High Performance Processor Architecture

André Seznec
IRISA/INRIA
ALF project-team

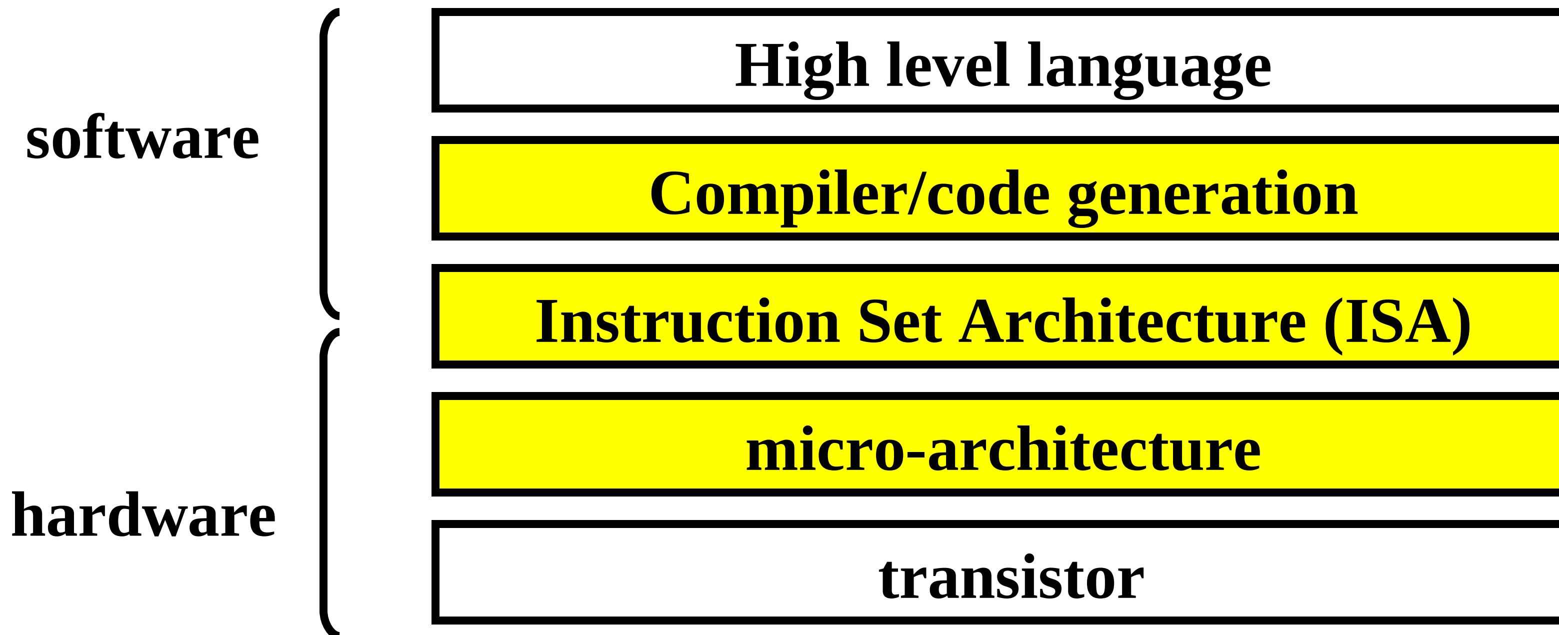
Moore's « Law »

- Nb of transistors on a micro processor chip doubles every 18 months
  - 1972: 2000 transistors (Intel 4004)
  - 1979: 30000 transistors (Intel 8086)
  - 1989: 1 M transistors (Intel 80486)
  - 1999: 130 M transistors (HP PA-8500)
  - 2005: 1,7 billion transistors (Intel Itanium Montecito)
- Processor performance doubles every 18 months
  - 1989: Intel 80486 16 Mhz (< 1inst/cycle)
  - 1993 : Intel Pentium 66 Mhz x 2 inst/cycle
  - 1995: Intel PentiumPro 150 Mhz x 3 inst/cycle
  - 06/2000: Intel Pentium III 1Ghz x 3 inst/cycle
  - 09/2002: Intel Pentium 4 2.8 Ghz x 3 inst/cycle
  - 09/2005: Intel Pentium 4, dual core 3.2 Ghz x 3 inst/cycle x 2 processors

Not just the IC technology

- VLSI brings the transistors, the frequency, ..
- Microarchitecture, code generation optimization bring the effective performance

The hardware/software interface



Instruction Set Architecture (ISA)

- Hardware/software interface:
  - The compiler translates programs in instructions
  - The hardware executes instructions
- Examples
  - Intel x86 (1979): still your PC ISA
  - MIPS , SPARC (mid 80's)
  - Alpha, PowerPC (90' s)
- ISAs evolve by successive add-ons:
  - 16 bits to 32 bits, new multimedia instructions, etc
- Introduction of a new ISA requires good reasons:
  - New application domains, new constraints
  - No legacy code

Microarchitecture

- macroscopic vision of the hardware organization
  - ➔ Nor at the transistor level neither at the gate level
  - ➔ But understanding the processor organization at the functional unit level

What is microarchitecture about ?

- Memory access time is 100 ns
- Program semantic is sequential

- But modern processors can execute 4 instructions every 0.25 ns.

  – **How can we achieve that ?**

high performance processors everywhere

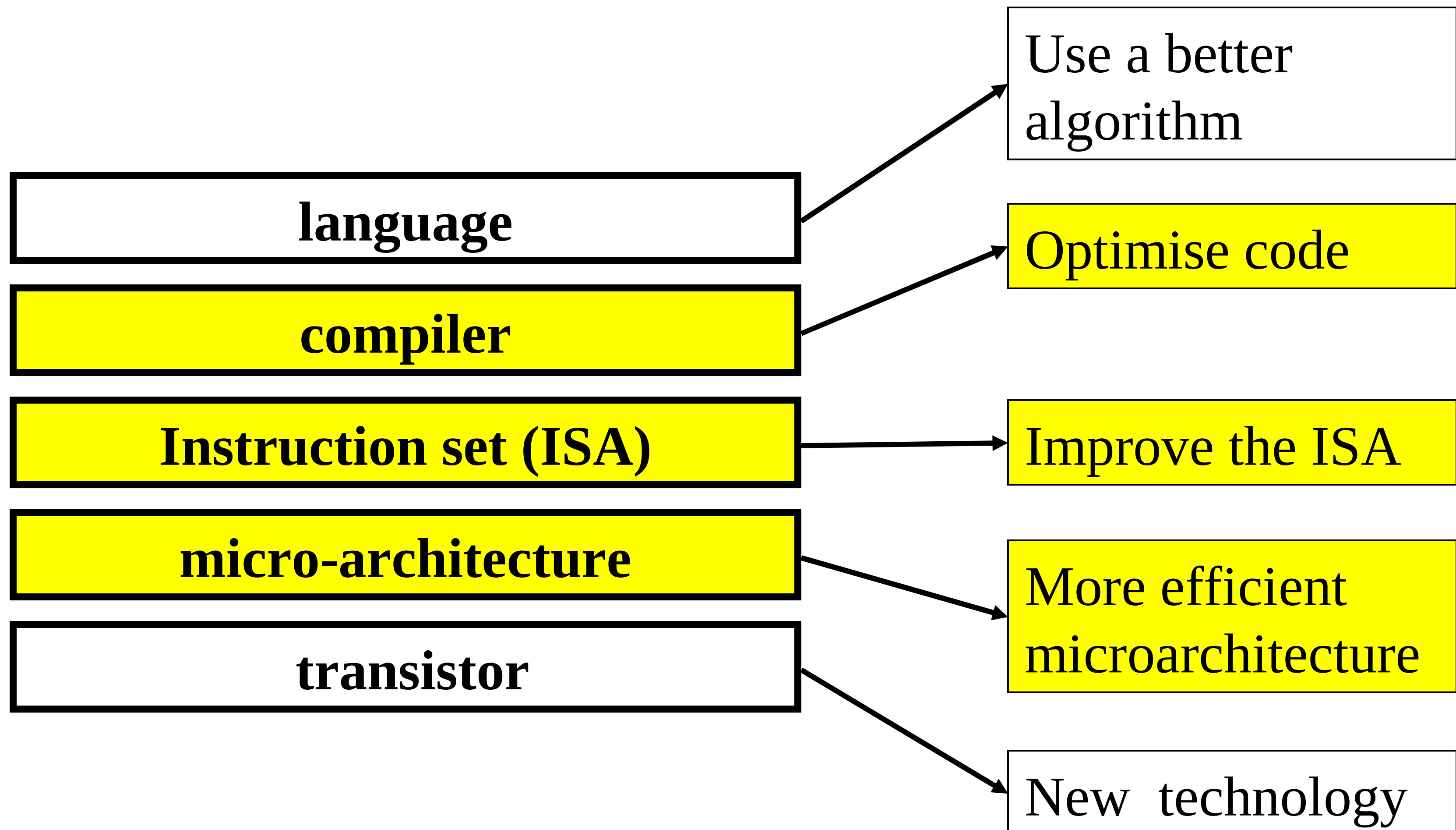
- General purpose processors (i.e. no special target application domain):
  - ➔ Servers, desktop, laptop, PDAs
- Embedded processors:
  - ➔ Set top boxes, cell phones, automotive, ..,
  - ➔ Special purpose processor or derived from a general purpose processor

Performance needs

- Performance:
  - Reduce the response time:
  - Scientific applications: treats larger problems
  - Data base
  - Signal processing
  - multimedia
  - …
- Historically over the last 50 years:

Improving performance for today applications has fostered new even more demanding applications

How to improve performance?



How are used transistors: evolution

- In the 70's: enriching the ISA
  - Increasing functionalities to decrease instruction number
- In the 80's: caches and registers
  - Decreasing external accesses
  - ISAs from 8 to 16 to 32 bits
- In the 90's: instruction parallelism
  - More instructions, lot of control, lot of speculation
  - More caches
- In the 2000's:
  - More and more:
  - Thread parallelism, core parallelism

A few technological facts (2005)

- Frequency : 1 - 3.8 Ghz
- An ALU operation: 1 cycle
- A floating point operation : 3 cycles
- Read/write of a registre: 2-3 cycles
  - ➔ Often a critical path ...
- Read/write of the cache L1: 1-3 cycles
  - ➔ Depends on many implementation choices

A few technological parameters (2005)

- Integration technology: 90 nm – 65 nm
- 20-30 millions of transistor logic
- Cache/predictors: up one billion transistors
- 20 -75 Watts
  - 75 watts: a limit for cooling at reasonable hardware cost
  - 20 watts: a limit for reasonable laptop power consumption
- 400-800 pins
  - 939 pins on the Dual-core Athlon
-

The architect challenge

- 400 mm2 of silicon
- 2/3 technology generations ahead
- What will you use for performance ?
  - → Pipelining
  - → Instruction Level Parallelism
  - → Speculative execution
  - → Memory hierarchy
  - → Thread parallelism

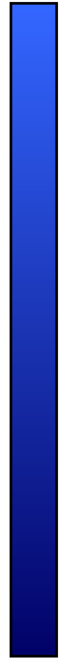
Up to now, what was microarchitecture about ?

- Memory access time is 100 ns
- Program semantic is sequential
- Instruction life (fetch, decode,..,execute, ..,memory access,..) is 10-20 ns

- **How can we use the transistors to achieve the highest performance as possible?**
  - ➔ **So far, up to 4 instructions every 0.3 ns**

The architect tool box for uniprocessor performance

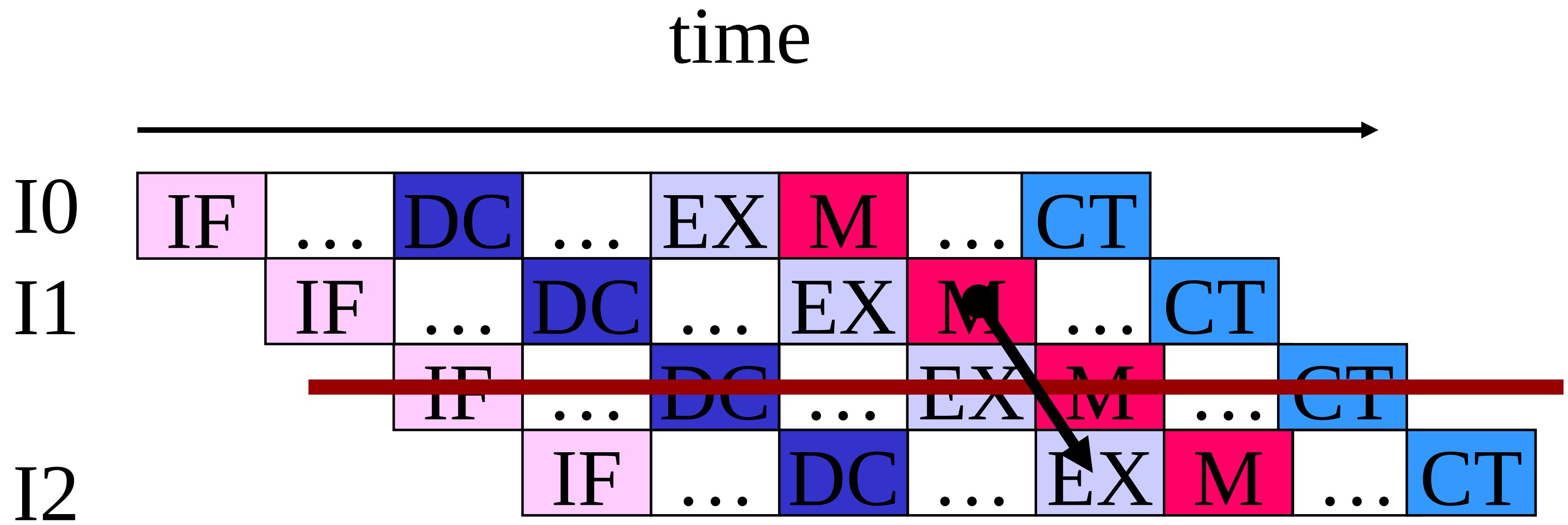
- Pipelining
- Instruction Level Parallelism
- Speculative execution
- Memory hierarchy



