Master Informatics Eng.

2021/22

A.J.Proença

Optimizing sequential code

(revision: most slides from an undergrad course)

Improving code performance to explore ILP: an example from the Computer Systems course

The following slides are a selection from CS.

The originals (in Portuguese) are in:

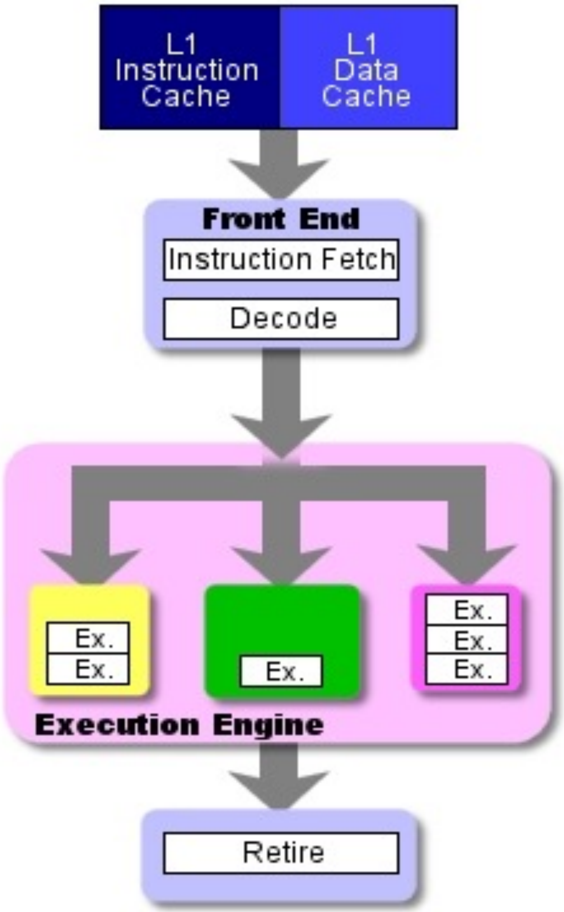
- http://gec.di.uminho.pt/mei/cp/slides_sc.zip

