
10    fs1 = fopen(file1, "r");
11    fs2 = fopen(file2, "r");
12    if( fs1 == NULL || fs2 == NULL ) {
13        perror("Error ");
14        printf("Press any key to exit...\n");
15        getch();
16        exit(EXIT_FAILURE);
17    }
18    ft = fopen(file3, "w");
19    if( ft == NULL ) {
20        perror("Error ");
21        printf("Press any key to exit...\n");
22        exit(EXIT_FAILURE);
23    }
24    while( ( ch = fgetc(fs1) ) != EOF )
25        fputc(ch,ft);
26    while( ( ch = fgetc(fs2) ) != EOF )
27        fputc(ch,ft);
28    printf("Two files were merged into %s file successfully.\n",file3);
29    fclose(fs1);fclose(fs2);fclose(ft);
30    return 0;
31 }
```

Check if a string is a vowel

Brute force: check with an if

```
1 #include <stdio.h>
2
3 int main() {
4     char ch;
5
6     printf("Enter a character\n");
7     scanf("%c", &ch);
8
9     if (ch == 'a' || ch == 'A' || ch == 'e' || ch == 'E' || ch == 'i' ||
10        ch == 'I' || ch == 'o' || ch == 'O' || ch == 'u' || ch == 'U')
11         printf("%c is a vowel.\n", ch);
12     else
13         printf("%c is not a vowel.\n", ch);
14 }
```

How to do this better?

Print a decimal number as binary

```
1 #include <stdio.h>
2
3 int main() {
4     int n, c, k;
5
6     printf("Enter an integer in decimal number system\n");
7     scanf("%d", &n);
8
9     printf("%d in binary number system is:\n", n);
10    for (c = 31; c >= 0; c--)
11    {
12        k = n >> c;
13
14        if (k & 1)
15            printf("1");
16        else
17            printf("0");
18    }
19    printf("\n");
20    return 0;
21 }
```

What needs to be changed if n was an unsigned char?

Recursive functions

- Is a function that calls itself to improve code readability & allow elegant implementations of some problems

```
1 void func()      <-----|<---|
2 {                |         |
3     . . . . .    |         |
4     func();      >---- recursive call----|--->|
5     . . . . .
6 }
7 int main()
8 {
9     . . . . .
10    func();      >-----|-----|
11    . . . . .
12 }
```

- To prevent infinite recursion, any recursive function **must** implement if ... then ... else ... (or similar)
 - ▷ Explain why this is not that good?
- Against: you need to pay a lot of attention when writing recursive; you pay some performance penalty since lot of function calls that add execution overhead; You should know why
 - ▷ You can always use loops to mimic recursion

Power of a number using recursive function

```
1 #include <stdio.h>
2 int power(int n1, int n2);
3 int main()
4 {
5     int base, powerRaised, result;
6
7     printf("Enter base number: ");
8     scanf("%d",&base);
9
10    printf("Enter power number(positive integer): ");
11    scanf("%d",&powerRaised);
12
13    result = power(base, powerRaised);
14
15    printf("%d^%d = %d", base, powerRaised, result);
16    return 0;
17 }
18 int power(int base, int powerRaised)
19 {
20     if (powerRaised != 0)
21         return (base*power(base, powerRaised-1));
22     else
23         return 1;
24 }
```

What prevents infinite recursion in the code above?

9. Assembly primer

Overview

- Every CPU “speaks” his own **machine language** composed of **low-level** instructions; the actual functionality of these functions has been decided during microprocessor HW design-time
- Instructions are binary encoded & usually expressed in software interfaces with hexadecimal base (for human & machine convenience); instruction binary code is called **opcode**
- Opcodes are used to **derive the appropriate control signals using decoding logic circuitry** that will drive different functional HW blocks of the microprocessor (more on this later on)
 - ▷ **Can you explain decoding at circuit level?**
- Opcodes are hard to process by humans; to simplify things each opcode has a string equivalent called **mnemonic**
- Mnemonics are translated into opcodes using a relatively simple computer program – **assembler** – that produce the executable file; assembler is not exactly the same thing as a compiler
 - ▷ **You should be able to explain why**

Meaning of life ...

- *Why learning assembly language (today)?* is a valid question! especially knowing that we have **so many** compiled/interpreted languages that can easily abstract low-level (machine) language instructions & that assembly is hard to learn & deploy, error prone, difficult to debug, etc.
- **Still, there are few good reasons to use assembly!**
- With assembly you will:
 - ▷ learn how CPU works & be able to write more efficient code; compilers are indeed getting better & better, but abstraction always come at a price of **efficiency** (think of Matlab)
 - ▷ write **device drivers**, i.e. programs that control specific HW components (any computer is full of it ...)
 - ▷ write compilers & more generally any **HW dedicated software**
 - ▷ write **embedded software** where memory footprint & efficiency with limited computational power are the key values
 - ▷ **debug** some very specific issues & learn how to **profile** the existing code (performance bottleneck identification)

Basic assembly instruction

- Assembly program is a text file, in which each line contains one assembly instruction, here based on NASM/MASM syntax:
 - ▷ **NASM** – Netwide assembler/disassembler for Intel x86 used for 16, 32 (IA-32) & 64-bit (x86-64) programs (also **Linux**, open source)
 - ▷ **MASM** – Microsoft Macro Assembler for Intel x86 & **MS-OS**
- General assembly instruction format:
`[label] mnemonic [operands] [;comment]`
 - ▷ [label] – used to identify a specific place in our program (tag)
 - ▷ [mnemonic] – will be translated into an opcode; note that instructions that use different operands will have different opcodes
 - ▷ [operands] – could be without, or with one, two & sometimes even more operands
- Data flow direction from right to left, but it could be inverse (this is platform/compiler dependent):

```
1 mov dst,src;      move one register into another  
2 add dst,src;     do some arithmetic
```

Intel x86 operating modes – memory management

- Real mode

- Real mode
 - ▷ From 8086 and later x86-compatible CPUs, used for boot sequence after 80286
 - ▷ So, no multi-tasking support, nor memory protection (**watch out!**)
 - ▷ Uses 20 bits per address, chunks of 64kB of memory and notions of **near, far & huge** pointer, to manage memory demanding SW

- Protected mode

- Protected mode
 - ▷ From 80286, expands addressable physical memory to 16MB and virtual memory to 1GB
 - ▷ Provides protected memory, which prevents programs from corrupting one another (thus less sensitive to wrong memory accesses, crashes current exe, not everything)
 - ▷ All user programs are run in this mode for safety reasons

- Long mode

- Long mode
 - ▷ Overcomes limits of addressable space that now becomes 64 bit (x86-64); came outside Intel (AMD) that had to adopt the change

Registers 1/3

- Operands can be a memory location or a register
 - ▷ Memory location refers to central memory
 - ▷ Register – set of Flip-Flops, or a very fast SRAM memories close to arithmetic unit to allow fastest possible access to data
 - ▷ Limited number of registers is grouped into a **register file**
- In x86 we speak of **General Purpose Registers** (or GPRs)
- General mean that they can hold data or address

Bits	64	56	48	40	32	24	16	8
64					R*X			
32	Unused					E*X		
16	Unused						*X	
8	Unused						*H	*L

In the above * substitutes: A, B, C, D (ex. EAX, AH, RDX ...)

Nice trick to maintain compatibility of different CPU generations

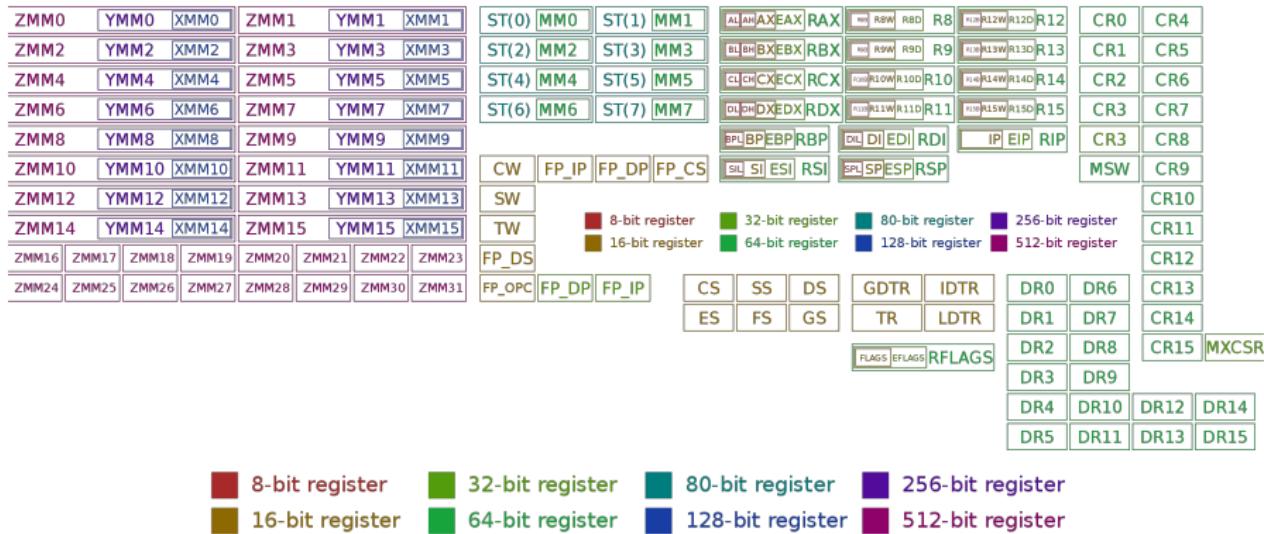
Registers 2/3

- A side GPRs we also have other registers: **segment, index, instruction**
 - ▷ **Segment Registers** – store addresses, typically organized in three distinct **segments** (important for assembly programming):
 - Program instructions
 - Variables (data)
 - Stack (local function variables & function parameters)
- **Instruction Pointer (EIP)** – contains the address of the next instruction to be executed; some instructions manipulate EIP (ex. branch in the program), but most of the time simple increment (next instruction in the program)
- **Flag register** – describe **current CPU state**; these are fundamental for the appropriate operation and are rarely used in high-level debugging, but are essential for low-level debugging (see next slide)
 - ▷ Initially only 16 bit wide (called FLAGS)
 - ▷ That became 32 bits (re-baptized EFLAGS)
 - ▷ Today 64-bit (re-baptized RFLAG)

Registers 3/3

- **Control Flags** – control CPU's operation: interrupt when arithmetic overflow is detected, defines operating mode (e.g. protected mode) etc.
- **Status flags** – indicate outcomes of arithmetic/logical ops
 - ▷ Carry flag (CF) – set when result of an **unsigned** arithmetic operation is **too large** to fit into destination
 - ▷ Overflow flag (OF) – set when the result of a **signed** arithmetic operation is **too large or too small** to fit into the destination
 - ▷ Sign flag (SF) – set when the result of an arithmetic or logical operation generates a **negative result**
 - ▷ Zero flag (ZF) – set when the result of an arithmetic or logical operation generates a **result of zero**
 - ▷ Auxiliary Carry flag (AC) – set when an arithmetic operation causes a carry from bit 3 to bit 4 in an 8-bit operand
 - ▷ Parity flag (PF) – set if the least-significant byte in the result contains an even number of 1 bits; otherwise, PF is clear; used for error checking of altered or corrupted data

Register file in x86 architectures



- Note extra regs GPRs: R8, R9, R10, R11, R12, R13, R14, R15
- We will discuss ZMM*, YMM*, XMM* later on
- Note register size(s) (color code legend)
 - ▷ How do you explain this?

Example and assembly program structure

- Simple piece of code:

```
1 inc count      ; Increment memory variable COUNT
2 mov total, 48  ; Transfer 48 to memory variable TOTAL
3
4 add ah, bh    ; Add content of register BH into AH register
5 and mask1, 128 ; Perform AND operation with a const
6
7 add marks, 10  ; Add 10 to the variable MARKS
8 mov al, 10     ; Transfer the value 10 to the AL register
```

- The above will not compile as such, the program is incomplete
- Typically any assembly program is structured in **segments** that define different memory blocks that will contain specific information (everything is in memory, remember?)
- In MASM we speak about **logical segments** containing different program components
- Typically we speak about **code, data, and stack** segments

MASM program structure

- Assembly program using simplified segment directives:

```
1 ; This is the structure of a main module simplified segment directives
2 .model small, c ; This statement is required before you
3 ... ; can use other simplified segment directives
4 .stack ; Use default 1-kilobyte stack
5 ...
6 .data ; Begin data segment
7 ... ; Place data declarations here
8 .code ; Begin code segment
9 ...
10 .startup ; Generate start-up code
11 ... ; Place instructions here
12 .exit ; Generate exit code
13 ...
14 end
```

- ▷ .model defines size of code & data pointers (have to chose one)
- ▷ C, BASIC, FORTRAN, PASCAL, SYSCALL, or STDCALL sets calling & naming conventions for procedures & public symbols
- ▷ Stack used for: a) temporary & local variables, b) pushing or popping registers c) storing return address for subroutine calls

Addressing modes (1/2)

- Register addressing – one of the operands is register

```
1 mov dx, var1 ; Register in first operand  
2 mov count, cx; Register in second operand  
3 mov eax, ebx ; Both the operands are in registers
```

- Immediate addressing – constant value or an expression

```
1 byte_value db 150; A byte value is defined  
2 word_value dw 300; A word value is defined  
3 add byte_value, 65; An immediate operand 65 added  
4 mov ax, 45h      ; Immediate const 45H to AX
```

- Direct memory addressing – operands specified in memory

```
1 add byte_value, dl ; Adds reg in memory location  
2 mov bx, word_value ; Operand from memory added to reg
```

Addressing modes (2/2)

- Direct offset addressing – arithmetic operators modify an address

```
1 byte_table db 14, 15, 22, 45      ; Tables of bytes
2 word_table dw 134, 345, 564, 123 ; Tables of words
3 mov cl, byte_table[2]           ; 3rd element of byte_table
4 mov cl, byte_table + 2         ; 3rd element of byte_table
5 mov cx, word_table[3]          ; 4th element of word_table
6 mov cx, word_table + 3         ; 4th element of word_table
```

- Indirect Memory Addressing – uses the arithmetic operators to modify an address explicitly

```
1 my_table times 10 dw 0 ; List: 10 words, 2Bytes each = 0
2 mov ebx, [my_table]      ; Effective add. of MY_TABLE in EBX
3 mov [ebx], 110           ; my_table[0] = 110
4 add ebx, 2                ; ebx = ebx +2
5 mov [ebx], 123           ; my_table[1] = 123
```

- In case we modify the address, what do we need to verify?

Control flow

- **Unconditional jumps** using `jmp` instruction transfer control unconditionally to another set of instructions
- Single argument contains the address of the target instruction

```
1 ; Handle one case label1:  
2 ...  
3 jmp continue  
4 ; Handle second case label2:  
5 ...  
6 jmp continue
```

- **Conditional jumps** – two-step process:
 - ▷ Test some condition that provides Boolean output
 - ▷ Then jump if the condition is true or continue if it is false

```
1 cmp ax, bx ; compare ax and bx  
2 jg next1    ; same as ( ax > bx ) goto next1  
3 jl next2    ; same as ( ax < bx ) goto next2
```

Loops 1/2

- Simplest form: use label to mark a place in the program, and then loop instructions with label reference
- When loop is executed, the content of cx register is automatically decremented at the end of each iteration
- Thus the number of iterations is controlled with the content of cx register: if 1 we continue to loop, if 0 the loop ends
- Code size within the loop is limited: no more than 128 bytes from label to loop instruction
- Example:

```
1 ; The LOOP instruction: for 200 to 0 do task
2
3 mov cx, 200      ; Set counter
4
5 next:
6     ...          ; Instructions here
7 loop next       ; Do again
8                 ; Continue after loop
```

Loops 2/2

- Loops while equal (or not equal) – check both `CX` and the state of the **zero flag** that will be set if the result of an operation is 0
- Many variants (check documentation):
 - ▷ `loopz` ends when either `CX=0` or **zero flag is clear**, whichever occurs first
 - ▷ `loopnz` ends when either `CX=0` or **zero flag is set**, whichever occurs first
- Interpret the code example below:

```
1 ; The LOOPNE instruction: While AX is not 'Y', do task
2 mov cx, 256      ; Set count too high to interfere
3                      ; But don't do more than 256 times
4 wend:             ; Some statements that change AX
5 ...
6 cmp al, 'Y'       ; Is it Y or too many times?
7 loopne wend      ; No? Repeat
```

Subprograms

- These are independent pieces of code, used (and re-used) anywhere in the program
- One could use labels & jumps inside the main program, but this is not practical since there is no stack management (you should remember this from C!); also this will result in spaghetti code
- In assembly subprograms use instructions `call` & `ret`

```
1 _start:                      ; tell linker entry point
2     mov ecx, '4'
3     sub ecx, '0'
4     mov edx, '5'
5     sub edx, '0'
6     call    sum               ; call sum procedure
7     mov [res], eax
8     ...
9 sum:
10    mov eax, ecx
11    add eax, edx
12    add eax, '0'
13    ret                     ; return once done
```

Final remark

No lecture can substitute personal efforts deployed to understand programming

(this is true in for any programming language)

So, pick your favorite editor, compiler and try it yourself

This is the only way to really learn, and if you learn C, many other programming languages and concepts will be far more accessible (so the effort spent is a well worth)

ELEC-H-473 – Microprocessor architecture

Th03: Basic processor architecture and the execution model

Dragomir Milojevic
Université libre de Bruxelles

2023

Today

1. Basic computer architecture concepts
2. Minimum system architecture
3. Instruction execution cycle
4. Different ISAs: RISC vs. CISC
5. Instruction execution cycle & circuit
6. Pipeline execution

1. Basic computer architecture concepts

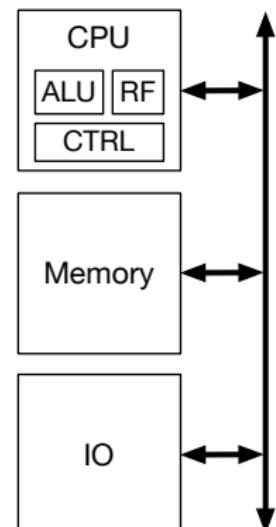
What really defines a computer?

- Definition of a computer made with respect to **electronic calculator**:
*"As opposed to electronic calculators, computers **store electronically the information that controls the computational process**"*
 - ▷ What is specific to electronic calculators?
- The above definition of computers is due to **John von Neumann**, mathematician
- He co-authored "*First Draft of a Report on the EDVAC*" in 1945, an unfinished & unpublished paper of 101 pages; there is some controversy about actual authorship ...
 - ▷ Concepts described in the above paper used to design ENIAC (USA) & Colossus (UK), machines considered to be the first computers



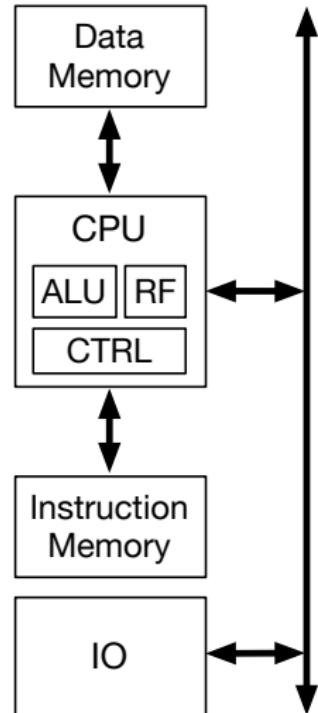
Von Neumann architecture: HW view

- First draft defines von Neumann architecture, a model of a computer structure (HW), term widely used today to describe different computer architectures (IoT, mobile & supercomputer)
- Central Processor Unit (CPU) – executes instructions defined in the program
 - ▷ Arithmetic-Logic Unit (ALU) + Control Unit (CTRL) + Register File (RF)
- Memory stores both instructions & data – a key concept that defines a computer, since instructions can change, just like normal data
 - ▷ What is the consequence of it?
- Input/Output (IO) – connects to outside world
- Communication – typically using buses that allow one pair of nodes to exchange data at a time



Harvard architecture

- Variant of Von Neumann architecture model in which Data & Instruction memories are **different** functional block instances!
- Motivation – instruction & data memories could be **different** circuits, even if implemented on the same PCB or within the same IC; then they can have:
 - ▷ different geometrical properties (capacity)
 - ▷ different electrical properties (R/W time)
 - ▷ but most importantly they allow **concurrent access, instruction & data access can overlap**
- Pure Harvard architectures with fully separated instruction & data memories are not mainstream; used very occasionally in dedicated DSPs & micro-controllers



Modified Harvard architecture

- Today computer architectures most often use **modified Harvard model**, a combination of pure Von Neumann & Harvard approach
- This is because memory is implemented using **cache hierarchies** to overcome the problem of CMOS SRAM/DRAM vs. logic scaling discrepancy (we will look deeper into this later on)
- Modified Harvard architectures take the advantage of von Neumann model, where instructions are treated as any other data:
 - ▷ everything is stored in the same memory, simple to control in HW
 - ▷ run-time compilation since same memory store data & instructions
 - ▷ it is even possible to write self-modifying code (exotic, but cool!)
- ... with the advantages of the Harvard architecture:
 - ▷ concurrent instruction/data access for more throughput
- In practice computers will have **separate** L1 for Instructions & Data, and for higher cache levels (L2, L3, etc.) and central memory they will use the same, often called **unified** memory

CPU

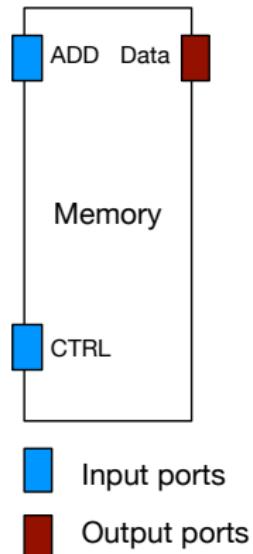
- ALU execute **limited number of arithmetic & logical operations** decided at CPU design-time, i.e. then can't be changed afterwards; they are hardwired in HW & IC manufacturing process
 - ▷ Unless you implement your CPU in FPGA circuit, and modify the micro-architecture on the fly – this exist, and is called **reconfigurable computing** that can even be dynamic (config generated at run-time)
- CPU may contain multiple ALUs that are composed of different sub-circuits specialized for specific arithmetic & logic operations and data types (typically integers, floating points)
- Depending on the operation ALU may require one or two (rarely more) **operands**, or arguments stored in memory or RF
- Access to operands through RF is preferred because of faster R/W access times to data; RF sometimes allow simultaneous access to few words in parallel (especially for READ)
- ALU writes back the result in RF, or memory

Main system memory

- Main (central) memory stores data & instructions in unified way
- Data & instructions are binary data, machine can't make a difference between the two by content: **is 0x32 opcode or data?**
- Thus, instructions are placed in a specific region of memory, often protected in access to prevent data corruption
 - ▷ Accidental modifications of program data will most likely cause the system to crash; **Can you explain why?**
- Memories are accessed through **ports** – collection of wires connected to pins, electrical interfaces of the IC to external world
- Ports are grouped based on functionality of the pins:
 - ▷ **Address port ADD** – location where we want to read/write
 - ▷ **Control port CTRL** – drives memory control logic
 - ▷ **Data port DATA** – actual data stored at address ADD
 - **Read** – data becomes available for other components in the system
 - **Write** – data is written in the memory

Practical memories

- **Address** – ADD [n-1:0] is input port with n pins, i.e. address bits → they provide 2^n addressable locations
- **Control** – CTRL [c-1:0] is also input port, but has only few pins (e.g. CLK, EN, R/W')
- **Data** – D [m-1:0] m pins that will act as input or output port depending on the operation direction (W and R respectively); sometimes R/W ports could be separated, this comes at cost of increased pin number at circuit level
- Typical memory “locations” at circuit level store 1 bit
- Actual memories are assembled from few memories connected in parallel, so that one address access multiple bits & allow **word-** or **bit-level parallelism** (often 1 byte, but could be more)



On memory performance

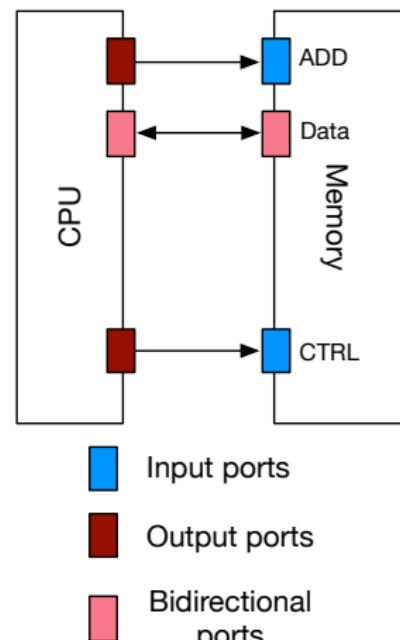
- Key memory parameters are **capacity** and **R/W access time**; both are intertwined: more capacity mean longer R/W access times and inversely, less capacity mean faster memories (important)
- Often memory performance is indicated using **number of transfers per second**, correlated with R/W access times of the memory, but also correlated to the speed of the interconnect between the logic and the memory (or R/W frequency F_{RW})
- Number of transfers per second multiplied by the transferred word width (so m) give us **memory bandwidth** $BW = F_{RW} \times m$
- We see that we can get more BW either by reducing the R/W access time or with wider words in a single access
- Improving R/W time of the memory and the interconnect became difficult to achieve over the past years; this is why today we see memories with very wide words (could be thousands of bits!)

On masters and slaves

- Masters can **initiate** transfer operations on one (one-to-one, the most frequent case) or more slaves (one-to-many)
- Slaves can only **accept**, or **refuse** access operations requested by a master, i.e. they can not initiate transfers themselves
- In our model CPU is the master & the memory is the slave
- CPU need some time to initiate & execute memory access; during memory access CPU is not used to compute something really useful (rather than computing, CPU is used to move data around)
- Today most **CPUs don't deal with memory accesses**, they outsource this task to other components in the system, they are just passing the information or what should go where to dedicated HW blocks that only move data around – **Direct Memory Access (DMA)**; this allows CPU to concentrate on what they should do: **useful computation!** (and not data transfers)

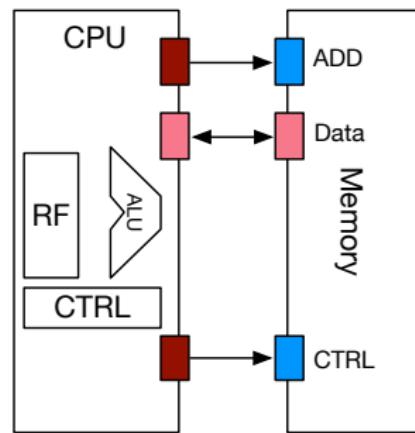
Putting it together

- On the memory side we have to read & write; Data port is typically **bi-directional** (on the CPU side ports are mirrored)
- Data port can be **input OR output** (exclusive), depending on the control logic (seen from CPU) – **single ported memory** – only one address/data couple is accessed at a time (1 clock cycle)
- CPU translates **R/W instructions** into appropriate HW operations
 - ▷ CPU initiates transfers by setting Ctrl signals & ADD to correct value
 - ▷ Memory accepts transfer (Ctrl)
 - ▷ HW decides on data direction depending on operation (R/W)



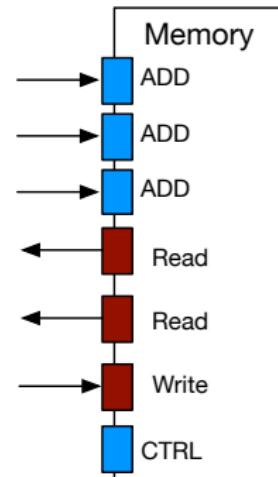
Closer look into the CPU: RF + CTRL

- ALUs implement adders, multipliers, dividers, hardwired mathematical operations (dedicated circuits) or microcoded complex operations (kind of a sub-programs for more complex operations for which dedicated HW would be too expensive)
- Register File is connected to ALU through a set of multiplexers circuits that will select the right paths
- Registers can be source (SRC) or destination (DST)
- ALU, RF & associated logic need control signals to steer their operation
- These will be generated by the CTRL unit, a sub-system that interprets instruction being executed and translating this into HW operation



Register Files (RF)

- RFs are memories built using SRAMs or set of FFs; simplest RFs are single ported, i.e. only one read OR write (or is exclusive) can be performed at a time & at one address location
- Typical RFs are **multi-ported memories**, i.e. they allow multiple data to be read or written in the same time; each port is then addressed with a separate address bus; this will impact memory pin count, and thus routability of the IC
 - ▷ Multi-ported RFs allow more BW to ALUs, assuming same memory/interconnect speed which is the case
- Practical RFs have few read ports & only **one** write port to avoid **data coherency problems**
 - ▷ In case of multiple write ports we need to handle eventual simultaneous writes to the same location → arbitration



On HW performance

Assume that CPU runs at certain frequency and can process N instructions per second, then:

- **Memory should deliver** necessary data & instructions to CPU, i.e. we need to R/W data from memory fast enough to feed computation
- **Wires between memory & CPU** should be fast enough to transport the above data with minimum **delay** and **latency**
 - ▷ Wires (e.g. Cu conductors) have **parasitic** resistance, capacitance & inductance (RCL) that at higher frequencies will **act as filters** defining the F_{max} at which the wires could still operate correctly
 - ▷ RLC depend on wire material, geometry & distance: shorter wires, smaller delay, better F_{max} ; this is why DRAMs are placed close to CPU
 - ▷ If delay is too big, FFs area inserted at expense of **increased transfer latency** – i.e., the N^0 of clock cycles to go from A to B
- CPU to Memory communication is designed to match the speed of memory and CPU interfaces, but today this is complex to achieve

If the above is not the case, the **CPU will be stalled**, i.e. the **computation will stop**, **CPU time wasted**, this is not good!

On SW performance

- Because of the HW reality (RLC), any SW execution time will be **bounded** either by:
 - ▷ **Computation** – there are many complex computations that run on small data sets; memory is not accessed often, the bottleneck is in computational part of the CPU (ALUs)
 - ▷ **Memory** – few simple computations on different data; memory is accessed often; CPU waits for memory to deliver data; bottleneck is in memory & memory-logic interconnect, not computation
- Good operating point for any SW is when computation, memory accesses and communication are in perfect **balance**; this is hard to achieve, but this is what programming art is all about; knowledge on HW could & will help to eventually reach this point
- Today CPUs compute fast thanks to high speed logic & parallelism (more on this later); major problem is in memory & memory-to-logic interconnect – compute systems face **memory wall** (performance & power limitations)

Computer architecture is not just a HW model

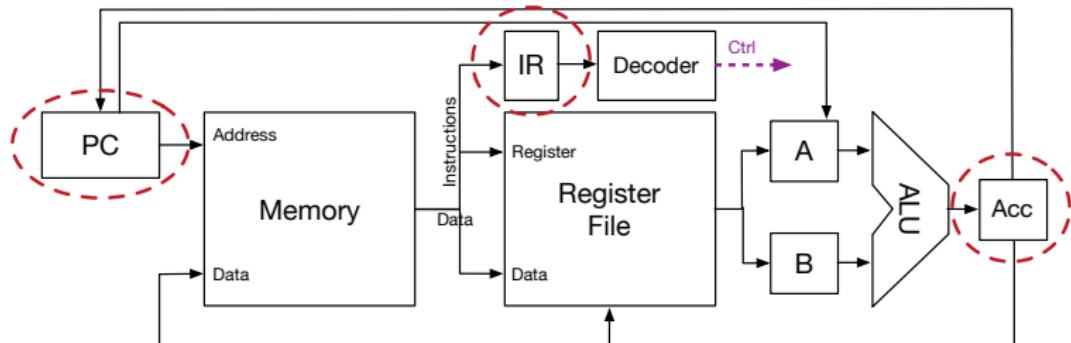
Descriptions of ALU, RF, CTRL logic & memory are not enough to make a computer → we still miss a few (key) ingredients!

- Architectural details necessary to make a real computer; next section will introduce minimal system architecture, a (very) simplified model of the actual computer architecture
- Instruction execution model, or how instructions in the context of a computer program are handled during execution (at run-time); this is also part of Von Neumann model
- Instruction Set Architecture (ISA) – actual specification of instructions, abstracted model of the computing system
 - ▷ ISA defines supported instruction, data types, local registers, central memory & how it is managed, general input/output; ISA defines **what** a system can do, not **how** does it do it; you could write an ISA simulator of a computing system using another architecture (ISA simulators or run-time binary translator such as Rosetta)

2. Minimum system architecture

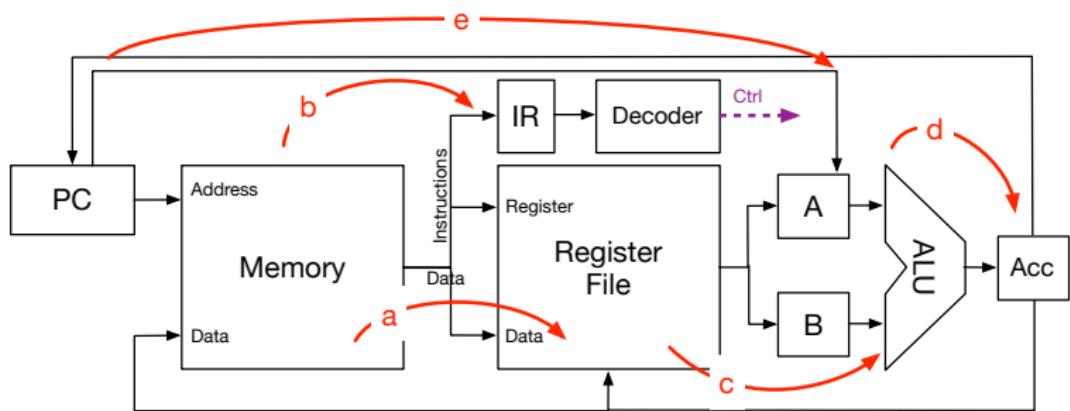
Extra components added to the previous model

- **Program counter (PC)** – register that stores the address of the next instruction to be executed; after instruction execution, PC is incremented to access next instruction in the program, or set to an arbitrary program address in case of a jump or branching
- **Instruction register (IR)** – stores the **opcode** of the instruction that will be executed; this opcode is decoded to generate control signals for other blocks in the system (we have seen simple decoder)
- **Accumulator (ACC)** – an optional register that can be R/W directly from ALU (or bypassed for a direct connection to the register file)

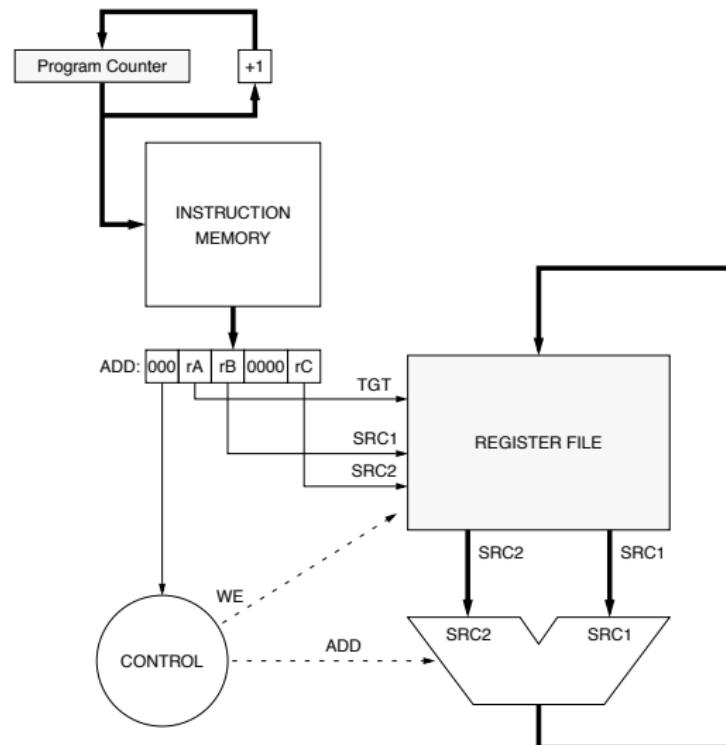


Different data exchange paths

- a) From central memory to the Register File (RF)
- b) From central memory to the Instruction Register (IR)
- c) ALU can compute from RF (or from the central memory)
- d) or it could use the result from previous operation (ACC)
- e) PC can be updated using a computation from ALU (addresses are used as any other data, just think of pointers in "C" language)



RISC16 used in labs ... or x86



Not that much different!

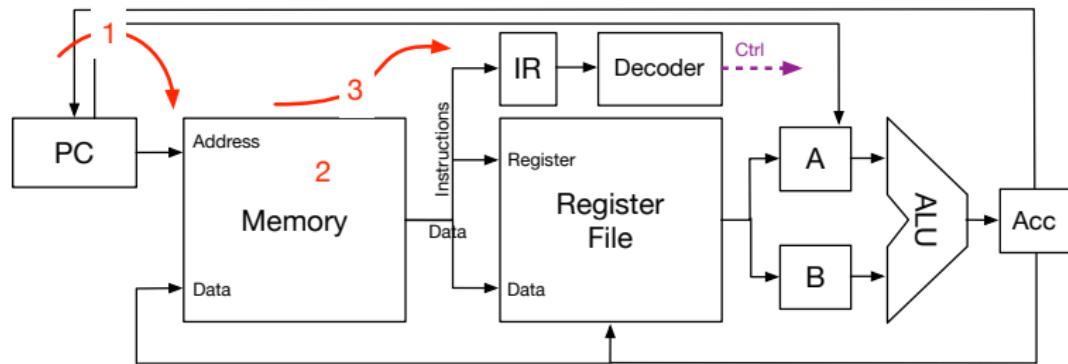
3. Instruction execution cycle

Instruction execution model

- Program is a list of instructions read and executed one by one in a **sequential** manner, and in the order specified by the program
- Von Neumann specification defines **instruction execution model**:
 - ▷ **Fetch instruction** – Current instruction is fetched from the memory, i.e. the instruction opcode is transported from the Instruction Memory to the Instruction Register (IR)
 - ▷ **Decode instruction** – Internal logic interprets the instruction opcode, i.e., it generates the appropriate control signals for all associated logic that will be involved in the execution of this operation
 - ▷ **Fetch operands** – Memory read for data; necessary only if data is not in the RF (for simplicity we merge this operation with next step)
 - ▷ **Execute instruction** – Perform actual operation on operand(s)
 - ▷ **Write result** – memory or RF, depending on instruction
- Every instruction executes in 4 steps: **F, D, Ex, W** in simplified model & notation (fetch operand & execute steps are merged)

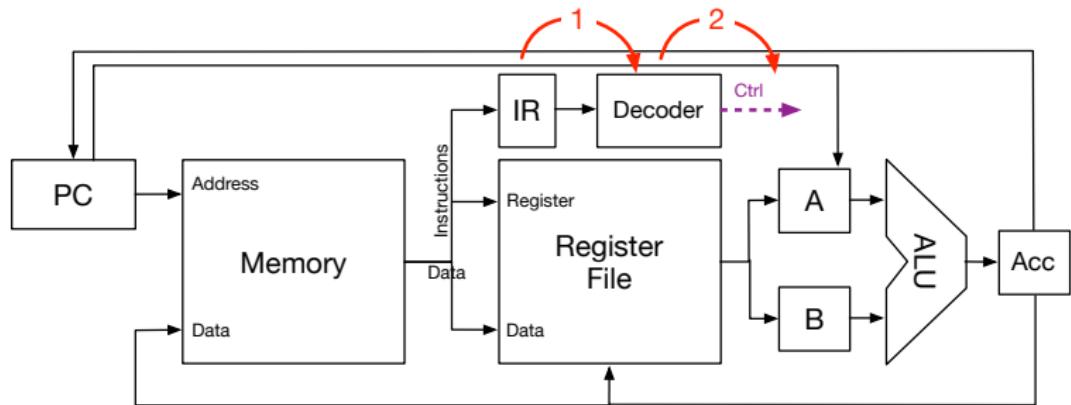
A. Instruction fetch

1. Control logic (not shown on the figure) places the value of the PC on the address bus of the system
2. Control signal will generate read memory access; data read is placed on the **data bus** – this is the opcode of the instruction stored on a given address
3. IR knows that the data on the bus is to be taken thanks to the control logic, it reads the opcode from the bus and store it for further processing in the local register



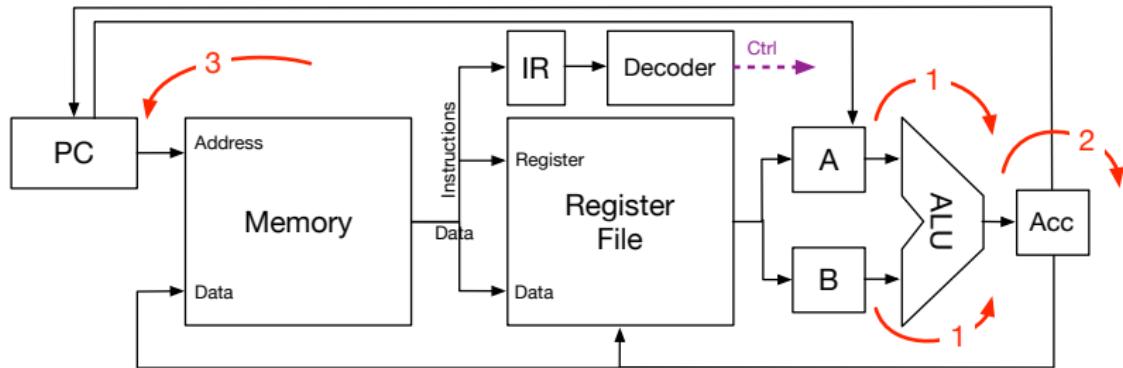
B. Instruction decode

1. Control logic reads the content of IR & passes it to **decoder circuit**
2. Depending on the opcode, decoder will generate different control signals for all other units in the system (e.g. ALU operation selection & operands preparation)
 - ▷ In the simplest form this is just a $x : y$ decoder (**What are x, y ?**)
 - ▷ Decoders are simple **combinatorial circuit**
 - ▷ We have seen the example in Th01



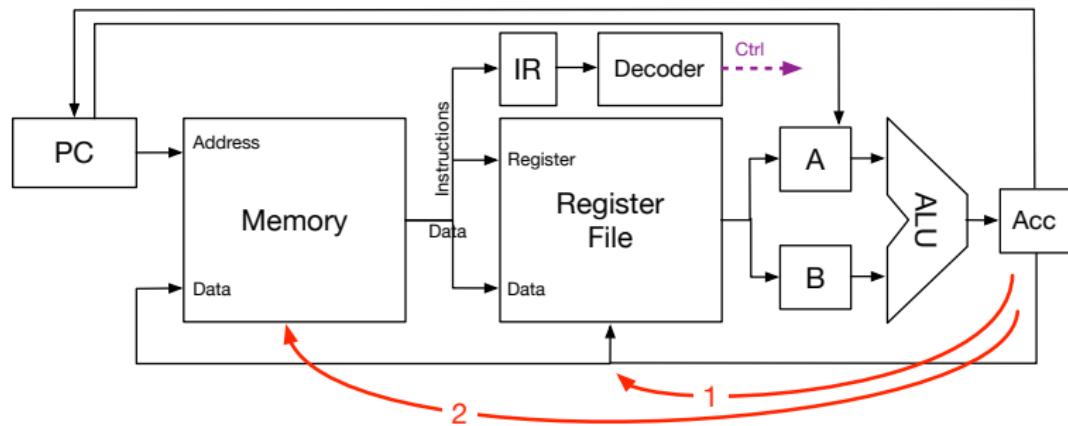
C. Instruction Execute

1. ALU can now execute the selected operation (logical AND, arithmetical + etc.) on the operand(s)
2. The result is written in the destination register (or accumulator)
3. PC is updated according to the executed instruction
 - ▷ If arithmetic/logical operation on data we will have $PC+1$; instruction flow control instructions could set PC to any value



D. Write

- Depending on the architecture and the *instruction* used the result is written to the appropriate destination: that can be either RF (1) or memory (2)
- It is the instruction opcode that specifies result destination
- Note that we said that RF is faster than memory ... you should understand trade-off between two write destinations



Different architectures depending on memory R/W

Depending on the possible combination(s) of the source/destination operands and/or result **location** (RF, memory, ...), we can have following computer architectures:

- **Pure Load/Store architectures** – operands must be in the RF, so any computation is always preceded with an explicit R/W operation from/to memory to/from RF
- **Register/Memory architecture** – operands can be in either in RF or memory (and this is an exclusive or, both RF and memory in one instruction is not possible)
- **Register + Memory** – any operation can have operands being in memory and/or RF (this is the general case)

According to you, what these differences really mean,
and what are the trade-offs?

4. Different ISAs: RISC vs. CISC

Instruction Set Architecture – ISA

- Until now we have defined (minimal) computer **micro-architecture** and the way it operated (instruction execution), but we didn't say anything about **WHAT** this machine can really do!
- **ISA** is the (missing) link between the micro-architecture and the programmer that writes programs in assembly or any other high-level language to be compiled into executable file ...
- **ISA is about definition of WHAT machine can do, as opposed to HOW this is done**
 - ▷ After all programmer doesn't need to know how the addition is done
- ISA typically defines:
 - ▷ Native data types (integer and floating point sizes)
 - ▷ Register number (names), sizes & types, type of instructions
 - ▷ Addressing modes, memory architecture, interrupts, exception handling and external I/O
- Computer is defined as: **micro-architecture + ISA**

CPU instructions classes

- **Data movement** – from any to any point including:
 - ▷ Load (from memory) or Store (to memory)
 - ▷ Move memory-to-memory or register-to-register
 - ▷ Move from memory to I/O devices (printer, screen, mouse etc.)
- **Arithmetic ops** – depend on data type due to encoding differences
 - ▷ Integer operations
 - ▷ Floating point operations
 - ▷ How do we encode floating point / integer?
- **Shifts** – at word level (e.g. sleft, sright)
- **Logical operations** – e.g. and, not, set, clear
- **Control instructions** – jump/branch conditional or not
- **Subroutine handling** – call, return
- **Interrupt handling**

Addressing modes

Not all modes shown below are implemented & syntax may vary

Register	add R4,R3	$R4 \leftarrow R4 + R3$
Immediate	add R4,#3	$R4 \leftarrow R4 + 3$
Register Indirect	add R4,(R1)	$R4 \leftarrow R4 + \text{Mem}[R1]$
Displacement	add R4,100(R1)	$R4 \leftarrow R4 + \text{Mem}[100+R1]$
Indexed/Base	add R4,(R1+R2)	$R4 \leftarrow R4 + \text{Mem}[R1+R2]$
Direct/ Absolute	add R4,(1001)	$R4 \leftarrow R4 + \text{Mem}[1001]$
Memory indirect	add R4,@(R3)	$R4 \leftarrow R4 + \text{Mem}[\text{Mem}[R3]]$
Auto-increment	add R4,(R1)+	$R4 \leftarrow R4 + \text{Mem}[R1]$ $R1 \leftarrow R1 + c$
Auto-decrement	add R4,-(R1)	$R4 \leftarrow R1 - c$ $R1 \leftarrow R4 + \text{Mem}[R1]$
Scaled	add R4,100(R2)(R3)	$R4 \leftarrow R4 + \text{Mem}[100+R2+R3*c]$

\leftarrow used to note the assignment of the result

Classification of ISA

- Depending on the way how we implement instruction set, we can have different flavors of ISA (not that many variants though)
- In computing, most of the time computer executes **same operations again & again**: memory move, some basic logic/arithmetic operation etc.
- Table shows instructions execution frequency: first 3 account for 48%, while only 10 instructions represent 96% of all executed instructions in a program

Rank	Instruction	Average[%]
1	load	22
2	condition. branch	20
3	compare	16
4	store	12
5	add	8
6	and	6
7	sub	5
8	more Reg-To-Reg	4
9	call	1
10	return	1
	Total	96

Classification of Instruction Sets

- Conclusion → **only few instructions are used most of the time**; this is the main motivation for two orthogonal approaches to ISA:
 - ▷ Reduced Instruction Set Computer – RISC
 - ▷ Complex Instruction Set Computer – CISC
- Both HW & SW communities have been fighting for decades to prove that one approach is better than the other
- In reality both are cool, and have their advantages and disadvantages, though following observations can be made:
 - ▷ CISC are more complex architectures, require more silicon and aim general purpose CPUs in High Performance Computing (HPC), so servers, supercomputers; adopted by companies that make their own silicon; **Can you guess why?**
 - ▷ RISC tend to be simpler, used for general purpose servers and of course mobile computing that flourished over the past decade

RISC philosophy

- Instructions are simplified in content and therefore can be implemented more efficiently in HW: most RISC instructions will require only one clock cycle to execute
- More complex instructions will be implemented as “subroutines”, called on per need basis
- RISC computers are often **load/store architecture**: if operation is to be done on memory, it will be split into separate load, execute and store instructions at assembly level
- Efficient instructions will have higher throughput and the system as a whole would be more performant
- In other words if 99% of instructions to be executed are faster, we can forget about the other 1%; instruction set appear to be **SMALL**, with instructions **HIGHLY** optimized in HW

CISC philosophy

- As opposed to RISC, CISC CPUs have many instructions & some of them can be very complex
- In CISC you could possibly load from memory, compute and store in a single instruction; or you could perform some dedicated math operation using specialized HW unit (CPU vendors sell more IC area, so better profit, and we get better systems)
- Obviously any operation directly from/to memory will be much slower than the same operation in the RF because we still need to perform the memory transfer; question is who is doing this: computer HW (CISC) or SW (RISC)?
- Rich instruction sets with many different instructions can not be implemented with fixed cost in terms of computation cycles
- This will have a strong impact on performance, as we will see later with actual architectures!

RISC/CISC example