Pipelining

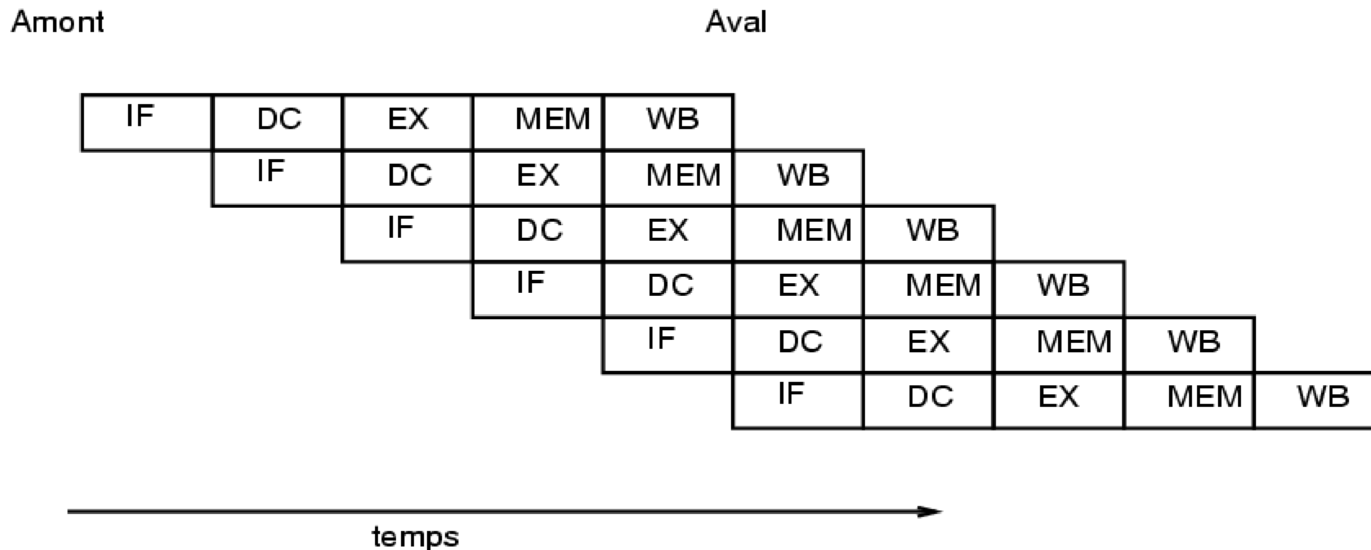
Pipelining

- Just slice the instruction life in equal stages and launch concurrent execution:



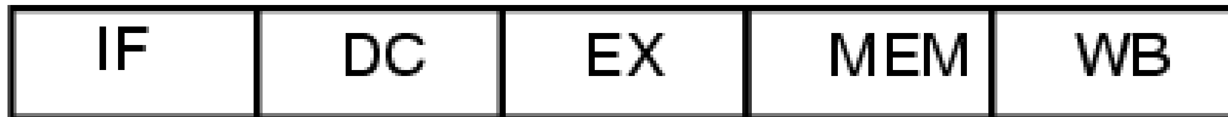
Principle

- The execution of an instruction is naturally decomposed in successive logical phases
- Instructions can be issued sequentially, but without waiting for the completion of the one.

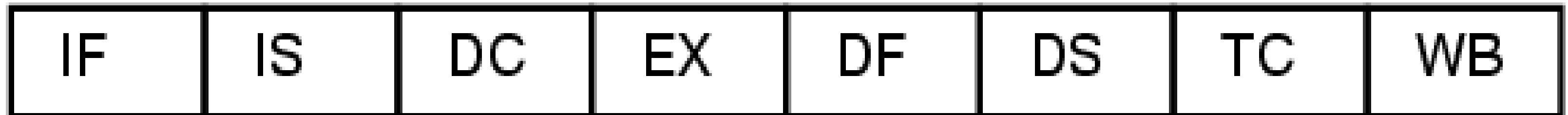


Some pipeline examples

- MIPS R3000 :



- MIPS R4000 :



- Very deep pipeline to achieve high frequency
  - ➔ Pentium 4: 20 stages minimum
  - ➔ Pentium 4 extreme edition: 31 stages minimum

pipelining: the limits

- Current:
  - ➔ 1 cycle = 12-15 gate delays:
    - Approximately a 64-bit addition delay

- Coming soon ?:
  - ➔ 6 - 8 gate delays
  - ➔ On Pentium 4:
    - ALU is sequenced at double frequency
      - a 16 bit add delay

Caution to long execution

- Integer :
  - → multiplication : 5-10 cycles
  - → division : 20-50 cycles

- Floating point:
  - → Addition : 2-5 cycles
  - → Multiplication: 2-6 cycles
  - → Division: 10-50 cycles

Dealing with long instructions

- Use a specific to execute floating point operations:
  - E.g. a 3 stage execution pipeline

- Stay longer in a single stage:
  - Integer multiply and divide

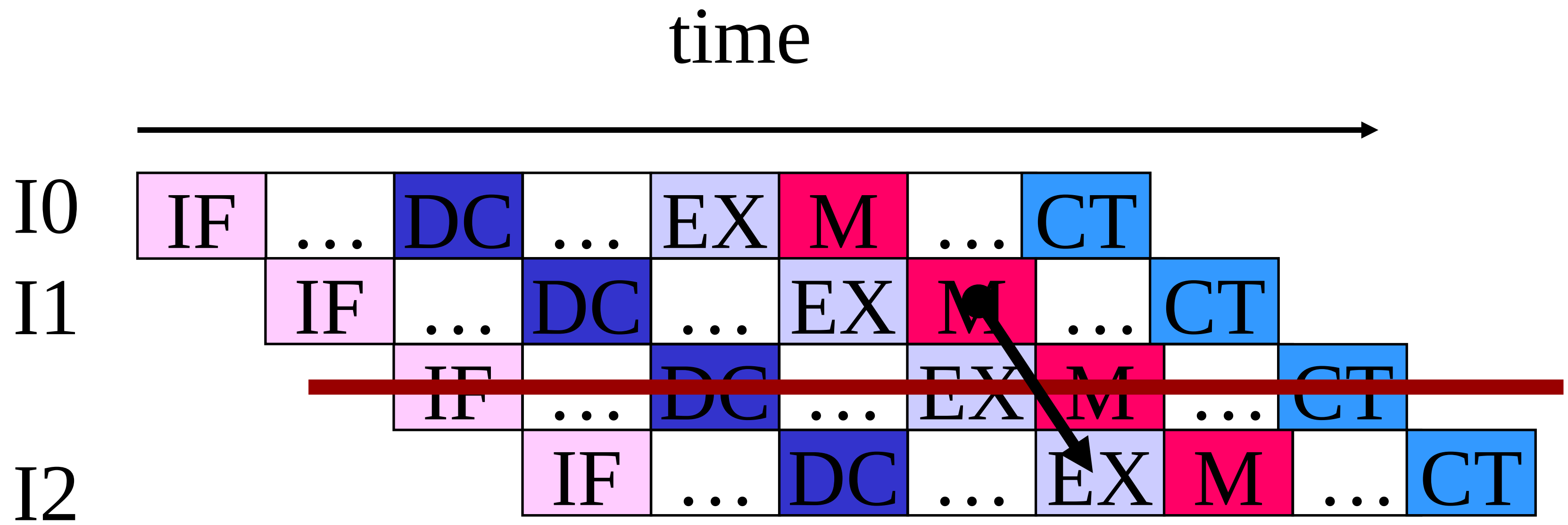
sequential semantic issue on a pipeline

There exists situation where sequencing an instruction every cycle would not allow correct execution:

- Structural hazards: distinct instructions are competing for a single hardware resource
- Data hazards: J follows I and instruction J is accessing an operand that has not been acccessed so far by instruction I
- Control hazard: I is a branch, but its target and direction are not known before a few cycles in the pipeline

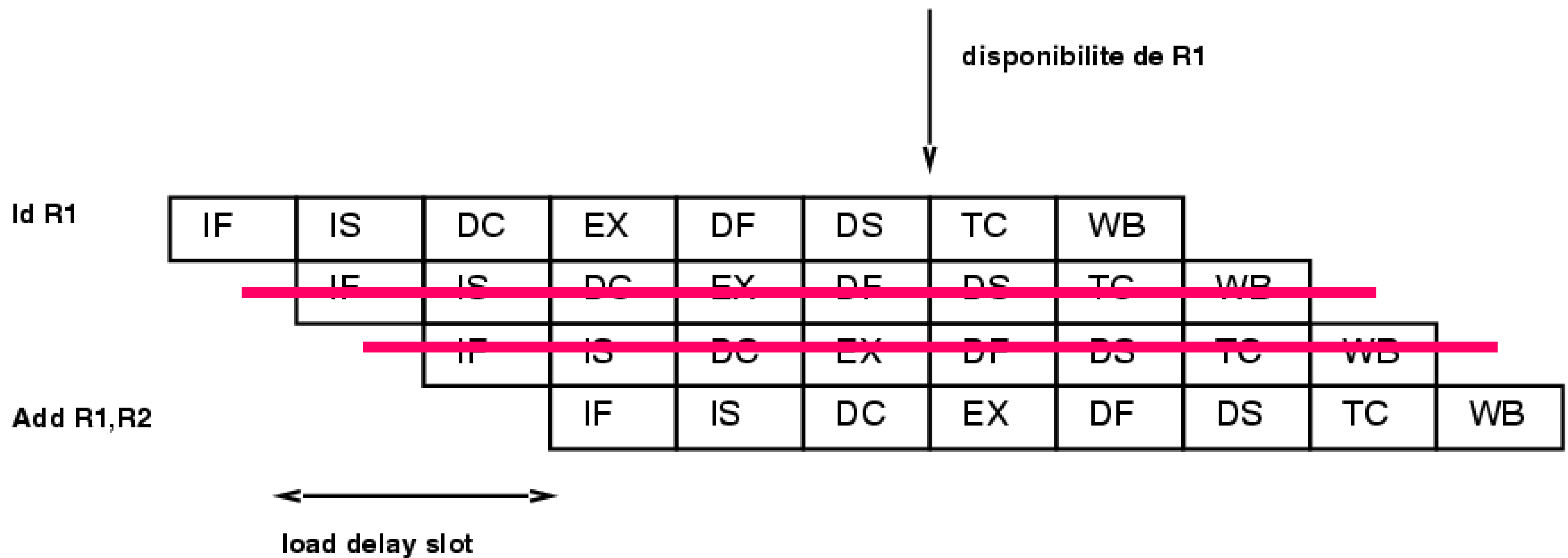
Enforcing the sequential semantic

- Hardware management:
  - ➔ First detect the hazard, then avoid its effect by delaying the instruction waiting for the hazard resolution



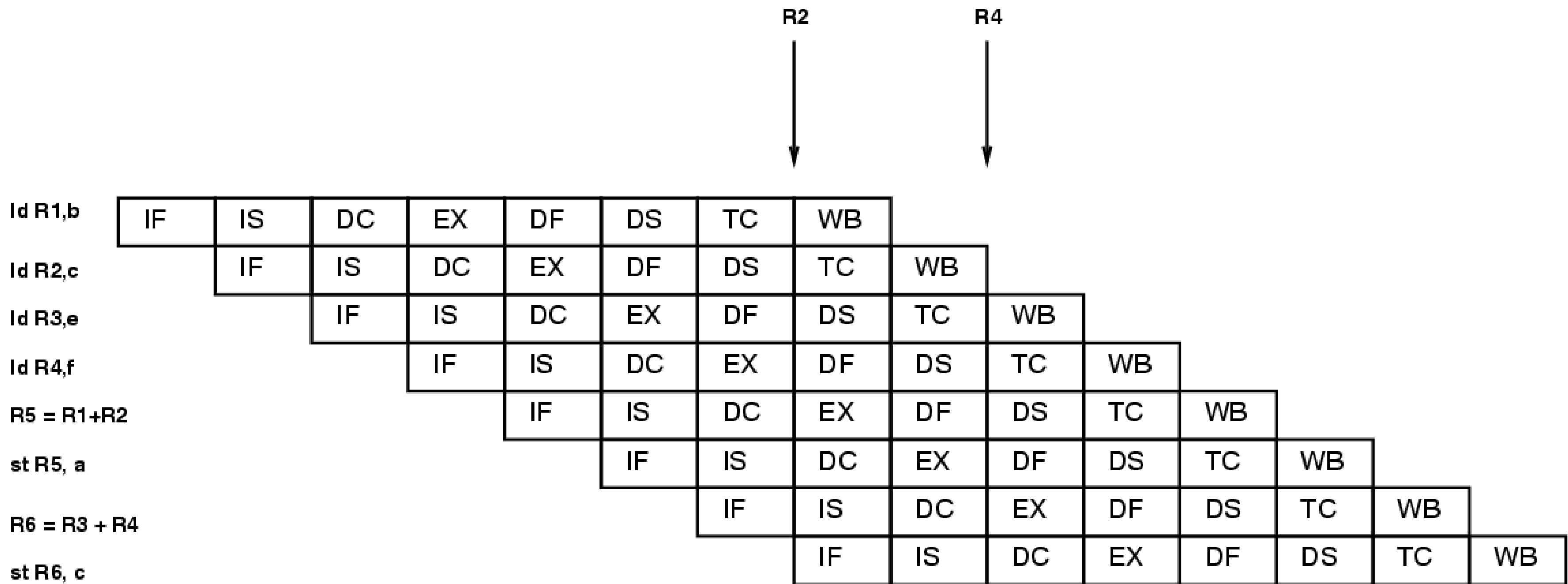
Read After Write

- Memory load delay:



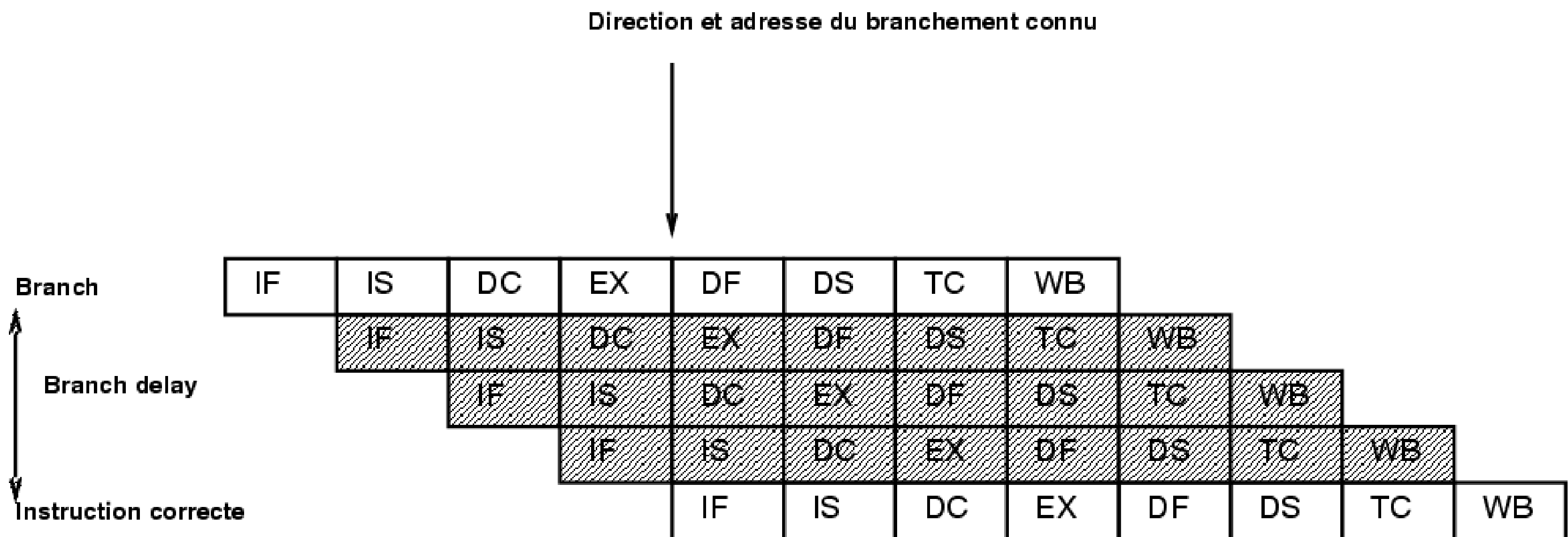
And how code reordering may help

- a= b+c ; d= e+f



Control hazard

- The current instruction is a branch (conditional or not)
  - ➔ Which instruction is next ?

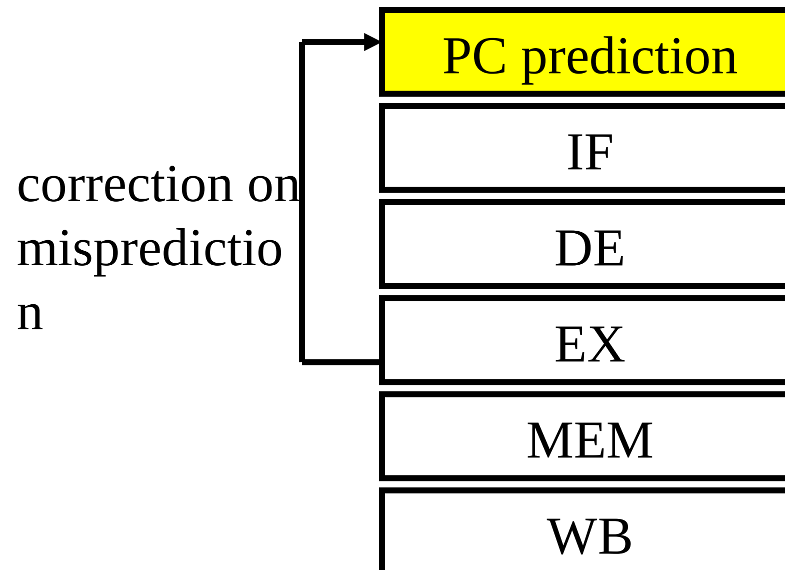


- Number of cycles lost on branchs may be a major issue

Control hazards

- 15 - 30% instructions are branchs
- Targets and direction are known very late in the pipeline. Not before:
  - Cycle 7 on DEC 21264
  - Cycle 11 on Intel Pentium III
  - Cycle 18 on Intel Pentium 4
- X inst. are issued per cycles !
- Just cannot afford to lose these cycles!

Branch prediction / *next instruction prediction*



Prediction of the address of
the next instruction (block)

Dynamic branch prediction: *just repeat the past*

- Keep an history on what happens in the past, and guess that the same behavior will occur next time:
  - essentially assumes that the behavior of the application tends to be repetitive
- Implementation: hardware storage tables read at the same time as the instruction cache
- What must be predicted:
  - Is there a branch ? Which is its type ?
  - Target of PC relative branch
  - Direction of the conditional branch
  - Target of the indirect branch
  - Target of the procedure return

Predicting the direction of a branch

- It is more important to correctly predict the direction than to correctly predict the target of a conditional branch
  - ➔ PC relative address: known/computed at execution at decode time
  - ➔ Effective direction computed at execution time

Prediction *as the last time*

direction	prediction	
1		
1	1	
1	1	
0	1	mipredict
1	0	mispredict
1	1	
1	1	
0	1	mispredict
1	0	mispredict
⋮		

```
for (i=0;i<1000;i++) {
    for (j=0;j<N;j++) {
      loop body
    }
}
```

2 mispredictions on the first and the last iterations

Exploiting more past: inter correlations

B1: if cond1 and cond2 …

B2: if cond1 …

cond1	cond2	cond1 AND cond2
T	T	T
T	N	N
N	T	N
N	N	N

Using information on B1 to predict B2

If *cond1 AND cond2* true (p=1/4), predict *cond1* true ⇨ 100 % correct

Si *cond1 AND cond2* false (p= 3/4), predict *cond1* false ⇨ 66% correct

Exploiting the past: auto-correlation

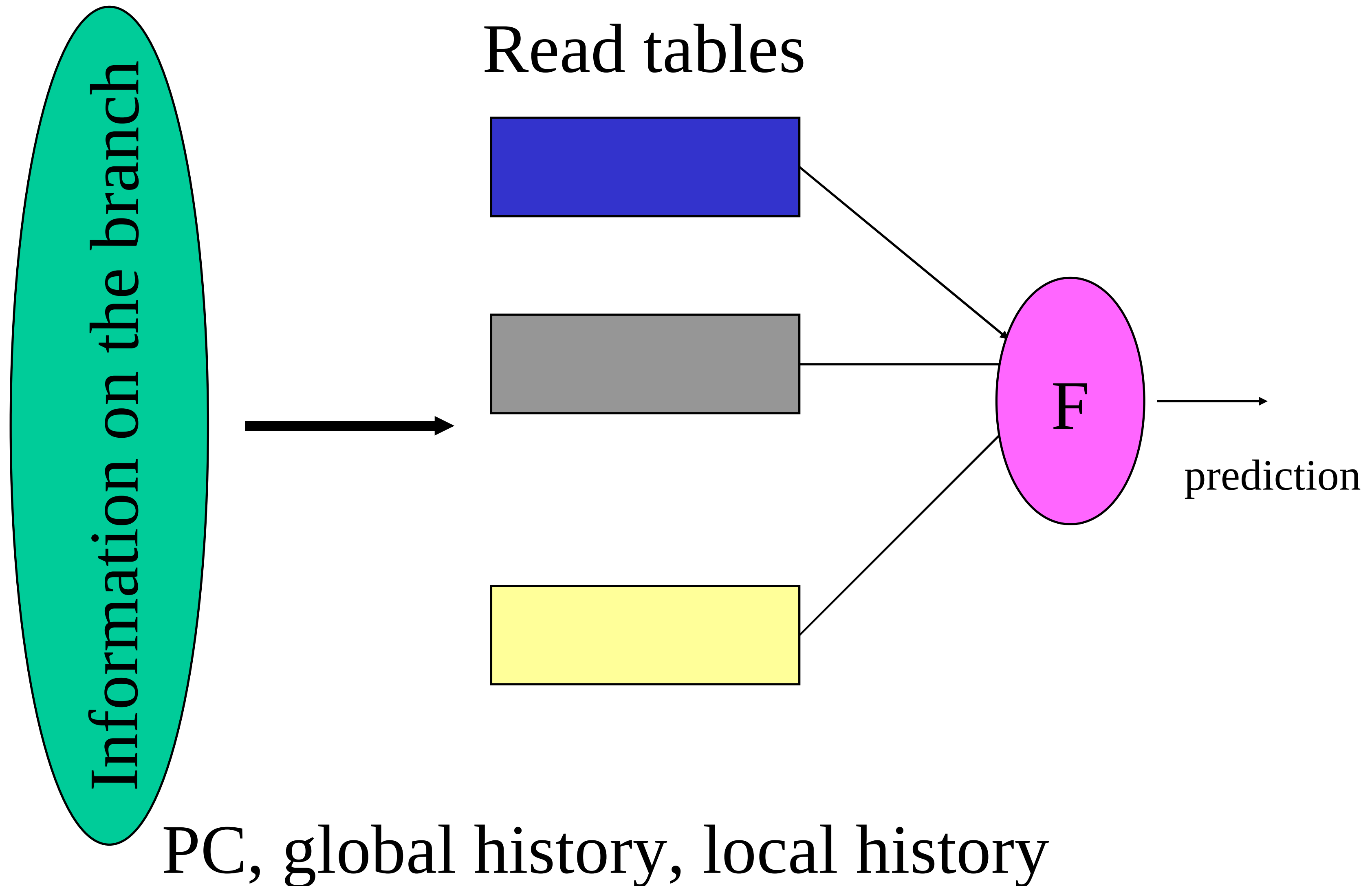
When the last 3 iterations are taken then predict not taken, otherwise predict taken

⇒ 100 % correct

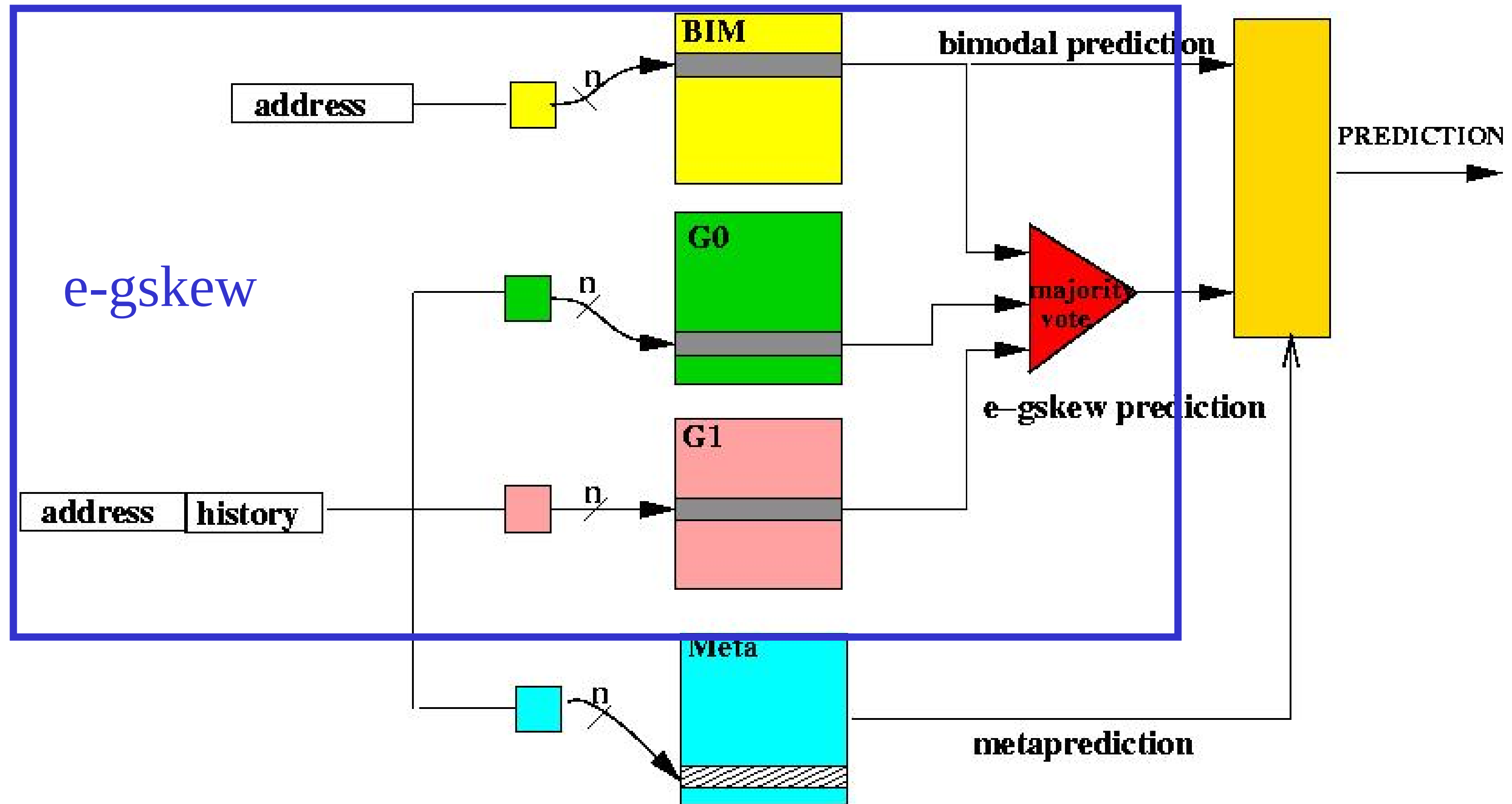
1
1
1
0
1
1
1
0
1
1
1
0

```
for (i=0; i<100; i++)
    for (j=0;j<4;j++)
        loop body
```