```
1 load  ax,a  
2 load  bx,b  
3 mul   ax,bx  
4 store a,ax
```

```
1 ; here we use a  
2 ; single instruction  
3 ; that does everything  
4 mul a,b
```

- Emphasis is on **software**
- Single-clock, reduced instruction only
- Register to register: LOAD and STORE are independent instructions
- Low cycles per second, large code sizes
- More RF storage resources

- Emphasis is on **hardware**
- Includes multi-clock complex instructions
- Memory-to-memory: LOAD and STORE incorporated in instructions
- Small code sizes, high cycles per second
- More computational hardware

RISC/CISC performance trade-off

- Software execution time t_{CPU} is given by:

$$t_{CPU} = N_{Inst} \times CPI \times \frac{1}{F_{Clk}}$$

where N_{Inst} is the number of instructions in the program, CPI is the number of cycles required to execute one instruction (on average) and F_{Clk} CPU clock frequency

- Having the above in mind:
 - ▷ **RISC** → increase N_{Inst} , but reduce CPI and F_{Clk} since optimized functions are more complex in silicon
 - ▷ **CISC** → do the opposite: they decrease N_{Inst} , but increase F_{Clk} , and CPI could be variable for different instructions!
- Which one reduces t_{CPU} will be algorithm, program, compiler & architecture dependent, so no quick judgments here

RISC in academia

RISC was and still is very popular in academia, few examples:

- **Berkley RISC – SPARC**: Scalable Processor Architecture
 - ▷ A lot of registers for local computation
 - ▷ Low-latency instructions (typically 1 cycle)
 - ▷ Still in action ... but less and less, we will see a concrete example
- **Stanford – MIPS**: Microprocessor without Interlocked Pipelined Stages (not the same as Million Instructions per Second)
 - ▷ Load/store architecture (works on registers only)
 - ▷ Found in game consoles (Nintendo 64, Sony PlayStation, etc.)
 - ▷ About to become open-source
- **RISC-V** – originally from Berkley, but with contributions from all around; highly configurable, fast, low-power & silicon proven
- **RISC16** – simple simulator for learning (used during labs)

Commercial RISC/CISC

- During '90 RISC were mostly associated with mobile applications:
ARM & ARM derived CPUs found in tablets & smartphones
- But not only → RISC based HPC machines:
 - ▷ Sun Microsystems (acquired by Oracle) SPARC line of CPUs from T1 to T5 (2013) targets servers 1CPU = 16 & 32 cores
 - ▷ Fujitsu K-computer – super-computer (2011) based on SPARC cores (10 Penta-flops, 80k 8-core processors @ 2GHz)
 - ▷ Fujitsu A64FX – fastest super-computer in 2020 uses 48 ARM-V8 cores per CPU in 7nm CMOS

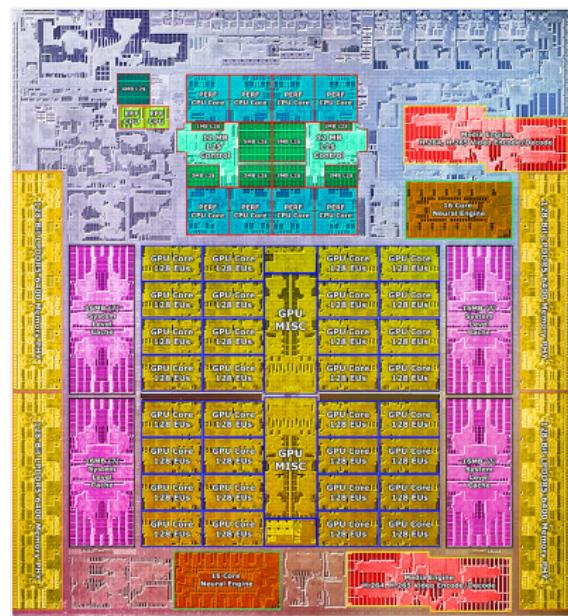


- Intel (obviously), AMD and IBM

Today

- RISC/CISC hardly relevant; tendency to abandon big, fat cores in favor of more **heterogeneous** architectures; this is true even in HPC/server communities; there are many low-power ARM based servers on the market

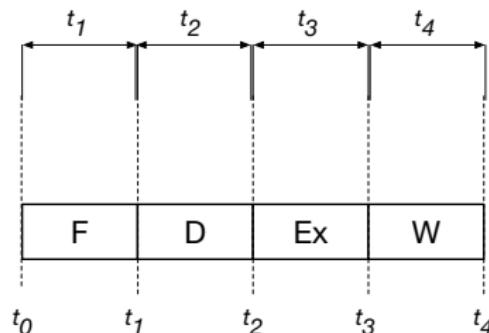
- Apple abandoned Intel → M1Max
- Highly optimized CPUs made using state of the art 5nm CMOS (57 billion transistors)
- 8 high-performance, 2 low-power cores; 32 GPUs; 2×16 cores neural engine; dedicated video encoder, decoders engines
- Loads of local memory



5. Instruction execution cycle & circuit

Instruction execution model

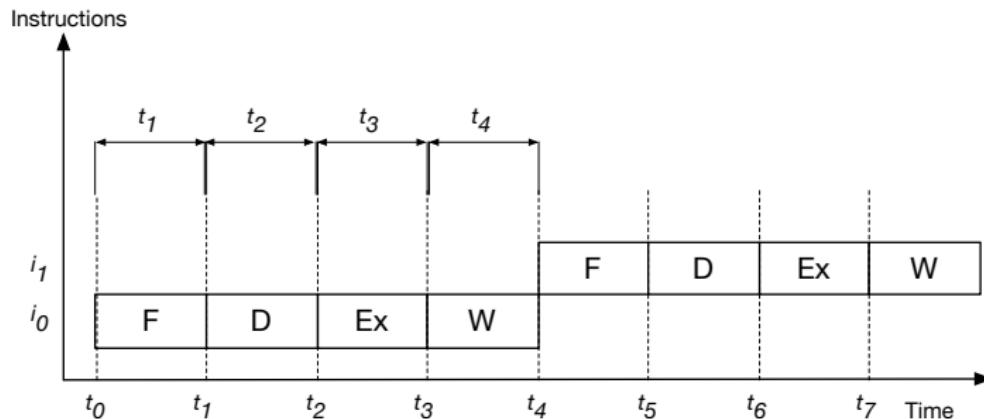
- Defines a sequence of steps that need to be executed **one after the other & in a given order** (we can't execute before decode)
- So the execution of F, D, Ex, W of a single instruction assuming different execution times per stage t_i :



- Valid question from HW perspective: how to execute these 4 steps **for subsequent instructions**? → different ways of doing this will result in *different computer architectures*

Single Issue Base Machine – SIBM (1/2)

- Simple approach or KISS – *Keep It Simple & Stupid*: next instruction starts to execute only after the previous has finished
- Proposed execution model refers to Single Issue Base Machine (SIBM); “single” → only 1 instruction can be issued & executed at a time (“single” will become “multiple” in next lectures)
- In the example below instruction i_1 will start execution only at t_4



Single Issue Base Machine – SIBM (2/2)

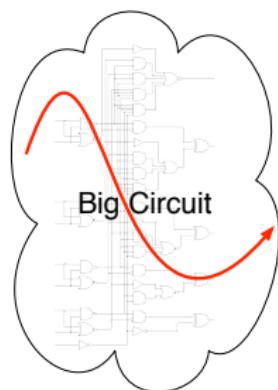
- Let's assume for now that each of 4 steps will take exactly the **same amount of time** t to execute:
 - ▷ While the above assumption is not necessarily easy to achieve in the real world, CPU designers will try very hard to come close to it! We will see why this is SO important in computer architectures
- So, time to execute any instruction will take $4 \times t$; a program with N instructions will take $N \times 4 \times t$ to execute; instructions can be issued and completed with a frequency of $F_{Clk} = \frac{1}{4 \times t}$
- In practical systems, such as CPUs implemented using CMOS ICs, F_{Clk} of SIBM will be very low
- Thus SIBMs are **not good at all** from performance perspective: we need to complete F, D, Ex, W for each instruction before issuing the next instruction for execution
- No practical computer will work like this, **performance would be way too low** → we need to (seriously!) improve things

CPUs and ICs

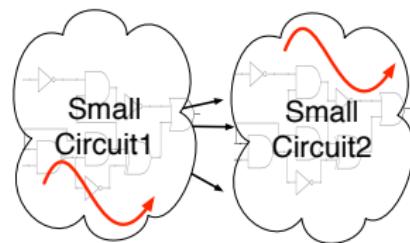
- Before going any further: do we agree on the following?
 - ▷ CPU is a **logic circuit** (and sequential one with FFs)
 - ▷ Any logic circuit has a **critical path** – path that exhibits biggest delay – sum of all gate and interconnect delays
 - ▷ Performance of the whole circuit (so performance of the CPU) is limited to the performance of the critical path
- If this is clear, then we should also agree on the following:
 - ▷ Bigger the circuit, better the chances are that critical path is longer, so larger t & lower F_{Cik} ; and inversely: smaller the circuit, smaller the critical path should be (so, smaller t & higher F_{Cik})
- We can then roughly conclude: **smaller circuits are FASTER** and **bigger circuits are SLOWER**; thus **making a CPU that will execute all 4 instruction steps as a “single” circuit doesn’t make sense as in SIBM**: it is better to divide CPU in multiple & smaller logic sub-circuits

Reducing the critical path delay 1/3

- Idea of the **link** between the circuit size & the critical path delay relation is used when designing any circuit: we can split one big circuit into 2 (or more) smaller sub-circuits
- Let's split a circuit so that 1/2 of the critical path is in one sub-circuit (SmallCircuit1), and the other 1/2 in the other sub-circuit (SmallCircuit2):



One big critical path

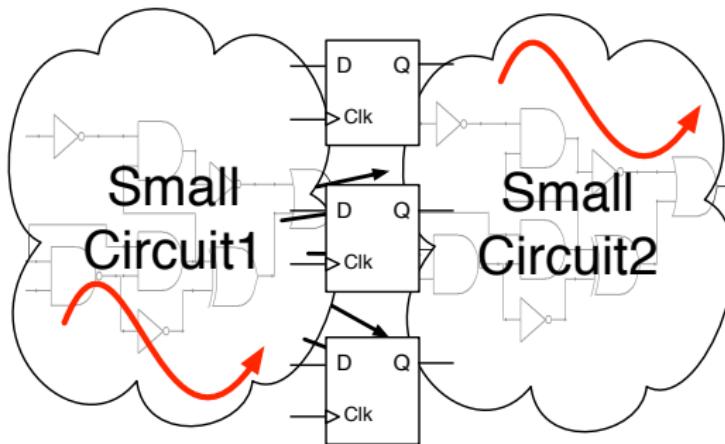


Two smaller critical paths

- But this doesn't change things, sum of delays remains the same ...

Reducing the critical path delay 2/3

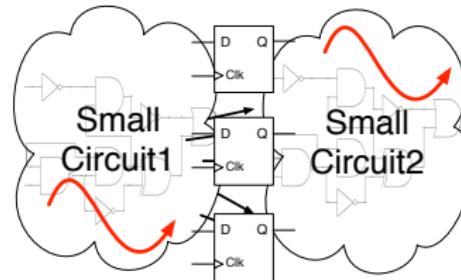
- To really gain for each wire connecting 2 sub-circuits we **insert a Flip-Flop (FF) – synchronous mem.** driven by raising **Clk** edge
- FFs will store the result of the operation of the SmallCircuit1:



- FF insertion makes two circuits **independent**, now they can operate in parallel; critical path is really cut in 2

Reducing the critical path delay 3/3

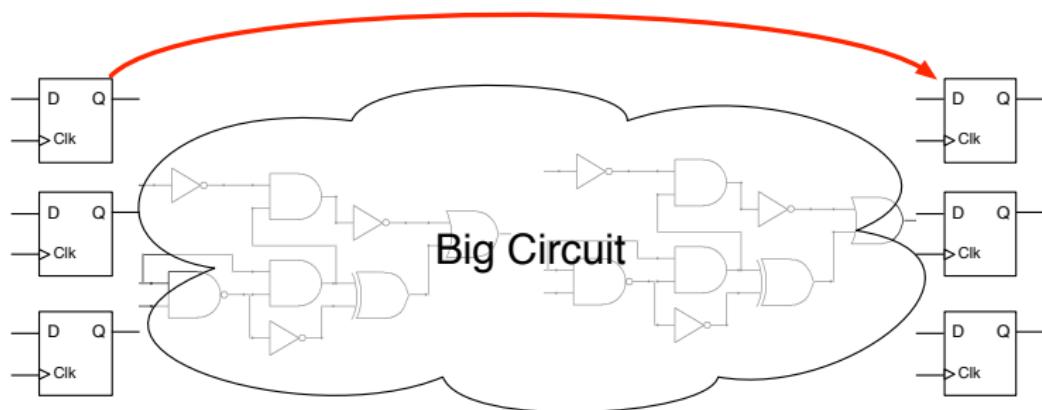
- This is possible because FFs will update their content only after certain amount of time (next Clk rising edge); “time distance” between the two circuits is 1 clock cycle
- In t_1 , SmallCircuit1 calculates a result from input data d_1 & SmallCircuit2 is idle) in t_2 SmallCircuit2 receives as input the result of SmallCircuit1 computation from FFs
- Since result of SmallCircuit1 computation on d_1 is stored in FFs, we can start computing d_2 with SmallCircuit1



Above circuit, compared to one big circuit is said to be pipelined with one stage

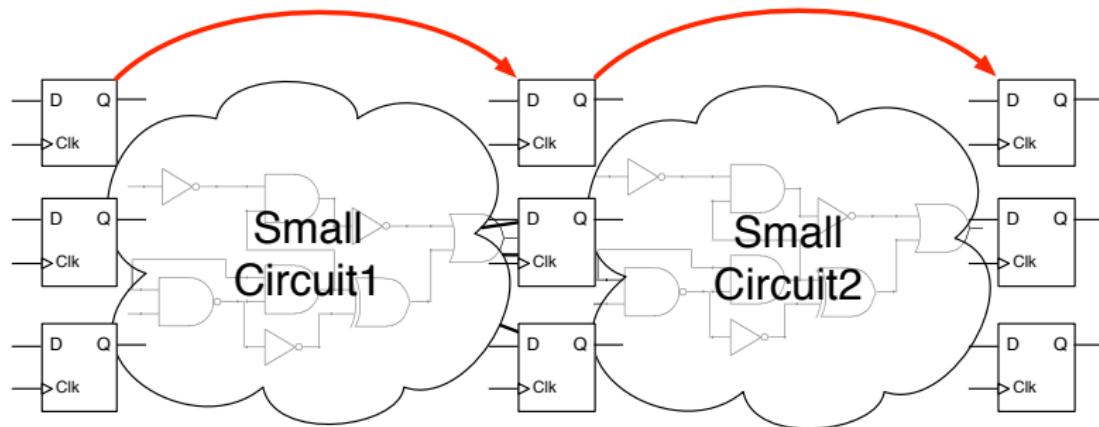
Logic circuits are sequential, CPUs too (1/2)

- They already have FFs to avoid race conditions since they use state machines (**I suppose you understand this?**); and do use FFs from Inputs to Outputs (RTL paradigm we saw in logic circuits)
- In other words they already operate from Register-to-Register
- In one big circuit (Single Issue Base Machine), result is **one cycle away**, with long cycle period because of circuit complexity:



Typical logic circuits are sequential, CPU too (2/2)

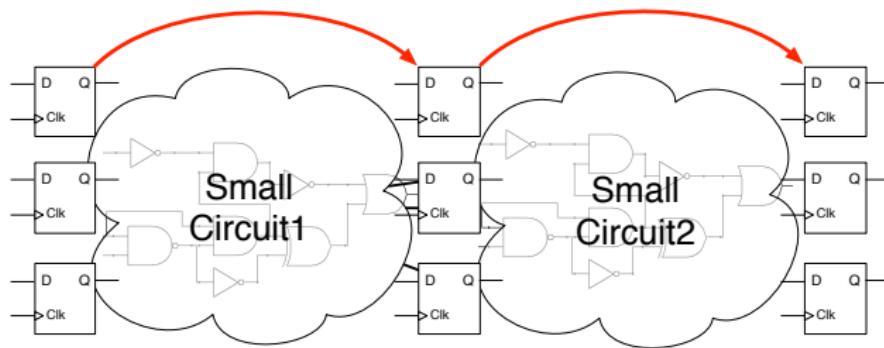
- In pipelined circuit, result is **two cycles away**, because we broke the critical path in two and inserted FFs between two sub-circuits:



- Functionally, "**the big**" & "**the broken in two circuits**" are identical ... if we allow enough execution cycles
- Difference is in the **timing references** of intermediate results

Pipeline stage insertion: consequence

- When inserting a pipeline stage in a circuit we hopefully break the critical path equally to reduce the delay in half for each sub-circuit
- This means F_{Clk} increases by a factor of 2
- But we do insert one clock cycle delay: **circuit latency**



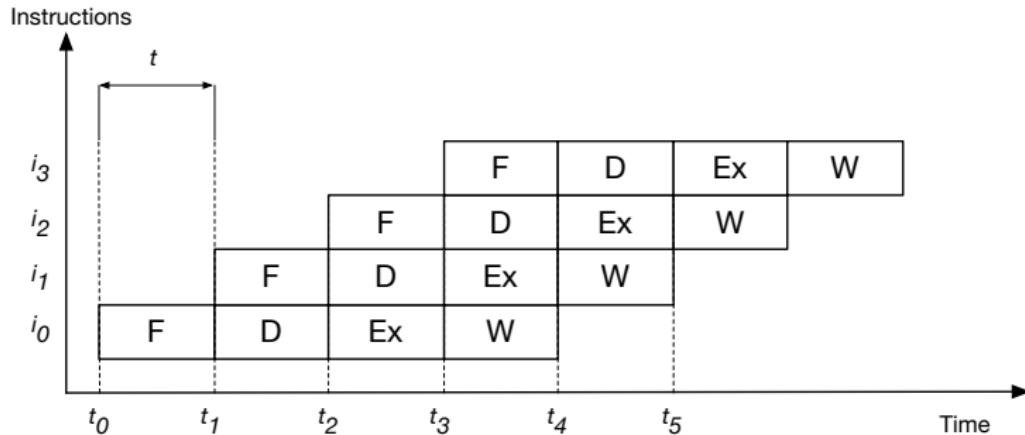
Pipeline circuits tend to increase the operating frequency, but introduce delay expressed in extra clock cycles

6. Pipeline execution

Pipelining and CPU instruction execution model

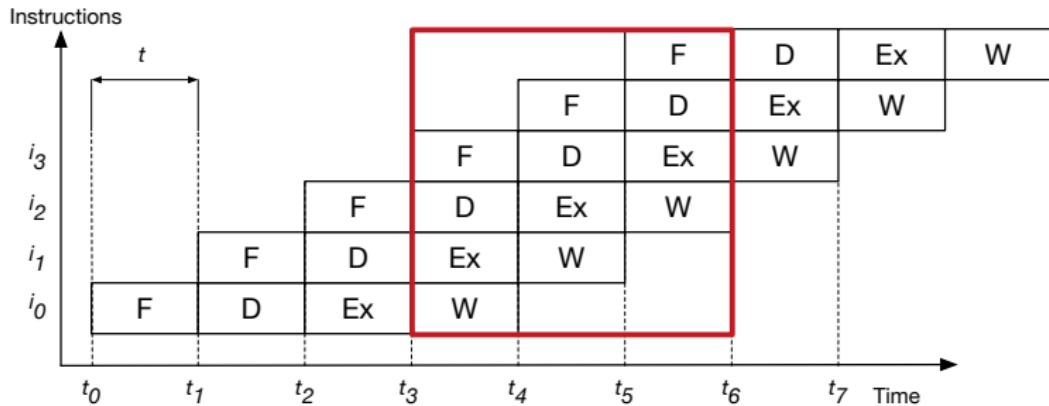
- Each execution model step (so F, D, Ex, W) could be implemented as a **distinct logic sub-circuit**; all sub-circuits are linked between them with FFs that isolate computation of different stages & pass results from one **pipeline** stage to another
- Because F, D, Ex, W are sub-circuits, they need less logic gates and wires; and if they are smaller they could be faster: this is because critical path can be “shorter” in smaller circuits because we have less gates & interconnect
- First instruction execution is therefore delayed 4 cycles, one clock cycle delay per pipeline stage; when instruction enters the **instruction pipeline**, the result will pop-up at output 4 cycles later
- We define instruction **latency** as number of clock cycles elapsed between instruction fetch (F), and operand write (W)
- **We want to minimize individual instruction latency of all instructions to maximize program performance**

Pipeline – timing diagram 1/2



- In t_0 – we fetch first instruction i_0
- In t_1 – i_0 fetch done, i_1 can be fetched AND (&) i_0 decoded
- In t_2 – we fetch i_2 , decode i_1 & execute i_0
- In t_3 – we fetch i_3 , decode i_2 , execute i_1 & i_0 write
- and so on, the pipeline is said to be full

Pipeline – timing diagram 2/2



- After t_3 , on every clock cycle, 1 new instruction is: **fetched, decoded, executed and written (terminated)**
- For the external observer, once the pipeline is full, every instruction will take only one cycle to execute
- Latency of the instruction remains however 4 cycles, so the total “time” instruction takes to execute by the CPU remains the same

Computing pipeline acceleration

- For N instructions and n pipeline stages, the total cycle count is:
 - ▷ for a Single Issue Base machine:

$$C_{sibm} = N \times n$$

- ▷ for a n-stage pipeline CPU:

$$C_{pipe} = n + N - 1$$

- The acceleration is then:

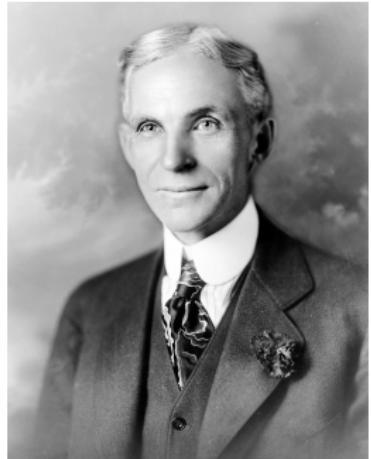
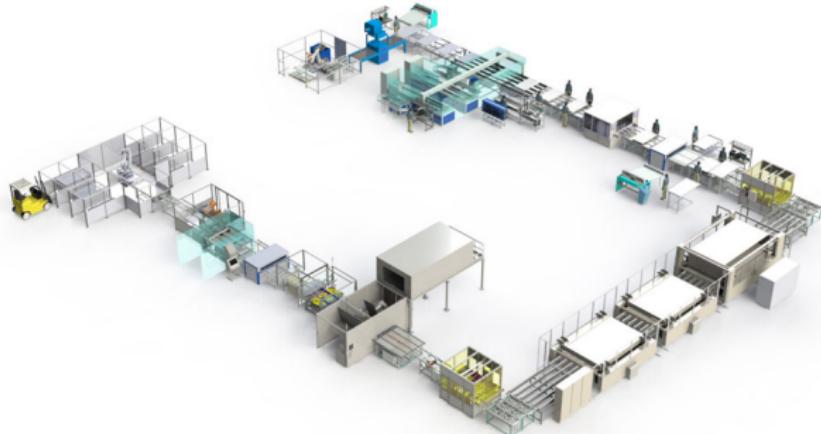
$$Acc = \frac{C_{pipe}}{C_{sibm}}$$

- If $N \gg n$ (N is big compared to n , true for big programs):

$$Acc = \lim_{N \rightarrow \infty} \frac{C_{pipe}}{C_{sibm}} = \lim_{N \rightarrow \infty} \frac{n + N - 1}{N \times n} = \frac{1}{n}$$

Acceleration is proportional to the number of pipeline stages!

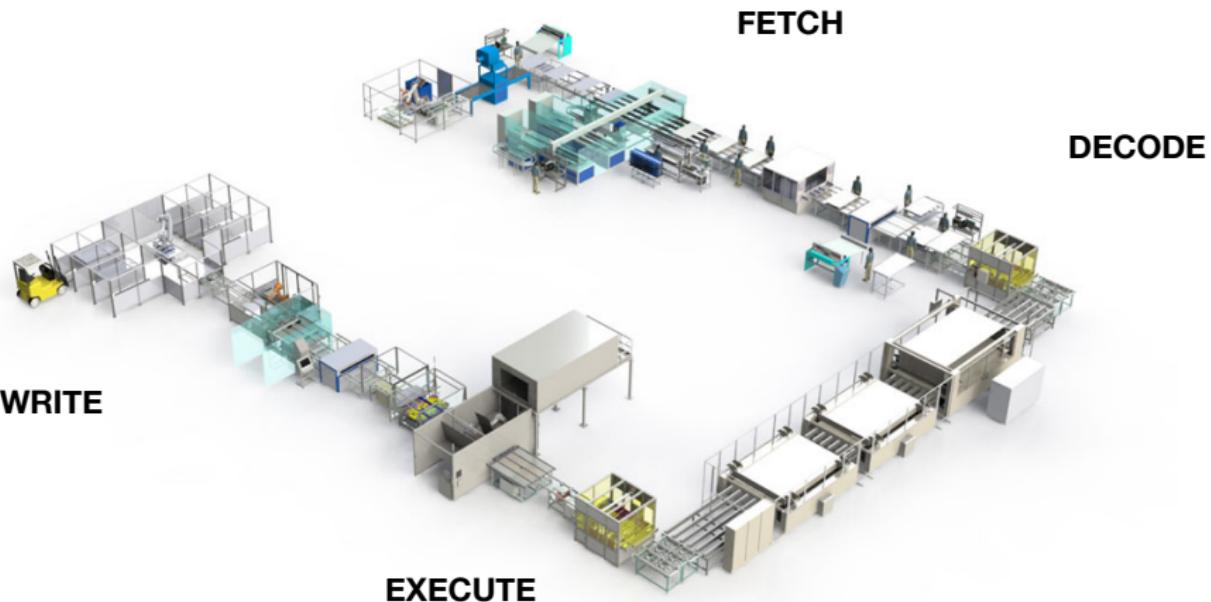
100 years old concept, due to Henry Ford!



- (1) Place the tools and the men in the sequence of the operation so that each component part shall travel the least possible distance while in the process of finishing.
- (2) Use work slides or some other form of carrier so that when a workman completes his operation, he drops the part always in the same place—which place must always be the most convenient place to his hand—and if possible have gravity carry the part to the next workman for his own.
- (3) Use sliding assembling lines by which the parts to be assembled are delivered at convenient distances.



Analogy with CPU



Explain the concept of latency in this picture

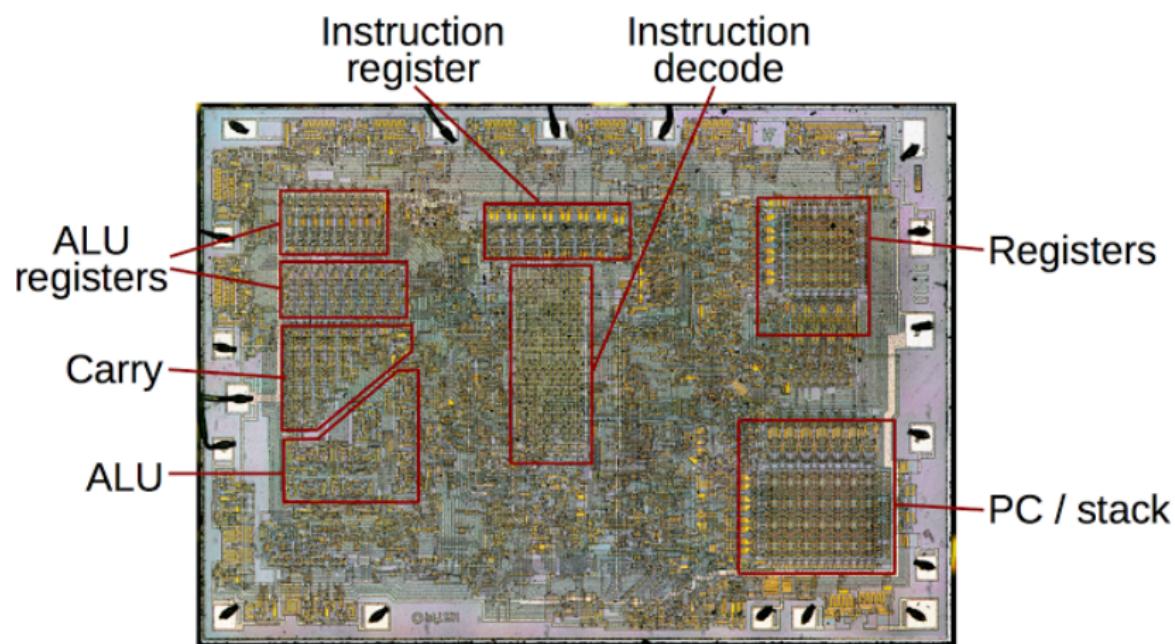
Pipeline & Super-pipeline

- If we speed-up things with deeper pipelines, should we insert infinite number of pipeline stages?

NEVER!
- Each extra pipeline stage adds extra **latency cycle**, infinite number of pipe-line stages would mean **infinite latency**!
- Acceleration factor of n is possible only if we execute a **huge number of instructions** taking **the same amount of time**
- Thus RISC ISAs could use pipeline better than other ISAs
- In practice number of stages is a trade-off between critical path depth (minimum period, so maximum frequency) and latency
- Some CPUs are **super-pipelined** – we can have few dozens of pipeline stages (not anymore these days)
 - ▷ Past Intel X86 CPUs have been heavily pipelined with >30 stages, good for marketing since F goes up, but latency too; these days <15

Core organization at IC-level – layout of 8008 CPU

Note the placement of different functional blocks (RF close to ALU, instruction register close to instruction decoding stage etc.)



Pipeline & the execution model 1/2

- Pipeline executions introduces something special → **parallelism**, i.e., computation are happening at the same time!
- Pipelining – one of few techniques said to exploit what is called **Instruction Level Parallelism (ILP)**; ILP means that we execute some things at the same time, true even in a single core, single ALU machine!
- In case of a pipeline, thanks to pipelined logic circuit operation, ILP is possible because we have overlapped execution of different stages of different instructions
- To measure the efficiency of pipeline we use **Instructions Per Cycle – IPC** (different from *CPI* that we already mentioned)
- For a system with single ALU, the *IPC* of 1 means that ILP is used at it best! In practice *IPC* is rarely at this value; in the next lecture we will see what cause $IPC < 1$

Pipeline & the execution model 2/2

- Pipeline execution is **handled automatically** by the HW of the CPU, you do not see this happening!
- But don't expect miracles, you can write the code that will be gentle to the execution pipeline ... as you could write code that will kill the pipeline execution! and the performance of your program will be poor
- There is a clear link between HW & SW: think of hypothesis we've made to calculate the acceleration of pipelined architecture!
- It is in general a good idea to check how well the pipeline is used during SW execution, especially in loops with many iterations
 - ▷ Why loops with many iterations?
- Because pipeline execution is hidden from us, we need specific tools (CPU simulators) that will help us to understand how well pipeline is used
 - ▷ Example: VTune profiling software for Intel processors

Things to take

- Even single CPU computing systems, with 1 core & 1 ALU are executing things in **parallel** because of:
 - ▷ Bit-level parallelism at word, operand level – e.g. adder circuit will add multiple bits of a word in a single clock cycle
 - ▷ Pipelined HW for F, D, Ex, W operations that enable ILP
- Pipelined execution of instructions:
 - ▷ Enables higher instruction throughput, with computation acceleration proportional to the number of pipeline stages; but to take most of it we need nicely filled pipeline stages
 - ▷ More pipeline stages enable higher operating F_{Clock} , since smaller critical paths linked to logic circuits in HW
 - ▷ But more stages increase instruction latency
- Computing systems today are most of the time using modified Harvard architecture, a blend of CISC/RISC ISA, with more or less pipeline stages, depending on the application they target

ELEC-H-473 – Microprocessor architecture

Th04: Super-scalar architectures and execution hazards

Dragomir Milojevic
Université libre de Bruxelles

2023

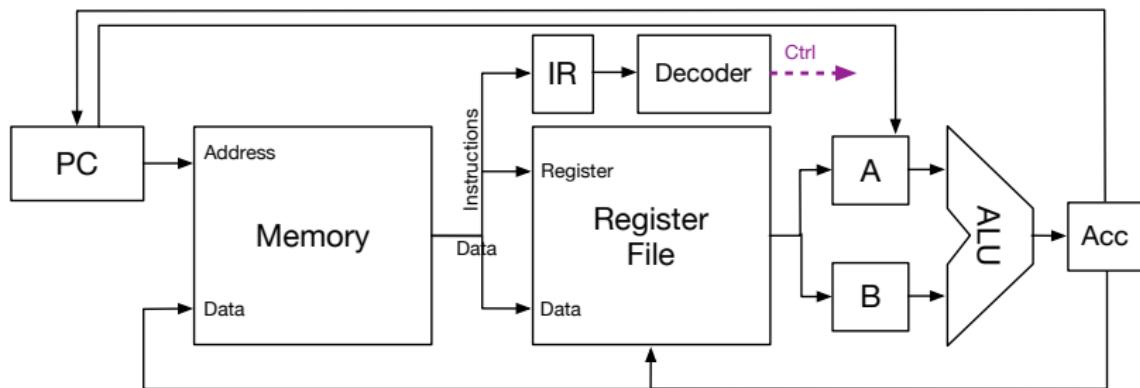
Plan for today

1. Pipelined, super-scalar architectures
2. Execution hazards – overview
3. Name dependencies
4. Data dependencies
5. Instruction dependencies
6. Branching and control dependencies
7. Loop unrolling

1. Pipelined, super-scalar architectures

Previously

- We defined basic computer architecture as a **combination** of **Von Neumann** and **Harvard architecture** to maximize **data & instruction throughput** between the CPU and the memory
- We saw a simplified model of minimal computer architecture:



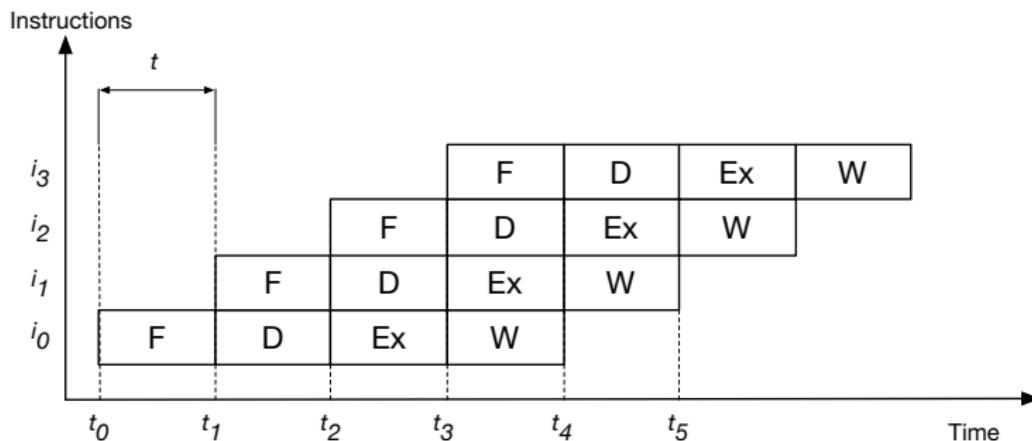
- Any (modern) CPU will look like this; it is of course a **VERY** simplified view, we will add things to this as we move along ...

Practical instruction execution

- Instruction execution model introduces different views:
 - ▷ Programmer's view – F, D, Ex, W
 - ▷ IC designer's view – tied to logic circuit critical path → it defines minimum period T_{min} , or maximum circuit operating F at which the logic circuit will operate correctly
- Pipelined execution is implemented to trade-off:
 - ▷ reduction of T_{min} , or increase of F due to shorter critical path as we "break" the circuit into smaller entities
 - vs.
 - ▷ latency – pipeline execution will delay the output of the instruction for the number of cycles equal or proportional to pipe-line depth
- Pipeline depth is a HW design choice, with two extreme options:
 - ▷ CPUs with few pipeline stages generally associated to RISC
 - ▷ CPUs with many pipeline stages generally associated to CISC

IPC of a pipelined architecture

- **IPC = Instruction(s) Per Cycle:** for a single ALU and even if different stages are executed in parallel, at its best, **the pipeline can achieve $IPC=1$** ; adding more ALUs could make $IPC>1$
- That is 1 instruction that completes at **each clock cycle**
- $IPC=1$ is possible only when pipeline is initialized (here from t_4 & on) & all execution steps take same amount of time ...

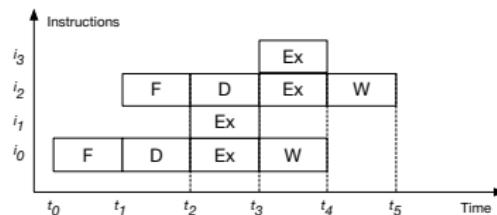
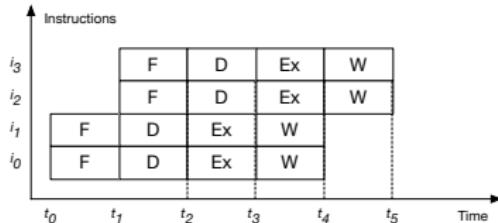


Super-scalar architectures

- In our simple architecture we have assumed **only one execution unit**; so only one instruction could be executed in one clock cycle
- ALUs are **not that expensive in HW** when compared to other components in the core/CPU (RF, other stages logic, etc.)
- Why not using multiple ALUs to enable higher instruction throughput – **super-scalar architectures**
- If we have multiple ALUs **and the program has data independent instructions**, we could have few executions in the same clock cycle, and thus $IPC > 1$
 - ▷ Note that in super-scalar architectures we don't replicate computation only, but memory access too; so when $IPC > 1$ we have computations **and/or R/W operations**
- Question is how to supply data to multiple ALUs, and make them as busy as possible all time? (i.e. how to fetch and decode more instructions per single clock cycle?)

Instruction execution in super-scalar architectures

- One approach is to duplicate complete pipeline stage, i.e. we **duplicate the whole pipeline** (left figure)
 - ▷ This is what **multi-core systems** do ... This obviously cost area since we replicate the complete pipeline HW
- Another approach would be to use the existing F/D stage to **fetch & decode more than one instruction at each clock cycle** to feed the pipeline back-end (right figure)



- Today CPUs use combination of two: at core level one F/D stage generates multiple instructions for Ex stage; further, multiple cores are implemented in a single CPU (single IC package)

Acceleration of a super-scalar architecture

- For N instructions, n pipeline stages and m execution units, the total cycle count for a super-scalar architecture and acceleration with respect to a normal pipeline architecture is:

$$C_{ssc} = n + \frac{N-m}{m} \rightarrow Acc = \frac{C_{ssc}}{C_{pipe}}$$

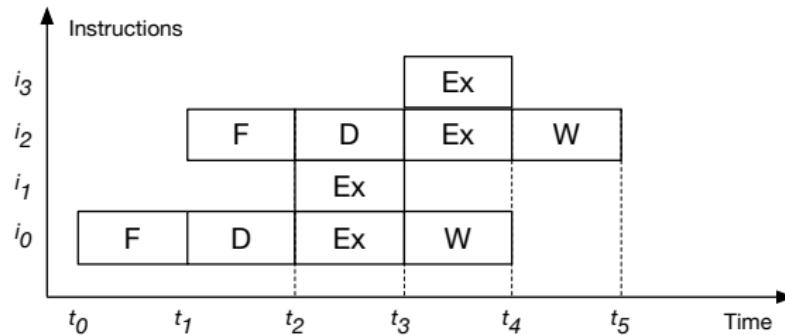
- For $N \gg n, m$ (N is big compared to n and m):

$$Acc = \lim_{N \rightarrow \infty} \frac{C_{ssc}}{C_{pipe}} = \lim_{N \rightarrow \infty} \frac{n + \frac{N-m}{m}}{N+n-1} = \frac{mn+N-m}{m(N+n-1)} = \frac{1}{m}$$

- Acceleration of a super-scalar architecture is proportional to the number of execution units!
 - Compared to Single Issue Base Line the acceleration is: $n \times m$

In-order issue & super-scalar execution

- Super-scalar processor fetch & decode **few** instructions at a time to build the instruction queue; instructions are pulled in the order in which they appear in the program – **in order instruction issue & execution** (we will see later how could further improve things)
- Different instructions are **then dispatched** to different ALUs



- Different ALUs in a super-scalar CPU do not have to be the same – often they have a common subset of functions + some specialized features (i.e. they are heterogeneous)

Resource conflicts

- Let's imagine that on top of normal functionality we have a dedicated HW multiplier in ALU1 and not in ALU2
- If two instructions are to be executed are add and mul, we are OK, both ALUs can be used at the same time (parallel processing) and we could have $IPC=2$; if two instructions are add, OK too (we still have $IPC=2$)
- **Problem arises if both instructions are mul** – if issued at the same time they will generate a **resource conflict** – two or more instructions try to use the same HW resource; if this is the case one instruction will have to wait ...
- This will cause pipeline stall and we loose clock cycles as before
- In another words **functional specificity** of ALUs will generate extra constraints for instruction scheduler that will become more complex to extract ILP to increase IPC (no free lunch)

2. Execution hazards – overview

Pipeline acceleration

- Execution time speed-up is proportional to the number of pipeline stages n is possible only iff:
 - ▷ All pipeline stages take same, fixed, predictable amount of time; not easy to achieve in HW & SW for obvious reasons
 - ▷ There is no dependency between consecutive instructions – this could be done in SW (programmer or compiler), but also not that simple (notion of instruction scheduling that we will see later on)
 - ▷ We have lot of instructions – this is possible in heavy loops
- First two assumptions are EXTREMELY important! If they are satisfied then the programmer is very lucky ... or he (she) understand computer architecture so well, that was able to write a perfect program!
- In practice this is (VERY) hard to achieve, even for a very good programmer that knows HW and SW very well

In the real world

- What happens if ideal pipeline pace can't be maintained?
 - ▷ Do you remember Charlie Chaplin in the movie "Modern Times"? (a must see movie by the way ...)
 - ▷ Our hero is too tired & distracted to follow other workers in the pipeline
 - ▷ He misses his turns, pipeline chain is broken ... and Mr. Ford doesn't become a millionaire !
- If this occurs in a computer architecture we call this: **pipeline stall, broken pipeline or bubbling** (bubble "enters the pipeline")
- If this occurs often (think of loops & this is where computers are good), **SW performance will be seriously degraded**, so high performance apps will try to minimize these as much as they can



Bubbling in computer architecture

- Fundamental problem of bubbling is that in any practical computer architecture we will always have what is called **execution hazards**
- Execution hazards occur:
 - ▷ When instructions to be executed in the pipeline **do not take the same amount of time to execute**
 - Use of non-optimized arithmetic or any exotic operations ...
 - ▷ When **something unpredictable occurs**
 - You need to write to a IO device that is busy, your write can't complete
 - You need to fetch one operand from RF and the other from memory
 - ▷ When there is **computational dependency between instructions**
 - This is something that we will have by definition since computing with machines is about serialization of computation – **ie. Turing machines**
- We need to enable HW management of such situations, even if it is advised that we manage pipeline stalls at SW level too; after all this is what makes a difference between good & poor program

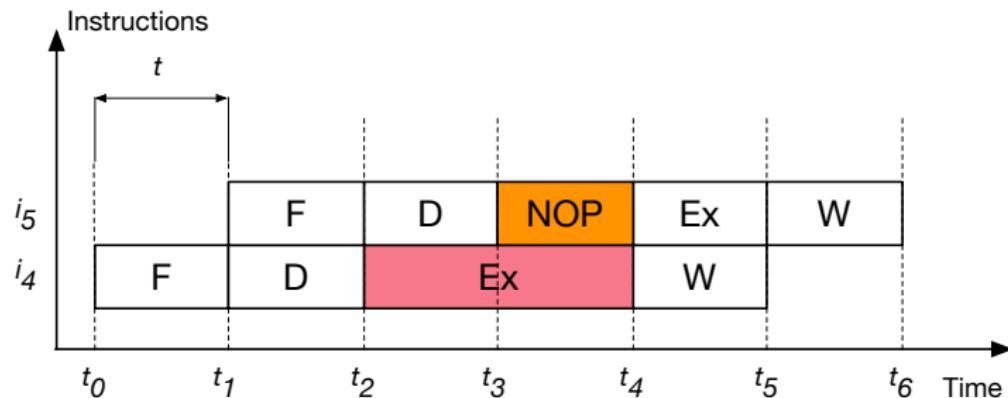
Pipeline stall – a simple way to handle hazards

- Instruction i_{n-1} is fetched and decoded; already at that stage the control logic determines if there is a potential hazard for the next instruction i_n
- If i_{n-1} will take more time to execute (and now that we know what is going to be executed we know how much cycle it will take), control logic will insert extra cycle(s) using **No OPeration – NOPs** instruction (AKA **wait states**) to slow down the execution of the instruction i_{n-1} ; the number of inserted NOPs will compensate for the difference in execution time
- Let's compute $D = (A/B) + C$ using:

```
1 mov ax,a;  
2 mov bx,b;  
3 mov cx,c;  
4 div ax,bx; -- this one takes two cycles  
5 add ax,cx; -- this one will have to wait
```

Bubble due to different execution times

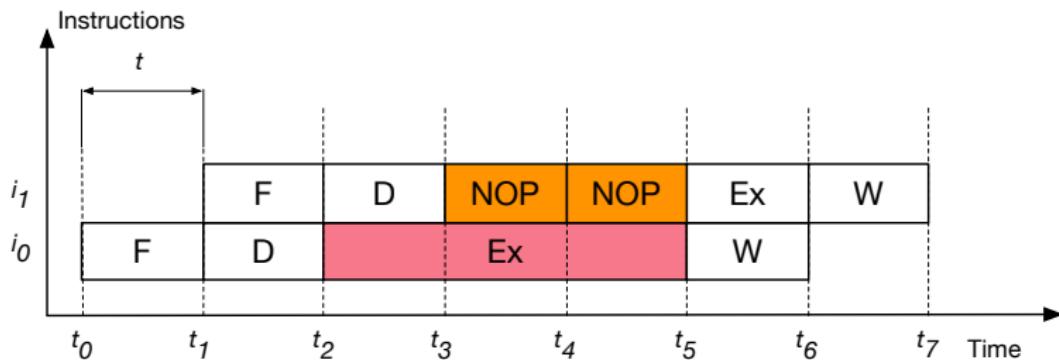
- In the previous code suppose that `div (i4)` takes two cycles to execute and since it occurs before `add (i5)` that takes one cycle we insert one NOP
- Instruction `i5` waits for 1 cycle, no write in t_3 , 1 cycle lost ...



- Here we lost only one cycle; but what will happen if i_4 takes much more ... this could impact program performance a lot

Big bubble

- In the example below the output of i_1 is delayed 2 cycles because execution of i_0 will take 2 cycles



- If the number of NOPs > 2 , the previous stages will not be able to proceed either; **bubble will propagate backwards to the input of the pipeline**
- Ultimately F stage will be blocked and next instruction will execute only when the first fully completes ... then the pipeline works as SIBM, and we saw that SIBM are far from being great!

Impact of bubbling on acceleration

- For a pipelined CPU, with a single ALU, we could have **IPC= 1 at most**, this would mean one (result) write at each clock cycle
- Because of execution hazards we have pipeline stalls, and thus the real IPC is always going to be <1
- Obviously to improve the execution time we need to reduce pipeline stalls as much as possible
- Pipeline stalls can be avoided at HW-level using more or less complex methods, some of which are used in various CPUs that surround us, including low-cost ones
- We will look into some solutions based on different types of pipeline hazards, that can be classified as follows:
 - ▷ Hazards due to different types of **dependencies**, typically we speak of: **Name, Data and/or Branching / Control &**
 - ▷ Hazards due to **resource conflicts**

3. Name dependencies

Name dependencies

- RF is a set of registers that store operands; RF registers are fast, multi-ported memories, expensive in silicon, so we typically limit their number as much as possible to **limit** the system cost
 - ▷ Note that RF performance could be also impacted with RF size
- Each register in the RF is identified with an address, or name in the ISA, to be used by the programmer (or compiler)
- Register name dependency occurs if two or more instructions destinations targets the same register address (name), **but if there is no actual data flow between instructions**
- If the number of registers is small, this could happen often
- These dependencies are not **true (data) dependencies**, since there is no real computational sequence between instructions (reason why they are also called **false dependencies**)

Register renaming

- If there is no data dependency, register names can be changed to avoid as much as possible dependencies & thus pipeline stalls → register renaming
- After all the programmer (or the compiler) does not care where exactly the data is stored; computer of course does care, as long as the memory references are consistent
- Obviously we need to preserve overall computation order imposed by the program to guarantee the right result on output
- We have different approaches depending on **when** and/or **who** exactly is performing the register allocation, typically:
 - ▷ Static register allocation
 - ▷ Dynamic register allocation
- While both have certain advantages (& disadvantages!) dynamic is a preferred choice whenever we want to have more performance

Static register allocation

- Static register allocation is done **during programming or at compile time** (and thus not at run-time)
- We have time to decide how to do it, i.e., we can spend some compile time to figure out the right allocation; since we have more time to decide, we could possibly get good results
- Who will do this allocation?
 - ▷ **Programmer** – when allocating registers for operands (assembly); for complex problems this can quickly become hard due to limited number of registers; programmer will try to minimize number of RF to memory transfers
 - ▷ **Compiler** – when translating from high-level (C/C++) into assembly; due to automation allocation could be possibly better
 - ▷ Whoever does it, once registers are allocated they can't be changed; change is possible only if we re-compile (not handy); note that bigger the RF is, simpler the problem of allocation is (but the RF performance is lower, as usual it is a trade-off)

Dynamic register renaming

- Logical (ISA) registers are mapped to physical registers using some kind of **dynamically allocated table** that specifies which logical register is mapped to which physical register; conversion table is called **Register Alias Table – RAT**
- Thus at run-time, any program that executes instructions will access physical registers of the RF **through** the RAT
- RAT is implemented as a **Look-Up Table (LUT)**, i.e. memory: address is the logical register and the content is the physical register; a single memory read operation is needed to see which logical register correspond to which physical register; this is fast!
- The content of the RAT is updated at run-time depending on the needs that can be anticipated for a given window of instructions to be executed
- This in general works quite well & is implemented in most CPUs these days since ratio benefits/cost is very favorable

Name dependency – example

Here logical and physical registers are the same:

```
1 mov cx, [mem1]
2 add cx, bx
3 mov cx, [mem2]
4 add cx, dx
```

- Both additions target same destination register cx
- There is no possible overlap of instructions
- Poor possibility for pipelining because everything points to cx

Logical and physical registers mapped through RAT, the actual execution of the code on the left could look like this:

```
1 mov cx, [mem1]
2 add cx, bx
3 mov ax, [mem2]
4 add ax, bx
```

- Second DST register is modified at run-time to target another free register slot (e.g ax)

4. Data dependencies

Example of data dependency

- Let's compute: $c/b+a$ assuming following register allocation:

```
1 mov cx,a  
2 mov bx,b  
3 mov ax,c  
4 div ax,bx ; this one is slow  
5 add ax,cx ; this one is fast needs previous output  
6 mov d,ax
```

- Suppose that arithmetic division in line 4. is going to take more time than addition in line 5.
 - This might not be true for high performance CPUs, because they will have fast divider circuit that may take same time as addition
- If these two instructions are to be pipelined: addition will have to wait until division ends first! → **pipeline stall**

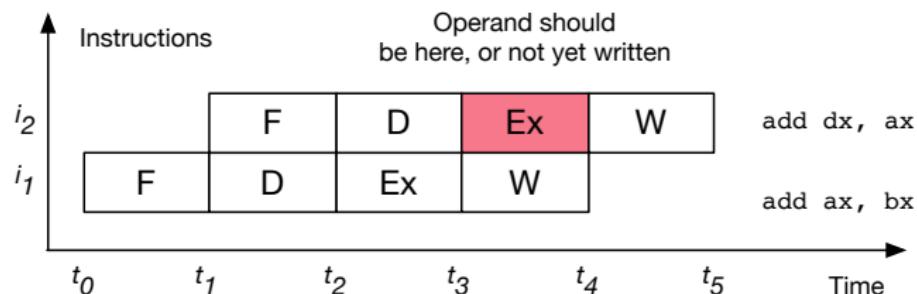
Example of data dependency

- It is possible to enumerate all possible types of data hazards!
- Two subsequent instructions have `src` and `dst` operands for which we could have read and write operations
- Thus 2×2 options give us 4 combinations, out of which following 3 are interesting:
 - ▷ Read-After-Write (RAW)
 - ▷ Write-After-Read (WAR)
 - ▷ Write-After-Write (WAW)
- Fourth or Read-After-Read (RAR) can be skipped since it involve read-only operations, and not write

Different types of data hazards (1/2)

Read-After-Write (RAW) – (true data dependency) one instruction reads a location after an earlier instruction writes new data to it, but in the pipeline the write occurs after the read (so the instruction doing the read gets old data)

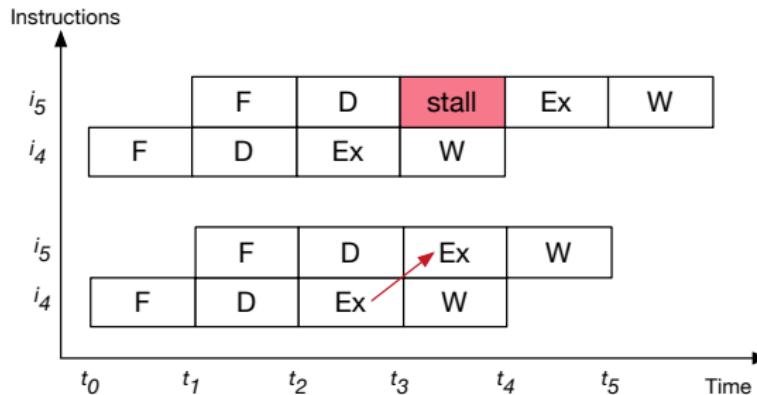
```
1 add    ax, cx;  
2 add    bx, ax; ax needs to be written first
```



How CPU handles this?

Simple solution for Read-After-Write hazards

- In RAW case if i_5 uses result from i_4 , it will be stalled even if i_4 takes only 1 cycle to execute
- This is because data needs to be written to RF **before** i_5 can use it (E stage of i_5 can proceed only after W of i_4 , hence a **bubble**)
- Or computed data is already available after E of i_4 : if this computed **data can be passed directly to i_5 (bypass RF)** we do not need to wait for W of i_4 → **data forwarding**



Different types of data hazards (2/2)

Write-After-Read (WAR) – (anti-dependence) if i_2 writes before i_1 uses it; a write occurs after a read, but the pipeline causes write to happen first (this can happen if we have multiple ALUs and i_2 terminates earlier, good order needs to be preserved)

```
1 add ax, cx;
2 add cx, ax; if cx is written before previous
               ; instruction has completed; could occur on
               ; architectures that change execution order
```

Write-After-Write (WAW) – (output dependence) both instructions write to same destination; this can & will happen if we have multiple ALUs and i_2 terminates earlier

```
1 add ax, cx
2 add ax, bx; both instruction executed in parallel
               ; 2nd terminates before 1st
               ; execution order is not right
```

5. Instruction dependencies

Instruction scheduling 1/2

- Two instructions are said to be **independent** if they can execute simultaneously (i.e. in parallel)
- If we have multiple ALUs, then multiple independent instructions **could** be executed in the same clock cycle and $IPC > 1$; if instructions are independent they will not cause pipeline to stall
- From the program perspective this would mean that the best is to put **as close as possible and as much as possible** independent instructions together!
- Inversely, data dependent instructions should be placed just far enough so that the result is available when **exactly** needed
- In another words **there is a preferred execution order of instructions** that maximizes the instruction throughput, i.e. minimizes pipeline stalls – **Instructions scheduling** – is about finding this preferred execution order of instructions

Instruction scheduling 2/2

- Scheduling is a well known computer science problem, although most often studied on a completely different level of abstraction
- Usually we speak about **task and/or thread scheduling** in **Operating Systems (OS)**
- For OS-level scheduling, many different techniques exist, the choice is made depending on what we want to do: please the user, or provide some guarantees on the execution time that could be hard (like in real-time systems)
- Difference between task & instruction scheduling is in **granularity** & amount of time available to compute the schedule; OS needs to make decision in mili seconds while instructions must be done in nano seconds
- Small amount of available time to compute schedule will have a strong impact on the way instructions are scheduled; **Can you tell what and why?**

In-Order execution

- Proposed simplified architecture follows the execution sequence imposed by the program itself: this is called **In-Order execution**
- Instructions are executed in **exactly the same order** as they appear in the executable code; simple from HW perspective ...
- But if the executable code is poor, hazards will occur
- If we use assembly language than the programmer is responsible for the instruction order
- If we use high-level programming language, this can be automated and the code quality will depend on the compiler quality
- Important: for in-order architectures, **correct scheduling is going to be ISA dependent**, one executable may execute differently on another ISA (or may not execute at all)
- Consequence – **performance is architecture dependent**

Out-of-Order executions vs. In-Order execution

- Another approach: make a **dedicated HW unit** that will perform **Out-of-Order** (OoO) execution (so, at run-time)
- Goal of OoO engine is to examine the code before the execution and **pack** instructions together to minimize potential pipeline stalls (i.e. maximize instruction throughput)
- Because it is a dedicated HW block OoO execution can adapt **dynamically** to the code that executes
 - ▷ Advantage: scheduling is compiler/user independent
 - ▷ Disadvantage: extra area (**significant!**) & minimal decision time, or scheduling problem is a very complex SW problem, so don't expect miracles (remember: Garbage-In, Garbage-Out)
- Even if OoO is fully automated, performance chasers will check eventual penalties (especially in loops)
- Most of the CPUs today are OoO, only small CPUs or μ controllers remain in-order

Out-of-order (OoO) execution example

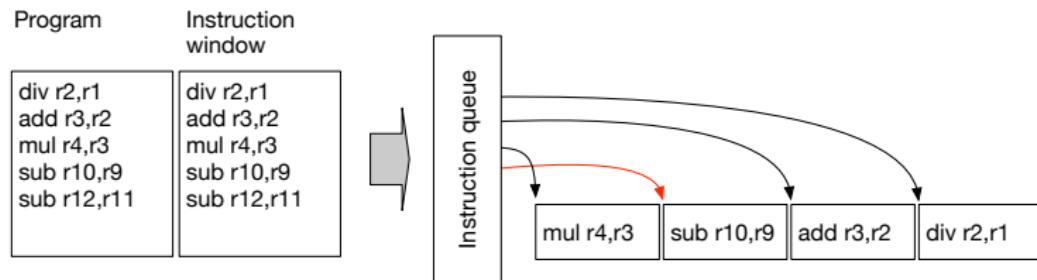
- First 3 instructions are data dependent, so no scheduling opportunities for div, add, mul; div takes 3 cycles to execute
- Last two sub operations are independent
- One of the sub could be executed earlier to hide the latency of mul

```
1 div r2,r1;
2 add r3,r2;      -- needs r2
3 mul r4,r3;      -- needs r3 --> serial computation
4 sub r8,r7;
5 sub r10,r9;
6 sub r12,r11;    -- all these are independent
```

Cycle	1	2	3	4	5	6	7	8
Div R2, R1	Fetch	Decode		Execute		Write		
Add R3, R2		Fetch	Decode	Wait		Execute	Write	
Sub R8, R7		Fetch	Decode	Execute	Write			
Mul R4, R3		Fetch	Decode	Wait	Execute	Write		
Sub R10, R9			Fetch	Decode	Execute	Write		
Sub R12, R11				Fetch	Decode	Execute		

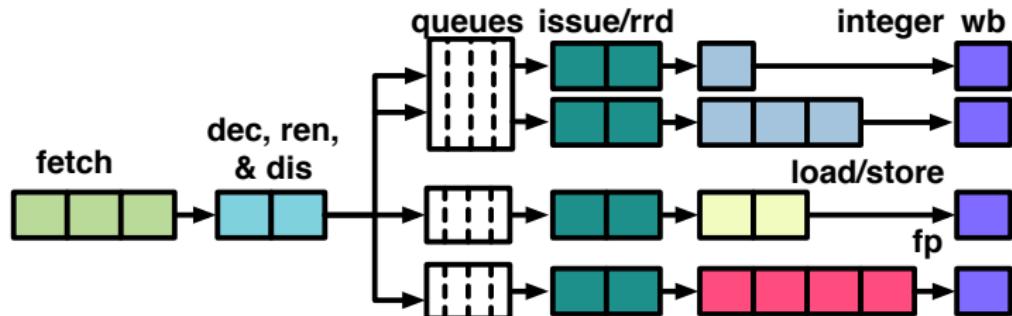
OoO: How it is done in practice?

- Instructions are fetched and dispatched into an **instruction queue** (instruction window) where they wait until their input operands become available
- Instructions are allowed to leave the queue before others; they are issued whenever an appropriate ALU is free
- Results are queued (in some memory), and only after all older instructions have their results written back to the register file, the end result will be retired from the output queue



OoO: Concrete academic example

- University of Berkeley → **Berkeley Out Of Order Machine (BOOM)** – open source, RISC-V spec CPU that is configurable (you can choose what is inside), supports multiple ALUs, it is silicon proven with performance comparable to competitors
- Described using high-level language descriptions (Chisel)
- Generates actual HDL for circuit synthesis, place & route



6. Branching and control dependencies

Branching with unconditional jumps

- Most of ISAs & programming languages enable **unconditional jumps**, with some kind of **jmp**, **goto**, **label** statements
- Problem with unconditional jumps is that very quickly they result in **difficult to read & debug programs**, not to mention possible **execution inefficiencies**; jumps will for sure cause pipeline to stall!

```
1 label1:  
2     ...; -- do something  
3     jmp label2;  
4     ...; -- some other code  
5 label2:  
6     ...; -- do something  
7     jmp label1;
```



- Programs with unconditional jumps are referred to as "**spaghetti code**": the analogy is perfect, it is hard to figure out which spaghetti goes where!

So what we should do?

- Alternative to unconditional jumps is **structured programming**:
A programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures, for and while loops – in contrast to using simple tests and jumps such as the go to statement which could lead to “spaghetti code” causing difficulty to both follow and maintain.
- Thus you should use functions & subroutines, **NEVER JUMPS!**
- Known for some time, this is what E.W.Dijkstra wrote in '68:

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code).

"Go-to statement considered harmful", Comm.ACM11(1968),3:147-148

Branching with conditional jumps

- If unconditional branches are banned from well designed computer program, **conditional branching** is essential to any SW since they provide mechanism to alter “normal” instruction execution flow depending on some Boolean conditions evaluated at run-time
- At **branching point**, the instruction control flow can chose from 2 different instruction sequences that are mutually exclusive
 - ▷ One **xor** the other, but never both since the condition is a Boolean

```
1 if ( BOOL ) then {  
2     // instruction sequence 1  
3 } else {  
4     // instruction sequence 2  
5 }
```

- In the above code, BOOL condition may be a (very) complex expression resulting FALSE (0) or TRUE (anything $\neq 0$)
 - ▷ Note that the condition will use our ALU to compute the result

Branch execution

- Look at the practical branch below:

```
1 if (a==35) then {  
2     // instruction sequence 1 BRANCH IS NOT TAKEN  
3 } else {  
4     // instruction sequence 2 BRANCH IS TAKEN  
5 }
```

- Depending on the condition $a==35$, the branch could be:
 - ▷ Not taken (no branching) – we execute the instruction sequence 1; the next instruction is stored just right after in the program memory, **we do not have to change the Program Counter**
 - ▷ Taken – we execute instruction sequence 2; the next instruction is stored somewhere further in memory, **we have to change the Program Counter!** and adapt the execution accordingly
- Taking the branch will have impact on performance, especially with pipelines that work well for normal execution flow in which the next instruction fetched will be the one to be executed

Branching and pipeline execution

- Problem of branching is linked to the next instruction address that will change to something arbitrary instead of just simply increment the PC (next instruction); this **can not be easily anticipated**
- Consequence of the above is that **next instruction can't be fetched** before we know which of the two paths we will have to follow!
- In the example below, i_4 has to wait until cycle 8 to be fetched; this is caused by cumulated delay due to divide in i_1 , add in i_2 (with data dependency to i_1) & **branch** i_3 instructions

INST	Cycle	1	2	3	4	5	6	7	8	9
i_1	div R2, R1	Fetch	Decode		Execute		Write			
i_2	add R3, R2		Fetch	Decode	Wait		Execute	Write		
i_3	branch			Fetch	Decode	Wait		Execute	Write	
i_4	add R4, R2								Fetch	Decode

How to practically deal with branching?

- Since branches are essential part of any SW & they have strong impact on pipeline performance: **we need to handle them smartly**
- Smart branch handling involves some kind of **prediction**, typically two things should be anticipated:
 - ▷ If the branch will be taken or not; this will drive if the Program Counter needs to change differently from a simple increment
 - ▷ New branching address computation to enable appropriate instruction fetch
- Summary of branch handling options:
 - a) **No prediction** – even in “*Keep It Simple and Stupid*” (KISS) approach, we still need to enable branch handling in pipelined CPUs
 - b) **Static prediction** – decide at CPU design-time the favorite branch outcome
 - c) **Dynamic prediction** – allow computer to decide at run-time; we speak of **speculative branch execution**

a) No prediction at all

- We first compute Boolean condition (we have only one ALU remember) and then we look into the result of this computation
- Next instruction can not be fetched before we know which of the two paths we will have to follow!
- Simplest scheme to handle branches is to **flush the pipeline**, i.e. hold or delete any instructions after the branch, **until the branch destination is known**
 - ▷ Flushing pipeline is in general a bad idea; **Can you quantify what exactly are we loosing?**
- This is simple to make in hardware and to use in software
- But as you would expect this is not efficient, especially when the programmer wrote a wrong test ... in a loop that will execute many, many times (so many, many pipeline flushes & lost CPU cycles)

b) Static branch prediction – default “Not Taken”

- **Static prediction** – during CPU design time we decide which is the preferred branch that will be **always** chosen by the HW (choice between take or not to take)
- When branch is encountered, the pipeline **continues** the execution, as if there was no branch in the code; we fetch the next instruction with the PC incremented as usual
- So, even if the result of the branch condition will be known later, the next instruction is **pushed into the pipe**
- Pipeline doesn't see the branch, the condition is calculated as any other instruction in the program
- When the result of the test becomes available we are either:
 - ▷ **OK** – program took the branch that it should, pipeline is not broken, SW & HW are in phase, perf is good; the result of the test can be discarded; it is like there was no branch in the SW
 - ▷ **Not OK** – we took the wrong way!

b) Measuring the impact of branching

- If we took the wrong path, we need to recover and the easiest way to do so in HW is to (once again) **flush the pipeline**:
 - ▷ Turn already fetched instruction into NOP (no operation)
 - ▷ Re-calculate the new address for the Program Counter (PC)
 - ▷ Restart instruction fetch using newly computed address stored in PC
- Taking the branch impacts CPI because of extra execution cycles:
 - ▷ $CPI=1+N_Branches \times N_TakenBranches \times CyclesLost_PerBranch$
 - ▷ Assume 10% branches, with 70% taken & 3 cycles lost per branch on average → $CPI=0.2 \times 0.5 \times 3=1.3$ and not 1
 - ▷ Meaning 30% performance loss: **this is huge!**
- If your CPU assumes “Not Taken” branching strategy, you could eventually try to reduce $N_TakenBranches$ to improve the CPI by writing smarter code (it is less likely that CPU implements branch “Taken” strategy, due to HW complexity)

b) Improving branching with SW

- For any if statement we could try to figure out: how often the condition is TRUE (or FALSE)?
- Any condition could be often TRUE, often FALSE, 50/50, or whatever...; if we know that, and if CPU uses “Not Taken” branching strategy, we could write HW dependent code – same code run on CPU with “Taken” branching will not work well!
- What can you say about values of a & b below, knowing that the programmer used variable statistics to write different tests?

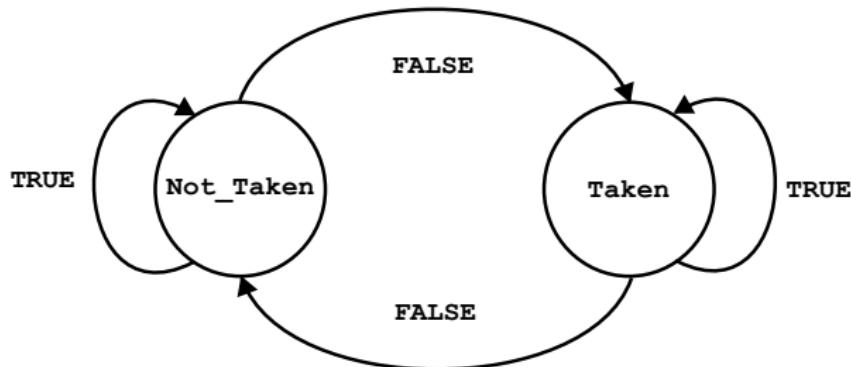
```
1  if (a!=35) then {  
2      // instruction sequence 1  
3  } else {  
4      // instruction sequence 2  
5  }  
6  if (b==47) then {  
7      // instruction sequence 3  
8  } else {  
9      // instruction sequence 4  
10 }
```

c) Dynamic branch prediction – motivation

- In static prediction, all decisions are made at CPU design time, **before** the actual program execution, and hence with zero adaptability during program execution
- Or, in real-life we could deal with **different input data sets**, and the run-time will be then data dependent
 - ▷ Imagine image processing SW that has to deal with mostly white or black pictures; you can't decide at compile time which one
- To allow dynamic adaptation to the SW, most of the CPUs introduce **dynamic branch prediction** – use past branch behavior (*branch history*) to somehow predict the future
 - ▷ Past outcomes are stored in **Branch Prediction Buffer** (BPB), a simple Look-up Table (LUT) that stores past outcomes
 - ▷ Based on previous results stored in BPB, prediction can be made using more or less complex algorithm (see next slides)
 - ▷ Prediction will depend on the behavior of the branch at run-time

c) Simple prediction algorithm: 1-bit prediction

- Simple dynamic branch **1-bit prediction scheme** uses **1-bit Finite State Machine (FSM)** with 2 states NOT_TAKEN & TAKEN
- Input for the state machine is the result of the current prediction: TRUE if prediction was good, FALSE otherwise



- In other words: prediction will be the same as long as the prediction made was good; however if there is misprediction, the FSM will flip the state, i.e. next prediction

c) 1-bit prediction & loops

- Loops use branches to decide if the loop body should be entered or not: taken (T) enters, Not Taken (NT) exits the loop body

```
1 // simple loop in C
2 for (i=0; i<7; i++){
3     // some computation
4 }
5 // loop in assembly ==>
```

```
1 .loop:
2     add    $1,edx. ; j++
3     ..... ; some computation
4     cmp    $7, edx ; if j<7 then goto L3
5     jne   .loop    ; BRANCH (T)
6     ..... ; (NT)
```

- Assume that when branch is executed for the first time, we predict that it is not taken, so NT
 - ▷ NT is initial state for the state machine on the previous slide
- If the loop has at least one iteration, the NT prediction will be wrong in the beginning of the loop, will turn to T until the last element in the loop where a misprediction will occur again (T instead of NT) → so NT (T), T, T, T, T, T, T (NT)
 - ▷ If the loop is huge, two mispredictions are not a big deal

c) 1-bit prediction – nested loops problem

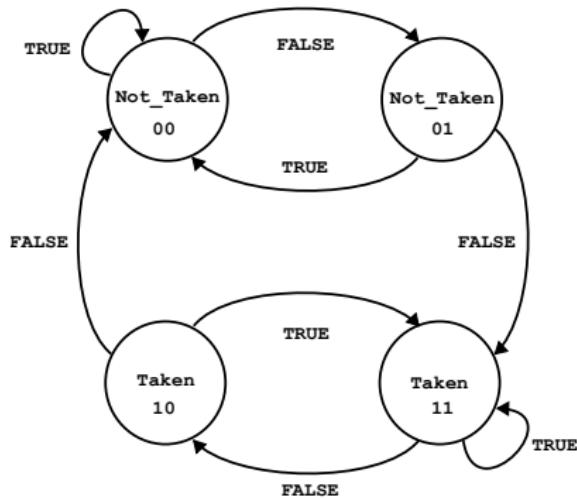
- Now let's analyze 1-bit branch prediction in a **nested loop**:

```
1 for (i=0; i<1000000; i++) {  
2     for (j=0; j<10; j++) {  
3         // some computation  
4     }  
5 }
```

- Due to loop nesting we will now have significant mispredictions count that could cause serious performance penalty due to frequent pipe-line stalls; in the example:
 - Inner loop will have 2M mispredictions:** 1M outer loop iterations \times 2 mispredictions per loop iteration
 - Outer loop has 2 mispredictions (that we can neglect here)
- If the computation in the loop body is simple, the execution will be inefficient since more cycles will be spent on loop management than the actual computation

c) 2-bit prediction

- Let's introduce 2-bit branch prediction scheme – the FSM now has 4 states
 - Note the state encoding indicated in the figure underlines the fact that there are two NOT_TAKEN states, so 00 & 01, and two TAKEN states, so 10 & 11
- Prediction flip will now occur only if there was a wrong prediction twice in a row
- 2-bit branch prediction scheme is a generalization of n -bit saturating counter approach; however the studies has shown that the extra benefits brought with $n > 2$ are not worth the effort



c) Dynamic branch prediction – downside

- Spectre vulnerability – exploit branch prediction to perform side-attack!



SPECTRE

Abstract—Modern processors use branch prediction and speculative execution to maximize performance. For example, if the destination of a branch depends on a memory value that is in the process of being read, CPUs will try to guess the destination and attempt to execute ahead. When the memory value finally arrives, the CPU either discards or commits the speculative computation. Speculative logic is unfaithful in how it executes, can access the victim's memory and registers, and can perform operations with measurable side effects.

Spectre attacks involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary. This paper describes practical attacks that combine methodology from side channel attacks, fault attacks, and return-oriented programming that can read arbitrary memory from the victim's process. More broadly, the paper shows that speculative execution implementations violate the security assumptions underpinning numerous software security mechanisms, including operating system process separation, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks. These attacks represent a serious threat to actual systems since vulnerable speculative execution capabilities are found in microprocessors from Intel, AMD, and ARM that are used in billions of devices.

Paul Kocher et al., "Spectre Attacks: Exploiting Speculative Execution"

7. Loop unrolling

Why loop unrolling?

- Let's look into the following loop:

```
1 // heavy loop starts here
2 for(i=0;i<1000000;i++) {
3     A[i]=A[i]+B[i];
4 }
```

If there is a **delay** when executing the computation in the loop on line 3 (e.g. data not being ready, complex operation, etc.), **pipeline stall** could occur in every loop iteration!

- Plus, at the end of each iteration: loop index has to be calculated (`i++`), & condition (`i<1000000`) needs to be evaluated
 - Note that condition evaluation is useful only 1 out of 1.000.000 times! For 999.999 times the test is TRUE, and for 1 it is FALSE
- Bigger the loop is, more time (& power) we loose; note that loop indexes could be easily more than $\times 1.000.000$ iterations

Loop unrolling: how and what does it do

- Consider the transformation of the previous loop:

```
1 for(i=0; i<250000; i++) {  
2     A[i+0*250.000]=A[i+0*250.000]+B[i+0*250.000];  
3     A[i+1*250.000]=A[i+1*250.000]+B[i+1*250.000];  
4     A[i+2*250.000]=A[i+2*250.000]+B[i+2*250.000];  
5     A[i+3*250.000]=A[i+3*250.000]+B[i+3*250.000];  
6 }
```

- Four computations are now **independent**, so:
 - if the CPU is super-scalar, multiple execution units could be used at the same time to parallelize computations
 - it is possible to use register renaming & order instructions to minimize pipeline stalls
 - compute less conditions & increment less (less loop overhead)
- However something else will go wrong with the code above, we will discuss this later, once we add memory view to our system

ELEC-H-473
Th04: Memory subsystem

Dragomir Milojevic
Université libre de Bruxelles

2023

Previously

- Pipeline architecture and reasons that cause pipeline inefficiency
 - ▷ Execution time variability (complex instructions, memory latency, ...)
 - ▷ Pipeline hazards due to data & control dependencies
- HW & SW techniques to prevent hazards & improve pipeline IPC
 - ▷ Register renaming
 - ▷ Data forwarding/out-of-order execution of instructions
 - ▷ Branch prediction
 - ▷ Loop unrolling
- We have until today abstracted the memory part

Or memory plays a very important role in the overall system behavior & performance

Today

1. Memory technologies
2. Memory organization
3. Cache memory & hierarchy
4. Implementing caches
5. Memory management

1. Memory technologies

Bit-cell technology

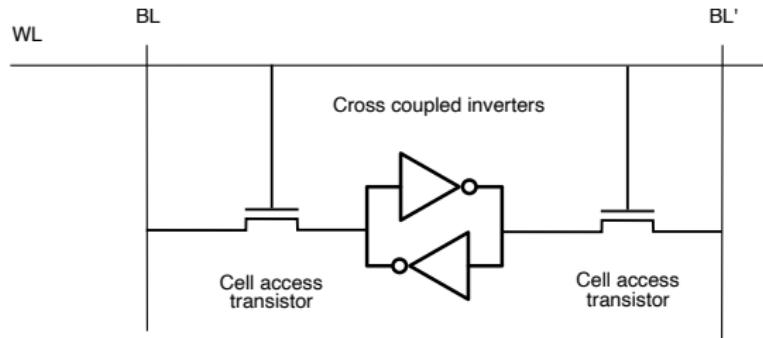
- Different technology options could be used to store 1-bit of information (and many **others in development such as MRAMs**)
- Two classes, depending if the information is lost on power down:
 - ▷ **Non-volatile** – don't need power to retain information and
 - These are gaining a lot of traction (SSDs, SD card, USB etc.)
 - ▷ **Volatile** – their content is lost on power down
- Key volatile technologies:
 - a) **Flip-Flops (FFs)** – Basic storage element for any sequential logic circuit; **fastest** of the three, but most expensive in terms of area (typically used to build the registers, register files, FSMs,...)
 - b) **SRAM** – Static Random Access Memory (random: any data address can be accessed in same amount of time); performance, price in the middle (used for smaller memory capacities, fast)
 - c) **DRAM** – Dynamic Random Access Memory low cost/bit but at the expense of high access time (good for massive storage, but slow)

a) Flip-Flops (& Latches)

- **Bistable** – sequential circuits having only two stable states (sequential logic circuit) controllable by their inputs
- If the bistable element is sensitive to the signal **level** of it's input (absolute voltage value) it is then called **latch**
- If the bistable element is sensitive to an **edge** (transition 0 to 1 & inverse), of the control signal it is called **Flip-Flop –FF** (we already used them to build the instruction execution pipeline)
- Difference between the two is in the state table and the way how they will be implemented in HW; typically latches are banned from HW design since they are sensitive to glitches
- For FF, the edge is produced by a rising/falling edge of the periodic signal, called **clock**
- **FFs are preferred memory devices for fast memory access** so local signal storage and RFs

b) SRAM cell

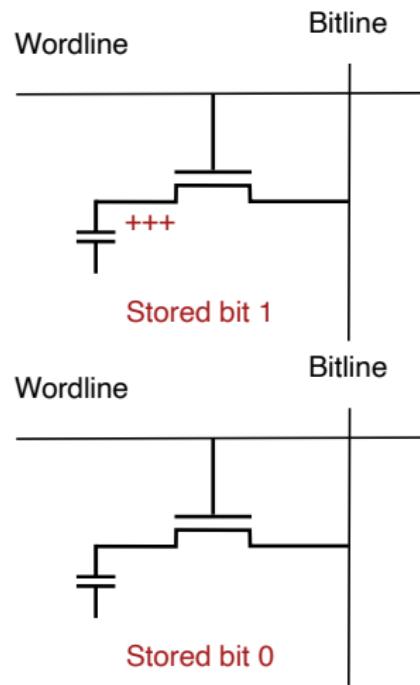
- One bit of information is stored in a pair of cross-coupled inverters



- SRAM cell can be in one of three states: Idle, READ or WRITE;
2 nMOS cell access transistors control the state:
 - when wordline (WL) is active, both nMOS are ON
 - the bitline (BL) state is transferred from/to inverter couple
 - this is called 6T(transistor) SRAM since $2 T$ for ctrl + $2 \times 2 T$ for INV
- SRAM cell is immune to noise because the inverters will restore the value that could be possibly lost – auto-refreshing

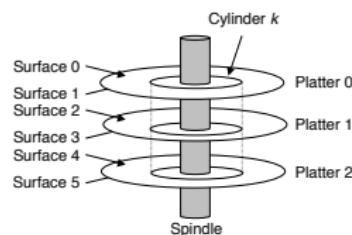
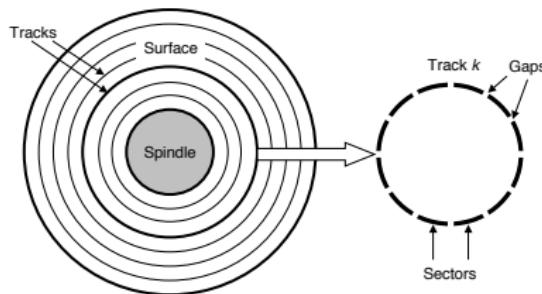
c) DRAM cell

- One bit of information is stored in the form of a **capacitor charge**
- Single nMOS transistor controls the charging process of the capacitor
- When the wordline is active nMOS is on, bit value is stored (WRITE) from, or placed (READ) on bitline
- Capacitors leaks current, i.e. **content will be lost** after some time, and after read operation (destructive read)
- **DRAM cell needs refreshment at regular rate even when not read**, hence the name; during refresh DRAMs can't be accessed (degrade access time)



Magnetic Hard Drive Disks (HDDs)

- Constructed from platters; each platter has both sides coated with magnetic recording material; few platters encased in a sealed container; platters spin at fixed rotational speed (up to 15kRPM)
- Each platter a collection of concentric rings called **tracks**; each track partitioned into a collection of **sectors**; each sector contains an equal number of bits (typically 512B)
- Sectors are separated by gaps where no user data is stored but formatting bits that identify sectors



Cross-comparison of different bit-cells

- These will heavily depend on IC manufacturing process
- Transistor count gives you the idea of area ratio

Technology	FF	SRAM	DRAM
Transistor count /bit	20	6	1
Access time	<0.5	1-10ns	100ns

- SRAMs performance, power & area are function of capacity; bigger SRAM capacity is, worse performance, power & area are
 - ▷ Note that these days SRAM have hard time scaling !

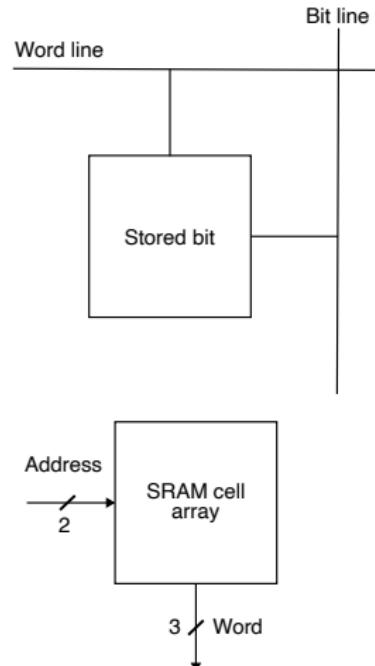
This is why computer systems today implement complex memory sub-systems in hierarchical way; designing memory sub-system is a complex task, because many parameters need to be considered

2. Memory organization

Memory structure and organization

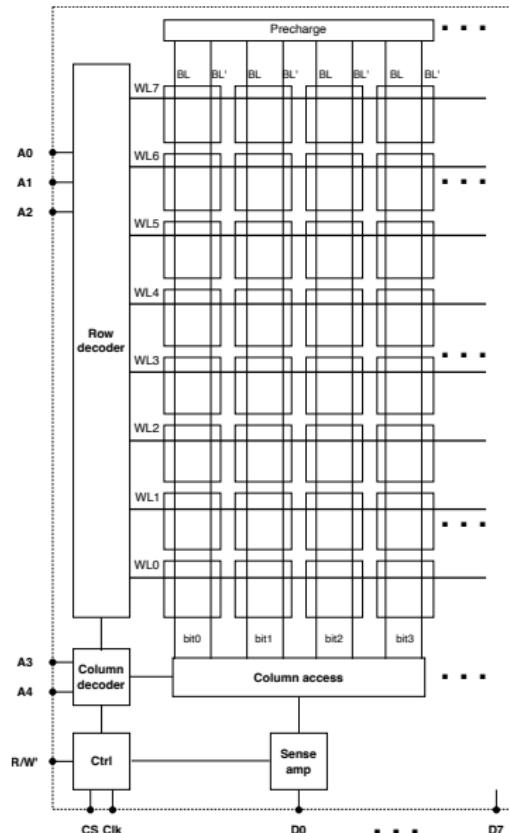
We introduced **bit-cell storage** element, i.e. the smallest memory component that stores 1 bit of information (various options)

- To build larger memories, single bit-cells are organized in **2D memory arrays** of say n rows & m columns; memory capacity is then $n \times m$ bits
- Each row is identified with unique address; only one row can be accessed at a time using **wordline** (WL); columns are accessed using **bitline** (BL) to create a word
- Address on the bus activates one row & one column of the array:
 - ▷ from data bus to array for **WRITE**
 - ▷ from array to data bus for **READ**



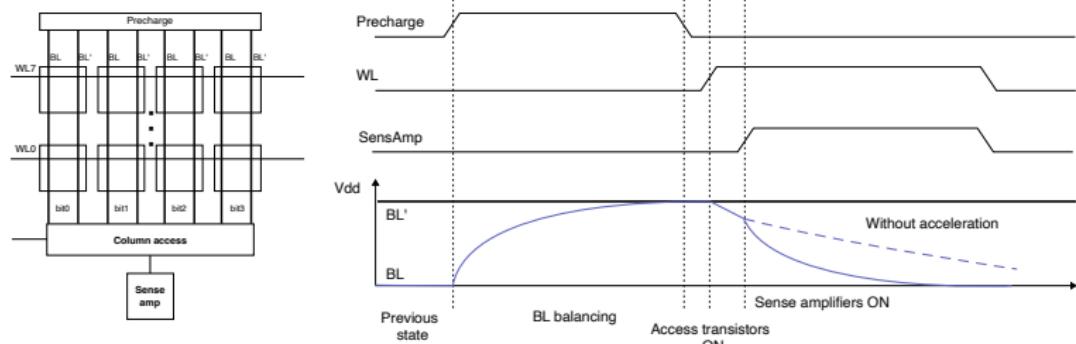
SRAM array

- Organization → HW design choice; many parameters trade-off; design is automated → **memory compiler** assembles **memory macro** generation using 1-bit cell and optimized logic
- **Row & column decoders** are used to perform bit selection within array
- Above is replicated to enable parallel access to one data word; could be anything from 8 to 512 bits
- Example: 8 lines × 4 columns give access to 1 out of 32 bits; replicated 8 times to get for 8-bit word
- Chip Select (CS) used to activate SRAM & Clk drive R/W' operation



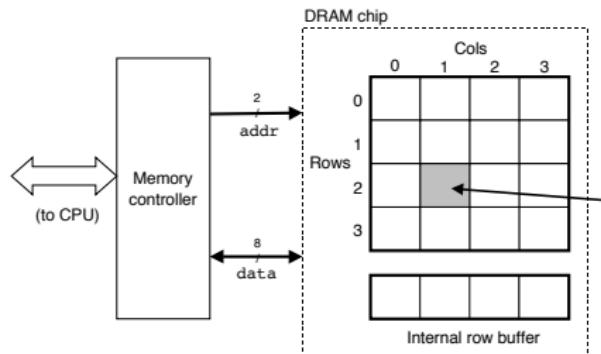
Improving SRAM timing

- As memory capacity increases the length of BL & WL increases as well as the number of pins connected to it
- Bigger SRAM capacity will have **more capacitance** that can't be driven with realistic circuits; **R/W delay, power will be too high**
- Current solutions to improve R/W time even in small macros:
 - Precharge circuit** – deal with BLs; dedicated circuits will bring voltage V_{dd} level **before** the actual R/W operation takes place
 - Sense amplifiers** – will accelerate & intensify any voltage disparity between BL and BL' (inverted BL)



DRAM array

- Just like in SRAMs, DRAM cells are organized in array connected to **memory controller**, logic that organizes transfers of w bits at a time
- To read cell (i,j) , controller first sends the row address i (here 2)
- DRAM then copies entire contents of row 2 into internal row buffer
- Next, the memory controller sends the column address (here 1)
- DRAM copies 8 bits from the row buffer to the memory controller
- Row, column addresses are called RAS & CAS for Row or Column Access Strobe that appear as DRAM IO pins

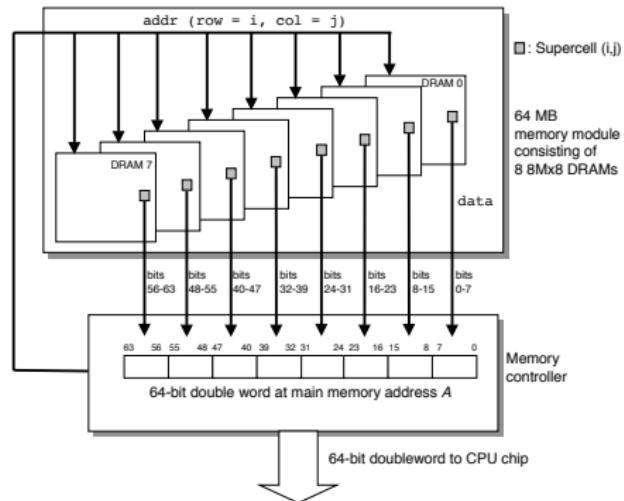


On single memory access size

- Any system architecture will have to define the **smallest unit of memory access**
- Choice is made at architecture level and is based on cost/performance trade-off; off-chip memories need IC-pins and these pins are routed at PCB level; their cost limits the pin count
- If the smallest unit of access is one byte, then in order to construct more complex data types, such as 64 bit integers or complex data structures, probably used more often than simple unsigned char, we need to perform **multiple memory accesses**
 - ▷ We need 8 memory access for a 64 bit integer or structure!
- **Not good!** especially for data dominant applications (often today)
- To increase memory bandwidth **more word parallelism is required**; bytes are in general packed in wider words so that single data access returns one large word (this is why 512-bits words)

DRAM as a full chip/module

- DRAM ICs are packaged in memory modules that plug into expansion slots on the main system PCB (motherboard)
- Single Inline Memory Module – SIMM with 72 pins for 32-bit chunks, or Dual IMM – DIMM with 168-pins, for 64-bit chunks
- Main memory built using few DIMM modules and 1 memory controller



Main memory – DDR SDRAMs

- DDR = Double Data Rate – we get 64 bits of data at both rising & falling clock edges; SDRAM = Synchronous Dynamic RAMs
- DDRs are standardized: access protocol and electrical properties:

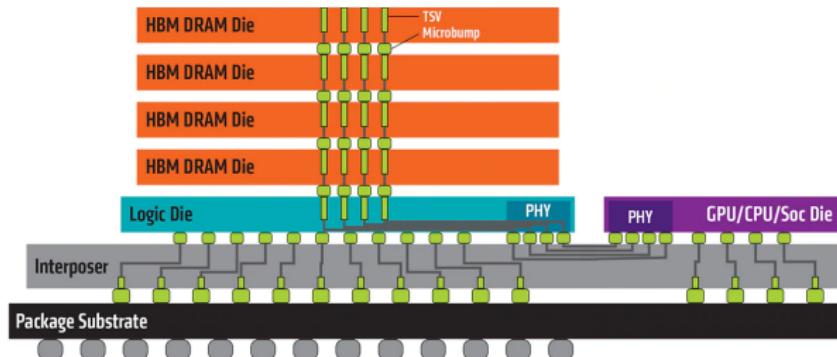
Names	Frequency	Transfer Rate	Max Bandwidth
DDR	100MHz	200 MT/s	1.6 GB/s
DDR2	400MHz	800 MT/s	6.4 GB/s
DDR3	800MHz	1600 MT/s	12.8 GB/s
DDR4	1600MHz	3200 MT/s	25.6 GB/s
DDR5	NA	6400 MT/s	NA

- Even if DDR maximum BW looks great this is not sufficient for most applications these days – computer systems are hitting memory wall; CPUs need more BW than this

Solution: 3D stacked DRAM memories

High bandwidth Memory

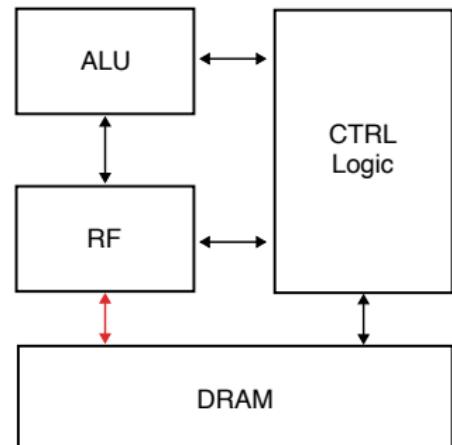
- Multiple DRAM dies are stacked on the top of each other using: **3D-IC stacking technology**; more memory capacity for same volume
- Bottom die implements control logic using wide data bus; **1024/2048 bits or more**; more pins, shorter connections (lower capacitance) allow higher operating frequencies and lower power → more bandwidth
- Complete system built using **silicon interposer** – IC level PCB; CPU & DRAM on the same die, i.e., in the same IC package; PCB macroscopic wires replaced with IC level interconnect



3. Cache memory & hierarchy

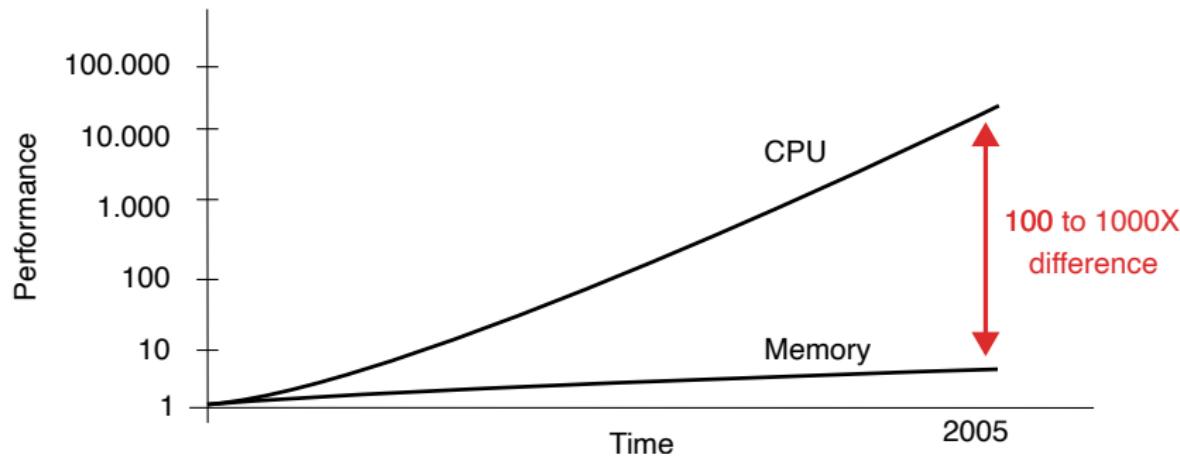
Direct CPU to central memory case

- We have seen a **datapath** between the RF and the main memory
- During Ex step of an instruction we will look for operands; if they are not in the RF, we need to fetch them in the central memory, most likely to be an off-chip DRAM; this is different CMOS process, so it can't be implemented on the same die as logic
- In order to balance execution & memory accesses, and hence not stall the processing, we need to **compute & access data equally fast**
- DRAMs are protocol based, slow and far away, and we know that CPUs=ALU + RF are fast
- But how big is the difference between the DRAM and logic?



Uneven scaling of logic vs. memory

Normalized performance of CPU vs. Memory:

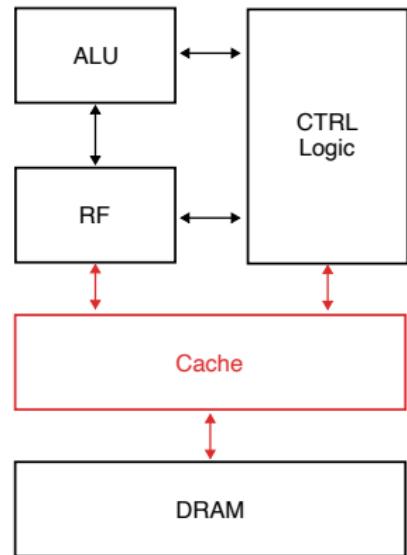


DRAM process scaling did not follow the same evolution pace as logic; over the years bigger and bigger gap grew between the two

→ CPUs are way more faster than memories; ALUs will have to wait for data, & we waste many CPU cycles

Solution for slow DRAMs

- Insert **smaller**, but **faster** memories (SRAMs) on instruction & data datapath between CPU & main memory – **cache**
- Cache holds a copy of main memory content and it can deliver it to CPU ALU/RF much faster than DRAM
- When CPU needs new data (or instruction) it first looks into the cache
- If data found – **cache hit** – **main memory latency is hidden from CPU**; CPU could be stalled, but just for limited number of cycles (acceptable loss)
- If not – **cache miss** – data needs to be fetched from main memory; CPU is stalled for significant number of cycles



Why cache memory work? (SW view)

- Cache hits are possible because of two types of **locality**:
 - ▷ **Temporal locality** – same data locations are accessed closely in **time** (example: loop counter; loop counters are incremented after in general small number of executed instructions)
 - ▷ **Spatial locality** – data that will be accessed in the near future is generally not that far away from the current data (good example are data arrays, in general we access arrays one element after the other; same goes for index counters, etc.)
- How **local** data is, will depend on how cache memory is controlled, but also the SW part:
 - ▷ the problem itself
 - ▷ the algorithm used to solve the problem
 - ▷ program implementation
- Understanding how caches work may help you write more efficient code, because you will be able to minimize the access time

Practical example of temporal & spatial locality

- Let's take a loop and array computation:

```
1 for (j=0 ; j<1000000; j++) {  
2     C[j] = A[j] * B[j];  
3 }
```

- Temporal locality – it is very much alike that from t_i to t_{i+1} index j becomes $j+1$
- Spatial locality – if $C[j]$ has been accessed recently, it is alike that $C[j+1]$ is “close” to $C[j]$
- Locality is ok in the example above because we access things in a “linear” fashion; but loops & indexes could be anything, like below:

```
1 for (j=0 ; j<1000000; j++) {  
2     C[j] = A[j] * B[j*100];  
3 }
```

- Watch out, some access patterns to arrays could reduce locality and thus program performance

Cache memory is a necessary evil

- Obviously: **more cache hits** mean less CPU stalls, so better performance; **more cache misses** on the other hand mean more CPU stalls, so worse performance
- Also **more cache memory capacity** mean higher cache hit rate probability (spatiality will work for more distant data)
- When designing a computer architecture we trade-off:
 - ▷ **Performance** – try to maximize cache hits,
 - ▷ **Cost** – but minimize cache size
- Implementing big on-chip caches is expensive, SRAMs are not cheap, **so the above two criteria are orthogonal!**
- Complex compromises need to be made, but we need more memory because applications are memory hungry
 - ▷ General purpose CPUs; memory is between 30 and 60% of total die area! For more dedicated applications (e.g. Deep Learning) this ratio could be even worse (5-10% of logic for up to 95% of memory)

Cache performance metrics

$$\begin{aligned} \text{mem_stall_cycles} &= n_instructions \\ &\times \text{cache_misses_per_instruction} \\ &\times \text{miss_cost} \end{aligned}$$

- *n_instructions* – simply the program size
- *cache_misses_per_instruction* – how often instructions will cause cache misses; depends on SW & cache capacity
 - ▷ More cache capacity is likely to reduce the number of misses
- *miss_cost* will depend on:
 - ▷ **Memory access time** – time required to get one data (technology & memory capacity dependent)
 - ▷ **Hit rate** – how many times memory access returned right data (this will depend on many things more on this later)
 - ▷ **Latency** – time between request & first data being available (memory & interconnect dependent)
 - ▷ **Bandwidth** – time to deliver block of data (interconnect dependent)

Measuring cache performance

- How to evaluate the above params?
- One can evaluate cache miss counts using **simulators** that for a given HW and SW simulate execution and log memory accesses
- Practical example → **CACTI** framework from HP labs

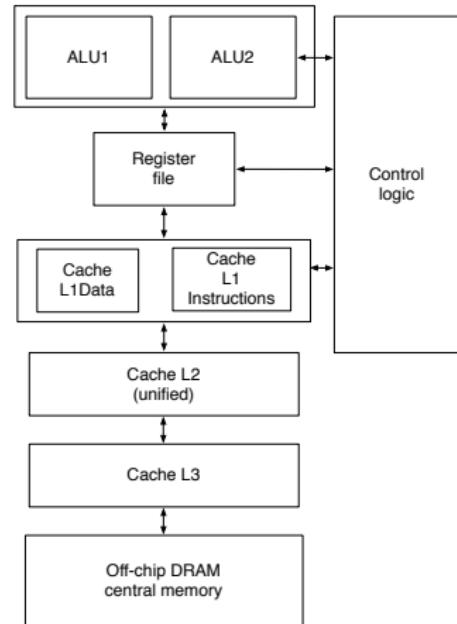
CACTI is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model.

By integrating all these models together, users can have confidence that tradeoffs between time, power, and area are all based on the same assumptions and, hence, are mutually consistent.

CACTI is intended for use by computer architects to better understand the performance tradeoffs inherent in memory system organizations.

System architecture update

- More detailed block-diagram of the system architecture, assuming super-scalar architecture with 2 ALUs
- L1 is split for instructions & data (Harvard architecture), processors are fast enough, work in pipeline & can have high throughput
- Upper cache hierarchy is structured in extra 2 levels of cache or even 3 for high-performances CPUs; **Last Level Cache – LLC** is then L4
- From L2 and on caches are **unified**, they store both instructions and data
- LLC talks to an off-chip DRAM



Memory hierarchy – properties

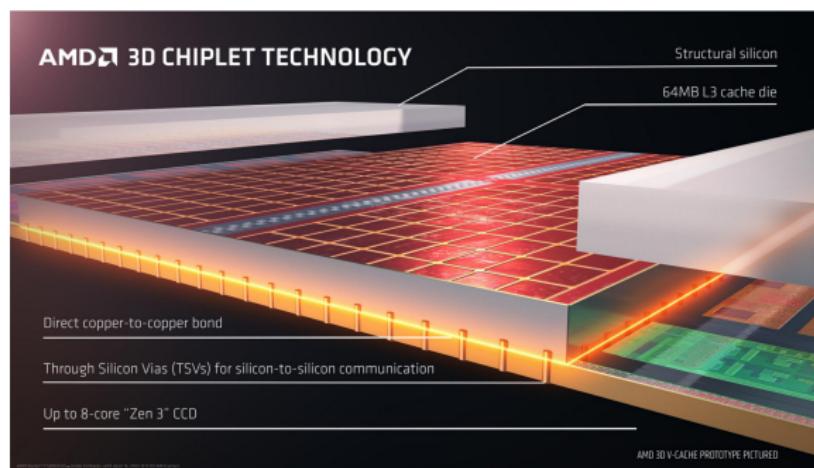
- We can not put some numbers for complete memory hierarchy
- Different memory levels trade-off area (memory size) vs. access time; all this is cost/performance driven

	RegFile	L1	L2	L3	Main	Disk
Size units	<1 kB	<256 kB	<2 MB	<24 MB	<512 GB	>1 TB
Time (ns)	<0.5	0.5-25		<250		5×10^6
Mngmnt	compiler core	HW	HW	HW	OS	OS

- There are orders of magnitude differences!

Stacking caches – AMD Ryzen with 3D VCache

- Novel 3D integration technologies allow stacking of cache SRAMs on top of the CPUs die, we speak of **Memory-on-Logic**
- SRAM dies as 8MB “slices” with 1024-bits interface to single CPU core are implemented on top of the logic die
- Tighter 3D interaction allows higher interconnect speeds & reduce memory access power → **Logic to Memory bandwidth >2TB/sec**



4. Implementing caches

Cache memory: basics

- Cache memory stores instructions and data **plus some extra information** that will help identifying data
- Cache data is structured in **cache entries** (for instructions & data)
- One cache entry typically has the following fields:
 - ▷ **data block** – the actual data
 - ▷ **tag** – a **part** of the data main memory **address**
 - ▷ **flags** – indicates status of the data block, it can be:
 - **valid** – if the valid data has been loaded
 - **dirty** – if the data has been modified by the CPU (this means that some kind of main memory update should take place)
- Data transfers from main to cache memory are done in data blocks called **cache lines** to optimize bandwidth
- Size of a single cache line depend on HW choices (single byte transfers are rarely efficient, we prefer to move data in blocks)

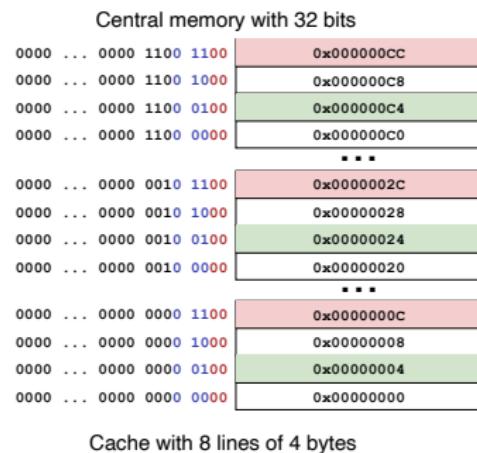
Important questions related to cache

- Where to place a given block of data in the cache?
 - ▷ Known as **block placement** problem – in other words we need to find a method to map main memory address space into cache memory address space
- How do you find a given data in the cache?
 - ▷ Known as **block identification** problem – this has to be fast and at lowest HW cost
- Which data block should be evicted to make room for new ones?
 - ▷ Known as **block replacement** problem – caches are finite and in general of small size, so we need to erase unused data to enable storage of newer and more useful data

Depending on how these three are done we have few different options ...

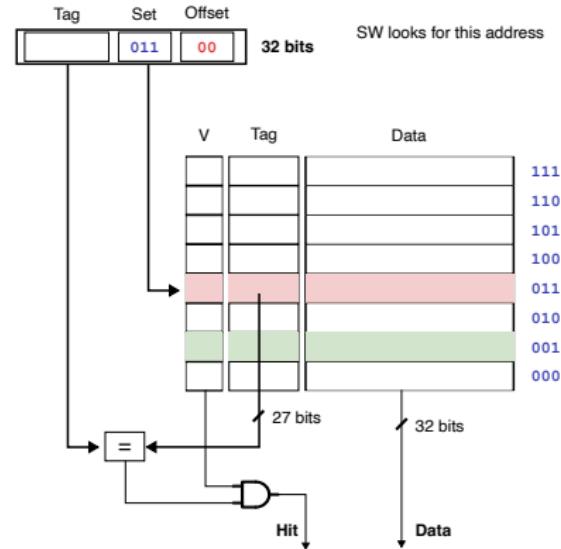
Simple way – Direct mapping

- Map 2^{32} address space into $2^3 = 8$ cache lines of 4B each
- Main memory address space is structured in words of 4B, so 1 address per 4 bytes; addresses are aligned at 4B boundary, so word addresses end with 0, 4, 8, C, etc.
- First two bits (from LSB) indicate **byte offset** within the cache line
- Next 3 bits are used to map the **set address** → all addresses that have these 3 bits the same will point to same set (red or green)
- Remaining 27 bits ($=32-2-3$) are called **tag** → this is what we need to store in order to find data



Direct mapping – block identification

- In the example each cache entry (or set) consists of: 32 bits user data, 27 bits for tag and 1 bit indicating data validity status
 - You should be able to compute complete cache capacity
- Cache memory is accessed using 32-bit address (from SW)
- Two LSBs (byte offset) are ignored at this stage
- Next 3 bits specify the entry (or set) in the cache where we should look for the data
- Load instruction reads the specified entry from cache and checks the tag and valid bits:



If the tag and valid bits are ok, bingo (hit)!

Problem with direct mapping

- Because there is only one set in which given address is mapped, only one location can be cached at the time
- If two consecutive memory accesses target the address pointing to the same set, there will be conflict! Newest data (targeting the same set) will evict the previous data saved at same location
- If this occurs in a loop, we will have a systematic conflict that will lead to many cache misses (and hence the performance loss)

```
1 mov    bx,35;  
2 mov    cx,10000000;  
3 loop:  
4 add    bx, [0x0004]; point to  
5 add    bx, [0x0024]; the same set  
6 sub    cx, 1;  
7 jnz    loop;
```

```
1 mov    bx,35;  
2 mov    cx,10000000;  
3 loop:  
4 add    bx, [0x004];  
5 add    bx, [0x008]; here we're ok  
6 sub    cx, 1;  
7 jnz    loop;
```

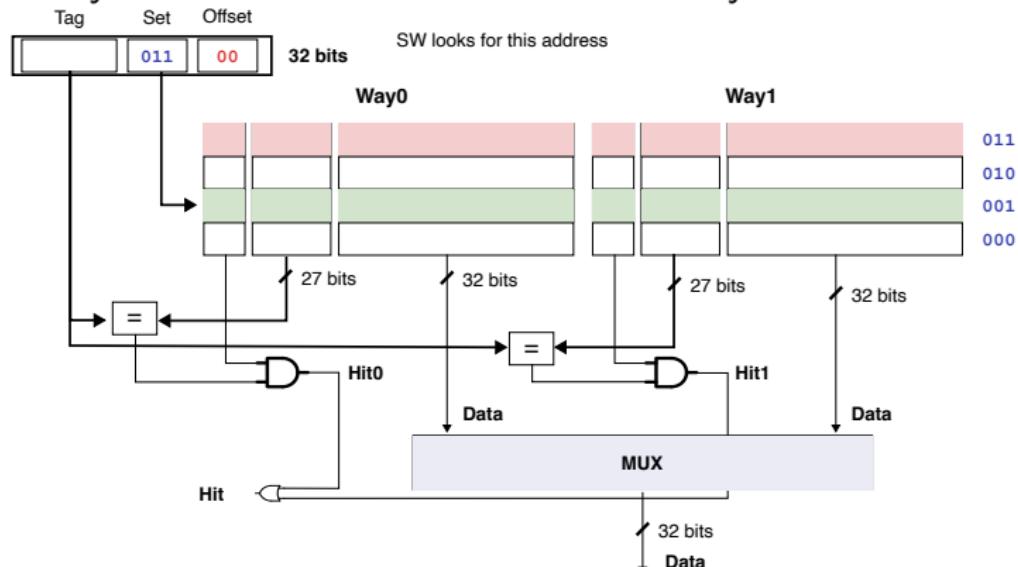
- In left code 0x0024 will evict 0x004 systematically, causing cache miss at every loop iteration → poor performance; right is better!

Solution for conflicts in direct mapped cache

- Instead of having only one place where we can put data stored at a given address, we provide multiple locations in cache; this is called **associativity**
- Cache is said to be ***m*-set associate** if data could be found in m different sets (one set being a group of addresses)
- We also say that m is the **degree of cache associativity**
- Depending on the value of m , we can have less to more associativity:
 - ▷ **Direct mapped cache** – 1-way set associate, given address is located at only one address in cache (we just saw this)
 - ▷ ***m*-set** – data could be found in one of the m sets
 - ▷ **Fully associate** – given data can be placed anywhere in cache
- This also means that in m -set associate we need to look for the data in m different places; it could make HW more complex

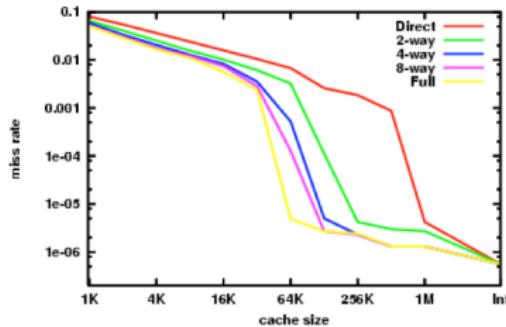
2-way set associate cache: example

- Cache reads blocks from both ways in selected sets & checks tags & valid bits (note that we keep total memory capacity the same)
- Upon a hit in one of 2 ways, multiplexer will select the right data
- In the example previous 0x004 & 0x024 could remain in cache, since they can be stored at different memory locations



Set-associative cache performance

- Bigger caches improve hit rate; higher the associativity, better probability to avoid address conflicts; but more associativity means more complex search
 - ▷ e.g. in a 8-way set associate cache we need to check 8 tags
- Running different benchmarks gives the idea of benefits:



Effect of doubling m is same as doubling the cache size !!!
Why this is interesting?

- Note that the effect is reduced for higher associativities !!!

Replacement policy

- Assume m-way set associative cache
- When cache miss occurs the controller will fetch a new cache line from main memory
- Then we need to decide **where** (in which way) we want to store this new cache line
- In direct mapped caches there is no choice since there is only one address where this line can be stored (data will be evicted anyhow, since only one place)
- In m -way cache we have m possibilities, so which m to chose?
- Since caches are of a small size, during the program execution this question is in fact: *Which existing line in cache to replace?*
- This decision is known as **replacement policy**

Popular replacement policy algorithms

- **Random** – pick the candidate randomly; this is very easy to implement, but you might evict the data that you will need soon
- **Least Recently Used (LRU)** – if **recently** used data is to be **reused**, **least** recent data is **less** likely to be used!
If this is true then erasing the block that has been unused for the longest time should be replaced
- **First In First Out (FIFO)** – LRU needs to keep track of what is going on increased HW complexity; FIFO emulates LRU behavior at lower implementation cost

LRU is the best for smaller caches, however no significant difference for bigger caches with random replacement policy!

Note that each of these methods has very different cost in HW

Caching write operations

- Same reasoning as for read operations applied to writes
- Rather than writing data to central memory directly, we first try to write new data in the cache
- Why we should avoid to write to the central memory directly, i.e. when the result is produced?
- So the variable address is first checked to see if it is cached
- Again two situations can occur:
 - ▷ Hit – the destination address is cached, the data can be first written into the cache more rapidly than to the central memory
 - ▷ Miss – the block of data corresponding to the address is not cached so it has to be written in the main memory (slow)

Policies for central memory update

- Depending when the main memory is updated, we have two different approaches
- **Write through** – the main memory is **immediately** updated after the data has been written to cache
 - ▷ This can be time consuming if the central memory is too slow (we need to wait for the process to complete before issuing other writes)
 - ▷ and becomes **very bad** if this location is constantly updated ...
- **Write back** – the data is kept in the cache as long as possible and the main memory is updated only once in a while
 - ▷ If there is a write to a block, the dirty bit associated with this block is set to 1
 - ▷ We know that this location has been modified and different from the one in the central memory! (important, we will see this later)
 - ▷ Main memory is updated only when the data from cache is evicted

Improving caches & multi-level caches

- Common optimization of write through technique is to insert a **write buffer** between cache and main memory, or **buffered write through** technique
- Buffer is filled by CPU and emptied by memory (like a FIFO)
- Since write buffer stores data and has the life on his own, it will not stall the CPU (unless the buffer is full)
- How good this solution is will depend of course on FIFO depth: more is better but we need to keep our memories small (memories come at cost penalty, especially fast ones)
- If we can't empty FIFO fast enough: buffer will saturate, CPU will not be able to write & we face the same problem as before
- Solution → insert L2 that can offload the buffer and match the operand generation frequency of the CPU
- **Main motivation for deeper cache memory hierarchy**

Instruction cache

- Previously we spoke about data caches only
- But there is **no reason not to apply** same principals to the instructions (after all no difference between the two)
- **What does the instruction cache improve and how?**
- The only difference is that for instructions we are only in the READ mode (because you are not supposed to write over your own code, so those who want to write self-modifying codes need to hack this restriction – this is doable!)
- Also instruction caches are split from data caches (different physical memories) because of the higher BW (Harvard architecture) and different access patterns of instructions compared to data (so cache control could be different)

Further reduction of miss rate

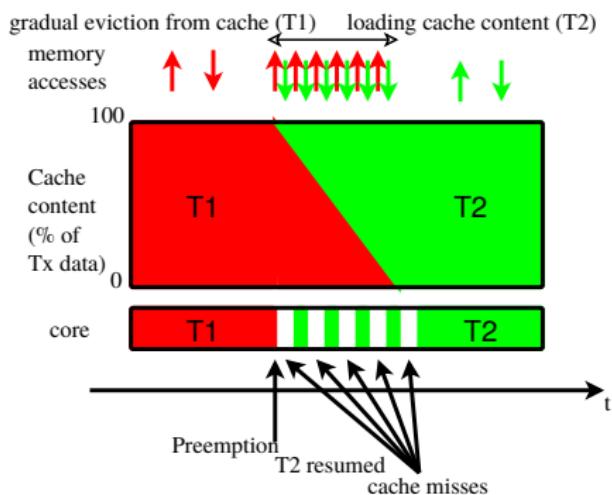
- After all we still have cache misses because cache misses are unavoidable (at least some of them) and this is because we have:
 - ▷ **Compulsory cache misses** – cache memory needs to be initialized; this is unavoidable (unless you explicitly **prefetch data** in cache); it generally accounts for a small proportion of the total miss rate
 - ▷ **Conflicts** – there will be more requests than ways, this is application dependent but also HW implementation trade-off
 - ▷ **Capacity** – if the cache size is too small, data blocks will be systematically evicted and then reloaded, this one is the most important
- Whatever is done in HW is a choice, in general cost driven
- But the application (SW) has to know about this choice and use it at best ...

Caches & context switches

- Complete CPUs (IC-level) are expensive & we are greedy
- CPUs are time-shared devices: different users/applications share the same resource though OS
- OS arbitrates and assigns available resources (CPUs/cores) to consumers at thread level
- In multi-tasking OS regular ticks cause **threads to re-schedule**
- Typical tick period range from 1 to 100ms to ensure some fairness (for general purpose systems; this may vary for some dedicated computer platforms)
- **OS scheduling involves thread migration**; important for desktop computing, but **crucial** for servers/data-centers and any multi/many core system
- Real issue is the performance penalty of the thread switch, since we have to go through complete memory hierarchy

Improve context switches

- Imagine task T1 is preempted by a task T2; content of T1 could be kept & gradually off-loaded to next cache level rather than flushing it completely
- While in the same time load of T2 data (multi-ported memory), this will inevitably cause bunch of misses...
- Solution? Make memory twice (or more) bigger
- Comes at cost; look at cache size of current big CPUs & their cost and especially performance since big chips
- 3D stacked memories: less footprint & interconnect



Understanding caches at SW level

- Take matrix multiplication: if matrix is too big, many cache misses, because of physical non-adjacency of data:

```
1 for (i=0; i<n; i++)  
2     for (k=0; k<n; k++)  
3         for (j=0; j<n; j++)  
4             C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