General principle of branch prediction



Alpha EV8 predictor: (*derived from)* (2Bc-gskew)

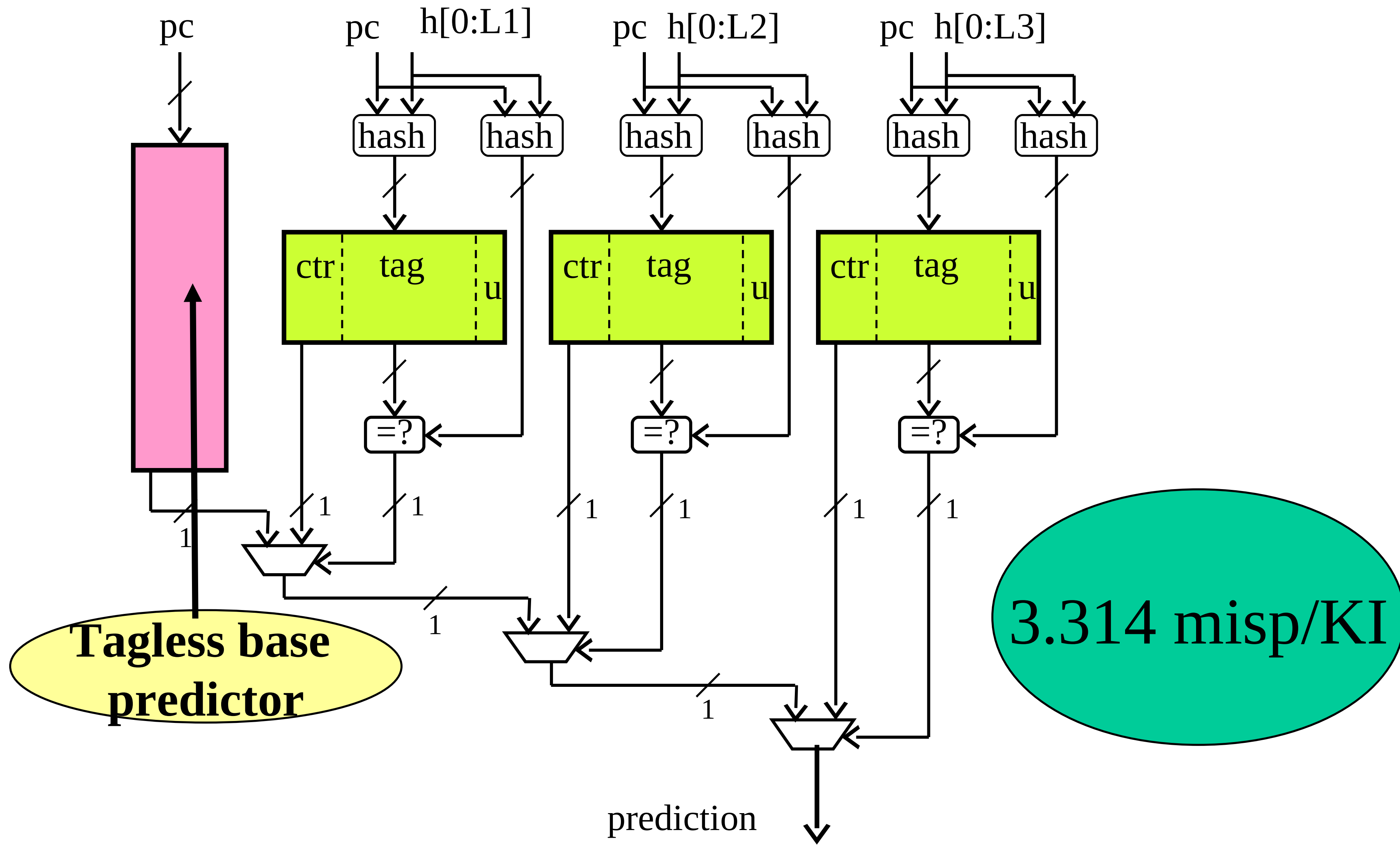


352 Kbits , cancelled 2001

Max hist length > 21, ≈35

Current state-of-the-art

256 Kbits TAGE: Geometric history length (dec 2006)



ILP: Instruction level parallelism

Executing instructions in parallel : supercalar and VLIW processors

- Till 1991, pipelining to achieve achieving 1 inst per cycle was the goal :

- Pipelining reached limits:
  - ➔ Multiplying stages do not lead to higher performance
  - ➔ Silicon area was available:
    - Parallelism is the natural way

- ILP: executing several instructions per cycle
  - ➔ Different approachs depending on who is in charge of the control :
    - The compiler/software: VLIW (Very Long Instruction Word)
    - The hardware: superscalar

Instruction Level Parallelism: what is ILP ?

- A= B+C; D=E+F; → 8 instructions

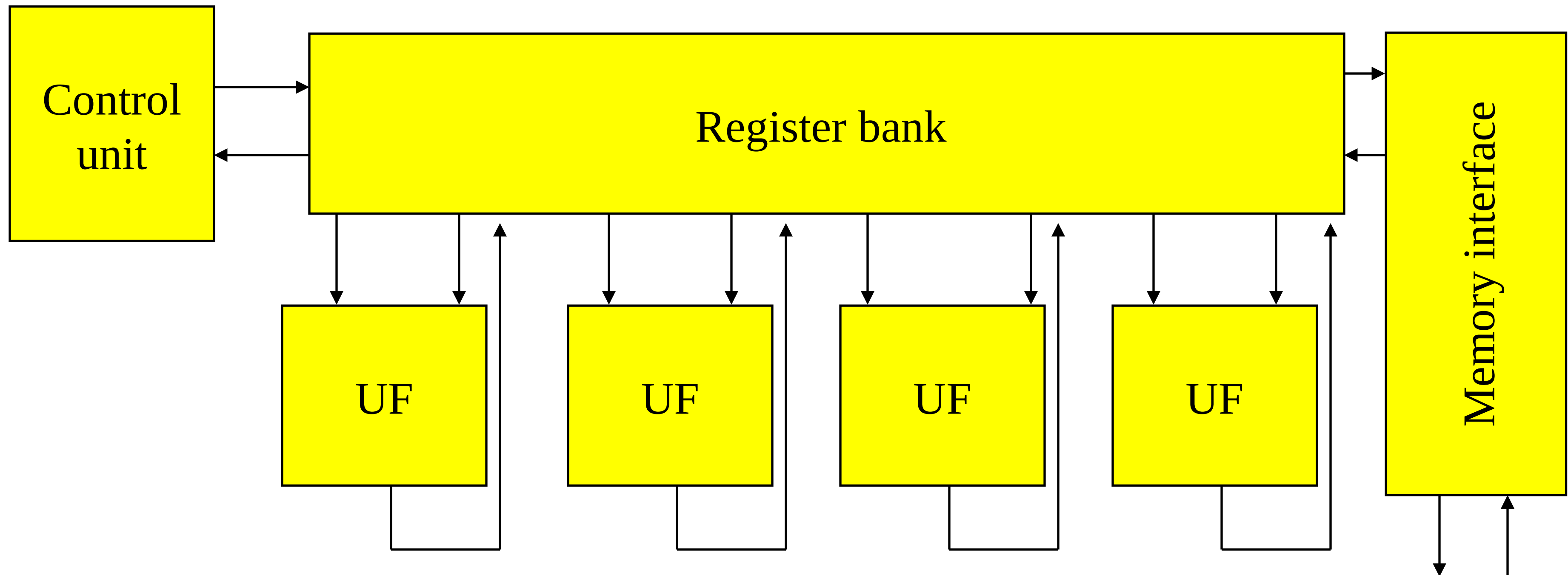
1. Ld @ C , R1 (A)
2. Ld @ B, R2 (A)
3. R3 ← R1+ R2 (B)
4. St @ A, R3 (C)
5. Ld @ E , R4 (A)
6. Ld @ F, R5 (A)
7. R6 ← R4+ R5 (B)
8. St @ A, R6 (C)

- (A),(B), (C) three groups of independent instructions
- Each group can be executed in //

VLIW: Very Long Instruction Word

- Each instruction controls explicitly the whole processor
  - ➔ The compiler/code scheduler is in charge of all functional units:
    - Manages all hazards:
      - – resource: decides if there are two competing candidates for a resource
      - – data: ensures that data dependencies will be respected
      - – control: ?!?

VLIW architecture



VLIW (Very Long Instruction Word)

- The Control unit issues a single long instruction word per cycle

- Each long instruction lanches simultinaeously sevral independant instructions:
  - ➔ The compiler garantees that:
  - ➔ the subinstructions are independent.
  - ➔ The instruction is independent of all in flight instructions

- **There is no hardware to enforce depencies**

VLIW architecture: often used for embedded applications

- Binary compatibility is a nightmare:
  - ➔ Necessitates the use of the same pipeline structure

- Very effective on regular codes with loops and very few control, but poor performance on general purpose application (too many branchs)
- Cost-effective hardware implementation:
  - ➔ No control:
    - Less silicon area
    - Reduced power consumption
    - Reduced design and test delays

Superscalar processors

- The hardware is in charge of the control:
  - ➔ The semantic is sequential, the hardware enforces this semantic
    - The hardware enforces dependencies:

- Binary compatibility with previous generation processors:

- All general purpose processors till 1993 are superscalar

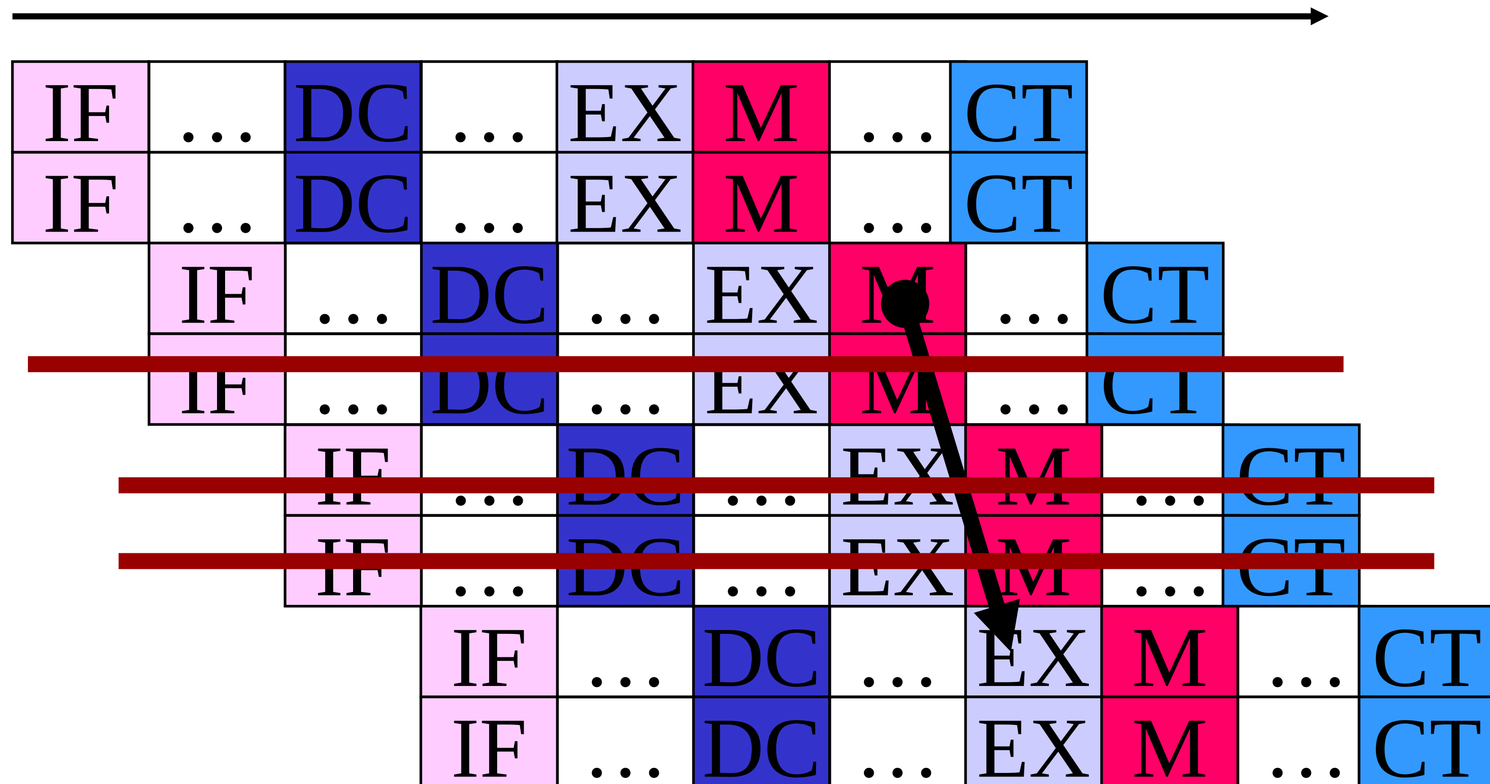
Superscalar: what are the problems ?

- Is there instruction parallelism ?
  - On general-purpose application: 2-8 instructions per cycle
  - On some applications: could be 1000's,

- How to recognize parallelism ?
  - Enforcing data dependencies

- Issuing in parallel:
  - Fetching instructions in //
  - Decoding in parallel
  - Reading operands iin parallel
  - Predicting branchs very far ahead

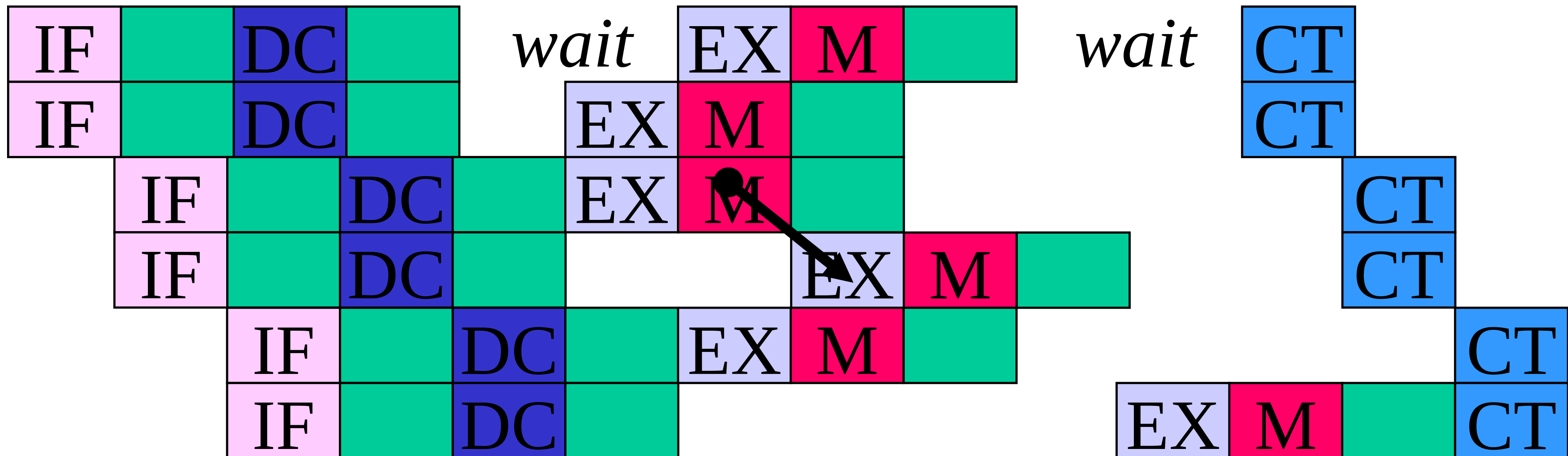
In-order execution



out-of-order execution

To optimize resource usage:

Executes as soon as operands are valid



Out of order execution

- Instructions are executed out of order:
  - ➔ If inst A is blocked due to the absence of its operands but inst B has its operands avalaible then B can be executed !!
  - ➔ Generates a lot of hardware complexity !!

speculative execution on OOO processors

- 10-15 % branches:
  - On Pentium 4: direction and target known at cycle 31 !!
- Predict and execute speculatively:
  - Validate at execution time
  - State-of-the-art predictors:
    - ≈2-3 misprediction per 1000 instructions
- Also predict:
  - Memory (in)dependency
  - (limited) data value

Out-of-order execution: Just be able to « undo »

- branch misprediction
- Memory dependency misprediction
- Interruption, exception

- Validate (commit) instructions in order
- Do not do anything definitely out-of-order

The memory hierarchy

Memory components

- Most transistors in a computer system are memory transistors:
  - Main memory:
    - Usually DRAM
    - 1 Gbyte is standard in PCs (2005)
    - Long access time
      - 150 ns = 500 cycles = 2000 instructions
  - On chip single ported memory:
    - Caches, predictors, ..
  - On chip multiported memory:
    - Register files, L1 cache, ..

Memory hierarchy

- Memory is :
  - either huge, but slow
  - or small, but fast

  The smallest, the fastest

- Memory hierarchy goal:
  - Provide the illusion that the whole memory is fast

- Principle: exploit the temporal and spatial locality properties of most applications

Locality property

- On most applications, the following property applies:

  ➔ **Temporal locality** : A data/instruction word that has just been accessed is likely to be reaccessed in the near future

  ➔ **Spatial locality**: The data/instruction words that are located close (in the address space) to a data/instruction word that has just been accessed is likely to be reaccessed in the near future.

A few examples of locality

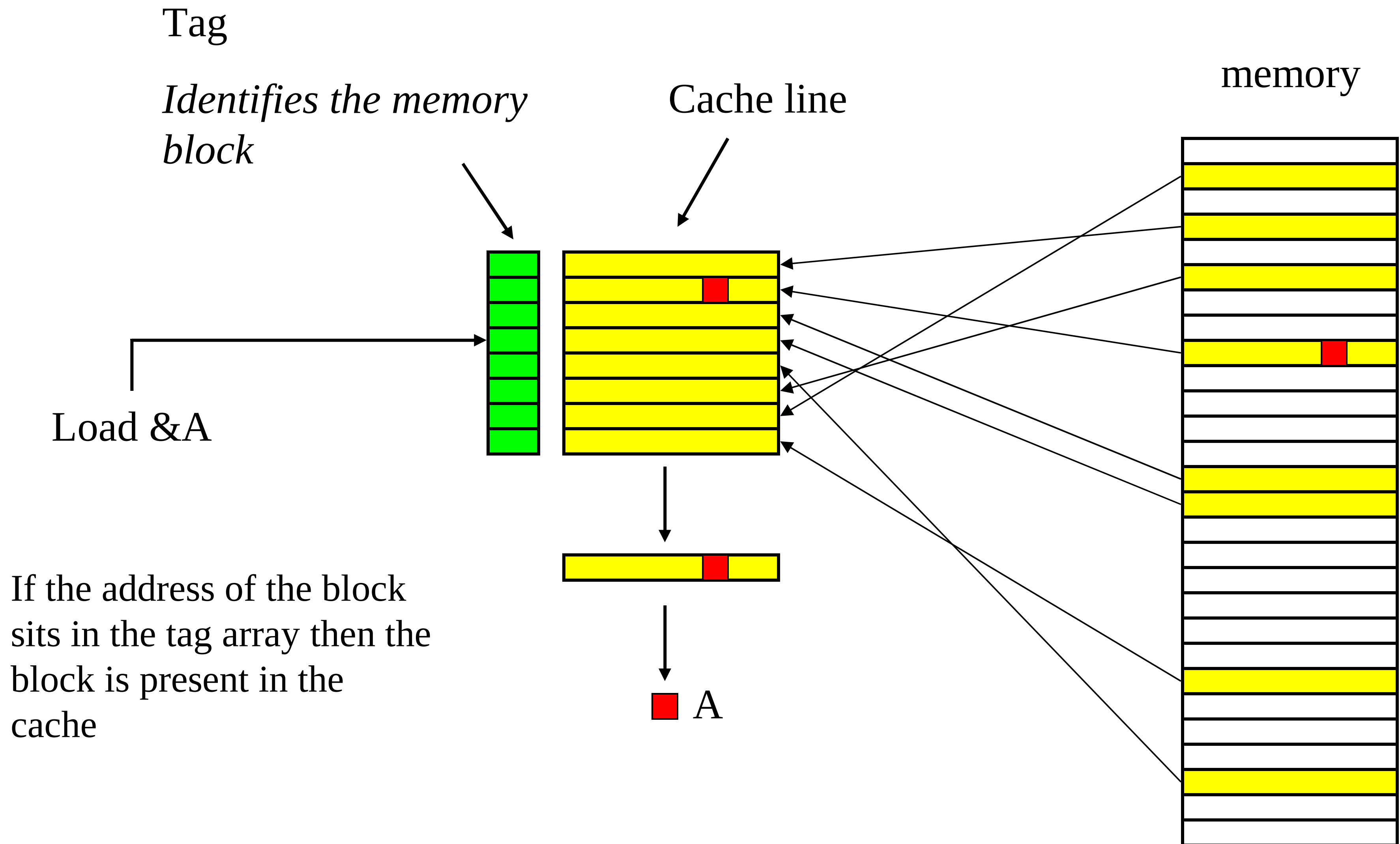
- Temporal locality:
  - Loop index, loop invariants, ..
  - Instructions: loops, ..
    - 90%/10% rule of thumb: a program spends 90 % of its excution time on 10 % of the static code (often much more on much less ☺)
- Spatial locality:
  - Arrays of data, data structure
  - Instructions: the next instruction after a non-branch inst is always executed

Cache memory

- A cache is small memory which content is an image of a subset of the main memory.

- A reference to memory is
  - 1.) presented to the cache
  - 2.) on a miss, the request is presented to next level in the memory hierarchy (2nd level cache or main memory)

Cache



If the address of the block sits in the tag array then the block is present in the cache

Memory hierarchy behavior may dictate performance

- Example :
  - 4 instructions/cycle,
  - 1 data memory acces per cycle
  - 10 cycle penalty for accessing 2nd level cache
  - 300 cycles round-trip to memory
  - 2% miss on instructions, 4% miss on data, 1 reference out 4 missing on L2

- To execute 400 instructions : 1320 cycles !!

Block size

- Long blocks:
  - ➔ Exploits the spatial locality
  - ➔ Loads useless words when spatial locality is poor.
- Short blocks:
  - ➔ Misses on conntiguous blocks

- Experimentally:
  - ➔ 16 - 64 bytes for small L1 caches 8-32 Kbytes
  - ➔ 64-128 bytes for large caches 256K-4Mbytes

Cache hierarchy

- Cache hierarchy becomes a standard:
  - L1: small (<= 64Kbytes), short access time (1-3 cycles)
    - Inst and data caches
  - L2: longer access time (7-15 cycles), 512K-2Mbytes
    - Unified
  - Coming L3: 2M-8Mbytes (20-30 cycles)
    - Unified, shared on multiprocessor

Cache misses do not stop a processor (completely)

- On a L1 cache miss:
  - ➔ The request is sent to the L2 cache, but sequencing and execution continues
    - On a L2 hit, latency is simply a few cycles
    - On a L2 miss, latency is hundred of cycles:
      - – Execution stops after a while