- Solution for big data sets: cut them in blocks of the cache size and do block-by-block computation to avoid cache capacity problems; then merge temporary results

```
1 for (ii = 0; ii < SIZE; ii += BLOCK_SIZE)  
2     for (kk = 0; kk < SIZE; kk += BLOCK_SIZE)  
3         for (jj = 0; jj < SIZE; jj += BLOCK_SIZE)  
4             for (i = ii; i < ii + BLOCK_SIZE && i < SIZE; i++)  
5                 for (k = kk; k < kk + BLOCK_SIZE && k < SIZE; k++)  
6                     for (j = jj; j < jj + BLOCK_SIZE && j < SIZE; j++)  
7                         C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

- Explain what happens & how caches help? How about loop unrolling?

5. Memory management

Memory space as seen from SW

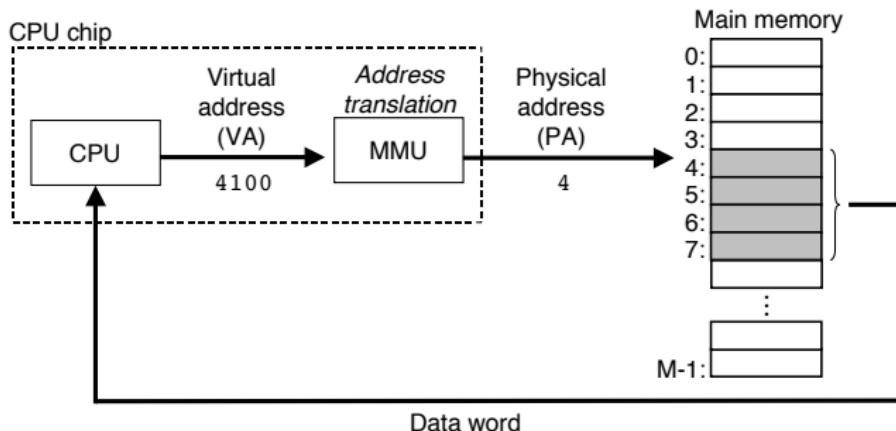
- Program and data are stored in **physical memory**: DRAM ICs for main memory and SRAMs for on-chip cache hierarchy
- Physical memory has **unique, contiguous, physical address (HW) space**, one address points to unique location in memory
- Let's assume that SW manipulates the actual HW address; logic analyzer placed on the address bus will shows the actual pointer
- **This is not a good idea at all!**
- Following reasons support this:
 - ▷ OS will have to allocate some fixed memory for each newly generated process; **How to decide on the amount of memory?**
 - ▷ We should be able to run the same SW on normal and high-performance computer with loads of memory; **How to cope with systems that have different memory capacities?**
 - ▷ Programs do have different memory needs; **How to adapt?**

Virtual & physical memory

- Instead of having same SW & HW views of the address space, why not **making different address spaces: 1 for SW & 1 for HW**
- Address seen by the program are not real physical addresses, i.e., they are now **virtual**
- We can then decide how to **map** the virtual to actual physical memory space at any time, even at run-time though HW and OS interaction; **this is better**
- If two systems have different memory capacity, OS could then easily determine and inform the application if yes (or no) it is possible to allocate memory space for data
 - ▷ **Do you remember `malloc` function?**
- Further, any program will see the memory space that corresponds to the complete addressable memory space, no matter what the actual physical size of the memory is

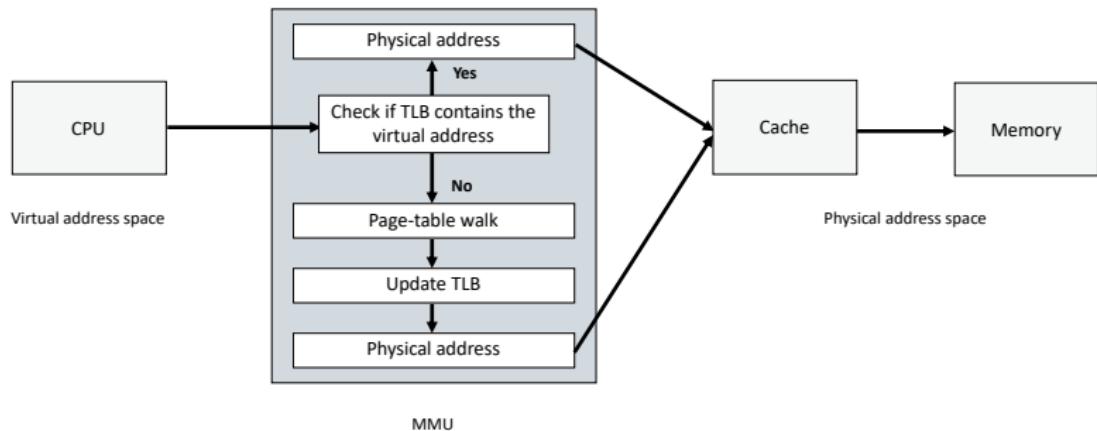
Page address translation

- CPU accesses main memory using **virtual addresses** (VA) that converted to **physical address** (PA) using **address translation** process
- Since address translation occurs at run-time it has to be very efficient! LUT could be a good choice
- **Memory Management Unit** (MMU) translates VA to PA at run-time using dynamically updated LUT managed by OS



Translation Lookaside Buffer (TLB)

- However the LUT previously could be quite BIG!!! So hierarchical LUTs, but then we increase search time → solution cache these addresses
- TLB is a cache of page translations, providing quick access to recent virtual to physical address conversions
- If there is a hit, we can forward the physical address; but if there is a miss, we need to go through page tables to find the translation



Virtual memory: use DRAM as HDD cache!

- On the SW side:
 - ▷ all virtual memory space is allocated to every program running
 - ▷ virtual memory is divided into contiguous byte-sized arrays called **virtual pages** (VPs) stored on HDD
 - ▷ because of HDD access cost, page size is from 4KB to 2MB
- On the HW side:
 - ▷ physical memory is partitioned into **physical pages** (or page frames), of the same size as virtual pages
- **Contents of the HDD are cached in main memory**
- So any virtual page can be found **either in main memory or on HDD**
- Whenever a programs access a virtual page:
 - ▷ Either it is in memory, so no penalty, or on
 - ▷ HDD, page needs to be transferred from HDD to main memory, so penalty due to the transfer (non-negligible even with SSDs)

Advantage of virtual memory: example

- Consider the following C-code, compiled and executed on a machine with 4GB central memory and 64 bits address space:

```
1 int main() {  
2     int size = 64 000 000 000;  
3     int *ptr;  
4     ptr = (int *)malloc(size * sizeof(int));  
5     if (ptr == NULL) {exit(0);}  
6     ...  
7 }
```

- What will happen during execution of the program above?
- In a system where OS sees only physical memory, this will never work, the amount of requested memory is more than the size of the central memory ($64G \times 4B$)
- If we assume HDD as a central memory, cached using DRAM (in a way), then the above becomes possible
- Address space is 2^{64} , virtual memory takes all addressable space

ELEC-H-473 – Microprocessor architecture

Th06: Data parallelism – SIMD

Dragomir Milojevic
Université libre de Bruxelles

2023

Previously

- Instruction Level Parallelism (ILP) introduced to allow parallel execution of different instruction phases to improve throughput
- Execution hazards reduce instruction throughput, different HW techniques proposed to avoid pipeline stalls that cause performance loss; but we need to addressed these in SW too!
- Memory hierarchy with chain of smaller, but faster SRAM memories has been added to compensate for speed difference between cores & central (DRAM) memory; memory management techniques introduced to improve data & instructions throughput
- Super-scalar architectures with multiple ALUs for multiple Ex steps in the same clock cycle; plus Load/Store units for overlapped data transfers & computations → **parallelism**
- But data could be processed even more in parallel ...

Today

1. Data parallelism: motivation
2. Parallel processing: old classification, still up-to-date
3. SIMD extensions in Intel CPUs
4. Identifying the CPU
5. Memory alignment
6. SIMD programming for Intel CPUs
7. Automated compiler vectorization

1. Data parallelism: motivation

On ALU and data types

- ALUs are designed to be efficient on computations for **most commonly used data types** for various applications
- Common data types are **integers** and **floating point** of different bit-widths, could be anything between 8 and 128 bits
- ALU complexity (silicon area) depends on the operand width, so the CPU architects will trade-off: ALU perf. vs. area (i.e. cost)
- If we need simple computations most of the time, 16-bit ALU(s) may be enough; this is what you have in simple CPUs, micro controllers etc. note low-power (mobile) CPUs are already 64-bit
- If we use a CPU with a simple ALUs & if application requires bigger operands **occasionally**, the ALU could support that at the expense of performance: the computation on more complex operands is decomposed into a sequence of smaller operations, so these operations could take few cycles to complete

GPP ALUs

- General Purpose CPUs have ALUs designed to accept **largest** data type possible & therefore can use **any smaller data type**
- This is possible thanks to better IC integration (CMOS scaling)
- Doubling of minimum operand size over the years: from 4 bits in first CPUs to 64 bits today, and more (we will see...)
- Let's look into the inverse problem:
 - ▷ What happens with execution efficiency for smaller data types?
- Take an example of a 32-bit ALU doing operations on two unsigned integers encoded with 8 bits:

Operation	CPU
Min : 0x00	0x000000B5
Max : 0xFF	+ 0x00000011

6 hex digits (24-bits out of 32) on the left are not unused!

How inefficient this is?

- Consider following program & assume more realistic 64-bit ALU:

```
1 unsigned char A, B, C;  
2 int i;  
3 for (i = 0 ; i < 1000000; i++) {  
4     C[i] = A[i] + B[i];  
5 }
```

- ALU usage efficiency will be only 12.5%
- What will happen if we assume 128-bit ALU and same data type?
- Computations over smaller data types using ALUs designed to work for wide data types are highly inefficient ...
- And it is not only about performance, power efficiency is poor too:
 - What will happen in code above if A, B, C were signed integers?

Solution: pack more data to use all 64 bits

- We could do the following using super-scalar CPU with 4 ALUs:

```
1 unsigned char A, B, C;           // data type here is a single byte
2 int i;
3 for (I = 0 ; I<10000000; I+=4) {
4     C[I+0] = A[I+0] + B[I+0];    // 0xB5 + 0x11 for I=0
5     C[I+1] = A[I+1] + B[I+1];    // 0xA0 + 0xAB for I=0
6     C[I+2] = A[I+2] + B[I+2];    // 0xBB + 0xFF for I=0
7     C[I+3] = A[I+3] + B[I+3];    // 0x10 + 0x30 for I=0
8 }
```

and compute 4 sums in parallel in each iteration! these are independent data, so this is doable, remember loop unrolling

- Even better: let's do the computation with a **single 64-bit ALU!**
- How? **Pack data, use same ALU & stop propagating carry;** the above first iteration can be done in one Ex cycle

Bits	31-24	23-16	15-08	07-00
Iter.	i+3	i+2	i+1	i+0
Op.1	0x10	0xBB	0xA0	0xB5
Op.2	0x30	0xFF	0xAB	0x11

This solution has benefits ...

- Better computing performance

- Better computing performance
 - ▷ Sets of data are **vectors**: we speak of **vector processing**
 - ▷ One could design vector ALUs, or if we have wide operand ALU, smaller data types AND need to (or can) do vector operations, **we could modify our ALU** so that it can work as vector processor
 - ▷ This is also known as **sub-word parallelism**: we modify the ALU to work on vectors of smaller operands; one Ex operation still done in one clock cycle, but over the vector of values!

- Better memory access

- Better memory access
 - ▷ Instead of reading individual vector elements we read sets of data; not only we compute faster, but we transfer more efficiently data from memory to Register File (& the other way)
 - ▷ Bus transfers between central-memory & cache and caches & CPU, are **not 8 bits transfers** but more than that (64 bits between DIMM and CPU packages); atomic transfers should use this amount of data
 - ▷ But memory access have some issues that we will cover in Section 5

2. Parallel processing: old classification, still
up-to-date

Sequential processing

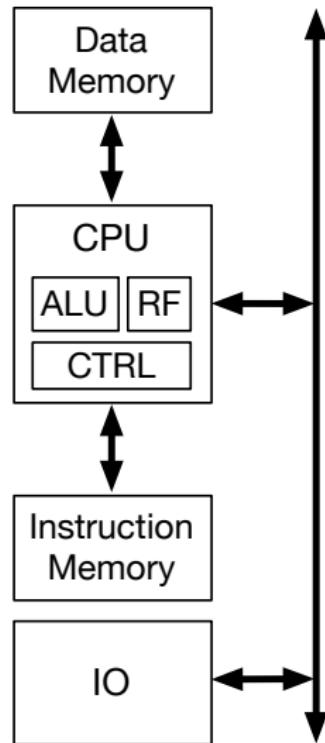
- In our simplified computer architecture model we assumed that the ALU exhibits **bit-level parallelism**; that is all bits in operands will be processed in parallel, concurrently
- Thus, any elementary arithmetic/logic operation will be executed in parallel for all bits in the arguments, and would take let's say 1 Clk cycle to complete (this may vary, but let's keep it simple)
- To keep the complexity of the ALU low, the number of operands is limited to 2 or 3 arguments, rarely more
- Complex operations and/or computations on more operands, are done using a **sequence of simple computations**
- We speak of **sequential execution**, instructions are executed on limited set of data, one after the other in time (Turing-machine)

Parallel processing

- As opposed to sequential processing, if we do some computations concurrently we can speak of **parallel processing**
- Because instruction execution is a lengthy process, it is decomposed in **steps** (F, D, Ex, W) and they are executed in a pipeline → **Instruction-Level Parallelism (ILP)**; once pipeline is full, 4 steps are executed in parallel for different instructions
- This is yet another execution in **parallel** (other than ALU operating in parallel on operand bits)
- We also added multiple execution units to increase the parallelism and allow data independent instructions to be computed in parallel – **super-scalar**
- **There are TOO many parallel things going on, so let's try to put some order in it ...**

Options for parallel computations

- Sequential vs. parallel computation: **how to classify computer architectures?**
- Look at Harvard architecture & the number of occurrences of **instruction** & **data** memories (duplicated)
- How many instruction & data are handled at each execution cycle of our architecture: could be **single** or **multiple**
- So, instructions & data could be single or multiple → there are 4 combinations all
- **Michael Flynn** did this in 1972, to classify different computing systems; his classification (or taxonomy) is still in use



Flynn's classification – four combinations

- Single Instruction Single Data (SISD)
At each clock cycle **one instruction** executes over **one pair of operands**; this is what we already have in our model
- Single Instruction Multiple Data (SIMD)
At each clock cycle **one instruction** executes over **sets of operands**; this is vector processing, sub-word parallelism we spoke about in previous section
- Multiple Instruction Single Data (MISD)
At each clock cycle **multiple instruction** work on **same data**
- Multiple Instruction Multiple Data (MIMD)
At each clock cycle **multiple instruction** can execute over **sets of operands**
- Any computing system today will fall into 1 of these 4 categories

Flynn's classification – practical systems today

- SISD – most common, versatile since they can compute anything that is computable (sequence of computation, c.f. Turing); trade-off execution time for system complexity; **they depend heavily on system speed, and thus CMOS, to deliver performance**
- SIMD – very common in '70 and become popular in past decades: array processors, Graphical Processing Units (GPUs), extensions to traditional CPUs; **omnipresent today in high-performance, but also low-power CPUs**
- MISD – specific machines, not that many examples; good for fault tolerance (Space Shuttle) or systolic arrays (used for Deep Neural Networks, though not really pure MISD)
- MIMD – Any multi, many core/CPU system from desktop computer to computing cluster to internet based computation, data centers, cloud computing, super-computing etc.

SIMD computers

- Used in '60, '70 and '80 to build super-computers with **dedicated CPUs** built only for these machines & not as off the shelf components!
 - ▷ CRAY Research Co. – ULB had one, you can see it from Av. Buyl
 - ▷ Thinking Machines – Connection Machines CM1 to CM5



- During '90, super-computers started to use **general purpose CPUs** found in normal desktop computers to cut the cost, but adding special HW features (e.g. Silicon Graphics started using Intel CPUs)
- From that time and on SIMD moved into General Purpose computing

Why SIMD moved into CPU architectures?

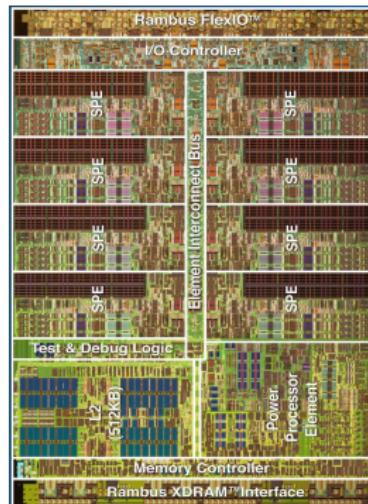
- Multimedia data: audio, image, video ... and combinations
- All these use Digital Signal Processing (DSP) algorithms and these algorithms need high-performance CPUs; examples:
 - ▷ Speech compression, filters & recognition algorithms
 - ▷ Video display and capture routines
 - ▷ Rendering routines & 3D graphics (geometry)
 - ▷ Image and video processing algorithms
 - ▷ Spatial (3D) audio
 - ▷ Physical modeling (graphics, CAD)
 - ▷ Encryption algorithms, complex arithmetics
- Most DSP algorithms (true for 1, 2 or 3D) have similar properties:

Huge arrays of small data types on which we should do the same thing (i.e. apply same instructions)

Looks like a perfect candidate for SIMD!

SIMD is found everywhere!

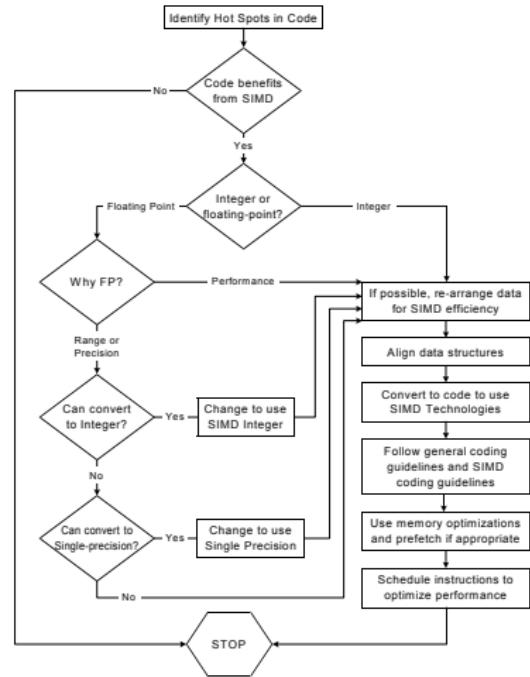
- HPC general purpose CPUs (Intel, AMD, Apple M1, etc.) & dedicated processors (GPUs from Nvidia, gaming CPUs etc.)
- Mobile processors: ARM's NEON (supported by Apple silicon); MIPS MDMX (MaDMAx) and MIPS-3D
- Example: **Cell Processor in Playstation**
 - ▷ Synergistic Processing Elements (SPE): 8 fully functional, but simplified cores, next to one PowerPC Processing Element (PPE) general purpose core
 - ▷ Each SPE has 7 execution units **including SIMD floating point unit**; SPEs are dual issue with max. 2 instructions issued in parallel in each cycle; no branch prediction
 - ▷ No cache, but 512KB of local memory



Is your application suited for SIMD?

Intel proposes the following chart to see if the SIMD is for you:

- Do we need to speed-up our code? If yes identify SW bottlenecks ...
- Do we use FP? (left branch)
 - ▷ Do we really need FP computations for precision? If not, convert to int
 - ▷ What is the FP size? Do we REALLY need FP of that size?;
- Can SW be vectorized? (right branch)
 - ▷ Could single instruction be applied to different data sets at the same time? If yes: **bingo** → program could be implemented in SIMD; note that this question is not that simple to answer, see Section 7



OM15156

SIMD usage overview

- **Disadvantages**

- ▷ Not all algorithms can benefit from data parallelism; once written, program is most likely to be architecture dependent, so less portable
- ▷ Register files & SIMD HW support in cores cost area; but looking at everything else (e.g. memory hierarchy) the overhead is acceptable
- ▷ If the program is referencing data in non-contiguous manner, this is a BIG problem (sometimes it can be solved, but not always)
- ▷ **No guarantees of automated use** – code vectorization is possible but not guaranteed see Section 7
- ▷ Programming can be difficult; even simple programs could take anything from little to significant development time (nobody likes that); time to market can be accelerated using pre-built libraries

- **Advantages**

- ▷ Computational speed → **10× improvements** are not uncommon!

3. SIMD extensions in Intel CPUs

Overview

- Intel has introduced SIMD computational paradigm in their general purpose CPUs for quite some time now; different SIMD extensions to standard instruction set have been proposed:
 - ▷ **MMX** – introduced in 1997 with Pentium MMX CPU
 - ▷ **SSE** – from 1999 to 2007 in NetBurst & Prescott CPUs
 - ▷ **AVX** – 1st and 2nd generation in 2016 until now
- These SIMD extensions differ in:
 - ▷ **Operand size** – increased from initial 64-bits to 256, and now 512 bits; this is $8\times$ in 20 years!
 - ▷ **Instruction set** – more instructions added to simplify SW; more HW, since more gates per cm^2 of an IC
 - ▷ **HW resources** – each extension introduces more memory & more dedicated computational resources
- All Intel SIMD extensions are **backward compatible**: CPU with AVX2 will run MMX, but it doesn't work the other way around
 - ▷ **What happens if you execute instruction that doesn't exist in ISA?**

MMX – MultiMedia eXtension

- Stands for **Multiple Math eXtension** and **Matrix Math eXtension**
- Idea: reuse FP ALU & Register File for SIMD; RF is the same as the regular one; just register name aliasing to save resources
- 12 FP registers are 80 bits wide, so 64 bits store **packed data**:
 - ▷ Assume double precision IEEE floating point format (64-bits)
 - ▷ Any of $2 \times 32, 4×16 -bits, 8×8 -bits integer$
 - ▷ Exclusive FP or SIMD operation, so no concurrent execution of FP & SIMD instruction (normal, no dedicated HW for SIMD)

$\leftarrow 64 \text{ bits} \rightarrow$								Vector
X								$1 \times 8 \text{ Bytes (FP)}$
X X								$2 \times 4 \text{ Bytes}$
X X X X								$4 \times 2 \text{ Bytes}$
X	X	X	X	X	X	X	X	$8 \times 1 \text{ Byte}$

SIMD instructions & data types

- Imagine that we have a CPU that can work in SIMD mode with data vectors of 64 bits (width of the SIMD ALU and registers)
- ISA provides `mov` instruction, that loads 64 bit word from memory into one of 12 `mmx` registers; memory transfers are data-type agnostic, they just manipulate words of bits
- **ALU can operate in 8 or 16-bit modes** (so different data-types) and let's assume add operation using some kind of SIMD instruction `padd`
- Because we have two different data types and no way to distinguish between single or 2-bytes operations, we need two different instructions to perform addition:
 - ▷ `paddb` – does the add operation on 1-byte elements
 - ▷ `paddw` – does the add operation on 2-bytes elements
- Decoded instructions generate ctrl signals to configure SIMD HW

SIMD instructions & data types: example

EDX and EBX point to 64-bit arrays (8×8 -bit words)

edx →	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1	0	mm0	mov mm0, [edx]
1	2	3	4	5	6	7	8												
7	6	5	4	3	2	1	0												
ebx →	<table border="1"><tr><td>15</td><td>13</td><td>11</td><td>9</td><td>7</td><td>5</td><td>3</td><td>1</td></tr></table>	15	13	11	9	7	5	3	1	mm1	mov mm1, [ebx]								
15	13	11	9	7	5	3	1												
	<table border="1"><tr><td>2055</td><td>1541</td><td>1027</td><td>513</td></tr><tr><td>1798</td><td>1284</td><td>770</td><td>256</td></tr></table>	2055	1541	1027	513	1798	1284	770	256	mm1	paddb mm1, mm0								
2055	1541	1027	513																
1798	1284	770	256																
	<table border="1"><tr><td>3853</td><td>2528</td><td>1797</td><td>769</td></tr></table>	3853	2528	1797	769	mm0	mov mm0, [edx]												
3853	2528	1797	769																
		mm1	mov mm1, [ebx]																
		mm1	paddw mm1, mm0																

Make sure you understand the result of the code above !

SSE – Streaming SIMD Extension

- MMX evolution, introduce 128-bit wide registers: xmm0-xmm7
- Not just new names: they are not any more aliased FP registers but a **dedicated Register File** (thanks CMOS scaling!)
- Extra floating point instructions added to ISA, so:
 - ▷ Data movements: MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS
 - MOVAPS, MOVUPS – aligned & non-aligned memory accesses we will discuss this more in details in Section 5
 - ▷ Arithmetic: ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS
 - ▷ Compare: CMPSS, COMISS, UCOMISS, CMPPS
 - ▷ Data shuffle and unpacking: CVTSI2SS, CVTSS2SI, CVTTSS2SI
 - ▷ Bitwise logical operations: ANDPS, ORPS, XORPS, ANDNPS

SSE versions

SSE extensions evolved in time:

- SSE2 – 2001, Pentium4
 - ▷ New **math instructions** for double precision (64-bit) FP
 - ▷ SIMD operations on any data type: from 8-bit integer to 64-bit float entirely with `xmm` vector-register file, no need to use the legacy MMX or FPU registers
- SSE3 – 2005, Prescott (variant of Pentium4)
 - ▷ Added specific Digital Signal Processing & 3D graphics instructions
 - ▷ **More instructions that enable easy manipulation of the words inside `xmm` registers** (array elements)
 - ▷ Better transfer for **unaligned memory access**
- SSE4 – 2006 in various flavors
 - ▷ ... more general purpose instructions, not only multi-media

AVX – Advanced Vector Extensions 1st generation

- Next major evolution of SSE (2011)
- Main features:
 - ▷ Data path increases from 128 bits to **256 bits** with Register File of **16x256 words** named `ymm0` to `ymm15`
 - ▷ Before all instructions on 2 operands only with the following format:
 $xmm0 \leftarrow xmm0 + xmm1$
i.e., one source & one destination registers **must be the same**; the above will erase the content of `xmm0`; if you want to keep `xmm0` value, you need to copy its content in the Register File; data pollution will cause more frequent memory access & performance loss
 - ▷ **AVX introduces 3-operand instructions:**
 $yymm2 \leftarrow ymm0 + ymm1$
`ymm0 & ymm1 remain untouched since dst is now ymm2`
- AVX requires OS support (it will not work with older OSs)

AVX – 2nd generation

- AVX2 – yet another evolution (2013); introduces **512 bits** wide operands for even bigger vectors
- AVX2 became AVX-512 (2015)
 - ▷ And we will stop here!!!
- Register File is now composed of **32x512** words; this is double compared to normal AVX in terms of number of words and word size; register names: zmm0 to zmm31
- Previous 128-bit registers (xmm0 to xmm31) and 256-bit registers (ymm0 to ymm31) are sub-set of the above SIMD Register File
- Different instruction sets for different CPUs family; this is new since in the past entire SIMD instruction set of a given extension family has been supported by all CPUs in the same generation
- Supports 4-operands operation
- This is too many options! How to program becomes a challenge

4. Identifying the CPU

How to write portable SW for so many SIMD ISA?

- There is no magic: architecture dependent optimizations, those that use the most of a given CPU architecture & SIMD extension, do mean that you need to write a **piece of code for each CPU variant** you may encounter (i.e. SIMD extension)
- First, you need to identify your CPU exactly; for Intel CPUs you can use assembly instruction cpuid to identify the exact architecture you are running
- When called, this instruction will fill standard registers with codes that are unique for a given CPU architecture & model
 - ▷ Depending on the value of first 4 MSBs in EAX, different info may be gathered
 - ▷ If $EAX=0$ then you get a manufacturer ID string – 12 ASCII string stored in EBX, EDX, ECX
 - ▷ If $EAX=1$ then you will get extended processor info in EAX and feature bits in EBX, EDX, ECX

Intel CPU families and their exact identification

- Assume EAX=1 before cpuid call → following info found in EAX, 32-bits register (here we limit us to EAX content)

EAX															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				Extended Family ID								Extended Model ID			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CPU type		Family ID				Model				MaskID			

- Exact CPU model is derived from Model, Extended Model ID and Family ID fields (we have many options)
- Very low-level information since even mask set (MaskID) for wafer processing is specified!

Putting everything together

Using manual processor dispatch method:

- First: `__declspec(cpu_dispatch(cpuid, cpuid, ...))` is used to provide a list of targeted CPU architectures, along with an empty function body (AKA **function stub**)
- Then `__declspec(cpu_specific(cpuid))` is used to declare each function version targeting particular CPU architecture
- Intel CPU type is detected at runtime using `cpuid`, and the corresponding function version is executed

```
1 #include <stdio.h>
2 __declspec(cpu_dispatch(generic, future_cpu_16)) // list of functions
3 void dispatch_func() {}; // empty function stub
4 __declspec(cpu_specific(generic)) void dispatch_func() {
5     printf("Code for non-Intel processors and generic Intel\n");
6 }
7 __declspec(cpu_specific(future_cpu_16)) void dispatch_func() {
8     printf("Code for 2nd generation Intel Core processors goes here\n");
9 }
10 int main() {
11     dispatch_func();
12     printf("Return from dispatch_func\n"); return 0;
13 }
```

5. Memory alignment

On memory transfers – top-down

- Central memory (DRAM) is accessed using 64-bit words; because of DRAM technology and DDR protocols access to memory is more complicated than just simple R/W/Clk; we need a full blown digital circuit to enable DRAM access → **DRAM controller**
- DRAM pin count limited by the cost, BW limited by the interface speed; HBM_s introduce larger interfaces with more data pins since IC-package level integration
- On-chip SRAM cache interfaces may be larger, since implemented at IC-level (we can have more wires running at higher speed)
- For all these memories accessing single byte is far from being optimal; we want to perform **bursts of data transfers**; this optimizes transfer time/energy per bit (byte)
- Processors are designed so that **memory transfers from certain addresses are more efficient** (and we know memory transfers are important for SW performance because of latency involved)

Memory alignment

- Memory address A is said to be aligned if $(A + 1) \bmod n = 0$, where n is the width of the accessed data in bytes
- When a memory address is misaligned, the value $(A + 1) \bmod n$ determines the **offset** from the alignment: HW needs to “extract” data using this offset
 - ▷ Assume 1 Byte address and 8-Bytes address alignment; addresses $0x0$, $0x7$, $0x10$ are aligned, but not $0x3$

0x															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
↑			↑				↑								↑

- Intel CPUs (AVX-512) are designed so that **memory movements** are optimal if performed on 64-byte boundary aligned addresses

How to align data?

- Consider allocation of a static variable: the use of appropriate clause/attribute tells the compiler what to do with the address

```
1 __declspec(align(64)) float A[1000];
```

- For dynamic allocation you need to use dedicated malloc functions; in case of Intel compilers `_mm_malloc()` and `_mm_free()`; other compilers do provide similar functions (program is not only architecture, but also compiler dependent)

```
1 char *buf;
2 buf = (char*) _mm_malloc(bufsizes[i], 64);
3 ...
4 _mm_free(buf);
```

- Allocating aligned memory is not enough, compiler should now addresses are aligned: `__assume_aligned(a, 64);` **compiler can use instructions for aligned memory accesses!** (slide 26)

Aligning composite data types

- Consider following structure to be used in an array:

```
1 struct myStruct {  
2     short a, b;           // 2x2 Bytes  
3     int   c, d;          // 2x4 Bytes  
4     unsigned char e[4];    // 4x1 Byte  
5 };
```

- We know data will be allocated contiguously; some data elements will not be aligned, & their access slow; example with 8-bytes alignment:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
a	a	b	b	c	c	c	c	d	d	d	d	e	e	e	e
↑		↑		↑				↑				↑	↑	↑	↑

- Solution – insert “empty bytes” to enable aligned access to each element in the structure → **padding** (manual or compiler); we waste memory resources, but memory is sometimes cheap

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	...
a	a	0	0	0	0	0	c	c	c	c	0	0	0	0	d	...
↑		↑		↑			↑								↑	...

SIMD and memory alignment

- In SIMD efficient vector move is strongly desirable because of the operand sizes (AVX-512 deals with vectors of 512 bits)!
- Memory moves: aligned `movaps` & non-aligned `movups`
- Loop with aligned access

```
1 movaps xmm0, [b] ;           load 16 bytes from b[]
2 addps  xmm0, [c] ;           add xmm0 with c[]
3 movaps [a], xmm0 ;           store xmm0 in the a[]
```

- Loop with non-aligned access

```
1 movups xmm0, [b+2] ;         load 16 bytes from b[]
2 movups xmm1, [c+3] ;         load 16 bytes from c[]
3 addps  xmm0, xmm1 ;          xmm0 = xmm0 + xmm1
4 movups [a+1], xmm0 ;         store xmm0 in the a[]
```

- Important note: **use of aligned memory on non-aligned address will most likely cause your program to crash**

Importance of aligned access

- Table shows MMX (Pentium3) vs. SSE (Pentium4) aligned memory accesses acceleration against non-aligned accesses, for different data types & array size¹

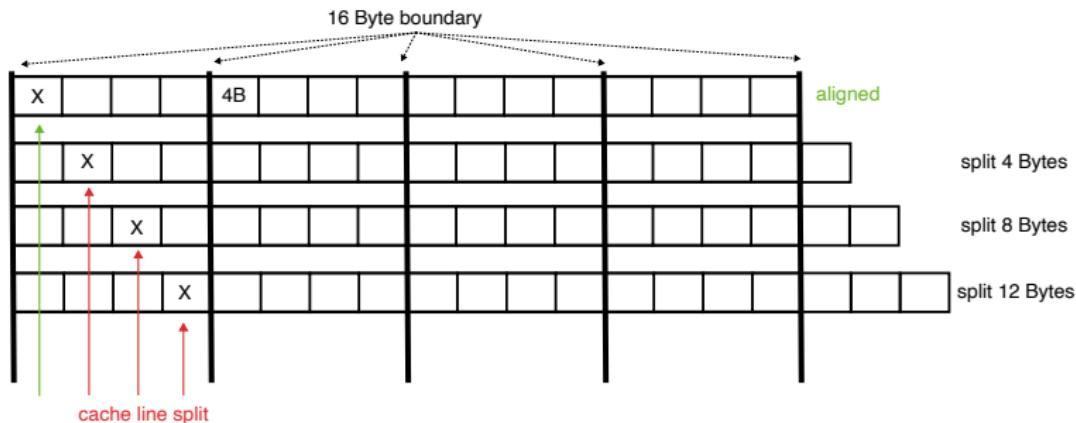
Array size	char		short		integer		float	
	MMX	SSE	MMX	SSE	MMX	SSE	MMX	SSE
256	1.46	1.64	1.58	1.8	1.05	1.71	3.37	4.73
512	1.58	1.78	1.66	2.01	1.1	1.91	3.72	5.46
1024	1.66	2.02	1.74	2.24	1.08	2.04	4.00	3.96
32768	1.13	2.7	1.12	2.10	1.1	1.91	1.41	2.47
64000	1.28	2.71	1.23	2.20	1.16	1.04	1.45	1.13
1048576	1.15	1.83	1.11	1.99	1.1	1.25	1.36	1.53
4194304	1.15	1.94	1.11	2.00	1.08	1.15	1.38	1.24
16777216	1.14	2.24	1.12	1.89	1.09	1.15	1.37	1.23
Average	1.32	2.11	1.33	2.03	1.09	1.52	2.26	2.72

Up to 3× → this is significant!

¹Source: *Performance Impact of Misaligned Accesses in SIMD Extensions*

Memory alignment & caches

- Assume 64-byte wide cache line and 16 byte address alignment; 4 data blocks form 1 cache line; 1 cell below is 4Bytes wide
- If data accesses at 64 byte boundary, cache is well used because read operation is at the boundary and will use **all 64 bytes**
- If not, then **two cache line reads are needed** to allow the access to the data – **cache line split**; this will affect performance



6. SIMD programming for Intel CPUs

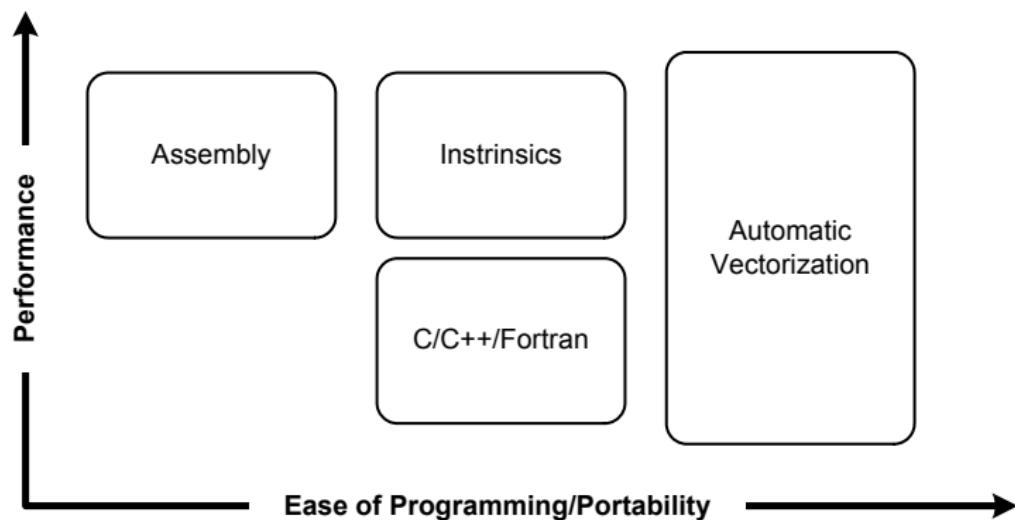
How to use SIMD in Intel CPUs?

After all there are not that many options:

- a. **Assembly** – do it the hard way! use assembly instructions and program CPU directly; instructions could be CPU architecture dependent: you may use more recent SIMD extensions with new instruction set, but the code will not run on older CPUs
- b. **Intrinsics** – assembly instructions are abstracted with C primitive functions; close to assembly, but just a little bit more user friendly & portable; just mentioned here ...
- c. **Intel Performance Libraries** – already implemented functions, supplied in libraries; you just need to interface them and compile your code; generic functions, could be further optimized, but best possible time-to-market option for SIMD
- d. **Automatic vectorization** – some compilers can do this, but don't expect miracles (we will discuss this a little bit more)

Trade-off for different SIMD

SIMD programming options → **performance vs. ease of use:**



As usual → no free lunch – assembly best performance, but worse ease of use; automated vectorization could reach good results but poor ones too! (other solutions in between)

a. Assembly & in-line assembly

- You could use assembly & compiler, but more user friendly solution is **in-line assembly** → insert assembly at any point in C/C++ code
- Example – simple loop in C:

```
1 void add(float *a, float *b, float *c) {  
2     for (int i = 0; i < 4; i++)  
3         c[i] = a[i] + b[i];  
4 }
```

becomes using in-line SIMD assembly:

```
1 void add(float *a, float *b, float *c) {  
2     __asm {  
3         mov    eax, a  
4         mov    edx, b  
5         mov    ecx, c  
6         movaps xmm0, xmmword ptr [eax] // note new move instructions  
7         addps  xmm0, xmmword ptr [edx] // note new processing instruc.  
8         movaps xmmword ptr [ecx], xmm0 // Pointers must be aligned!!!  
9     }  
10 }
```

- This is what we are going to do ...

b) Intrinsics

- **Intrinsics** – predefined C functions that map directly assembly instructions for easier use (no need for assembly, compiler is OK)
- Performance could be as good as assembly, but easier to write
- Compatible across different compilers (if they support intrinsics)

```
1 void add(float *a, float *b, float *c) {  
2     for (int i = 0; i < SIZE; i++) { !\h1{//Loop in C}#!  
3         c[i] = a[i] + b[i];}  
4 }
```

```
1 #include <xmmmintrin.h>  
2 void add(float *a, float *b, float *c) {  
3     _m128 t0, t1;           // XMM 128 bit registers  
4     t0 = _mm_load_ps(a);    // load 128 bit data  
5     t1 = _mm_load_ps(b);  
6     t0 = _mm_add_ps(t0, t1); // SIMD add  
7     _mm_store_ps(c, t0);  
8 }
```

Note `xmmmintrin.h` header file! Why do we need this?

Practical assembly & intrinsics

- We have seen that SIMD instruction sets become richer with every new extension; old instructions are kept in ISA for backward SW compatibility
- If in the past you missed a function & you blew your brains out to figure out the vector (SIMD) algorithm for it (personal experience), in the next SIMD generation this function might appear as a single instruction with appropriate HW support
- There is a chance that the above HW will further improve the SW performance & not just ease up code writing
- **Advice → you need to be up to date with most recent developments: you need to read documentation**
- Intel is very good in making documents, assuming that you do the effort; there is lot of documentation with many, many pages, with loads of useful information; soft-skill: how to browse documentation quickly & find what you are looking for

c) Intel Integrated Perf. Primitives – IPPs (2019)

- What it is?

Highly optimised SW library that provides a comprehensive set of application domain specific plug-in functions to outperform any compiler specific optimisation; use of Streaming SIMD Extensions (SSE), Advanced Vector Extensions 2 (AVX2), and Advanced Vector Extensions 512 (AVX-512) instruction sets; target different Intel processors: Atom, Core, & Xeon

- How to use?

With Intel Parallel Studio XE and System Studio (Intel IDEs)

- What do they cover?

Low-level building blocks for signal (1D) signal processing & (2D, 3D) image processing, computer vision and data processing (data compression / decompression and cryptography) applications

Example of IPPs usage

- Function `ippsTriangle_16s` generates 1D triangular signal with specified frequency `rFreq`, phase pointed by `pPhase`, and magnitude `magn` argument
- Function computes `len` samples of the triangle, and stores them in the array `pDst` (pointer to a pre-defined data type)
- Triangle: asymmetric if h in range $(-\pi, \pi)$; symmetric if $h = 0$

```
1 void func_triangle_direct() {
2     Ipp16s* pDst;           // predefined specific data types
3     int len = 512;
4     Ipp16s magn = 4095;
5     Ipp32f rFreq = 0.02;
6     Ipp32f asym = 0.0;
7     Ipp32f Phase = 0.0;
8     IppStatus status;
9
10    status = ippsTriangle_16s(pDst, len, magn, rFreq, asym, Phase);
11    if(ippStsNoErr != status)
12        printf("Intel(R) IPP Error: %s",ippGetStatusString(status));
13 }
```

d) Automated vectorization

- Some compilers provide support for **automated SIMD code generation** out of standard C statements
- Note that this **can** be done only in specific cases, when the C-code is written in such way that the compiler can effectively vectorize the code
- We will see how SIMD is sensitive to algorithm; compiler can't change the algorithm, or SIMD is always about another way of doing things
- Automated vectorization is often used in conjunction with compiler directives, i.e. **user defined hints** to compiler to simplify vectorization process (just think of memory alignment)
- Bottom line → don't expect that something like:
`compile -auto_vectorise input.c` works out of the box, even if some compilers may have the switch

7. Automated compiler vectorization

Background

- Almost all compilers provide some kind of `-O` switch to turn on **automated optimization** with the aim to improve performance of the produced executable
- Take Intel C++ Compiler – ICC (similar with `gcc` or `MSVisualC`):
 - `-O0` No optimization
 - `-O1` Optimise for size
 - `-O2` Optimise for speed, note exclusivity with above
 - `-O3` Enable O2 + intensive loop optimisations
 - `-msse3` Enables SSE for non-Intel CPUs
- Thus, optimizations may include **SIMD code generation** – ICC will look for vectorization opportunities **mostly on loops** whenever you compile with `-O2` or higher switch; works for Intel & non-Intel CPUs, but possibly much better for Intel micro-architecture, Intel knows ☺
- Vectorization may be explicitly disabled with `-no-vec` to make direct performance comparison of the executable
- Vectorization report may be generated with `-vec-report`:

MultArray.c(92): (col. 5) remark: LOOP WAS VECTORIZED

1. Loops must be countable

- Loop count must be known only at the entry to the loop, so at run-time, but not at compile-time
- Counter could be a variable, but the variable **must remain constant for the duration of the loop**
- This also implies that exit from the loop **must not be data-dependent**

```
1 SIZE = z*y;
2
3 for (j = 0; j < SIZE; j++) {
4     b[j] = a[j] * x[j];      // this should be ok
5 }
6 for (j = 0; j < SIZE; j++) {
7     b[j] = a[j] * x[j];
8     SIZE = b[j];            // NOT OK, SIZE can't be mod
9 }
```

2. Single entry and single exit

- Implied by previous; code below can't be vectorized due to the second data-dependent exit (branch):

```
1 void no_vec(float a[], float b[], float c[]) {  
2     int i = 0;  
3     while (i < 100) {  
4         // this should be ok  
5         a[i] = b[i] * c[i];  
6     }  
7     while (i < 100) {  
8         // but this is NOT ok  
9         // data-dependent exit condition:  
10        if (a[i] < 0.0) break;  
11  
12        // Can you write a condition on single SIMD element?  
13        ++i;  
14    }  
15 }
```

3. Straight-line code

- Different iterations MUST have same control flow, i.e. they must not branch; however, if statements may be vectorized if they can be implemented as **masked assignments**: the calculation is performed **for all data elements**, but the **result is stored only** for those elements for which the mask evaluates to true

```
1 #include <math.h>
2 void quad(int length, float *a, float *b,
3 float *c, float *restrict x1, float *restrict x2) {
4     for (int i=0; i<length; i++) {
5         float s = b[i]*b[i] - 4*a[i]*c[i];
6         if ( s >= 0 ) { // skipped in SIMD, we compute everything
7             s = sqrt(s) ;
8             x2[i] = (-b[i]+s)/(2.*a[i]);
9             x1[i] = (-b[i]-s)/(2.*a[i]);
10        } else { // and this is then masked
11            x2[i] = 0.; // make sure you understand how
12            x1[i] = 0.; // this can be done
13        }
14    }
15 }
```

4. Nested loops must be independent

- There must not be **loop carried dependence** – when code in a loop iteration depends on the output of previous loop iteration
 - ▷ This is a general problem in loop parallelization

```
1 // True even in a single loop, look at example below
2 // 1st iteration: a[2]=a[1];
3 // 2nd iteration: a[4]=a[2] -> uses a[2] modified previously
4 for (i = 1; i < N; i++)
5     a[2*i] = a[i];
6
7 // Both indexes will create the above situation
8 for (i = 0; i < N; i++)
9     for (j = 0; j < N; j++)
10        a[i+1][j-2] = a[i][j] + 1;
```

5. No function calls

- It is not possible to call other functions from the loop; even a `printf` will make a loop non vectorizable

```
1 for (i = 0; i < N; i++) {  
2     c[i] = a[i] * b[i];  
3     printf("%d", c[i]);  
4 }
```

- In the above case you may be prompted with a message:
nonstandard loop is not a vectorization candidate
- Exceptions:
 - ▷ Intrinsic math functions (they are already SIMD)
 - ▷ In-line functions – these are “hard” copies of functions inserted directly in the compiled code; for such functions there will be no function calls and push/pop → so no function call overheads

Obstacles to vectorization 1/4

1. Non-contiguous Memory Accesses – SIMD vectorization killer

- ▷ 64 or 128 bits can be loaded directly from memory in a single SSE instruction **only if they are adjacent**; and this is efficient
- ▷ If not, we need multiple load instructions (slows down everything)
- ▷ Examples: non-unit step or indirect memory access; compiler rarely vectorizes such loops, unless the amount of computational work is large compared to the overhead from non-contiguous memory access

```
1 // arrays accessed with step 2
2     for (int i=0; i<SIZE; i+=2)
3         b[i] += a[i] * x[i];
4 // Inner loop accesses array a with inverted indexes
5 // This will also guarantee cache miss if SIZE is big
6     for (int j=0; j<SIZE; j++) {
7         for (int i=0; i<SIZE; i++)
8             b[i] += a[i][j] * x[j]; // i,j indexes inverted
9     }
10 // Indirect addressing of x using index array
11     for (int i=0; i<SIZE; i+=2)
12         b[i] += a[i] * x[index[i]]; // x[index[i]]
```

Obstacles to vectorization 2/4

2. Data dependencies – vectorization will change execution order, so it can do so only if it preserves results of computations!

Simplest case is when data elements that are written do not appear in any other iteration; all the iterations of the original loop are independent of each other, and can be executed in any order, without changing the result

More generally we can have four options depending on Read or Write order (this is known from pipeline hazards)

- 2.1 Read-after-Write – next step uses result from the previous one:

```
1 A[0]=0;
2 for (j=1; j<MAX; j++)
3     A[j]=A[j-1]+1;
4 // Loop moves into this direction -->
5 // A[1]=A[0]+1; A[2]=A[1]+1; A[3]=A[2]+1; A[4]=A[3]+1;
6 //   ^_____
7 //           ^_____
8 // ...
```

Obstacles to vectorization 3/4

2.2 **Write-after-Read** – When a variable is read in one iteration and written in a subsequent iteration, this is a write-after-read dependency (AKA anti-dependency):

```
1 for (j=1; j<MAX; j++)
2   A[j-1]=A[j]+1;
3 // A[0]=A[1]+1; A[1]=A[2]+1; A[2]=A[3]+1; A[3]=A[4]+1;
4 //                                     ^---> written after being used
```

- ▷ Above is not safe for general parallel execution, since write may occur before the read, if we have OoO execution for example
- ▷ However vectorization is safe since no iteration with a higher value of j can complete before an iteration with a lower value of j
- ▷ Following may not be safe: vectorization might cause some elements of A to be overwritten by the 1st before being used in 2nd SIMD instruction

```
1 for (j=1; j<MAX; j++)  {
2   A[j-1]=A[j]+1;
3   B[j]=A[j]*2;          // this one is problematic
4 }
5 // A[0]=A[1]+1; A[1]=A[2]+1; A[2]=A[3]+1; A[3]=A[4]+1;
6 // B[1]=A[1]*2 ---^
7 // ---> A[1] may be written after being used 2nd SIMD instruction
```

Obstacles to vectorization 4/4

- 2.4 **Read-after-Read** – not really dependency, will not prevent vectorization; if variable is not written, it doesn't matter how often it is read
- 2.5 **Write-after-Write** – same variable is written to in more than one iteration; this is unsafe for parallel execution !

```
1 // This is ok because we can accumulate the sum
2 sum=0;
3 for (j=1; j<MAX; j++)
4     sum = sum + A[j]*B[j];
5 // Write the SIMD code yourself (after next session)
6
7 // Anti-example: typical case of vectorisation
8 for (i = 0; i < size; i++) {
9     c[i] = a[i] * b[i];
10 // The above will work only if c, a and b
11 // are non-overlapping pointers
12 // If c, a and b are possibly overlapping
13 // compiler will not be able to vectorise
14 // you need to supply a hint; see next slide
15 }
```

Pragmas – or how to help compiler vectorize

- Compilers can't make miracles ... the above examples show the degree of complexity they have to deal with
- No compiler could solve all problems by itself: welcome to **pragma directives** – user specified hints to improve vectorization and minimize compile time; **why these two relate?**
- By inserting **#pragma ivdep** before the loop; compiler will know that it can safely ignore any potential data dependencies

```
1 #pragma ivdep // we know what are we doing
2 for (i = 0; i < size; i++) {
3     c[i] = a[i] * b[i];
4     // The above will work !
5                         // a, b, c non overlapping pointers!
6     // i.e. data is independent, compiler can safely ignore
7 }
```

- The compiler will not ignore proven dependencies
- Use of this pragma when there are in fact dependencies may lead to incorrect results; **suggest an example?**

Examples of other hints

- `#pragma loop count (n)` – used to advise the compiler if the loop is worth while considering for SIMD optimization; note that it is not efficient to vectorize small loops
- `#pragma vector` – forces compiler to vectorize the loop if it is safe to do so, whether or not the compiler thinks that the vectorization will improve performance
- `#pragma vector align` – asserts that data within the following loop is aligned; generally to 16 byte boundary, for SSE instruction sets
- `#pragma novector` – asks the compiler not to vectorize a loop; you will use this when you know that it is not worth trying (and you want to save compile time)
- `#pragma vector nontemporal` – gives a hint to the compiler that data will not be reused, and therefore to use **streaming stores** that bypass cache and accelerate memory accesses

Streaming stores

- In vector processing input data is intended for one time-usage: once a part of the vector is processed, we do not need it any more
- As opposed to **temporal data**, data will be used again (& the reason why we have caches) we have **non-temporal data**
- Keeping all, so non-temporal data too, in the cache would make no sense (we speak about **cache pollution**)
- This motivates so called **non-temporal streaming stores**
- We have a set of **Streaming Load/Save Buffers** close to CPU (just like L1); fast memory, accessed directly from the main memory & bypassing cache hierarchy (we can do that since data is used once)
- If data access can be anticipated early enough, these buffers provide continuous stream of data to CPU with improved bandwidth
- Streaming stores performed using non-temporal move instructions:
MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, MOVNTPD (**for ref. only**)

Some references

Some pointers to explore optimization aspects more in details:

- *Intel 64 & IA-32 Architectures Optimization Reference Manual* – general optimization guidelines, good even if you program in high-level language such as C; makes a link between SW & HW (it is Intel specific, but some tricks applicable to AMD)
- *Intel SSE4 Programming Reference* – provides details about SIMD programming and execution with SSE; you can assume SSE4 being widely supported
- *Developer Guide for Intel Integrated Performance Primitives* – there is a collection of documents covering different application domains from 1D to 2D and security; enumerates all functions, their interfaces and how to use them

And of course there is much, much more ... you have to search

ELEC-H-473

Th08: Multi-processors & Thread Level Parallelism

Dragomir Milojevic
Université libre de Bruxelles

2023

Context

- Until now we focused on **uni-processors** system architecture: that is one execution pipeline, one CPU assuming one IC package
- We saw that even uni-processors (SISD) implement parallelism:
 - ▷ **ILP** – at instruction level
 - ▷ **Super-scalars** – multiple ALUs in the same core
 - ▷ **Data-parallelism (SIMD)** – vector computation with one ALU
- We also defined instruction/data relations and introduced Flynn's taxonomy of computer architectures:
 - ▷ **SISD** – standard basic architecture
 - ▷ **SIMD** – vector computations adapted for signal processing
 - ▷ **MIMD** – i.e. parallel computers, these days even general purpose multiprocessors desktops/laptops, servers, clusters & data-centers

Last group was not covered ...

(Or they are all around us, since the core operating F stopped scaling)

Today

1. Multiprocessor architectures
2. Interconnects for multiprocessors
3. Cache coherency
4. Snooping cache coherence protocols
5. Directory cache coherence protocols
6. Processes and threads
7. Simultaneous Multi-Threading

1. Multiprocessor architectures

From Single to Multiple

- Basic Von Neumann computation model uses a **single instance** of the Computational, Control and Memory modules in a box that we called a CPU
- If you want to provide more computational power, and we always want more of it (to sequence human genome or run some AI), we can eventually **multiply instances** in the above model and exploit **parallelism** at higher architectural levels – we thus introduce **multiprocessor systems**
- Parallel computing is not a new kid in town! Highly parallel machines were built already in the early '60
- Parallel computing became de-facto standard even for low-end domestic computers since early 2000, to compensate for poor frequency scaling of the CMOS
- Just look at your smartphone or smartwatch!

Terminology

So far terminology used was vague so let's first agree what is what and let's use this from now on:

- **Processors** – generic term (e.g. Von Neumann model)
- **Core** – complete instruction pipeline (F, D, Ex, W), with one or more ALUs (super-scalar) with ILP and eventually SIMD
- **CPU** – is a single **integrated circuit package**
 - ▷ Interconnects with the external world using pins and macroscopic wires that are slow, how slow will depend on wires and distance
 - ▷ Interconnects within the package are microscopic wires, so much faster than macroscopic ones!
 - ▷ Multi CPUs exist at different levels: from few IC packages (CPUs) on the same motherboard connected with visible but short wires, to data-centers and very long wires (or optical wires for speed)

CPUs integrate multiple cores and enable
multi-/many-cores computational paradigm that
materializes the MIMD model

Core centric classification

Depending on the architecture of different cores in the system

- **Homogeneous systems** – all cores have same functional specification, so same HW components, same ISA, same instruction set; any program could run on any of the cores
 - ▷ To be effective we need a constant, predictive computational load
 - ▷ Good for high-performance servers, super-computers, computing intensive apps, although load balancing is a problem of its own
 - ▷ Generally big integrated systems with very regular physical structure
- **Heterogeneous systems** – CPUs have different functional specifications, so different HW components, ISA, instruction sets
 - ▷ Good for variable load and where the power is of major concern
 - ▷ Ideal for mobile applications (from idle to YouTube video decoding)
 - ▷ But then look at Apple M1 used in standard desktop computers: they follow the same approach – power reduction, application driven core choice, ease of ecosystem handling (same core architecture for mobile & desktop)

Communication centric classification

Depending on the interconnect and the memory accesses:

- **Symmetric Multi-Processors (SMP)** – if all CPUs are equal AND have exactly the same access time to the memory
 - ▷ In general assumes homogeneous core architecture
 - ▷ OS treats all cores equally, meaning that all cores access all devices equally (no functional specialization)
 - ▷ This is what is happening in your computer with a multi-core CPU
- **Asymmetric Multi-Processing (ASMP)** – if we have some kind of functional preference
 - ▷ Core architectures are specialized for some functions, so we speak of heterogeneous systems: e.g. one device acts as IO processor only, or floating point processor

While today systems are considered to be SMP most of the time, architecture heterogeneity is a must for low-power applications and recently in desktop computing (even in Intel CPUs)

Memory centric classification

- But (**very important**) question is: What happens to the system (central) memory (and of course caches)?
- This is tightly coupled to how different cores/CPUs handle the **address space** and how program sees it; we can have:
 - ▷ **Shared address space** – the same address (pointer) refers to the same data for ALL cores/CPUs (same physical address)
 - ▷ **Distributed address space** – each CPU has his own address space, i.e. same address points to different memory locations
- Following such address space handling we can then have:
 - ▷ **Shared Memory Systems** – all processor access one memory (the number of memory circuits is not important, it is the address space that counts and how cores/CPUs sees this space)
 - ▷ **Distributed Memory Systems** – each CPU has his own private memory corresponding to his own private address space

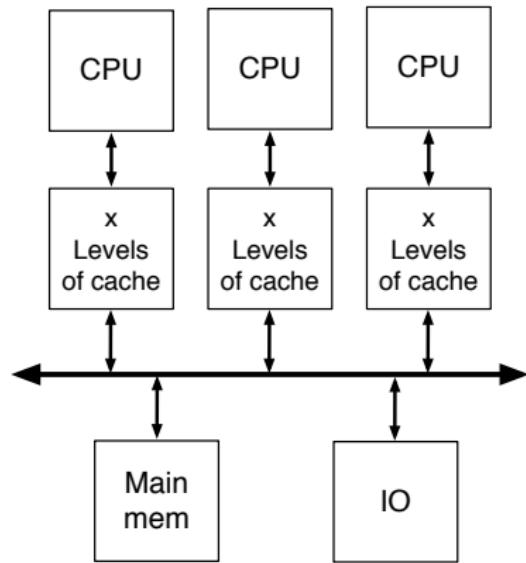
Different system variants are then possible ...

A. Centralized Shared-Memory (CSM) (1/2)

- Multiple processors are connected to a **single centralised memory** using some communication infrastructure
- If any CPU can access any central memory location with **the same latency** – **Uniform Memory Access (UMA)** machines
 - ▷ Nice for general purpose & time sharing applications with multiple users of computer programs
 - ▷ But pure UMA is not always possible, **Can you explain?**
- **Bad news for CSM:**
 - ▷ Clearly centralized access creates **bottleneck** at main memory (or at the communication infrastructure level)
 - ▷ We also need to add **communication latency** to the latency of each memory access which will cause additional loss of performance
 - ▷ **Scalability** issues with larger number of processors: good for small number of processors, becomes bad with the increasing number of CPUs (burden of any centralized architecture: centralized node is always a bottleneck)

A. Centralized Shared-Memory (CSM) (2/2)

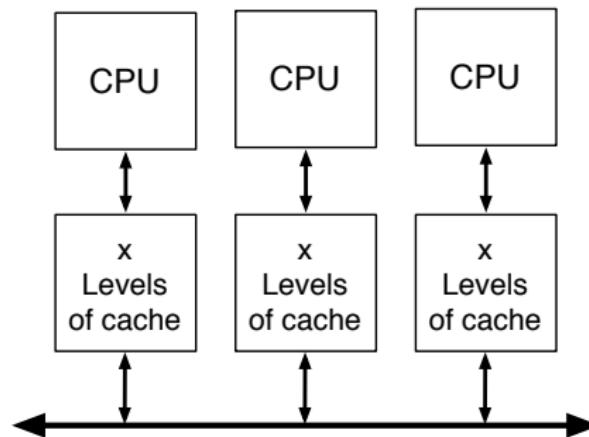
- Lower cache levels are embedded in cores (L1 in general) & **not shared** – **private cache**
- Upper cache layers are shared ...
- Since the address space is shared, **more than one cache** may have a copy of a given main memory location
- Causes **cache coherence problem**: different CPUs can write in their local caches that reference same location in the main memory
- Such architectures **must solve this** to ensure correct SW operation



Same content can be copied to multiple memories

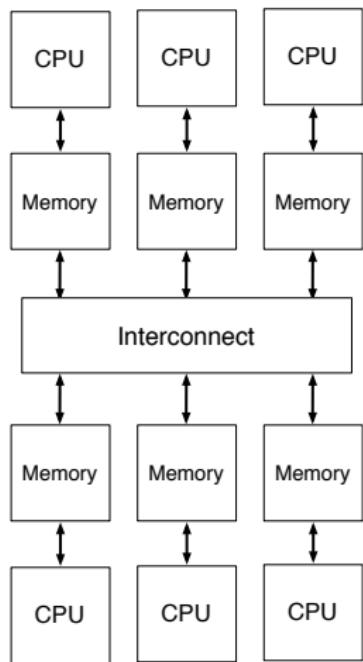
B. Cache-Only Memory Arch. COMA

- Simple solution to solve cache coherency problem: multiprocessor systems **without central memory**
- DRAMs are used as huge least level cache memory
- Motivation: remove redundant copies from memory hierarchy
- Data access to another node causes **data migration**
- More of a concept, cool for research, although some practical (hybrid) implementations have been reported in literature



C. Distributed Shared Memory systems (DSM)

- Each CPU has it's own Memory (and/or IO access point) but the address space is the same for all processors!
- CPUs interconnected using some kind of interconnect (more on this just after)
- Advantages: more bandwidth, not limited when scaled-up, less latency
- Problem: to get the data from one node to another involves inter-CPU communication that uses explicit load/store operations (complicated)
- Latency varies depending on the pair of nodes that communicate: Non-Uniform Access Machine (NUMA), as opposed to UMA (system size dependent)



D. Message Passing Machines

- Assume now that each CPU has his own **completely separated address space!**
- Communication between nodes is also explicit but uses **messages** to exchange data between a pair of CPUs
- **Process Granularity**=computation time/communication time
 - ▷ **Coarse** – each process holds a large number of sequential instructions and takes a substantial amount of time to execute
 - ▷ **Medium** – in the middle, communication overhead is reduced
 - ▷ **Fine** – each process contains a few sequential instructions
- Message passing multi-processor architectures target medium & coarse process granularity
- System design should focus on **communication infrastructure** since crucial for system performance; communication could be standardized (synchronous or asynchronous)

Summary of parallelism, today's perspective

- Classifications tend to create rigid placeholders & frontiers that rarely match what is really happening there
- Real life systems use mixture of the above concepts to trade-off different parameters and take the best of different worlds
- Let's identify the concepts on some real examples
 - ▷ Mobile processors
 - Are highly heterogeneous computer architectures
 - They integrate high-performance, low-power, GPU & AI dedicated, security HW components
 - Memory wise they combine CSM & DSM (GPU memory vs. general purpose cores memory)
 - ▷ Server processors
 - Most of the time homogeneous core architectures
 - Mostly SMPs, though some access times could vary
 - High level of parallelism

Structuring multiprocessors

- Is a matter of **scale** (i.e. the number of cores) and type of **interconnect** structure used
- Multiprocessors are organized in **hierarchical** manner to simplify individual IC implementation, full system deployment and maintenance
- Data-centers & supercomputers would have the following levels of hierarchy:
 - ▷ **IC** – currently circuits with up to hundreds of processing units (general purpose or specialized cores); parallelism limited by the die size of an IC (typically less than 700mm^2); implement multi-/many-core paradigm; limited by the size of cache memories
 - ▷ **Board/Rack** – from few to hundreds of ICs in the same housing with macroscopic, though short interconnects
 - ▷ **Building** – thousands of ICs built using multiple boards/racks in the same room/building; interconnect is long, so communication overheads become very important; cooling is a serious issue

Multiprocessors – the issue

- **Scalability** – question whether the system size can be expanded with a **linear increase** in performance & cost by simply increasing the number of processors used
 - ▷ We will see later a fundamental showstopper in the next slide !!!
- Different scalability attributes:
 - ▷ **Size** – measures the maximum number of processors a system can accommodate
 - ▷ **Application** – ability to run application with improved performance on a scaled-up version of the system
 - ▷ **Generation** – ability of a system to scale-up by using next-generation components (faster cores)
 - ▷ **Heterogeneous** – ability of a system to scale-up using HW, SW components supplied by different vendors
- Let's look closer into first two

Limits of parallel processing (1/2)

- Parallelism will not always work, there are limits to acceleration say's **Amdahl's law**
- Program p is defined by a computational load C_p as seen by a single core computing system (performance P_1)
- A part α of the load can be parallelized over N processors (P_N)
- **Meaning that $1 - \alpha$ part of the p can not be parallelized!**
- The acceleration then for a highly parallel system ($N \rightarrow \infty$):

$$A_{cc} = \frac{P_N}{P_1} = \frac{\alpha \frac{C_p}{N} + (1 - \alpha) * C_p}{C_p}$$
$$\lim_{N \rightarrow \infty} A_{cc} = \frac{\alpha + N * (1 - \alpha)}{N} = \frac{\alpha * (1 - N) + N}{N} = 1 - \alpha$$

Acceleration is going to be limited by the amount of non-parallelizable computations $1 - \alpha$!

Limits of parallel processing (2/2)

- The trouble is that you will always have this $1 - \alpha$ portion in any program that you will never be able to get rid off
- Simple example of a blocker for complete parallelism:

```
if (...)  
    then ...  
    else ...
```
- The above can not be parallelized no matter what you do ...
- Ratio between α and $1 - \alpha$ is application and algorithm dependent
- We can only try to reduce $1 - \alpha$, but it can't be 0
- This is going to have a very important consequence on system architecture these days, especially with cores that do not increase their frequency any more

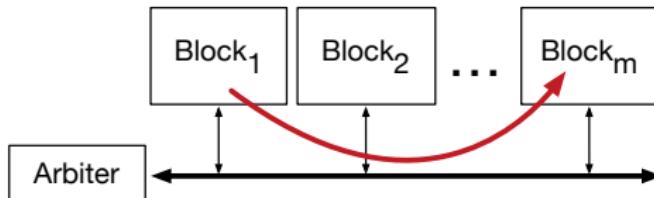
Practical scalability of multiprocessors

- Scalability in size is ok, building highly parallel systems is possible as we speak of **exascale computing** – systems capable of calculating at least 10^{18} floating point operations per second (1exaFLOPS)
 - ▷ E.g. Fugaku supercomputer (2020): 158,976 nodes using Fujitsu A64FX CPU with 48+4 cores (64-bit ARM architectures) per node; that is 8 billions cores! 1BUS\$, up to 80MW power!
- Scalability in application is another story: how SW exploits all these computing resources is a **BIG & COMPLEX** problem ...
- If the machine needs to serve a massive number of independent tasks that can be handled at OS level, this works quite well (data-centers & warehouse scale computers used by Amazon, Google, Microsoft, Facebook etc.)
- Some specific applications can be easily parallelized too, but this is not a general trend, especially when you have in mind different levels of parallelization

2. Interconnects for multiprocessors

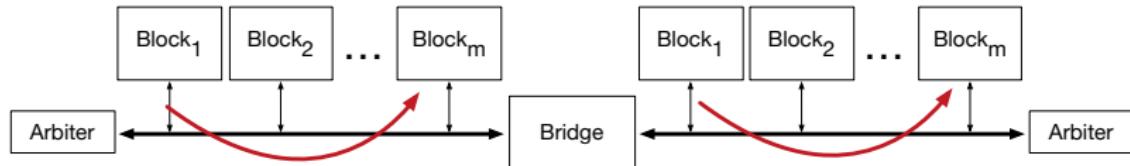
Busses

- Simplest possible scheme to interconnect multiple functional blocks: a set of **shared wires**, i.e. bus
- **Only one master-slave pair can communicate in the same time**
- In case of multiple masters, simultaneous master bus requests are possible, so **arbitration** is needed
- Since masters (cores) are fast compared to interconnect and slaves (memory), too many simultaneous requests will generate bus **contention** – this will kill the system performance
- Further buses do not scale with increasing number of nodes, so m is typically low



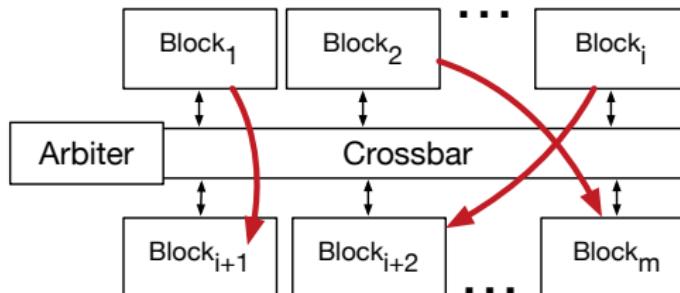
Hierarchical busses

- Parallelizing some of the traffic could help – so let's implement *multiple individual busses* with their own local arbiters to enable parallel communication
- Intra-bus communication still connects only one pair of blocks at a time, but few of them could run concurrently; total system bandwidth goes up; limitations are in inter-bus communications typically established through a **bridge** (poor scalability)
- Interesting for low-end computing systems (low-power, IoT, mobile etc.); good when there are huge differences in access time: slow vs. fast domains that can be separated without interferences



Crossbars

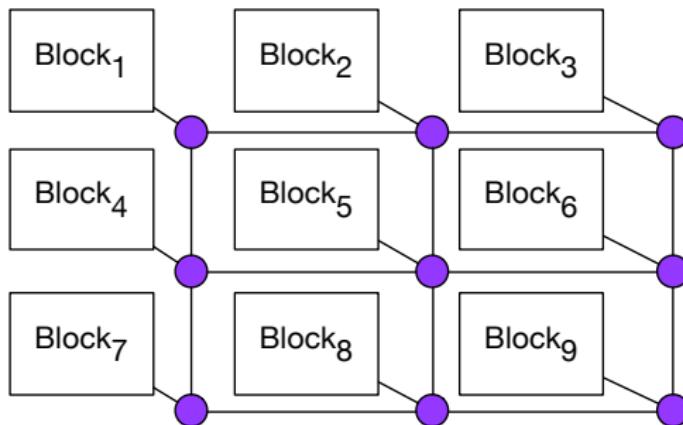
- Enable n masters to m slaves, allowing $(\min(n, m))$ concurrent parallel communication channels (we still may need arbitration)
- Contention is possible (two request target same output port), but if the access patterns are smart (at SW/OS level) we could have significant bandwidth (maximize the number of parallel communication channels); good for local high-performance communication (you will find this in multi-cores)



- Basic logic component behind crossbars are **multiplexors**, combinatorial circuits that scale poorly → limits the $m < 10$

Networks-on-Chip - NoC

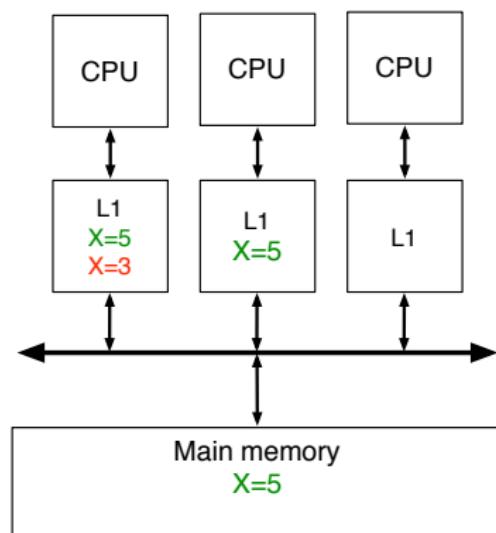
- To address the poor scalability of previous solutions, NoCs create variable node-to-node communication latency by introducing hierarchical crossbars – routers
- Each router has a limited number of I/O ports, implements data buffering and a crossbar with arbitration
- Multiple routers are implemented to reduce router-to-router delays at the expense of latency, but increase the total system bandwidth



3. Cache coherency

Cache coherence problem

- Two processors read same data from main memory; cache misses will cause central memory access; both caches have a **copy** of the same data; if one of the CPUs modifies data (i.e. writes), it will do so in its own cache
- So this value will be different from the original (main mem.) & any other cache copy that may exist (e.g middle CPU)!
 - ▷ How to decide which one to keep & which one to discard?
- Assuming that CPU on the left gets data first & modify it, are we supposing that the program on the right should use old or new (modified) data?
 - ▷ You can't say !!!



Cache coherency & CPU count

- Mono-processors – when introducing caches, we insured **consistency** between the cache and main memory data:
 - ▷ **Write-Through** – central memory is updated every time cache is updated (can be very slow if DRAM is very slow)
 - ▷ **Write-Back** – memory is updated only when the block in the cache is being replaced (to speed-up the DRAM upload process)
 - ▷ But life was easy: one original & copy (hence 1 cache line status bit)
- Multiprocessors – cache coherency is a **much bigger problem!**
 - ▷ **Write Through** – must keep reference of **ALL** modified data and keep **ALL** local caches up to date (this generates lot of traffic & monitoring activities in the interconnect)
 - ▷ **Write Back** – different data blocks (in different caches) can be replaced with different values (more than 2 cores do write operation);
How to decide on the order, or who updates the memory first?
- Multiprocessors will require complex tricks to ensure coherency

Solutions to cache coherency in multiprocessors

- Not easy problem to solve, and needs to be done at:
 - ▷ SW level – programmer (or a compiler) does the job, and assume that HW does not know anything about coherency
 - ▷ HW level – some dedicated device solves the problem & cache coherency becomes invisible to the programmer (magic)
- Which one should be preferred will depend on whom you talk to:
 - ▷ SW people → it should be the HW that solves it
 - ▷ HW people → it should be the SW that solves it
- In reality nobody wants to deal exclusively with it!
- Currently there is no perfect (and scalable) solution that solves cache coherency problem – we speak of different methods that adapt to application & architecture
- Practical solutions involve HW services that work hand in hand with SW (nothing is never pure black or white)

SW cache coherency solutions

- **Compiler** – complexity transferred from HW to SW completely
 - ▷ For – more time during compilation so potentially better solution
 - ▷ Against – difficult to implement, the compiler has to be very smart to do a good job (and this is hard), architecture dependent
- **Programmer** – complexity transferred from SW to us
 - ▷ Programmer does the analysis
 - ▷ He decides which data items may become unsafe for caching (e.g. global variables used & updated by different processes)
 - ▷ Mark them in such a way that they are not cached
 - ▷ Alternatively, determine the unsafe periods and insert code to enforce cache coherence
- In practice: both are complex to do, might lead to inefficient cache utilization, low performance of the memory system

HW cache coherency solutions

- Make use of **cache coherence protocols** that combine both algorithmic approach (SW) & HW devices to insure coherency
- Coherency needs to be solved at run-time, so we do not have that much time to make decisions (whatever needs to be put in place needs to be simple and efficient); advantages of HW cache coherency:
 - ▷ Dynamic recognition of potential problems
 - ▷ Transparent to programmer (which is the key property !!!)
 - ▷ Possibly allows more efficient use of cache
- Two main techniques:
 - ▷ Snoopy protocols
 - ▷ Directory protocols

4. Snooping cache coherence protocols

Overview

- Idea – make sure that every copy of the shared data itself, and their respective status bits in all caches *are systematically updated upon write operation*
- Since caches are connected to the cores using some kind of interconnect, tracking down the write operations could be done by systematically examining what is going on in the interconnect and identifying if the accessed data is in the local cache for every core in the system
 - ▷ This comes originally from the time when multiprocessors have been assembled at macroscopic level
- Since cache controllers know *exactly* the data they have, and are *connected* to the interconnect (core talks to cache controller) it is on them to monitor what is going on at the interconnect level – cache controllers **snoop** or **sniff** the interconnect

Snoopy coherence protocol 1 – Write update

Write update (or write broadcast) – updates systematically **all** cached copies of the data on **every** write operation: consecutive writes will cause every time the update of the modified word of data

- We are in the presence of one-to-many communication, “many” depending on the number of cores
- If the communication infrastructure is one-to-one this means n transfers for n cores causing tremendous bus contention
- Broadcast mechanism could improve this: data sent on the bus *once*, all controllers that identify the data being in their cache take a copy
- Despite broadcasting:
 - ▷ Write update protocols use a lot of system bandwidth
 - ▷ Solution doesn't scale with increased numbers of cores
- Modern multi- & many-core systems never use this, due to the above limitations, hence we need alternative solution (next slide)

Snoopy coherence protocol 2 – Write invalidate

Write invalidate – aims to minimize the amount of data that needs to travel through the interconnect

- Make sure that each processor has **exclusive access** to the data before the update occurs
- When write occurs, all other copies of same data are just *invalidated* & force the other cores to update their copies accordingly
- Note that invalidation concerns the whole block of data rather than a single word like in the write update scheme
- Consecutive writes to the same location invalidate the whole block only once (as opposed to write update that will update modified word many times)
- Cache data have **status bits** indicating state of each modified cache line

Status bits for a cache line in multiprocessors

- Depending on what CPU does to a cache line, status bit(s) are set to:
 - ▷ **valid** – data copy is the same as data in central memory (hence initially, when loaded for the first time, all cache lines are valid)
 - ▷ **dirty** – has been updated by Core_i, write operation took place, data in cache is not the same as the data in the central memory; this is necessary even in uni-processors to track the central to cache memory data coherency (we saw this when introduced caches)
 - ▷ **invalid** – data used by Core_i is in the cache, but it is not valid any more: Core_j modified his copy; Core_i **not good**
- When a cached data gets invalid bit set to 1, all data elsewhere will have to update; but this is not always required: sometimes data is not shared among cores
 - ▷ **shared** – such status bit is added to tag data exclusivity; if set to 0 data in cache is exclusive, and if modified it will not cause invalidation (since there are no copies) and save some bandwidth

MSI protocol 1/2

- Cached data status bits are updated with a **state machine** that will compute the new block state depending on the current state and the operation involved
- MSI protocol has the following states (that gave the name):
 - ▷ **Modified** – cached data copy is modified and inconsistent with original; controller must update this block in central memory if it has to be evicted from cache (cache have limited size)
 - ▷ **Shared** – block is unmodified and in read-only state in at least one cache; cache can evict the data without writing it to the central memory
 - ▷ **Invalid** – data not present in the cache or invalidated by a bus request, it must be fetched from memory or another cache if the block is to be stored in this cache
- State change in the above state machine will occur due to operation in the core and the interconnect (e.g. bus)

MSI protocol 2/2

- MSI state transition will occur based on inputs generated by core & interconnect side requests
- Core side requests
 - ▷ Core read operation
 - ▷ Core write operation
- Interconnect-side request (here bus):
 - ▷ BusRd: when a read miss occurs, core sends a BusRd request on the bus and expects to receive the cache block in return
 - ▷ BusWMiss: when a write miss occurs, core sends a BusWMiss request on the bus which returns the cache block and invalidates the block in the caches of other cores
 - ▷ BusInv: when a write hit occurs, core sends BusInv request on the bus to invalidate the block in the caches of other cores
 - ▷ Flush: Request that indicates that a whole cache block is being written back to the memory

Concrete example – a read miss

Let's analyze the example of a read, followed by a write operation on address X that is not yet cached

- Core1 gets a read miss from the local cache when trying to access the address X
- This read miss is translated to the central memory access and L1 gets a copy
- Now assume that Core2 access the same data X; this is also a miss (not yet cached for Core2)
- Since Core1 modified the data, it did set the state “invalid” bit (there was a modification on that data meanwhile), it will change the cache line state to “valid” **AND will send the copy to the requester** (and/or move the data to the main memory)
- This can be fine if small amount of data is shared between different cores, but if there is a lot of data exchange between cores, this will generate significant traffic, eventually bottlenecking interconnect & main memory

Serialization

- In the previous example we analyzed write-after-read, which is a race free situation since the order in which write occurs was given (first Core1 and then Core2)
- In general multiprocessors portions of code that run on Core1 AND Core2 are independent, they can run in parallel and could potentially **modify X at the same time – race condition**
- Race condition on writing X must be avoided, so serialization needs to be put in place to decide on the right write order
- The appropriate write sequence must of course insure the validity of the data
- We need to somehow implement the **serialization** – this will be done in SW (see section 6)

MSI Protocol variants

- By adding extra states to the state machine specific behaviors could be optimized for improved performance
- MESI protocol add **Exclusive** state
 - ▷ Goal is to minimize the transfers to main memory for write-back caches
 - ▷ It uses **shared** status bit of a cache line mentioned above to signal that cache line is present only in the current cache & that is clean (same as in main memory)
 - ▷ When write operation takes place on such exclusive block, no extra traffic will be generated to invalidate data; this can significantly offload the communication
- MOESI will add **Owned** state
 - ▷ Offloads central memory updates: if a block is in Owned state write-back could occur only when it needs to be shared

5. Directory cache coherence protocols

Idea

- Snoopy cache coherence protocols **does not scale well** because of **broadcast** mechanism: this is bad, especially if you imagine 1000 nodes and you want them to share a lot of data
- Hierarchical snoopy schemes can partially solve this problem, but will still suffer from the central memory bottleneck
- Alternative is **to avoid broadcasts** at any price
 - ▷ Keep the exact record on which processors are caching what locations, together with the state of the data in the cached locations
 - ▷ Tracking is kept in a **global directory** (hence the name)
 - ▷ System knows exactly **what** has been modified **where**, and & execute only **necessary point-to-point transfers**
- Approach is still centralized, but:
 - ▷ Point-to-Point communications can be more efficient and cheaper
 - ▷ Directories could be de-centralized (see next slide)
 - ▷ Forces systems to be structured hierarchically, which is good from physical integrated circuit design perspective

Directory based protocols: issues & solutions

- Directory is still centralized, and we know what does this means: poor scalability & bottleneck to this central node
 - ▷ Just imagine a highly parallel system: information will have to travel long distance plus all CPU have to communicate to this directory
- Solution to this problem must include *distribution*: directory is multiplied (just like cores and memories) and system should ensure that the system knows at top-level where is what
- Simplest way to achieve this is to store each directory in the highest cache level common to all cores that share that memory

6. Processes and threads

Multi-processors from HW to SW

- Now that we have a multiprocessor machine, the next question is:
How to use it, and preferably how to use it efficiently?
or in another words
Who is going to enable application parallelization?
- Not that many choices, *parallelization* could be done by:
 - Operating System** – preferable for every day usage, when we do not want to geek, just use the computer for multi-tasking (mail, web, music etc.) & run some background processes; it is fully **automated** – the user does not need to do anything
or
 - Programmer** – preferable for high-performance computing (scientific computations, AI, etc.); this is not easy, requires competence and it is not always possible to do since some SW problems can not be parallelized; anyhow this will take some time
- These days both approaches are used all the time!

Sharing the resource

- In the old days, CPUs were expensive, but good enough, compared to everything else in the computer architecture
- To allow better usage of the CPU, and thus reduce the cost of the program execution, CPUs have been shared among few “users”: **time-sharing** or **Time Division Multiplexing**; “users” could be different persons or even **programs**
- CPU will spend just a portion of time on each program or user – **main-frame/terminal paradigm** – high cost is now justified, and what become mainstream technology in [60 & '70]
- As CMOS scaled (to become cheaper) Personal Computing (autonomous & individual computers) paradigm replaced centralized approach in '80; and one can sell more CPUs!
- But these days with have revival of centralized computing through cloud, data-centers; some local computational power (smartphone, tablet, IoT device) plus lot of distant computation power (one sell even more CPUs)

Different ways to share the resource

- **Multi-programming** (in the early days)
 - ▷ Multiple programs loaded in memory and if the program needs to wait for a slow resource (e.g. IO) it can temporarily suspend the execution and allow other programs to execute
 - ▷ Problem: there is no fairness, if program takes CPU time & uses it, no other program can get the CPU before the first program ends
- **Cooperative time-sharing**
 - ▷ Program **suspends himself** to allow other programs to execute
 - ▷ Problem: if the program is poorly written (for example by a greedy programmer), we will have bad overall efficiency
- **Preemptive time-sharing** (most of the systems today)
 - ▷ Introduces “independent” decision maker that has the power of preempting the CPU – **scheduler**
 - ▷ Problem: how to decide on a good scheduling policy? (this is not a simple problem, no single solution that fits all applications, etc.)

Scheduling and Multi-tasking OS

- Since we need intermediate layer between HW and SW, it is natural to embed scheduling into OS
- When scheduling became integral part of the OS it enabled **multi-tasking**: multiple programs could run even on a single CPU machine (CPU time share)
- Each running executable is seen by OS as a **process** & multi-task OS aims to:
 - ▷ **Maximize process throughput** – increase as much as possible the number of executed processes per unit of time
 - ▷ **Minimize latency** – delay between data input & produced output
 - ▷ **Enable certain fairness** – avoid starvation of programs (or users)
 - ▷ **Allow smooth responsiveness** to the user (or other applications)
 - ▷ For other computer systems, objectives might be (very) different, typical example being real-time OS – guarantee that the execution will finish before a certain deadline

Processes and context switches

- To achieve the above objectives the OS needs to implement a mechanism to control process execution, i.e. it has to enable *switching* between different process – **context switch**
- Process switching implements the **preemption**: the scheduler decides which process should be **suspended** for execution to allow the execution of another process
- **Context switches do not come for free!**
- At each switch we need to save and restore the process state, so saving, loading registers & memory maps, updating various tables & lists etc.; this will generate a lot of memory access
 - ▷ It is good to optimize the context switch mechanism (e.g. make HW that does this as fast as possible) and the use of it (e.g. OS minimizes the number of preemptions)
- In multiprocessors there will be also notion of **migration**, task stops execution on one and resumes on another CPUs

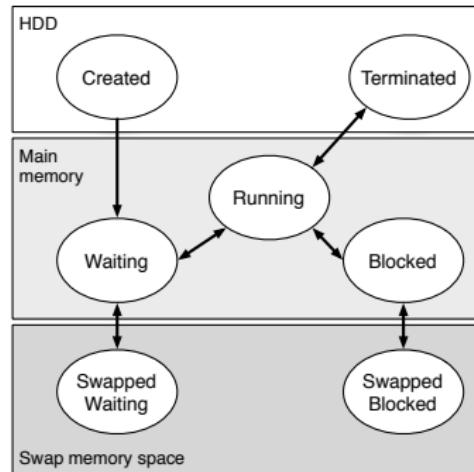
What defines a process

- Executable code of the program
- Memory – some region of virtual memory, including executable code and process-specific data (input and output)
- Call stack – to keep track of active subroutines & other events,
- Heap – for intermediate & significant data generated during run-time (dynamic memory allocation)
- OS descriptors – resources allocated to the process (file descriptors in Unix or handles in Windows)
- Security attributes – such as process owner and process's permissions (allowable operations)
- Processor state (context) – content of internal CPU registers, physical memory addressing, etc.
- CPU state – stored in registers when the process is executing, and in the memory otherwise

Life of a process (i.e. process states)

This is not standardized, but quite similar for different OS

- **Creation** – executable loaded from HDD into main memory
- **Waiting** – process is in memory but not active (not running on a CPU); if it is not used for a long time & memory is running low, it can be swapped out (i.e. moved to HDD) to save memory
- **Running** – when context switch occurs for this process, the context is loaded into CPU that starts executing instructions associated to it
- **Blocked** – if process needs to wait for a resource, we could free the CPU; process state is changed back to waiting (or running)
- **Terminated** – when all program instructions are executed



Inter-Process Communication (IPC) (1/2)

Processes are designed to be independent, but if needed they can communicate using one of the following IPC schemes:

1. **Shared memory** – two or several processes map a segment of their virtual memory space into an identical segment of physical memory; one of the most efficient way to ensure IPC, but it requires data **synchronization**
2. **Messages** – Data is copied from memory of the sender process into a message buffer of the system memory space; then copied again from buffer to memory space of the receiving process
 - ▷ Frequent data copying could stress the interconnect structure
 - ▷ In order to make messages work across the process boundary, the message buffers have to be named
 - ▷ Each process which creates a message buffer has to refer to the same message buffer name

Inter-Process Communication (IPC) (2/2)

3. Pipes

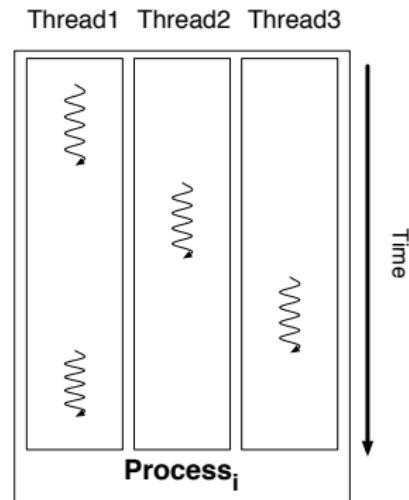
- ▷ Provide interprocess communication channel, implemented in the OS file system
- ▷ For efficiency, pipes are **in-core files**, i.e. they reside in memory instead on disk, as any other global data structure
- ▷ One file is reserved for reading & another for writing
- ▷ Reading/Writing from/to a pipe is performed in FIFO fashion
- ▷ Since the pipes have a limited size and the FIFO access discipline, the reading and writing processes are synchronized in a similar manner as in case of message buffers
- ▷ Access to pipes are the same as for files using some kind of `WriteFile()` and `ReadFile()` functions

Processes: pros & cons → threads

- Process are given in modern multi-tasking OSs
- However they do **have disadvantages**
 - ▷ They will generate lot of memory access & interconnect that is already heavily bottlenecked
 - ▷ Further, context switches in systems with Centralized Shared Memory and caches will require cache flushing
 - ▷ Analysis of typical applications shows that context switches introduce non-negligible performance overheads; how much will depend on application & data sets
- We want to minimize the context switch overhead
 - ▷ Reduce the amount of information required to define a process
- We want same executable to be able to issue multiple “sub-processes” (parent-child process)
- The above can be achieved using fine grain, light weight, processes called **threads**

Threads – properties

- Definition – smallest unit of execution (sequence of instructions) that will be recognized by the OS scheduler & scheduled for execution on one CPU at a time (or core, more details to come)
- Multiple threads can be issued from the same parent process; they share the process execution context
- Multi-process, multi-threading can occur on both uni-processor (time-sharing) & multiprocessors; on the last ones they enable **automated parallel execution**
- A thread consists of a stack, state of CPU registers, and an entry in the execution list of the system scheduler
- Implementation at SW, not OS level



Threads vs. Processes

- We said: threads issued from the same process share the same execution context → **they share same address & memory space**
- Inter-thread communication is much faster & simpler
- Thread's **context is much simpler**, the amount of information that fully describes the thread requires less memory
- Thus the context switch of a thread is cheaper than a process switch (less CPU time overhead for the context switch less communication)
- Processes are totally independent one from the other; threads issued from same process are:
 - ▷ Independent between themselves, and will be thus scheduled independently ...
 - ▷ but they are dependent on the process itself; they are just one level of hierarchy below
 - ▷ **Elegant way to implement application level parallelism**

Issues in multi-threading (1/2)

- Cache coherency deals with the problem of data **race conditions** (on data) that might occur in multiprocessor machines
- This is hidden from the programmer
- In the context of a multiprocessors & multi-threaded execution we will also have data race conditions!
- **This is because we do not know the thread execution order!**
 - ▷ Why it is so?
- Easiest way is to implement a **locking scheme, lockset**
- Lockset maintains a set of candidate locks for each shared memory location
- If a shared location is accessed when this set is empty, there has been a violation of the locking discipline!

Issues in multi-threading (2/2)

- Look at the example below: the value of `y` will depend on `x` that might be modified by another thread
- Result of Thread1 operation will depend on the order on which the scheduler decides to schedule Thread 1 & Thread 2
- In general you do not want to interact with the scheduler (but in some cases this is a very good idea and you could do that assign a given thread to a given CPU)
- In the example here you have somehow to lock value of `x` in order to get the things right (prevent F2 to mod `x` value)