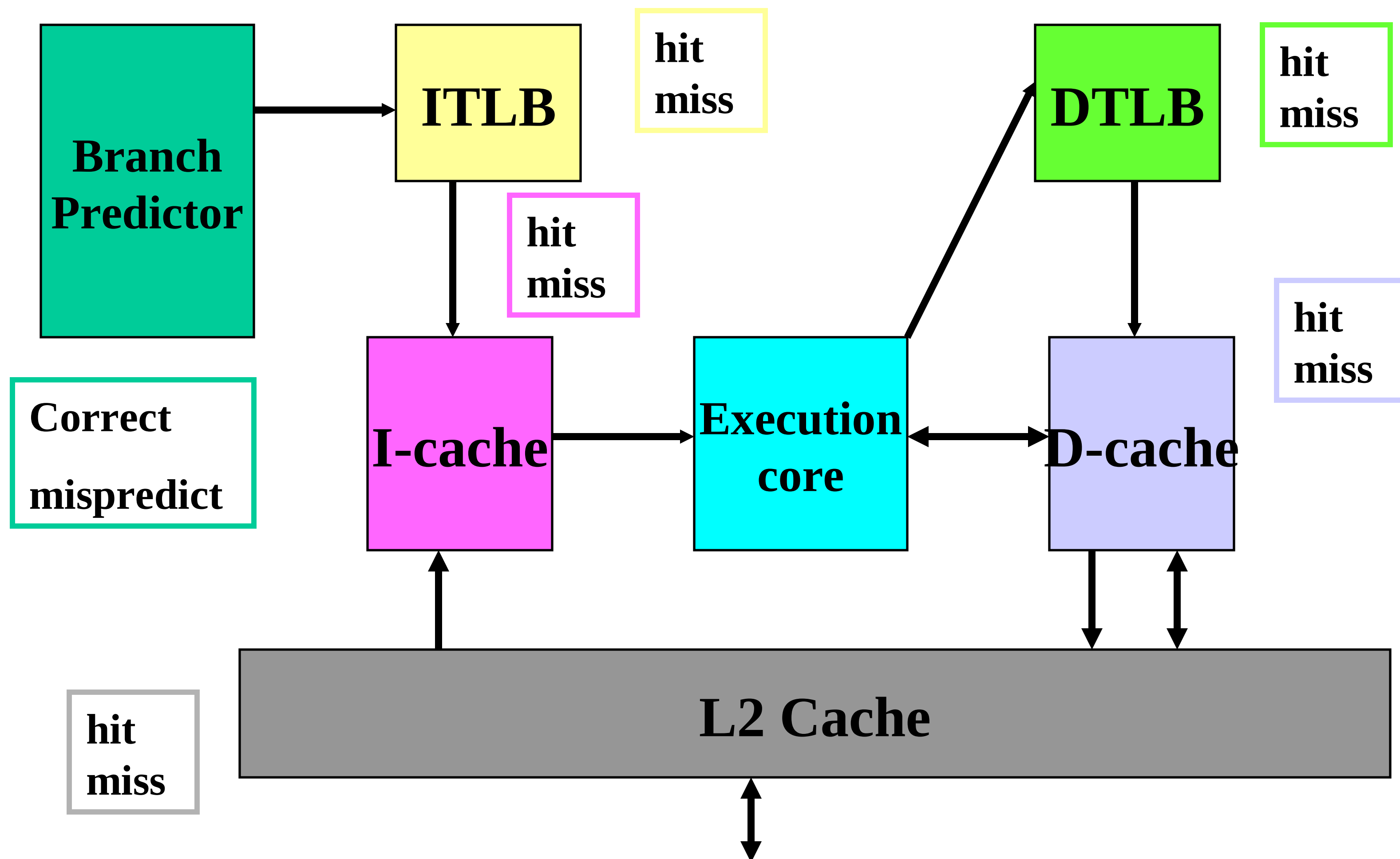
- Out-of-order execution allows to initiate several L2 cache misses (serviced in a pipeline mode) at the same time:
  - ➔ Latency is partially hiden

Prefetching

- To avoid misses, one can try to anticipate misses and load the (future) missing blocks in the cache in advance:
  - ➔ Many techniques:
    - Sequential prefetching: prefetch the sequential blocks
    - Stride prefetching: recognize a stride pattern and prefetch the blocks in that pattern

    - Hardware and software methods are available:
      - Many complex issues: latency, pollution, ..

Execution time of a short instruction sequence is a complex function !



Code generation issues

- First, avoid data misses: 300 cycles mispenalties
    - Data layout
    - Loop reorganization
    - Loop blocking
- Instruction generation:
    - ➔ minimize instruction count: e.g. common subexpression elimination
    - ➔ schedule instructions to expose ILP
    - ➔ Avoid hard-to-predict branches

On chip
thread level pararallelism

One billion transistors now !!

- Ultimate 16-32 way superscalar uniprocessor seems unreachable:
  - ➔ Just not enough ILP
  - ➔ More than quadratic complexity on a few key (power hungry) components (register file, bypass network, issue logic)
  - ➔ To avoid temperature hot spots:
    - Intra-CPU very long communications would be needed

- On-chip thread parallelism appears as the only viable solution
  - ➔ Shared memory processor i.e. chip multiprocessor
  - ➔ Simultaneous multithreading
  - ➔ Heterogeneous multiprocessing
  - ➔ Vector processing

The Chip Multiprocessor

- Put a shared memory multiprocessor on a single die:
  - → Duplicate the processor, its L1 cache, may be L2,
  - → Keep the caches coherent
  - → Share the last level of the memory hierarchy (may be)
  - → Share the external interface (to memory and system)

General purpose Chip MultiProcessor (CMP): why it did not (really) appear before 2003

- Till 2003 better (economic) usage for transistors:
  - ➔ Single process performance is the most important
  - ➔ More complex superscalar implementation
  - ➔ More cache space:
    - Bring the L2 cache on-chip
    - Enlarge the L2 cache
    - Include a L3 cache (now)

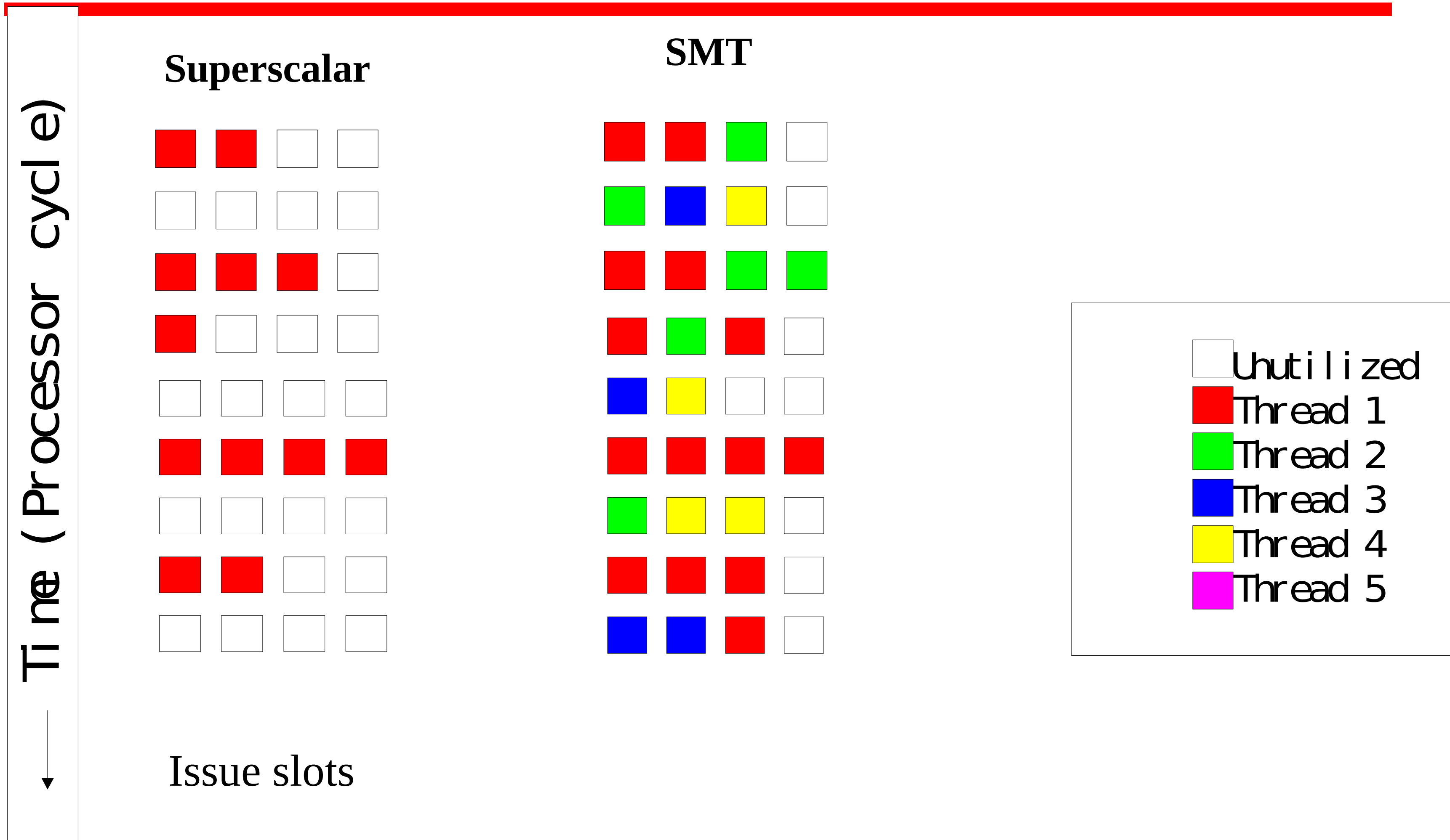
Diminishing return !!

Now: CMP is the only option !!

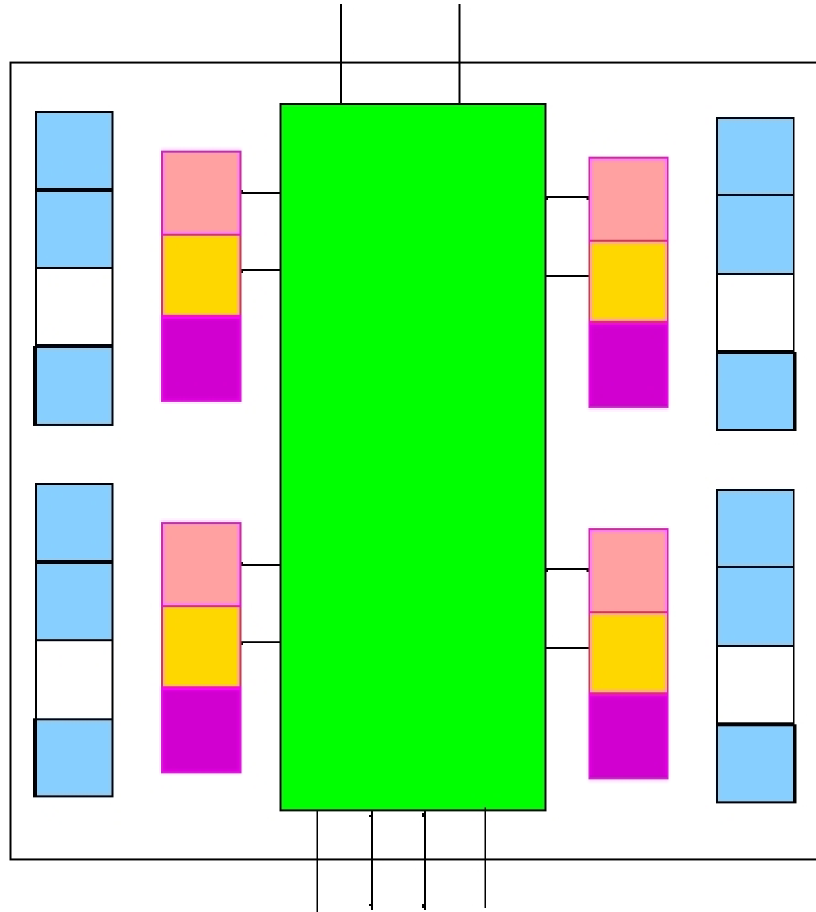
Simultaneous Multithreading (SMT): parallel processing on a single processor

- functional units are underused on superscalar processors

- SMT:
  - Sharing the functional units on a superscalar processor between several process
- Advantages:
  - Single process can use all the resources units
  - dynamic sharing of all structures on parallel/multiprocess workloads


Time (Processor cycle)
Superscalar
SMT
Unutilized
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Issue slots

The programmer view of a CMP/SMT !



Why CMP/SMT is the new frontier ?

- (Most) applications were sequential:
  - ➔ Hardware WILL be parallel
  - ➔ Tens, hundreds of SMT cores in your PC, PDA in 10 years from now (might be ☺

- Option 1: Applications will have to be adapted to parallelism
- Option 2: // hardware will have to run efficiently sequential applications
- Option 3: invent new tradeoffs ☺

There is no current standard

Embedded processing and on-chip parallelism (1)

- ILP has been exploited for many years:
  - DSPs: a multiply-add, 1 or 2 loads, loop control in a single (long) cycle
  - Caches were implemented on embedded processors +10 years
  - VLIW on embedded processor was introduced in1997-98
  - In-order supercsalar processor were developped for embedded market 15 years ago (Intel i960)

Embedded processing and on-chip parallelism (2): thread parallelism

- Heterogeneous multicores is the trend:
  - Cell processor IBM (2005)
    - One PowerPC (master)
    - 8 special purpose processors (slaves)
  - Philips Nexperia: A RISC MIPS microprocessor + a VLIW Trimedia processor
  - ST Nomadik: An ARM + x VLIW

What about a vector microprocessor for scientific computing?

Vector parallelism is well understood !