```
1 void F1 () {           // Thread1
2     if(x == 5) {       // Check
3         y = x * 2;    // Act
4     }
5 }
6 void F2 () {           // Thread 2
7     x+=1
8 }
```

```
1 lock(x);             // We lock the x
2 // Thread 1 call
3 F1();
4
5 unlock(x);
6 // Thread 2 even if lunched
7 F2();                 // x is locked
```

Solutions to solve data race conditions

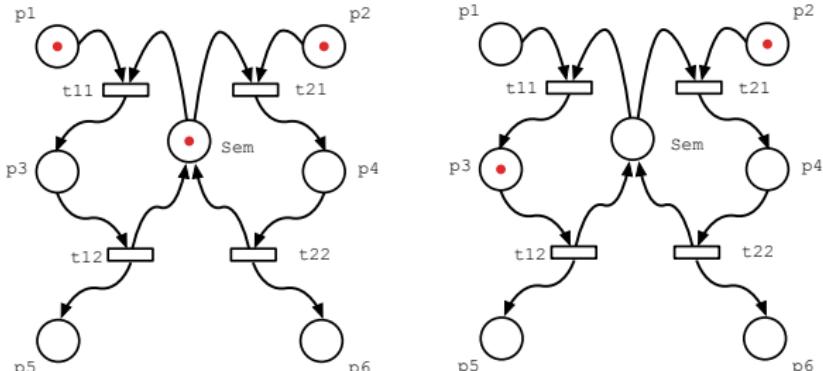
- No HW (or automated) solution to the problem of data race conditions in the context of multi-threaded applications with full READ/WRITE access to the data
- Data races are **easy to cause** and **hard to debug**
- To avoid these it is the programmer's responsibility to solve these situations using:
 - ▷ **Explicitly scheduled threads** – from software directly drive the scheduler using OS APIs
 - ▷ **Rendez-vous mechanism** – do the same as above using some SW tricks to insure the correct processing sequence on the data in time; this bypasses the scheduler and does implicit scheduling
 - ▷ **Mutually exclusive operations** to insure that there is no simultaneous modification of the data coming from two threads (although the order is not critical)
- Typical synchronization tools: **mutex** and **semaphore**

Mutex

- Mutex is a very general concept in computer sciences: used whenever we speak of concurrency
- Here we refer to an **OS feature** and the way how they can be used to insure synchronization between the threads (issued from the same process)
- Mutex is nothing else than a **shared boolean variable** that enables single access to a resource
- Mutex is first defined and it can be assigned one of the following two states:
 - ▷ **Not taken** – the resource is free, any thread can take it
 - ▷ **Taken** – the resource is occupied, if one thread wants to gain the mutex, it will have to wait until it is released
- Mutex guarantees mutual exclusion, hence the name, for a given **critical section** in the program, but doesn't guarantees the order ...

Practical illustration

- Assume two different processes have a section to be executed in **mutually exclusive fashion** – one & only one section executes at a time; modeled using **Petri nets** (similar to state machines)
- Red dot indicates that the state is active; the same goes for the shared mutex resource, or **lock Sem** (binary value in this example)
- States p1 & p2 fight for a **shared resource Sem**: tokens in p1, p2 & if transition t_{11} is enabled, the token from Sem; place p2 will have to wait until the resource Sem gets freed: when transition t_{12} is enabled, a token will be put back to Sem



Generalization – semaphores

- These are extensions to the concept of mutex: rather than Boolean variable, **semaphore** is a **counter**, that will count the number of different threads using given resource
- It is an integer variable, accessed through 2 atomic operations:
 - ▷ down / wait – decrement, block until semaphore is open
 - ▷ up / signal – increment, allow another thread to enter
- Rather than binary taken or not we can now queue threads:
 - ▷ When thread is launched it calls `wait`:
 - If semaphore is open ($\text{value} > 0$), thread continues and decrements the value of the semaphore
 - If semaphore is closed ($\text{value} = 0$), thread blocks in the queue
 - ▷ When thread finishes it calls `signal` that opens the semaphore (increment value):
 - If there are threads in the queue, the thread is unblocked
 - If no threads are waiting in the queue, the signal is remembered for the next thread
 - ▷ In other words, we keep the track of the “history” using the counter

Programming support for multi-threading

- **Bad news** – no standardized support for multi-threading!
- Whatever you do is OS/compiler/architecture dependent
 - ▷ **OS dependent** – because it will depend on OS APIs to provide the link to the associated HW services; you could use the OS APIs directly, but this is not that handy in practice
 - ▷ **Compiler dependent** – will provide an extra support layer to ease-up the usage of APIs in the OS: a) Create threads; b) Synchronize resource access among threads (inter-thread communication); Terminate threads
 - ▷ **Architecture dependent** – you will most likely have to re-write and re-compile the code for different target architecture
- There are initiatives to enable cross platform capability
 - ▷ **OpenMP** – API for multi-platform shared-memory multiprocessing programming built on the top of C/C++/Fortran
 - ▷ Available for many platforms, ISAs and OSs
 - ▷ Consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior

Example of thread generation for Win32

```
1 #include <stdio.h>
2 #include <windows.h>
3 #include <process.h>      // needed to include for _beginthread()
4
5 void  silly( void * );
6
7 int main()
8 {
9     printf( "Now in the main() function.\n" );
10
11    // Create a thread and ask it to start in the silly()
12    _beginthread( silly, 0, (void*)12 );
13
14    // This main thread can call the silly() function if it wants to.
15    silly( (void*)-5 );
16    Sleep( 100 );
17 }
18
19 void  silly( void *arg )
20 {
21     printf( "The silly() function was passed %d\n", (INT_PTR)arg ) ;
22 }
```

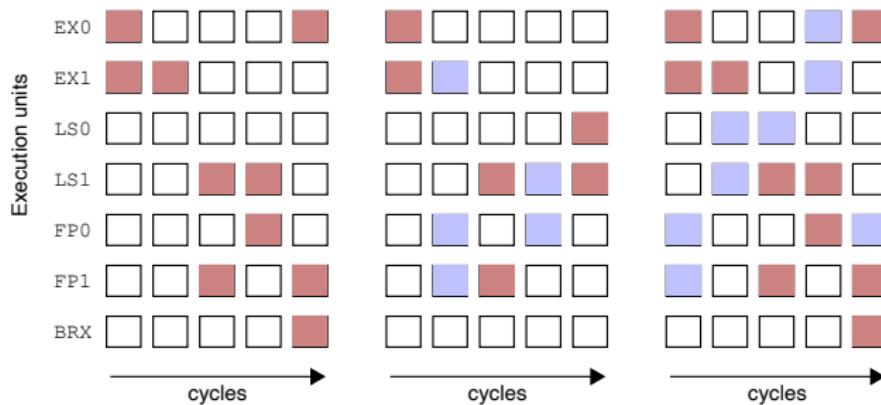
7. Simultaneous Multi-Threading

Simultaneous Multi-Threading (SMT)

- Once a thread gains core time, it gets **whole core & all of its resources for a certain amount of time until next context switch**
- If a core is a super-scalar unit, few instructions in this thread *could* be executed in parallel; we say *could* since the program has to have independent instructions that could be executed in parallel
- Most of the cores implement separate integer & floating point units; if the thread instructions use only integer operations, floating point unit sits there not doing anything
- Simultaneous Multi-Threading (SMT) is about allowing super-scalar cores to execute multiple threads simultaneously**
- From HW perspective this means that the core execution resources could be split in so called **logical cores**; e.g. one physical core can contain two logical cores
- To allow more efficient execution we should increase register files to limit the memory accesses that increase the system cost

SMT in practice

- Left – single thread: only instructions from current process
- Middle – multithreading enabled; from one to another cycle few threads share execution units; in the same cycle only instructions of the same thread are executed on different execution units
- Right – SMT: 2 threads use available resources different execution units in the same cycle



Better usage of available HW

SMT Past & Today: many examples

- **Alpha 21164** – four-issue superscalar microprocessor capable of issuing a maximum of 4 instructions per clock cycle to 4 execution units: 2 integer and 2 floating-point ops; this one has been issued in 1998!
- **SPARC** – originally Sun, later acquired by Oracle; still active with Oracle M7 core (& M12 these days) – relatively simple core architecture (2 ALUs) but with 32 cores each capable of executing 8 threads for a total of 256 threads on a single die (we will cover this CPU more in details later on)
- **Intel CPUs** do SMT too, but with another name: **hyper-threading**
 - ▷ Done by duplicating storage of the CPU state, not the execution resources (Intel is a super-scalar architecture anyhow)
 - ▷ Typically allows OS to see two logical cores from one physical core (so a dual core CPU will show 4 logical processors)
 - ▷ But do not expect 4X acceleration out of the box!

ELEC-H-473

Th09: Practical processor architectures

Dragomir Milojevic
Université libre de Bruxelles

2023

Goal

- Apply different concepts that we have seen on practical, state of the art architectures and real life products
- **Intel CPUs are very good example ...**, they integrate:
 - ▷ CISC with very deep instruction words
 - ▷ Heavy ILP (deep pipeline)
 - ▷ Super-scalar and SIMD computational models
 - ▷ Multi-core for thread-level parallelism
 - ▷ Complex memory hierarchy
- Short overview of low-power CPUs using the example of ARM processors (omnipresent today)
- Somehow in depth overview of **SPARC** architecture, different from Intel and used in servers and super-computers
- And a recent developments in low-power AI/ML architectures with **many-core paradigm**

Today

1. Microprocessors & CMOS
2. Intel
3. ARM
4. Another interesting architecture: SPARC
5. Recent many-core architecture

1. Microprocessors & CMOS

Processors & IC technology

- It is the **CMOS scaling** that enabled tremendous increase in processing power – **Moore's law**: every 18 months transistor density doubles; area is divided by 2 because linear dimension is scaled by 1.4 for full **node-to-node scaling**
- For **half-node scaling** ($14 \rightarrow 10 \rightarrow 7 \rightarrow 5 \rightarrow 3 \rightarrow \dots \text{nm}$) factor is 0.7 (i.e. 30% less area for each new tech generation)
 - ▷ But smaller transistors mean not only lower cost, and bigger functional density for the same Si manufacturing cost, but also & especially better performance & lower power dissipation
- If only same scaling factor could be applied to cars, today:



Speed	$180 \times 10^6 \text{ km/h}$
Fuel	0,04L/100km
Price	0,0003\$

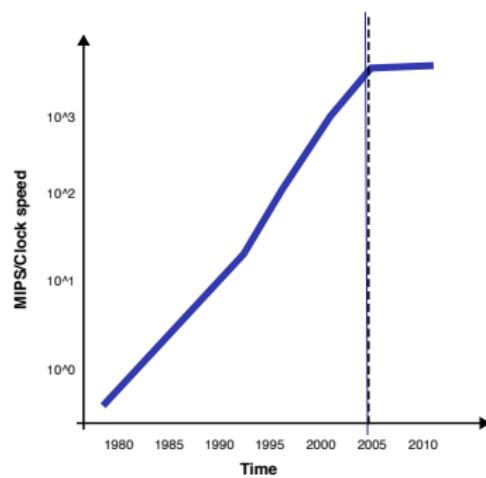
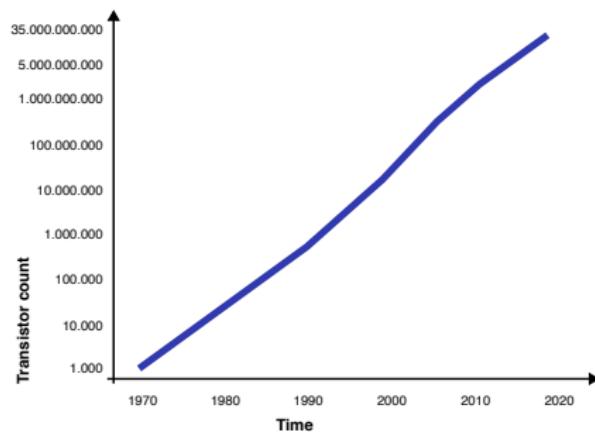
Basics of CMOS scaling

- Node-to-node area of transistors reduces by 50% so $\frac{1}{2}$
- **Capacitance C** – reduced by 30% ($\frac{1}{3}$) because it varies with area over distance
- **Voltage V** – assuming constant electric field, reduced by 30%, $\frac{1}{3}$ (because voltage is field times length)
- **Current & transition times** – scaled down by 30%, due to their relationship with capacitance and voltage
- **Circuit delay** – assumed to be dominated by transition time, so it is reduced by 30% too
- **Maximum frequency** – increases by about 40% ($1.4\times$), because frequency varies with one over delay
- **Power dissipation** – for individual transistor decreases by 50%, because dynamic power is

$$P_{dyn} = \sim C \times F \times V_{dd}^2$$

Scaling & Moore's law

- Previous has been true for quite some time – [Moore's law](#) (1965)
- Known as “*Happy days of scaling*” – things were predictable; IC manufacturing industry relied on photolithography improvements to shrink smallest IC feature: basic rectangle; **this was given** and everybody knew what will come next



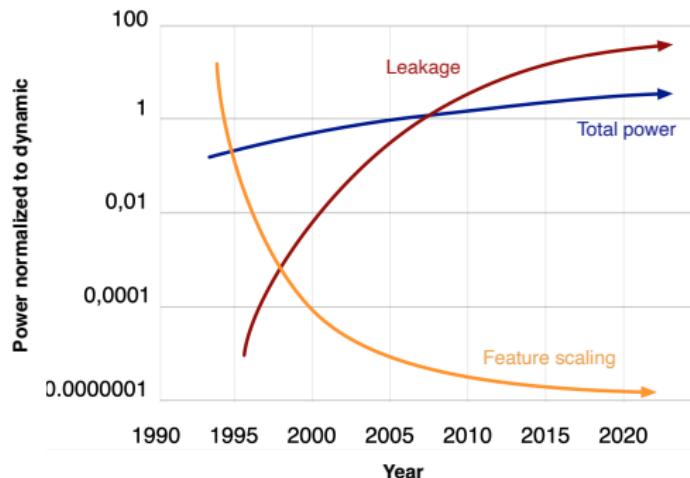
- However paradigm change since ~2000: abandon of frequency race in favor of more parallelism – **multi-core era** was born

Dennard scaling

- Scaling reduces area **and** power with the same factor $\frac{1}{2}$
- But when we move from one CMOS node to another we also want more CPU functions per mm^2 (think of all that HW that we added: OoO, branch prediction, FP, SIMD, wide and deep RegFiles etc.), **so we want to keep IC area constant**
- If we keep IC area constant, we have $2\times$ more transistors dissipating $2\times$ less power, so total power per IC remains the same
 - ▷ **Power density in the IC increases with CMOS scaling**
- We know we must keep junction temperature $<125^\circ C$
- To avoid devices breaking up (or wearing out too soon) we need to improve cooling; **but cooling improvements do not evolve at the same pace as transistor scaling**
 - ▷ Despite some fancy techniques such as micro-fluidic & forced fluid coolers etc.

Different power components scaling

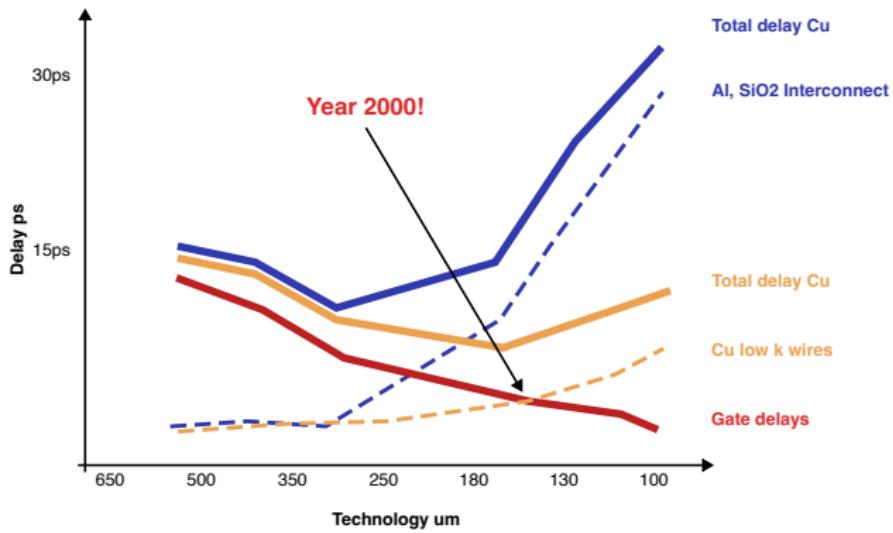
- In the good old days, power has been burned to compute things!
- Once again starting 2000, leakage power (one dissipated even when the gate doesn't toggle) is increasing to reach the same amount as dynamic power; **to toggle or not has the same power price**



- Total system power goes up by little since packaging & cooling solutions do not progress that swiftly

Interconnect scaling

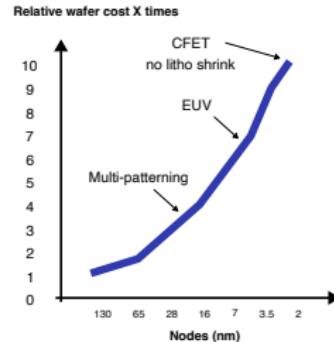
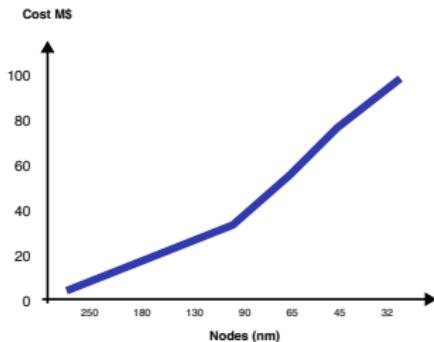
- In the good old days, wire delays could be neglected since the critical path delay has been dominated by logic gates
- Change in interconnect material (from Al to Cu) enabled significant reduction of the interconnect delay, but not for long



- Since 2000 wire delays rise, while gate delay continue to scale down; today wire delays are more & more dominant → serious bottleneck

Cost (non)scaling

- Design associated costs increase, more resources are required to place all that logic → people, computers, EDA, litho masks etc. (left graph)
- For advanced CMOS nodes manufacturing costs are increasing exponentially → 10× from 130nm to 2nm (right graph)



- To compensate for such significant investments (& still generate profits!) sales volume need to increase; this is why we see these days paradigms such as IoT, IoE, edge computing, wearables, pillcams etc.

Before physics, economy is most likely to be the strongest showstopper for future IC technologies

Scaling is dead, long live the scaling

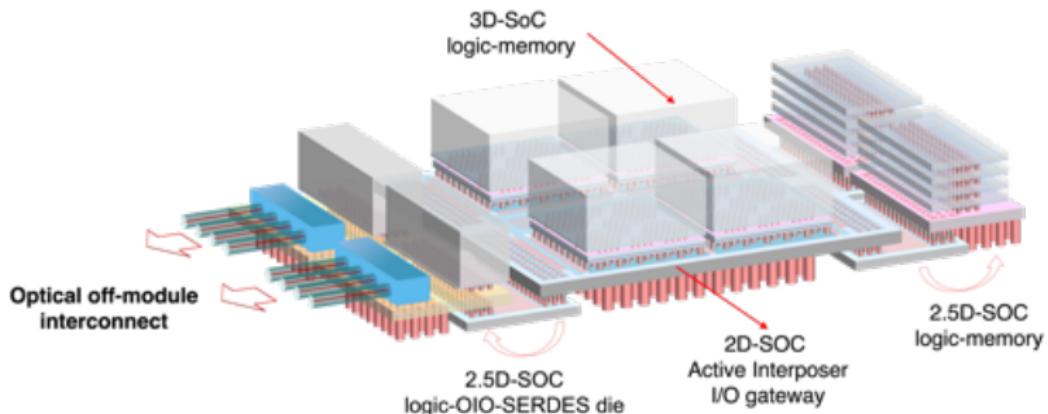
- IMEC technology roadmap – scaling will continue beyond 1nm¹
 - ▷ smart device (transistor) architecture, scaling boosters, new materials, standard cell design etc.; **but it will happen at slower pace**



¹Note the abandon of minimum feature size in favor of angstrom; single dimension scaling is not that relevant any more for future scaling

Complementary technology – advanced packaging²

- **Chiplets** – individually processed dies, assembled during packaging
- **Interposer** – active or passive (interconnect only) base die, hosting different circuits, including 3D stacked ones (microscopic PCB)
- Includes various vertical, horizontal & optical die-to-die interconnect



The above is known as 5.5D IC

²Source:imec

Key takeaway

- Impact of CMOS and IC manufacturing technology on microprocessor architecture design is tremendous
- Every single microarchitecture feature you see today in a CPU is implemented to improve SW performance of course, but also to tackle technology blockers mentioned above
- Future is in **System Technology Co-Optimization (STCO)** – holistic view of the complete stack: from lithography used for CMOS-IC manufacturing, to device and all the way up to architecture and SW
 - ▷ STCO is the evolution of **Device Technology Co-Optimization (DTCO)** paradigm used to describe co-optimization between lithography, device & gate (standard cell) design; **you may notice how holistic the whole thing is becoming in time**

Processing power

- Multi & even many-core paradigm to tackle need for higher processing power **without frequency scaling** (ever since 2000)
- At IC level we speak of **Systems-on-Chip (SoC)** or **Multi-Processor SoCs (MPSoCs)** – complex systems including many different functional blocks (heterogeneous) with complex communication infrastructure
- SMT and run-time overclocking with active temperature aware thread scheduling
- **Heterogeneous architectures** with high-performance cores implemented using high-performance, but power hungry transistors as opposed to low-power cores that will also exhibit low-performance; **new metric: performance/Watt**
- **Advanced Packaging Solutions** or **3D circuits** are used to implement technologically heterogeneous systems: IC CMOS process can be optimized for function: high-performance logic, memory, analog, low-cost etc. and thus improve PPA and cost

Addressing power & thermal issues

- Done with run-time management of the IC to adapt Frequency & Voltage of the system to the actual needs – **Dynamic Voltage & Frequency Scaling (DVFS)** at system and functional block levels
- Looking at dynamic power dissipation³:

$$P_{dyn} = \alpha \times F \times V_{dd}^2 \times C$$

we see that reduction of V_{dd} & F can help (a lot!); C we can't do much since fixed by the technology and the IC design; you can bet that during IC design minimizing C is one of the key optimization objectives

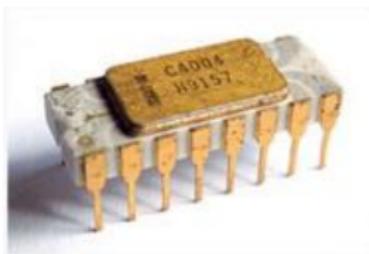
- Note that V_{dd} & F are correlated: for higher F , we need more V_{dd}
- FFs & clock network use considerable amount of power; for inactive parts of the circuit clock will not be propagated to FFs – **clock gating**
- But we saw leakage is as important as dynamic power; parts of the circuit that are not used are cut from the power supply – **power gating**

³Where α is the switching factor, i.e. how many times gate toggles.

2. Intel

Facts and figures

- Funded in 1968, designed first commercial CPU was 4004 (1971)



- Became largest and highest valued semiconductor company today
- Focus on high-performance CPUs, but they make other chips too: motherboard chipsets, controllers, memory, graphics, etc.; they have missed low-power train
- Big: 100k employees, 70B\$ revenue, 20B\$ net income (2018)
- Major competitor: AMD, smaller part of market share (~20%)
- Other competitors (servers & data centers): Fujitsu, Oracle, but they gradually abandon HW development; low-power cores are now 64-bit and they are in the server market

Strategy

- Intel combine all aspects of the CPU design:
 - ▷ Technology development for (proprietary) IC manufacturing
 - ▷ Micro-architecture development
 - ▷ Circuit design & manufacturingi.e., they can really do the STCO
- Business model in this industry splits activities: a) **fab** (e.g. TSMC, GF) from b) **chip design** (fabless, e.g. Qualcomm) & system **integrators** (e.g. Apple)
- Intel always ignored the above split model with success, did everything themselves and kept their technology only for their products
 - ▷ In 2015 they acquired Altera, 2nd largest FPGA company; **Can you guess what they are going to do with that?**
- They bet on the next big thing: **IoT** even if only 5% of the revenue is generated with IoT as of today (but don't forget IoT needs servers)
- They use **Tick-Tock** approach – develop microarchitecture & tech in parallel; products alternate tech & design: new architecture in node N; followed by the same design in N+1; then new design in N+1 etc.

Intel evolution & disambiguation

Many marketing names, not simple to disambiguate, some tips:

- Micro-architecture – key differentiator for a CPU since it defines the ISA; Intel has 2 major releases:
 - ▷ x86 – laptop/desktop/server depending on data/address path bit width, 32 and now 64 bit (x86-64), this is CISC
 - ▷ Itanium – servers/high-performance computing, VLIW architecture, remained low-volume; discontinued in 2020
 - VLIW – Very Large Instruction Word: pack few execute instruction into a single machine instruction; compiler looks for ILP rather than OoO execution unit – supposed better performance
- Micro-architecture code name
 - ▷ NetBurst ('00), Nehalem ('08), Haswell ('13), Raptor Cove ('23)
- Brand name – CPU (IC + package) you buy in the end
 - ▷ Pentium 4 (NetBurst); i3, i5, i7 (Nehalem, Haswell); Raptor Lake (Raptor Cove)

CPUs & technologies

- 8086 – First x86 processor
- 186 – DMA, interrupt controller, timers, and chip select logic
- 286 – First x86 processor with protected mode
- i386 – First 32-bit x86 processor
- i486 – 2nd gen. 32-bit x86, built-in FPU and pipelining
- P5 – Original Pentium
- ... and I am skipping things in between
- Nehalem (2008) – 45nm Core i3, i5, i7 (brand name)
(Incorporates the off-chip memory controller into the CPU die)
- Westmere – 32nm shrink some new features
- Sandy Bridge (2011) – 32nm
- Ivy Bridge (2012) – 22nm shrink
- Haswell (2013) – 22nm new architecture
- Broadwell (2014) – 14nm shrink
- ...
- TigerLake (2020) – 10nm
- RaptorLake (2022) – Intel 7 process

Evolution in features – IA32

- Note speed, transistor count and cache memory increase

IA32

CPU	Date	MHz	Trans	Registers	Bus	Max addr	Cache
8086	1978	8	29K	16GP	16	1MB	None
286	1982	12.5	134K	16GP	32	4GB	None
386	1985	20	275K	32GP	64	4GB	8KB on-chip
486	1989	25	1.2M	32GP/80FP	64	4GB	L1:8KB
Pentium	1993	60	3.1M	32GP/80FP	64	64GB	L1: 16KB
Pent Pro	1995	200	5.5M	32GP/80FP	64	64GB	L1: 16KB L2: 256/512KB
Pent II	1997	266	7M	32GP/80FP 64MMX	64	64GB	L1: 16KB L2: 256/512KB
Pent III	1999	500	8.2M	32GP/80FP 64MMX 128XMM	64	64GB	L1: 32KB L2: 512KB
Pent IV	2000	1500	42M	32GP/80FPU 64MMX 128XMM	64	64GB	L1: 12Kops L1: 8KB data L2: 256KB

Evolution in features – IA64

- Note increase in cache levels as we move into multi-core era; maximum addressable space follows DRAM scaling; L1 cache quite constant; significant increase in RegFile size (both word width and number of words) for SIMD; GPR remains the same

IA64

CPU	Date	MHz	Trans	Registers	Bus	Max	Cache
Pent M	2003	1500	42M	32GP/80FPU 64MMX	64	64GB	L1: 32KB L1: 1MB
Core Duo.	2006	1500 1600	228M	32GP/80FPU 64MMX	64	1TB	L1: 32KB L2: 2MB
Core 2 Duo/Quad	2006	1600 2000	410M 820M	32GP/80FPU 64MMX	2x64	1TB	L1: 32KB L2: 2/4 or L2: 6/12MB
Nehalem	2008	2660	731M	32GP/80FPU 64MMX 128XMM	3x64	256TB	L1: 32KB L2: 256KB L3: 4-12MB
Sandy Bridge (2-8) cores	2011	2800	1.2G(4) 2.7G(4)	32GP/80FPU 64MMX 128XMM 256YMM	4x64	256TB	L1: 32KB L2: 256KB L3: 8-20MB

12th (2021) & 13rd generation (2022)

- Highly scalable **System-on-Chip (SoC)** architecture
- Targets all computing segments from IoT to high-performance with power range from 9W to 125W
- 3 CPU classes: Desktop, Mobile & Ultra Mobile
 - ▷ Mobile different power dissipation groups: 45W, 28W & 9-15W
- Intel7nm CMOS, no reference to any dimension, just commercial
 - ▷ Enables ~100 million transistors per square millimeter
- **Heterogeneous architecture** with 2 types of cores – **Performance (P-Core)** and **Efficiency (E-Core)**
- Support for high-performance (DDR4 and 5), or low-power (LPDDR) DRAM standards
- Rich interfacing WiFi6E, Thunderbolt, PCIe, USB, SATA

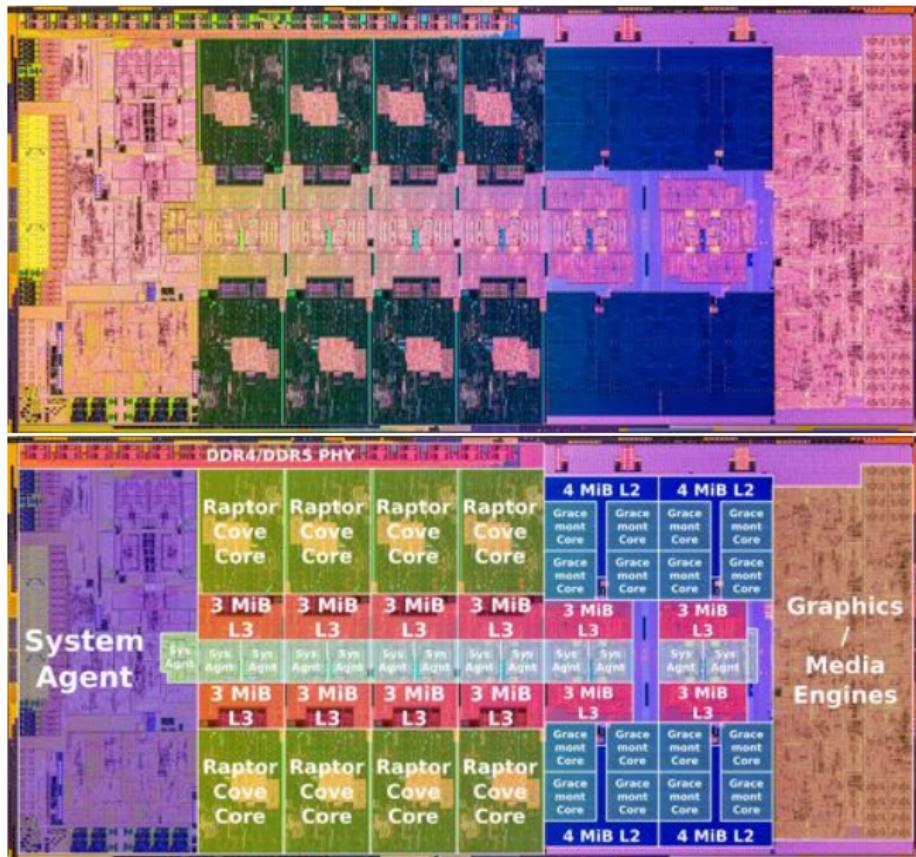
Example of CPUs (H-series, as high-perf)

Processor model	Cores Total	Threads	Cores P	Cores E	L3 MB	MaxF GHz	Power ⁴ W
i9-12900HK	14	20	6	8	24	5.0	45
i9-12900H	14	20	6	8	24	5.0	45
i7-12800H	14	20	6	8	24	4.8	45
i7-12700H	14	20	6	8	24	4.7	45
i7-12650H	10	16	6	4	24	4.7	45
i5-12600H	12	16	4	8	18	4.5	45
i5-12500H	12	16	4	8	18	4.5	45
i5-12450H	8	12	4	4	12	4.4	45

- Other product families (low-power mobile) include less resources

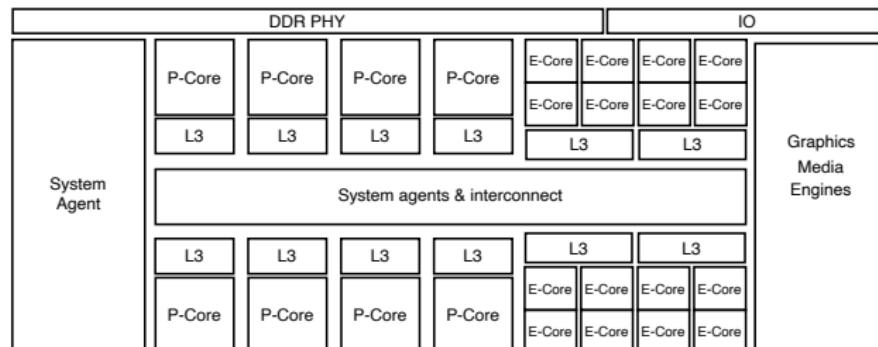
⁴This is base power, will be more when all resources run at max speed, <125W

13th gen. – Raptor Lake 8P/16E-Cores (2023)



SoC functional blocks

- High-performance (1.) & low-power cores (2.) (P- & E-cores)
- Cache sub-system (3.) with high BW Least Level Cache (LLC) L3 split for P and E-Cores
- SoC interconnect – as important as computational cores (4.)
- System Agent – SoC controller (5.)
- Embedded high-performance graphics & media engine (6.)
- Integrated DRAM controller (7.)
- SoC IO

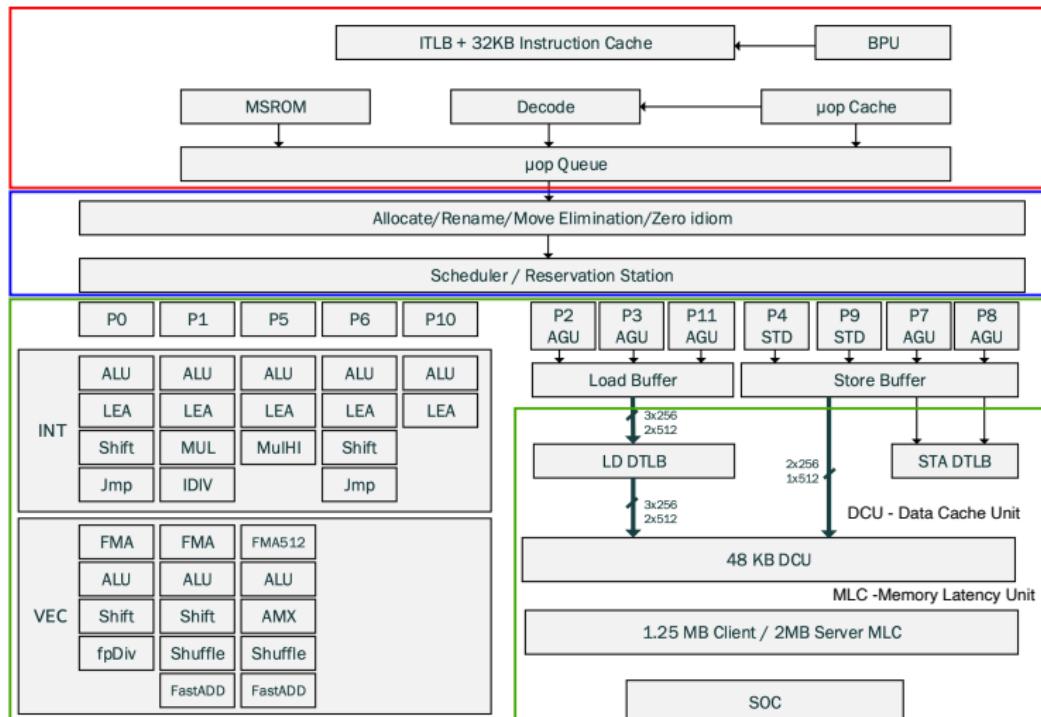


1. High-performance core features (Golden Cove)

- Improved instruction fetch – fetch bandwidth of 32B/cycles, 4-6 instruction decoders, increased micro-op cache size, and higher micro-op cache bandwidth
- Super-scalar with 12 execution ports, out of which 5 to 6 specialized execution ports (so ALUs); up to 8 load/store ports: with 3 loads & 4-8 wide store
- Greater capabilities per execution port; example: 5th integer ALU execution port with expanded capability & fast FP adder
- HW accelerator for matrix multiplication (Machine Learning)
- HW support to enable deeper OoO execution & expose more ILP & improved branch prediction (**Branch Prediction Unit – BPU**)
- Enhanced cached data prefetching for more memory parallelism
- **Mid Level Cache (MLC)** 2MB & 1.25MB for server & client CPUs

1. P-core architecture (Golden Cove)

- Front end, OoO+instruction scheduler, execution ports

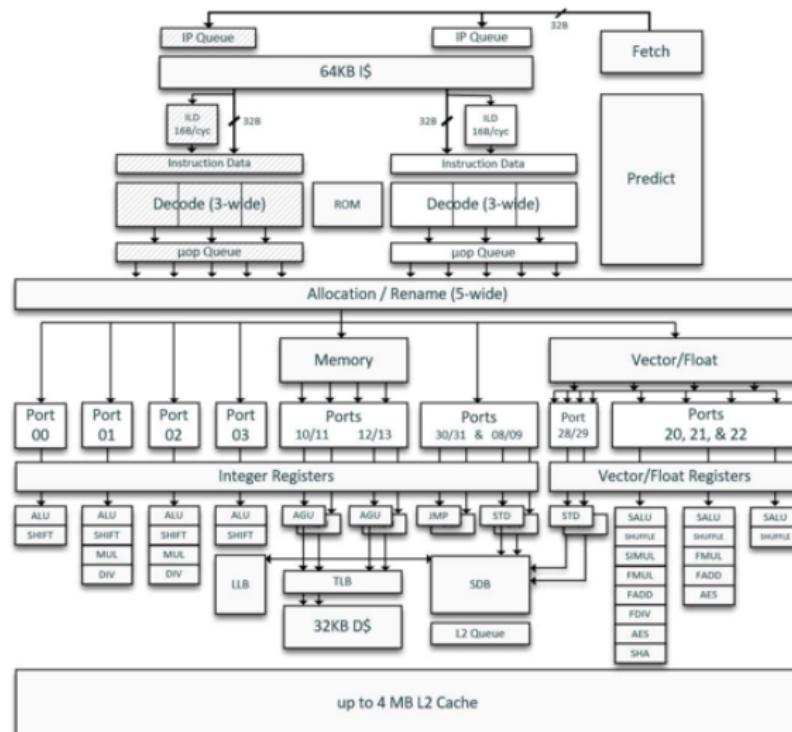


2. Low-power core features (Gracemont)

- Four integer ALU execution ports with expanded capabilities; two dedicated integer multiply/divide HW units
- Enhanced branch prediction unit
- 64KB Instruction Cache with dual 32B reads
- Dual load and dual store execution ports capable of 2 loads, 2 stores, or 1 load and 1 store per cycle
- HW support for security
- SIMD with 256-bit Advanced Vector Extension (AVX but not 512)
- Designed to support easy core clustering for multi-core operation

3. E-core architecture (Gracemont)

- Similar structure but simplified micro-architecture; from Raptor Lake layout, (slide 26) roughly 1/2 area of the P-Core



3. Memory subsystem

- P-Cores
 - ▷ First Level (DCU) – 48KB/8-way; cache line 64B; 5 cycles latency; writeback
 - ▷ Second Level (MLC) – 512KB/8-way; cache line 64B; 13 cycles latency; writeback
 - ▷ Third Level (LLC) – Up to 2MB per core/up to 16 ways; line 64B; latency depends on numbers of cores; writeback
- E-Cores
 - ▷ Two 16-Bytes loads & stores per cycle; load latency is 3–4 cycles
 - ▷ L1 – dual ported almost always fixed size
 - ▷ L2 – 2MB shared among four cores, 64B/cycle with 17 cycles latency
- Huge memory capacity takes a lot of IC area; logic requires massive wiring to enable circuit routing; this is not the case with SRAMs → 3D technology allows to split memory from logic; memory die can use different process and hence make economy on wires and active devices; AMD is already doing this (V-Cache)

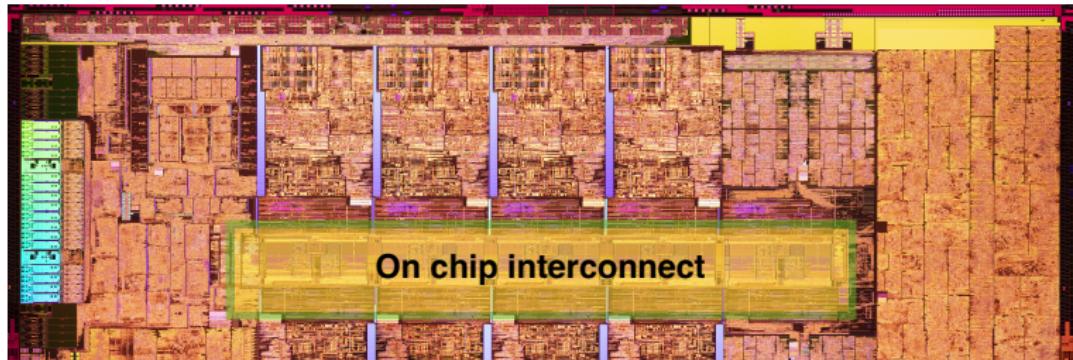
3. On-chip memory / cache sub-system

- More cache capacity mean more system performance; there is a strong correlation between the No of cores & total cache capacity –
multi-processor systems are cache hungry
- L1 capacity same for all
- L3 is more than $2 \times$ L2,
different SRAM cells (R/W
access time vs. density)
- Price follows total on-chip
memory capacity because it
depends on Si area; more
SRAM – more cost; HPC
more tolerant to higher cost
than mobile; compare 1st &
last: $10 \times$ cost diff. for only
 $5 \times$ more memory (profit)

Model	Cores	L2	L3	Price
		MB	MB	US\$
i9-12900	8P,8E	14	30	489
i9-12900F	8P,8E	14	30	464
i7-12700	8P,4E	12	25	339
i7-12700F	8P,4E	12	25	314
i5-12600	6P,0E	7.5	18	223
i5-12500	6P,0E	7.5	18	202
i5-12400	6P,0E	7.5	18	192
i5-12400F	6P,0E	7.5	18	167
i3-12300	4P,0E	5	12	143
i3-12100	4P,0E	5	12	122
i3-12100F	4P,0E	5	12	97
G7400	2P,0E	2.5	6	64
G6900	2P,0E	2.5	4	42

4. On-chip interconnect

- Intel says: **high-bandwidth, low-latency, modular** communication
- For recent architectures not that much information
- Previous iterations used **ring interconnect** to enable scalable communication between cores, GPU, LLC & System Agent
 - ▷ Separate rings: 32 Byte data-path, request, acknowledge & snoop
 - ▷ Fully pipelined at core frequency/voltage: bandwidth, latency and power scale with cores; enables concurrent transfers
 - ▷ Massive ring wire routing runs over the LLC with no area impact
 - ▷ Access on ring picks the shortest path to minimize latency



Other functional blocks

5. System agent

- ▷ SA, AKA uncore controls interconnect (core to core communication), access to LLC (L3), looks for cache coherency and interfaces DRAM controller

6. Graphics & media engine

- ▷ Highly parallel computing system dedicated to image, video, 3D processing & displaying up to 96 Execution Units (EUs)
- ▷ Enables low-power video encoding/decoding & filtering for image processing
- ▷ Graphics Architecture includes 3D compute elements, Multi-format HW assisted decode/encode pipeline, and Mid-Level Cache (MLC) for superior high definition playback, video quality, and improved 3D performance and media

7. DRAM controller and IO

Run-time power/performance management

- Previously done in BIOS (Basic Input/Output System) responsible for HW initialization during boot & **firmware**
- Today this operation is moved from firmware to OS, and makes use of a **dedicated HW core**; not a simple micro-controller since 1M transistors (2010) (equivalent to 486 CPU)
- Run-time management core implements **Advanced Configuration and Power Interface (ACAPI)** – an open standard for platform independent hardware discovery, configuration, power management and monitoring (Intel, HP, etc.)
- ACPI defines different CPU state classes:
 - ▷ **S-States** – running states with 5 sleep levels
 - ▷ **C-States** – on, off and 4 levels of idle states
 - ▷ **P-State** – performance states with 6 V_{dd}, F couples for DVFS
 - ▷ **T-State** – throttle States (little know about these ...)

S-States

S0	CPU fully powered & active it can be in any of P or C states
S1	Minimum live state
S2	CPU is off, but execution context is in DRAM that is powered on
S3	Some other chips on motherboard are turned off
S4	Hibernation – state is saved on HDD, practically no power dissipated
S5	Shut down – 0 power

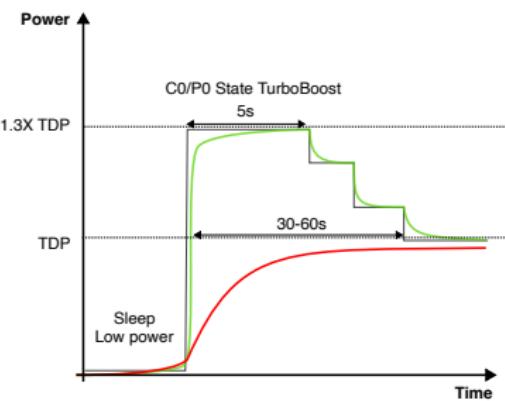
- If the system switches states top-down: we save more power, but moving back to more active state costs wake-up time from less than 1s up to few seconds; **so there will be performance penalty**

C-States & P-States

S0	CPU is fully powered		
	C0		Active mode
		P0	Max frequency (overclocking)
		P1	Guaranteed frequency (no overclocking)
		Pi	Various V_{dd}, F steps
		Pn	Energy efficient state
	C1		No execution, all Clk signals are off (Clk gating)
	C3		Caches are empty, V_{dd} is applied, core is active
	C6		No Vdd for core, system state saved in LLC (almost 0W)
	C7		L3 flushed

P0 – How can we safely overclock?

- Thermal Design Power (TDP) – max. amount of power IC package & cooling system are required to dissipate (given at design time)
- But thermal effects have inertia & are slow compared to clk speed
- Idea – after longer idle periods, the system accumulates “energy budget” that could be used for short high-power, high-performance operation (for up to a minute), after what it stabilizes on TDP, possibly at higher than nominal frequency – Intel TurboBoost technology



Pi-states – frequency

- Implements the concept of **Performance on Demand** to fight **Amdahl's law**: increase temporarily core F , but remain within Silicon limits (power, current & temperature constraints)
- How much more F will depend on **the number of active cores**
- Concrete example of Intel mobile CPU:
 - ▷ Base, nominal, or data sheet frequency of $F=2.5\text{GHz}$
 - ▷ F increased depending on how many cores run @ the same time
 - ▷ Highest possible frequency given to only 1 active core, other cores are used as heatsinks; there is a lot of metal in an IC, i.e. wires that are good thermal conductor and help spreading out heat to package and cooling (for advanced CMOS that is very expensive heat sink)

No cores	No of Turbo Steps	Max F [GHz]
1	10	$3.5 = 2.5 + 10 * 100$
2	9	$3.4 = 2.5 + 9 * 100$
3 or 4	7	$3.2 = 2.5 + 7 * 100$

Pi-states – power/performance trade-off

- P-States actually implement the concept of DVFS; each P-State defines a couple (V_{dd}, F) ; look at some real numbers:

P-States	F [GHz]	V_{dd} [V]	P [W]
P0	1.6	1.484	25
P1	1.4	1.420	17
P2	1.2	1.276	13
P3	1.0	1.164	10
P4	0.8	1.036	8
P5	0.6	0.956	6

For $2.7\times$ less F we get $4.2\times$ power savings!

3. ARM

ARM: company overview

- Created as Acorn RISC Machines (UK, 1983) to become Advanced RISC Machines (1990), both have ARM as acronym
- Today one of the biggest Intellectual Property (IP) providers; ARM does not provide silicon, one buys “soft” descriptions only; license gives access to RTL that is modified to meet specific needs
- Used in vast majority of portable embedded systems: architecture & design choices focus on performance/power ratio (OPS/W)
 - ▷ In 2005 75% of all 32b CPUs based on ARM architecture
 - ▷ Total of 2.5 billions of manufactured ICs licensed an ARM core!
 - ▷ For 2.14 billion mobile phones sold in 2005 ...
- Different core generations: ARM1, ARM2, ARM3, ...; ARM6 used in Apple Newton, first Personal Digital Assistant (PDA) precursor of smartphone; today 32/64 bit cores ARMv8 & ARMv9
- No longer UK company, acquired in 2016 by SoftBank, Japanese telecom giant for 32B\$

Common ARM CPU features

- Highly configurable core architecture (type of core and thus ISA, pipeline depth, HW resources for improved execution, cache sub-system etc.) with some common features
- Instruction compression from 32b to 16b for most frequent instructions (Thumb, Thumb2, ThumbeEE, referred as T); compression improves code density since less memory to store instructions; good for embedded systems with limited memory
- Hardware acceleration for Java programs (Jazelle, or J)
- Advanced DSP instructions (or E in short)
- SIMD extensions on data vectors of 8, 16, 32 et 64b (Neon)
- Extensions for simple et double precision floating point computations (Vector Floating Point Processor – VFP)
- Dedicated HW layer for security (TrustZone)

Processor configurability examples

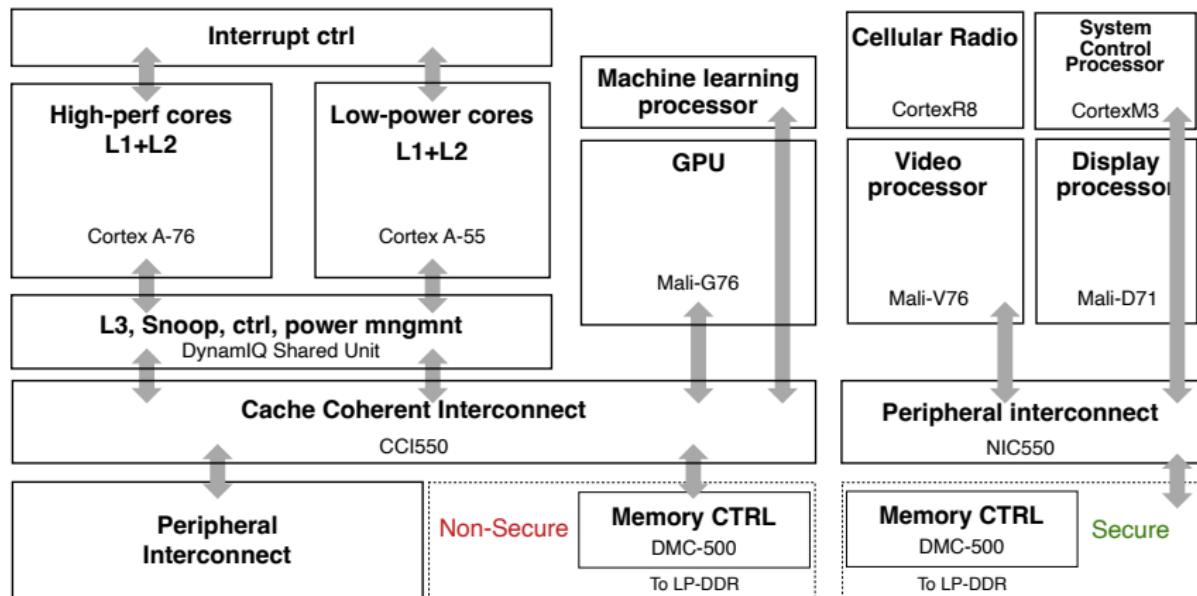
- Application profile – ARMv7-A → Cortex-A8/A9
 - ▷ Highest performance at low power (still with FPU/SIMD)
 - ▷ Memory management support (MMU)
 - ▷ Integrates TrustZone (safety) and Jazelle (HW Java support)
- Real-time profile – ARMv7-R → Cortex-R4
 - ▷ Protected memory (MPU)
 - ▷ Low latency and predictability execution, ideal for real-time needs
 - ▷ Evolutionary path for traditional embedded business
- Microcontroller profile – ARMv7-M → Cortex-M3
 - ▷ Entry point, low-end microprocessors, lowest gate count
 - ▷ Deterministic and predictable behavior is a key priority
 - ▷ Good for small embedded systems

Typical system requirements

- ARM targets low & middle-end market, SoC parameters are **cost driven**
 - CMOS die limited to $<100\text{mm}^2$ (HPC $<400\text{mm}^2$)
 - ▷ In 7nm CMOS this results in ~ 10 billion transistors; note that transistors used for logic are generally bigger than those of on-chip SRAM bit-cells; we want fast logic and dense SRAMs
- Aims high-performance at low-power for wide range of workloads: image/video processing, 2D/3D graphics, machine learning etc.
 - ▷ For mobile applications CPU power dissipation is limited to $\sim 5\text{W}$ and for short bursts of time
 - ▷ Above limitations are due to limited ability to dissipate heat (no fancy cooling mechanism in mobile) and limited battery capacity (these days $\sim 10.000\text{mWh}$)
 - ▷ Laptops & tablets can sustain higher power consumption since larger batteries & better cooling (even ARM based servers)
- Off-chip memory bandwidth $<50\text{GB/s}$ (Low-Power or LP-DDR5)

Typical SoC architecture

- System architects assemble top-level SoC out of many blocks



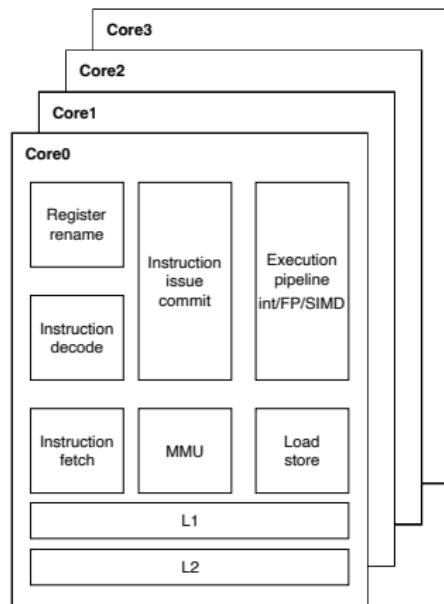
Core processing capabilities

- Processing resources use heterogeneous ISA, or Asymmetric Multi-Processing (ASMP) to trade-off performance & power
- Architecture big.LITTLE (now DynamIQ):
 - ▷ 2 types of cores: “big” cores designed for performance & “LITTLE” cores designed for power efficiency; both are RISC, variable pipeline depth, pure load/store
 - ▷ “big” cores are implemented using low V_t transistors that allow higher switching frequency, but also have higher dynamic & static (leakage) power; “LITTLE” cores use high V_t transistors and will run at lower F, V_{dd} pairs thus using less dynamic F and leakage power
 - ▷ threads with high priority & computational intensity are allocated to “big” cores; threads with less priority & less computational intensity (e.g., background tasks) are scheduled on “LITTLE” cores
- OS adjust dynamically V_{dd}, F at any time to trade-off power & performance; OS can also migrate tasks efficiently from big to LITTLE core to save energy or boost execution

Practical “big” core

A76 – designed to run up to 3GHz (depends on target CMOS tech)

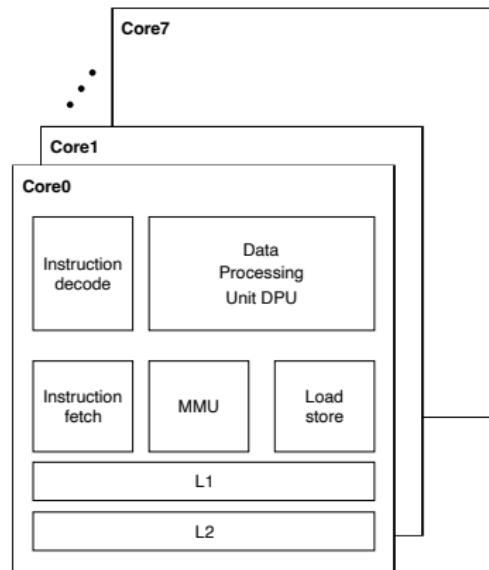
- 4-way superscalar, OoO, 128 instruction window, 13 pipeline stages
- 40-bit Physical Address, MMU, core bus width 128 or 256-bits
- L1 – private, split for instructions & data, configurable up to 64kB; data L1 is 4-way set-associative, with 64B cache line; pseudo-LRU cache replacement policy; 256-bit W/R interface to L2
- L2 – private (128, 256 or 512kB), unified, with configurable banking factor; cache lines have a fixed length of 64 bytes, implements MESI cache coherence protocol
- Up to 4 cores inferred in a cluster



Practical “LITTLE” core

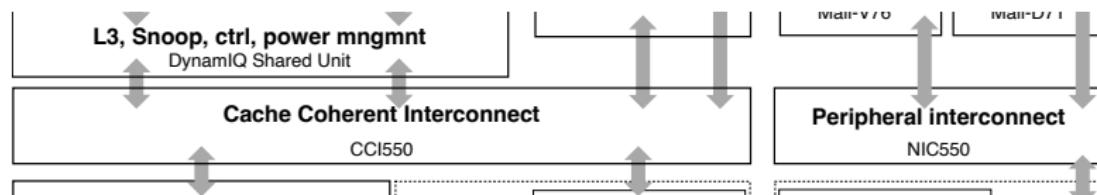
A55 – designed to run up to 2GHz (dependent on target tech)

- 2-way superscalar, in-order, 8 pipeline stages with optional floating point & SIMD ALU (DPU); possible HW security accelerators (AES); more simpler when compared to “big” core
- L1/L2 cache options similar as in “big” core: L1 up to 64kB, L2 up to 256; but control is simplified; typically have less capacity compared to “big” core cluster
- Optional L3 could be shared with “big” cores; if tasks migrate, data can be fetched from L3 rather than main memory
- Integrated MMU
- Cluster could infer up to 8 different core instances



System interconnect

- Many master & slave nodes are integrated in a single SoC; need for **high-efficiency communication infrastructure**
- One high-performance interconnect (CoreLink CCI-550); based on **AMBA protocol**: open-standard interconnect specification (ARM); only protocol specification is given, designers do actual implementation
 - ▷ Provides **cache coherent** interconnect between CPUs & other blocks
 - ▷ Implemented with crossbars, control logic & “snoop HW” to keep record of addresses stored in the caches of the attached cores
 - ▷ Resolves coherence requests without the need to broadcast to all nodes; reduces both system power & communication latency
- Less stringent interconnect used to connect peripherals (e.g. NIC-550)



Peripheral processing capabilities

- **Cellular Radio Processor** – complex communications standards are implemented using programmable processors (note plural) for flexibility; they also have **hard real-time** requirements to avoid dropping data
- **Sensor Processor** – GPS, accelerometer, gyroscope, touch, microphone, etc.; DSP capability; since “always-on” focuses on power efficiency
- **System Control Processor (SCP)** – controls clocks and resets, power state transitions & different power domains in the SoC; coupled with **Power-Management Integrated Circuit (PMIC)**
- **Video Processor** for (de)encoding using different standards
- **Display Processor** – scaling, rotation, composing layers, picture quality enhancements; limits GPU workload, for power savings; also backlight management, ambient light optimizations, etc.
- **GPU** – specialized core for high-quality graphics: 3D games, augmented & virtual reality; other compute intensive task could run on it
- **Machine Learning Accelerator** – for compute-intensive workload (e.g. Convolutional Neural Networks) that benefit from HW acceleration

Trust Zone – processing

- HW/SW support to protect valuable data; protected data **can't be copied, damaged, or made unavailable to genuine users**; aims
 - ▷ **Confidentiality** – data can't be seen for defined set of attacks
 - ▷ **Integrity** – protected modification by a defined set of attacks
 - ▷ **Authenticity** – data that is known not to modified externally
- Cores are **split in two bins: non-secure & secure**; non-secure cores access only non-secured resources; secured ones see all resources
- But to avoid truly dedicated cores, costly in silicon area & power TrustZone provides architectural extensions to allow one core to execute code from non-secure AND secure worlds; one physical processor appears as two virtual processors: secure & normal
- HW block (ARM call it Secure Monitor) acts as a gatekeeper for moving between the two worlds and makes sure that secured state remains inaccessible when moving back to normal state

Trust Zone – Memory

- Physical memory map is also partitioned into **normal** & **secure** memory spaces; as in processing, so:
 - ▷ virtual addresses from non-secure state can only map to non-secure physical addresses
 - ▷ virtual addresses in secure state can map to either secure or non-secure physical addresses
- Memory transactions have a security attribute to indicate secure or non-secure access
- Each memory transaction must indicate the security state of the address it is trying to access; note that in this implementation, caches carry information about which physical address space was accessed, NOT which Virtual Address (VA) space

4. Another interesting architecture: SPARC

History & overview

- SPARC – Scalable Processor ARChitecture; one of earliest RISC processors, developed at UC Berkeley in '80
- Commercialized by SUN Microsystems, Oracle & Fujitsu
- 1986: first SPARC processor implemented by Sun/Fujitsu
- 1989: SPARC International founded: SPARC Architecture becomes an open standard
- 1990: SPARC V8, 32-bit architecture (standard IEEE-1754)
- 1993: SPARC V9, 64-bit architecture
- 2009: Fujitsu, Venus core, used in K-supercomputer (2011)
- 2013: Oracle T5 multi-processors for high-performance servers
- 2015: Oracle M7 32 core SoC
- Today: Fujitsu M12 servers, will be discontinued in 2029, but supported until 2034

SPARC architecture – origins

SPARC is a CPU instruction set architecture (ISA), derived from a reduced instruction set computer (RISC) lineage. As an architecture, SPARC allows for a spectrum of chip and system implementations at a variety of price/performance points for a range of applications, including scientific/engineering, programming, real-time, and commercial. SPARC was designed as a target for optimizing compilers and easily pipelined hardware implementations. SPARC implementations provide exceptionally high execution rates and short time-to-market development schedules. For languages such as C++, where object-oriented programming is dominant, register windows result in an even greater reduction in instructions executed. Note that supervisor software, not user programs, manages the register windows. A supervisor can save a minimum number of registers (approximately 24) at the time of a context switch, thereby optimizing context switch latency.

SPARC ISA — main characteristics

- RISC with explicit load&store instructions to access memory & IO
- Only few addressing modes: “reg+reg” or “reg+immediate”
- **Triadic register addresses** – most of instructions work on 2 operands and place the results in the 3rd