A niche segment

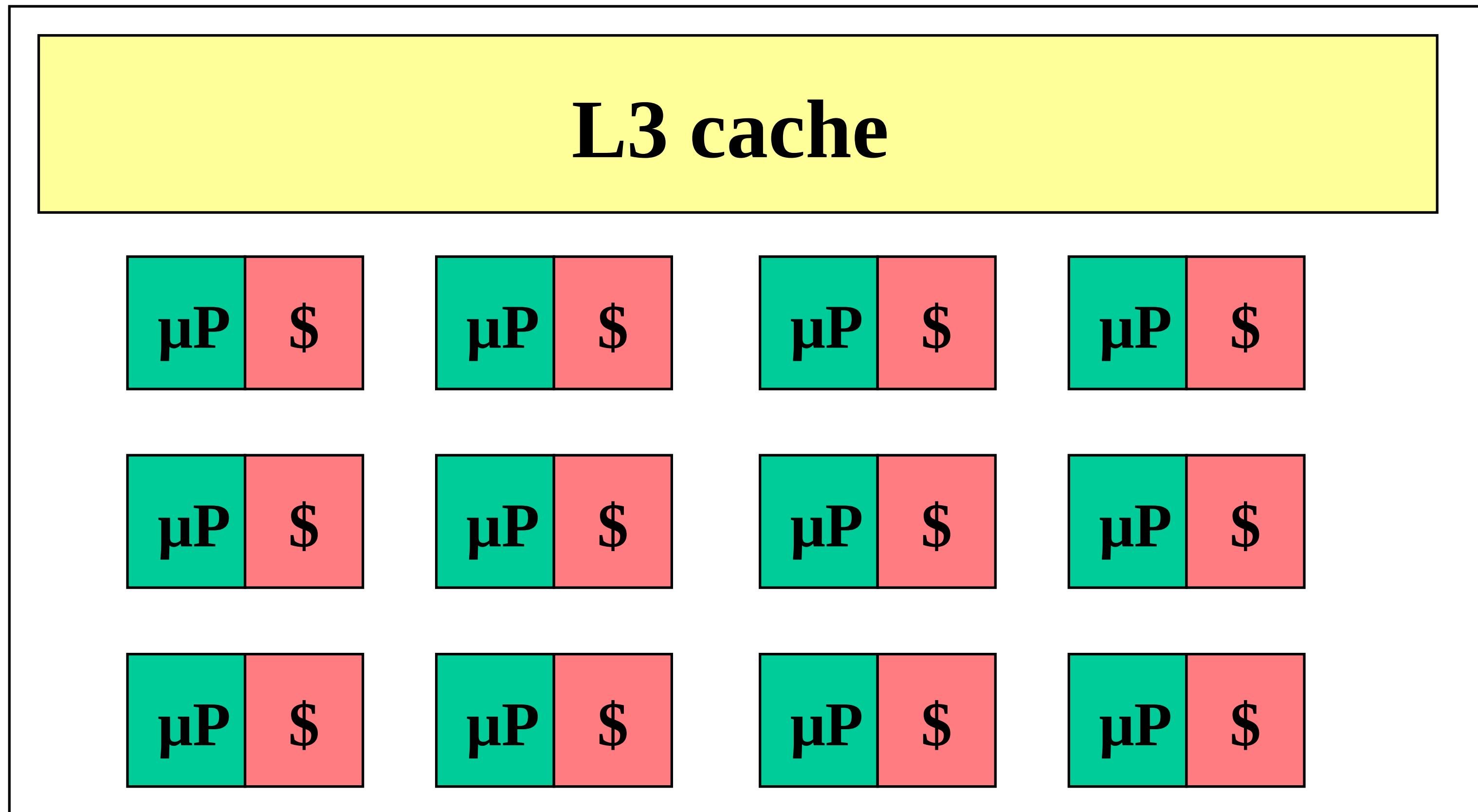
Not so small: 2B$!

Caches are not vectors friendly

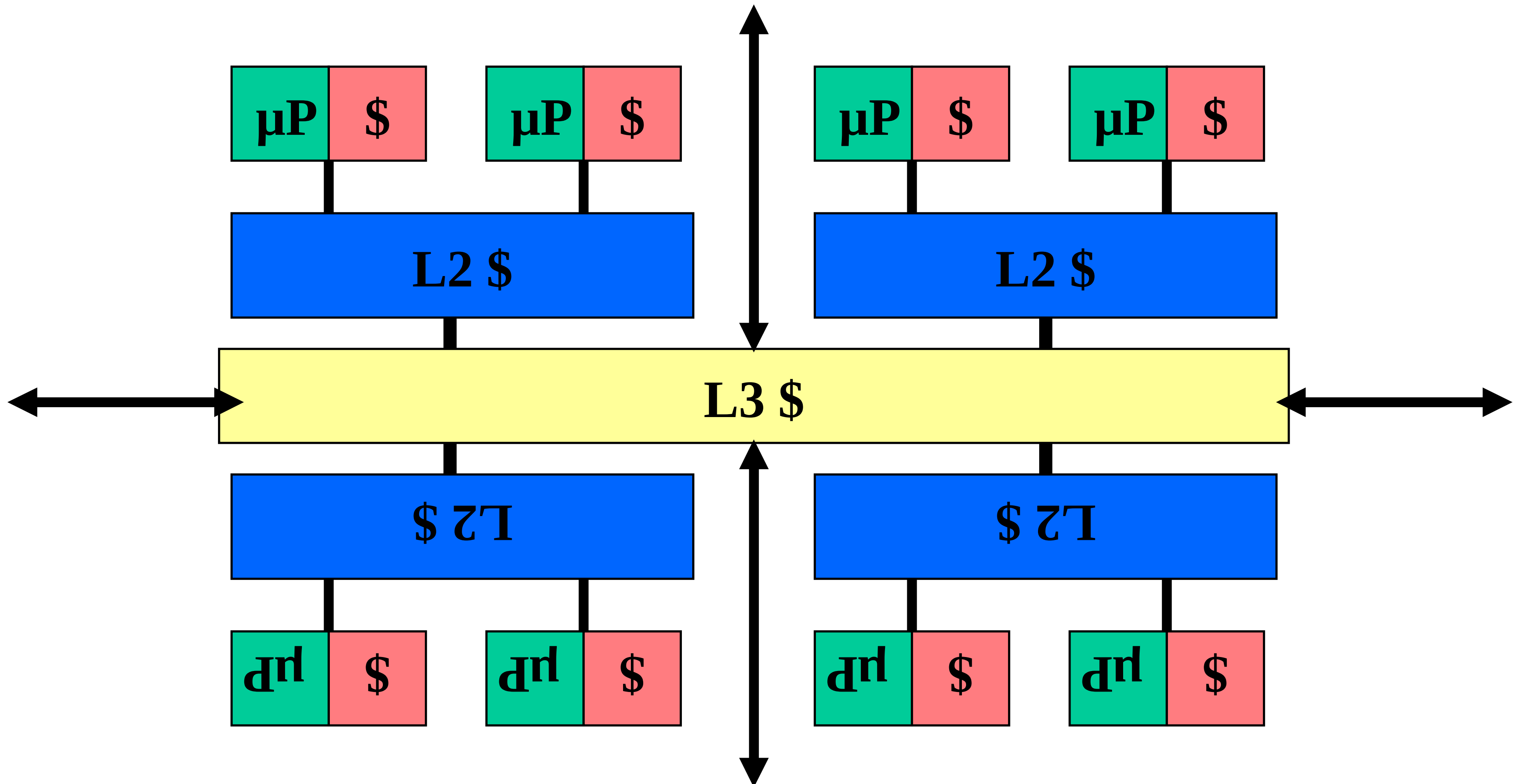
never heard about:L2 cache, prefetching, blocking ?

GPGPU !!
GPU boards have become Vector processing units

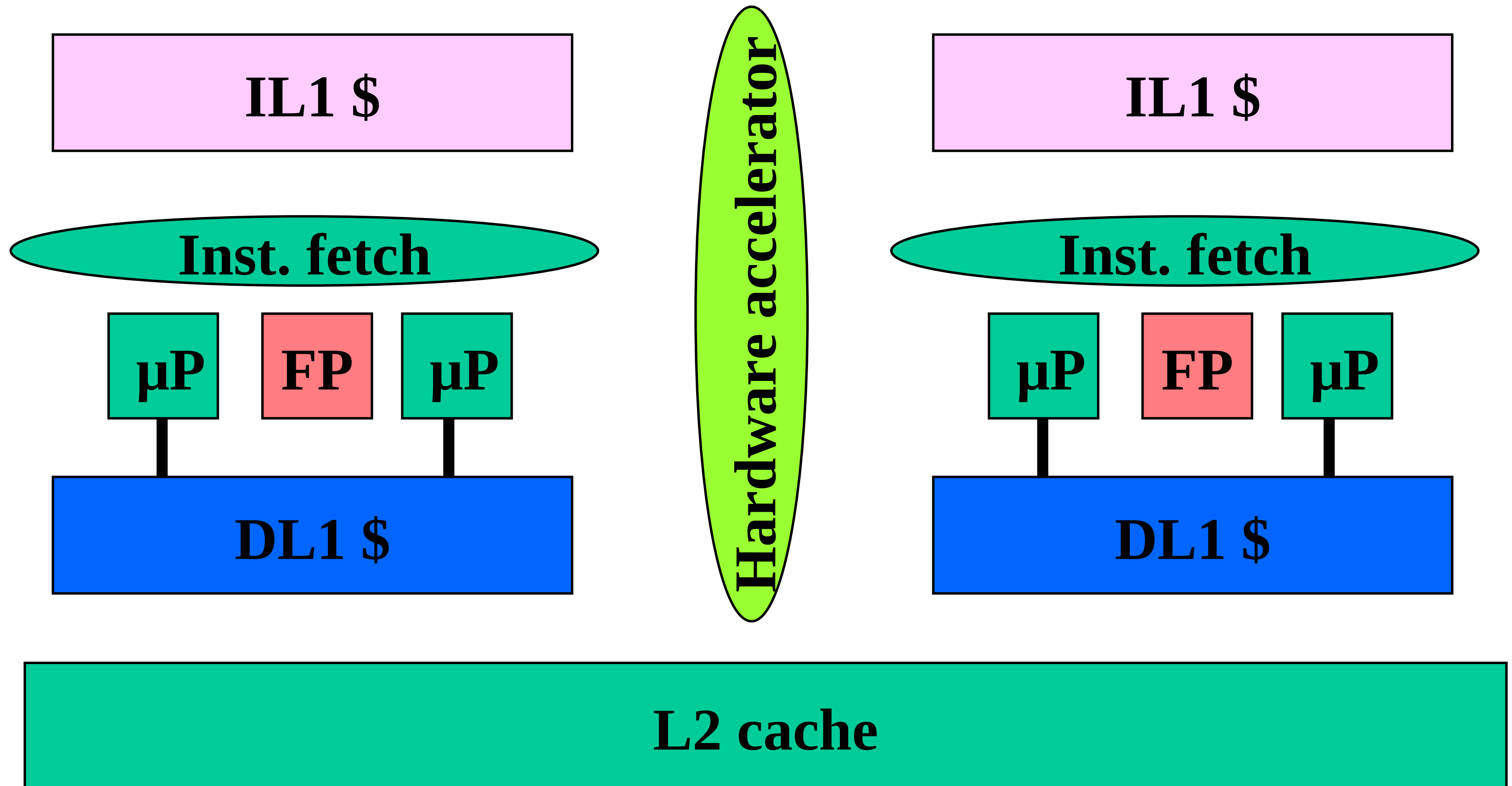
Structure of future multicores ?



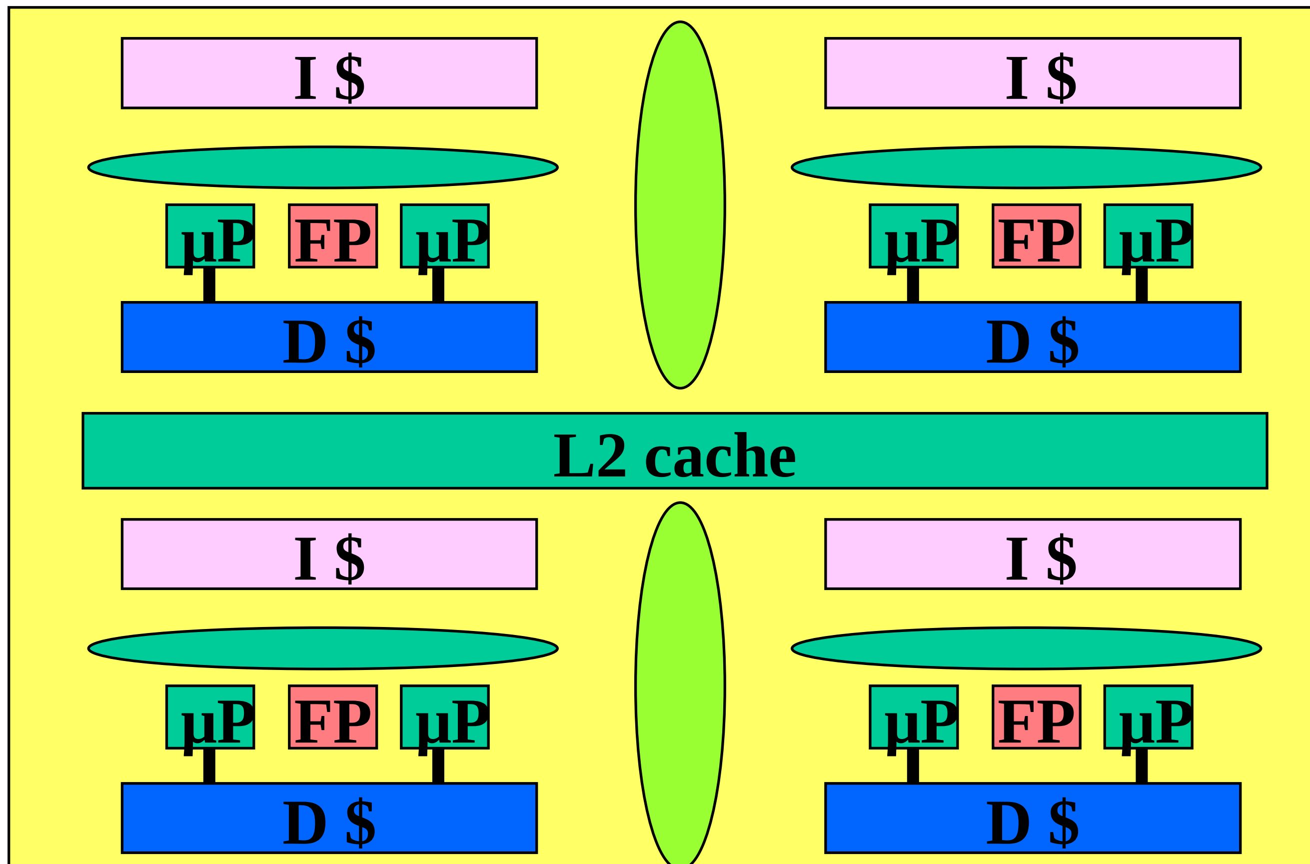
Hierarchical organization ?