```
1 add %g1, %g2, %g3 -- g3 = g1 + g2
2 sub %g1, %g2, %g3 -- g3 = g1 - g2
3 and %g1, %g2, %g3 -- g3 = g1 AND g2
4 neg %g1, %g2      -- g2 = - g1
```

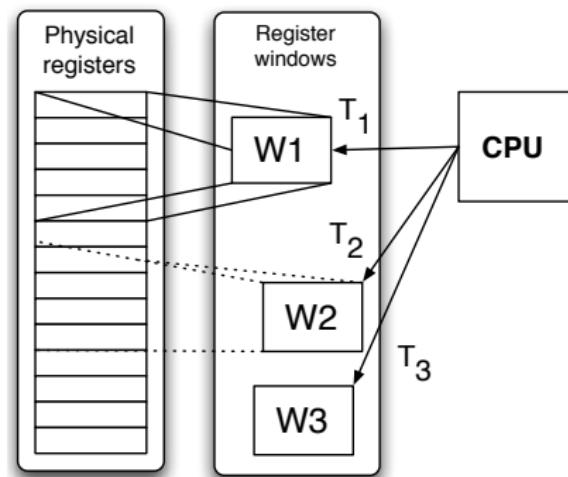
- **Register Windowing** – large windowed register file, the most prominent feature of SPARC architecture; allows stack content (so arguments, locales variables and addresses) to remain in the fastest memory to minimize overheads of function calls
- Separate register file for floating point data types depending on precision: simple, double, quad (32, 64 or 128-bits)

Register Windowing – motivation

- Central (global) memories are made using DRAMs for massive storage, they are **slow and energy inefficient**
- To keep up with increasing CPU speed typical systems use **memory hierarchy** using faster, but smaller SRAMs
- **What happens when we have a sub-routine call?**
(subroutines are mandatory to write complex, reusable programs)
- **PUSH & POP of the stack to save current execution context;
this means memory move into the global (slow) memory**
- If there are many nested function calls (imagine recursive functions) and very slow main memory, **performance will be poor**
...
- Same goes when system has to do multi-tasking and need to switch from one process to another (one thread to another), a typical use-case scenario in servers

Register Windowing – how it works

- Huge number of physical registers instantiated at design time (from 52 to 524!, hence scalable in the SPARC acronym)
- Few register grouped in **register window W** (example: 4 groups of 8 registers each)
- One register window (W_i) is attributed to one function call
- Function call assigns active W_i used by the CPU, so no PUSH & POP to central memory
- Problem: size of reg file & associated CMOS cost! Real platforms limit to 8 register windows at most

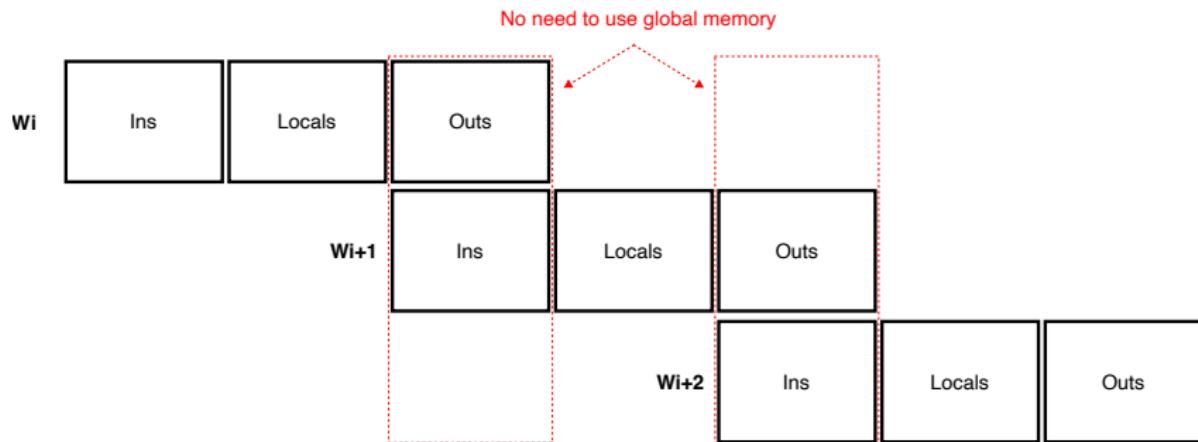


Register Windowing – usage

- At any moment, a programmer has access to 32 registers
- 8 registers are global, thus available to any function
- 24 other registers are part of the **register sliding window**
- A sliding window contains three types of registers:
 - ▷ **Input registers** – arguments are passed to a function using these registers, instead using the stack to end up in main memory
 - ▷ **Local registers** – used to store any local data
 - ▷ **Output registers** – when calling a function, one can put the arguments in these registers
- Programmers have access to 8 registers of each type
- Some of these registers have a special purpose and are not used to store local data

Function call: operation

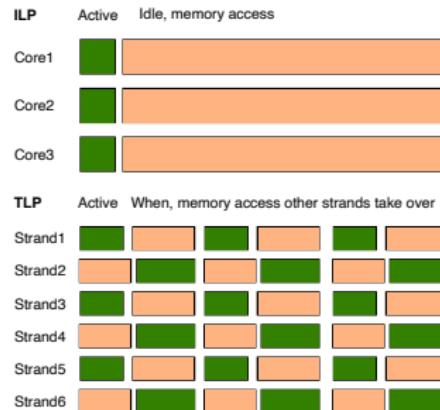
- When a function is called, the register window “slides”: the output registers of one function become input registers for called function
- A new set of local registers and output registers are provided
- When all available windows are taken, CPU frees oldest window by dumping its content to the stack



General purpose vs. high-perf server

- Server workloads are characterized by:
 - ▷ High Thread-Level Parallelism (TLP) – many threads of limited complexity
 - ▷ Low instruction-level parallelism (ILP) on per thread basis; limited opportunity to boost single-thread performance with complex cores
 - ▷ Majority of CPI is a result of off-chip misses – core computing capacities remain unused
- Traditional x86 architecture used in Chip Multi Processors (CMP)
 - ▷ Tend to maximise ILP
 - ▷ Super-scalar – many computational resources, but allocated on per thread basis (if a thread uses only integer pipeline, the floating point one remains unused), hence hyper-threading
- Not suited for this particular type of applications

ILP and TLP



- In ILP core is often idle waiting for memory (CISC like etc.)
- Idea: increase the number of small, single-issue cores working with multiple threads within the same IC (TLP) – **strands**

- Memory stall time is overlapped with execution of other threads on same physical processor core, and multiple physical processor cores run their threads in parallel
- ILP shortens time to execute instructions, but doesn't help much in overlapping execution with memory latency (out-of-order ILP hides some latency but limited to L1, L2, not central memory)
- With TLP **memory latency could be overlapped with execution of other threads** – fine grain parallelism

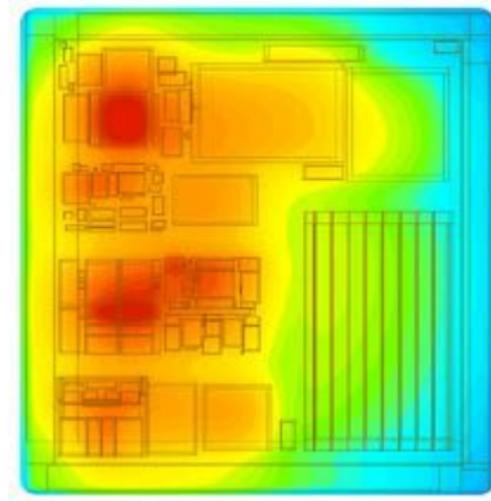
Single Chip MultiThreaded (CMT)

- CMP – one thread per core (before hyper-threading)
 - ▷ When a thread is stalled waiting for off-chip misses to complete, thread utilization of the core's execution resources is low
- Single Chip MultiThreaded (CMT) (as opposed to CMP) share CPU resources among all available threads, when one thread is stalled, the other can take over with minimum overhead time
- CMT idea: run few threads per core
 - ▷ Many simultaneous HW threads, **strands** in SPARC vocabulary, of execution per core: Simultaneous MultiThreading (SMT)
 - ▷ SMT enable multiple strands to share different resources within the core, namely the execution units (because cores are super-scalar)
 - ▷ SMT improves the utilization of key resources and reduces the sensitivity of an application to off-chip misses
 - ▷ Similarly, as with CMP, multiple cores can share chip resources such as the memory controller, off-chip bandwidth, and the level-2/level-3 cache, improving the utilization of these resources

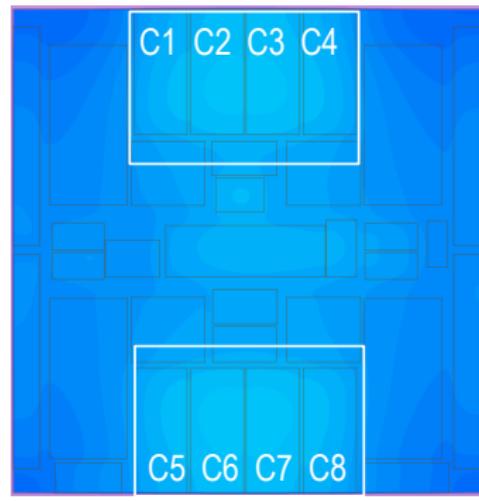
Benefits of the CMT

- More computational power per area, less CPUs per server
- Reduced power consumption; for equivalent computational load, CMT ICs run cooler, burn less power and require less cooling
- Total (chip+ownership) cost is lower than CMP solutions

Single-Core Processor

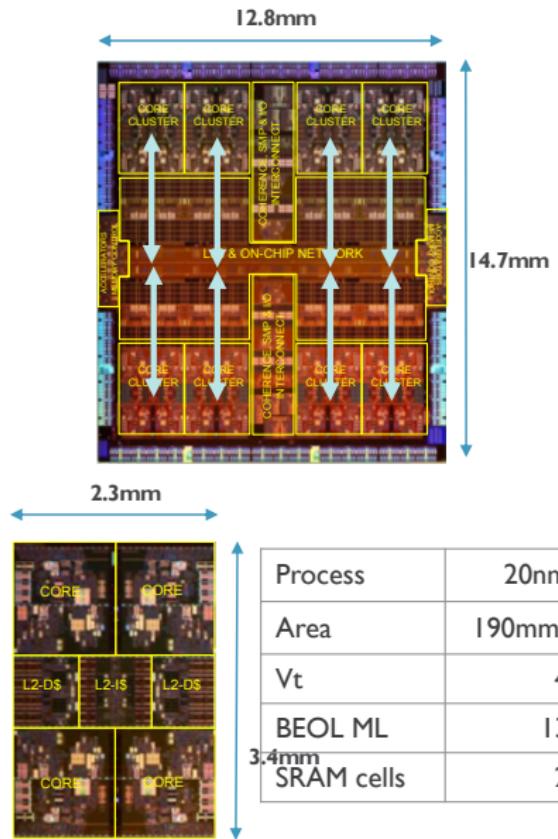


CMT Processor



SPARC these days: M7

- 8 SPARC Core Clusters (SCC) – 33% of total die area, 4 cores per SCC, 32 cores total
- L1 – 2MB total: 32kB instruction + 32kB data per core
- L2 – 6MB total: 8% of total die area 768kB per SCC
 - ▷ 1x256kB instruction (shared among all cores)
 - ▷ 2x256KB data (shared per core pair)
- L3 – 64MB total: 25% of total die area 8MB per SCC (fully shared)



5. Recent many-core architecture

Highly configurable SoC for low-power AI

- General purpose many-core SoC: up to 1024 cores!⁵
- Each core is simple RISC-V ISA 32-bit processors – single issue, in-order core needing only 12kGates/core (this is very simple!)
- SoC has 3 levels of hierarchy:
 - ▷ Tile – composed of 4 cores
 - ▷ Group – composed of power of 2 tiles
 - ▷ cluster – top-level IC build from four groups
- Each core has L1 with configurable number of banks; at group level power of 2 parallel DMA channels can be defined towards L2
- Fully shared memory system: any core can access any memory through inter-core interconnect designed to minimize communication latency:
 - ▷ Only 1, 3 & 5 cycles latency inside tile, group & cluster respectively

⁵M. Cavalcante, et al., MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect, DATE2021

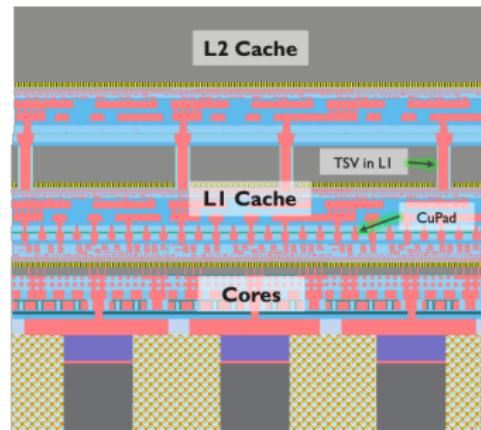
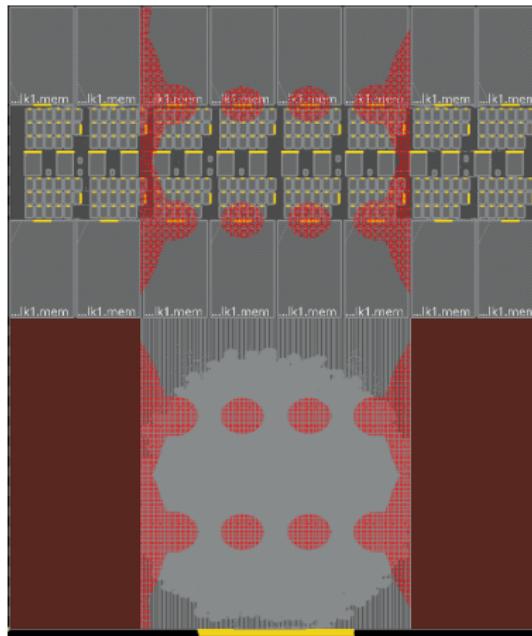
2D implementation example and layout

- SoC – 64 cores; 4 cores/tile, 4 tiles/group, 4 groups/cluster
- L2 is accessed using 16 parallel DMA channels, each channel carries 512 bits; each L2 macro delivers 512-bits in a single access
- **That is too many interconnect!**; real-life 2D design would probably use less channels and narrower data words



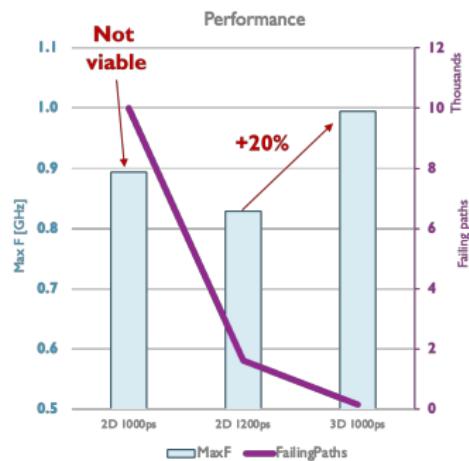
3D implementation

- Split all macros from logic & implement them on top
- But the memory is 2/3 of total silicon area
- More compact implementation is possible using 3 dies stack



2D vs. 3D Performance & Power

- 2D implementation with 1000ps minimum period doesn't work: too many paths don't reach timing (10k); 1200ps is more realistic
- In 3D 1000ps can be reached with only few failing paths



- Total power in 3D remains constant despite operating at 20% more frequency; 3D can either improve performance, or save power, as with half-node CMOS scaling

Why it works?

- Better performance/power is possible because 3D saves wires
 - ▷ 3D has 30% less total system wire length
- Collateral benefit: 3D design uses less area and is thus cheaper