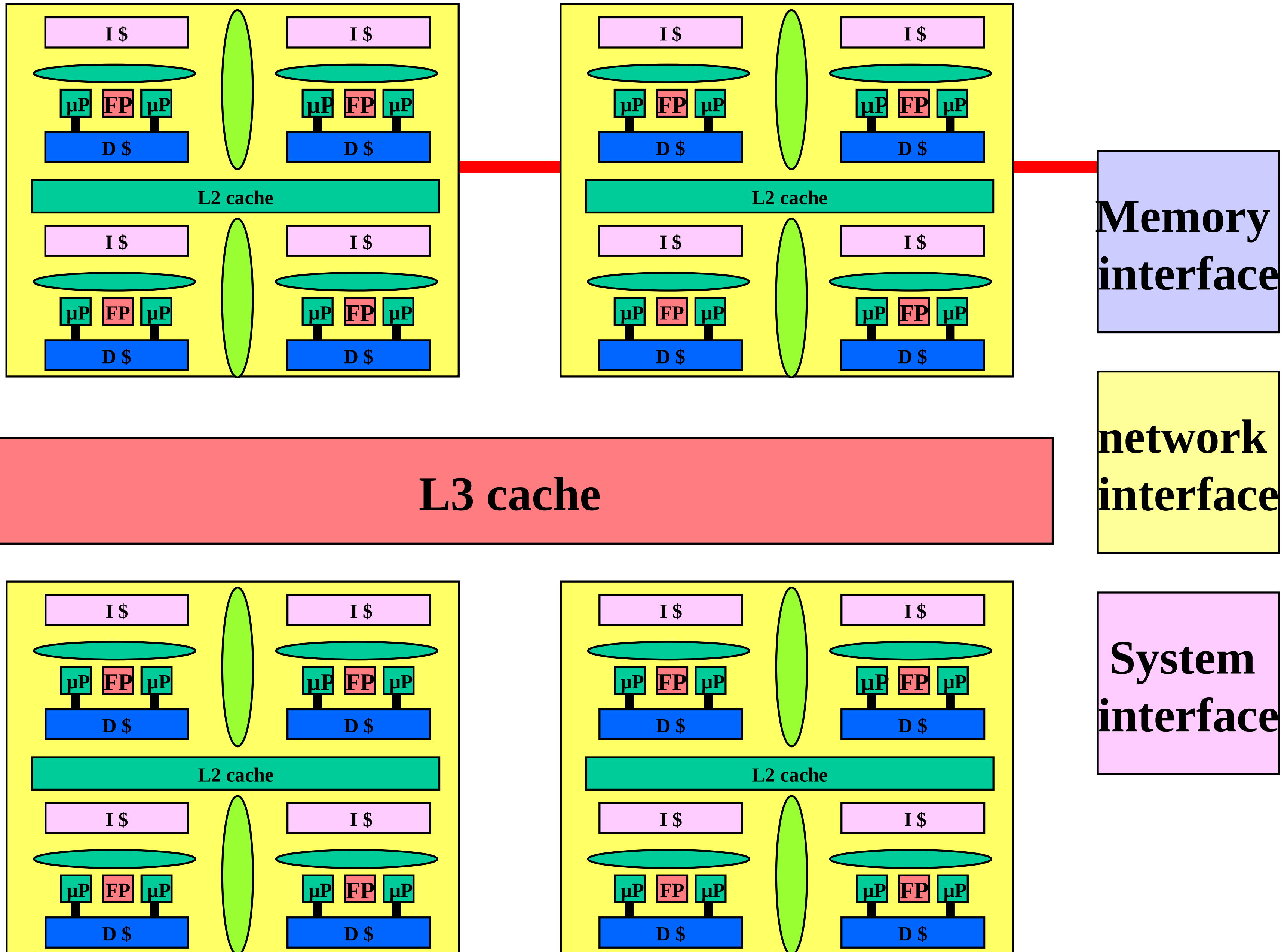


An example of sharing



A possible basic brick



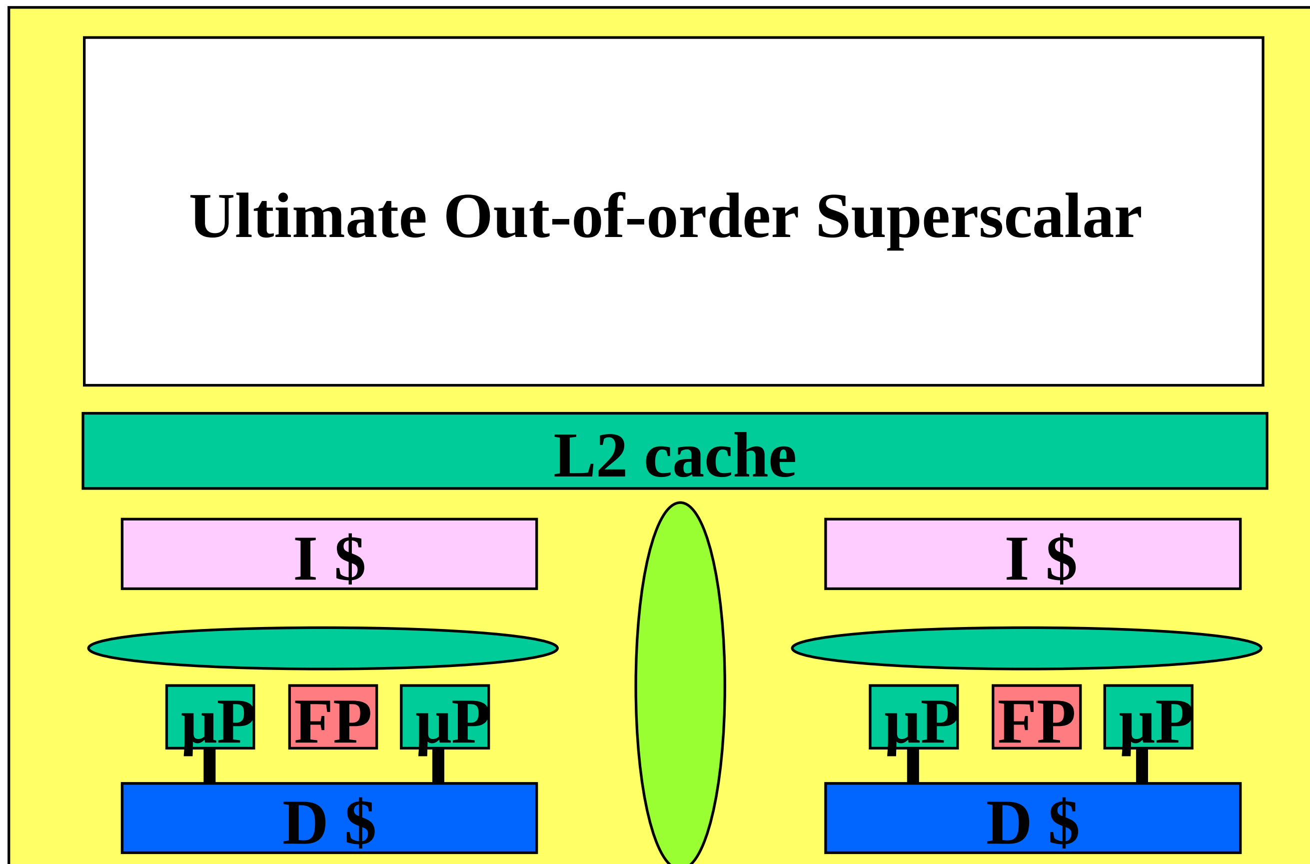

I $
µP
FP
µP
D $
L2 cache
Memory interface
L3 cache
network interface
System interface

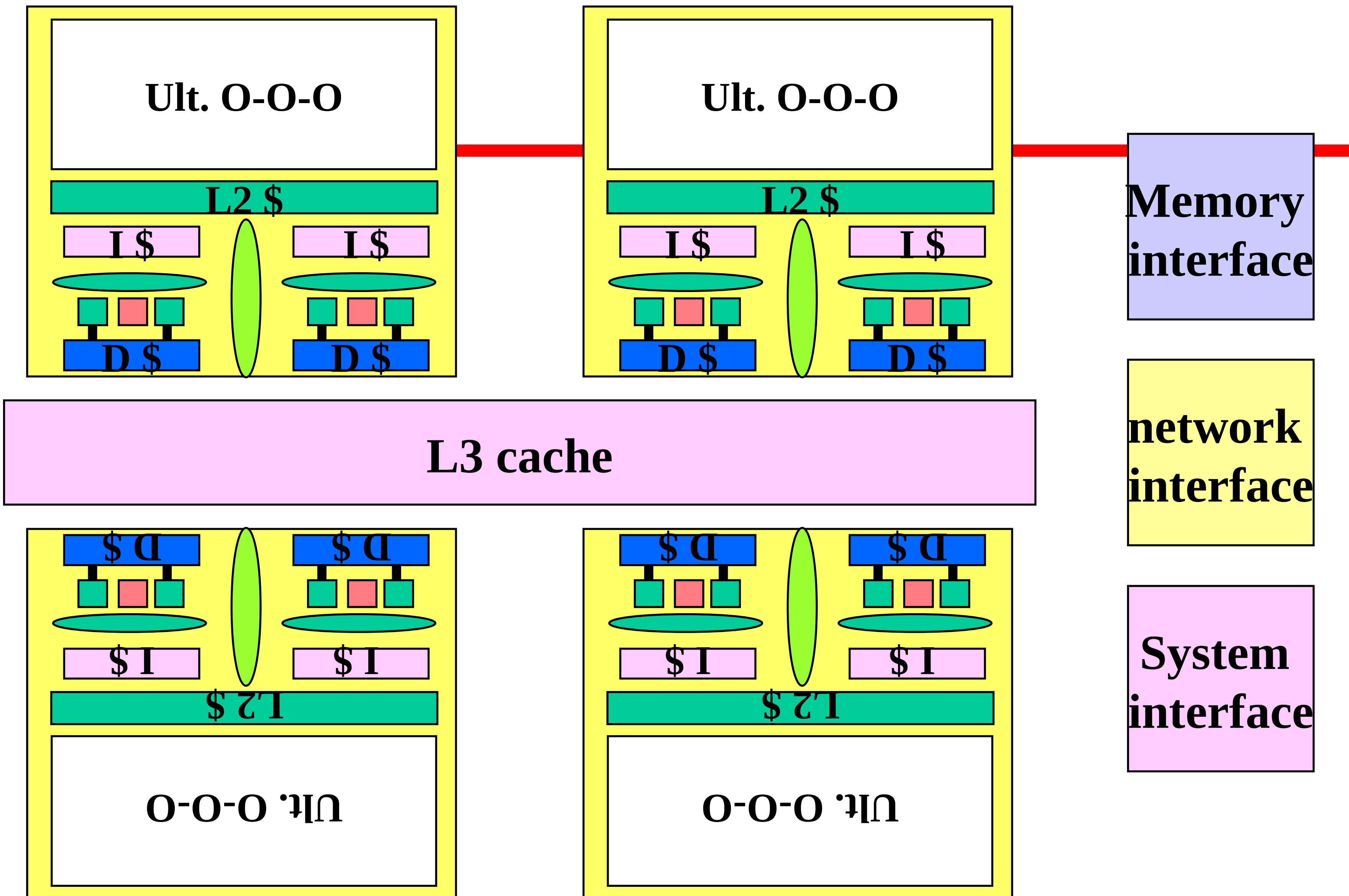
Only limited available thread parallelism ?

- Focus on uniprocessor architecture:
  - ➔ Find the correct tradeoff between complexity and performance
  - ➔ Power and temperature issues

- Vector extensions ?
  - ➔ Contiguous vectors (a la SSE) ?
  - ➔ Strided vectors in L2 caches (Tarantula-like)

Another possible basic brick





Ult. O-O-O
Ult. O-O-O
L2 $
L2 $
I $
I $
I $
I $
D $
D $
D $
D $
Memory interface
L3 cache
network interface
D $
D $
D $
D $
I $
I $
I $
I $
L2 $
L2 $
Ult. O-O-O
Ult. O-O-O
System interface

Some undeveloped issues ☺: the power consumption issue

- Power consumption:
  - ➔ Need to limit power consumption
    - Labtop: battery life
    - Desktop: above 75 W, hard to extract at low cost
    - Embedded: battery life, environment
  - ➔ Revisit the old performance concept with:
    - “maximum performance in a fixed power budget”
  - ➔ Power aware architecture design
  - ➔ Need to limit frequency

Some undeveloped issues ☺: the performance predictabilty issue

- Modern microprocessors have unpredictable/unstable performance

- The average user wants stable performance:
  - → Cannot tolerate variations of performance by orders of magnitude when varying simple parameters

- Real time systems:
  - → Want to guarantee response time.

Some undeveloped issues ☺: the temperature issue

- Temperature is not uniform on the chip (hotspots)

  - ➔ Rising the temperature above a threshold has devastating effects:
    - Defectuous behavior: transcient or definitive
    - Component aging

  - ➔ Solutions: gating (stops !!) or clock scaling, or task migration