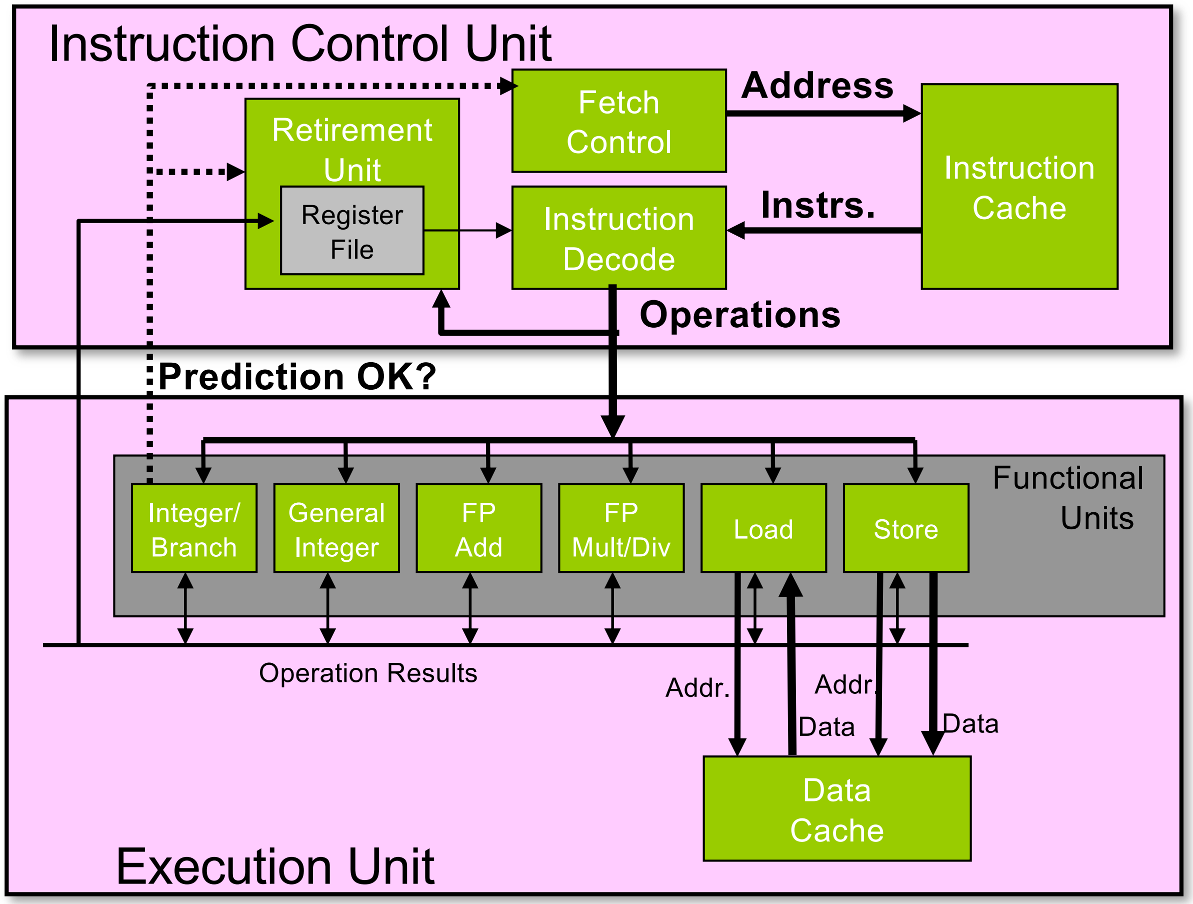
Last year lectures were recorded and the videos were placed on the e-platform; they are available here:

- http://gec.di.uminho.pt/mei/cp/videos_sc.zip

Internal architecture of Intel P6 processors

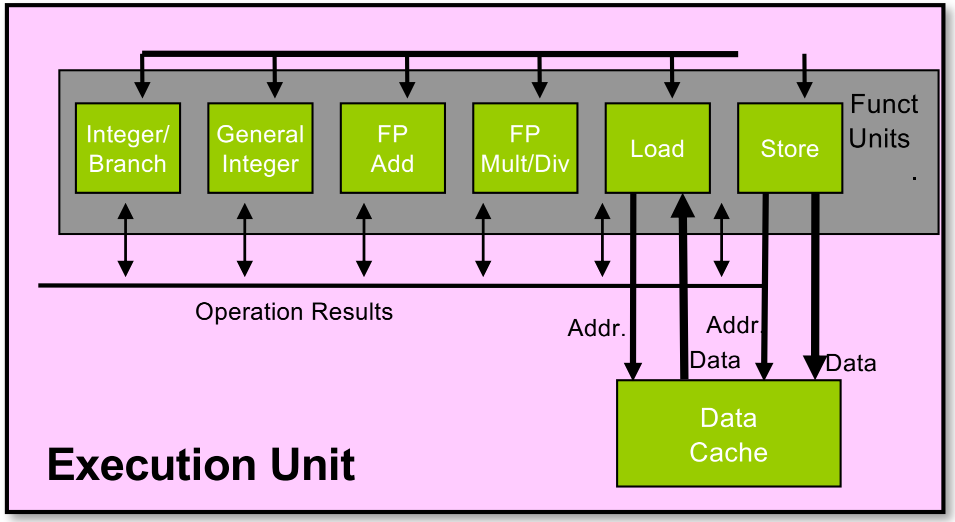
Note: "Intel P6" is the common μarch name for PentiumPro, Pentium II & Pentium III, which inspired Core, Nehalem and later generations





Some capabilities of Intel P6

- **Parallel execution of several instructions**
  - 2 `integer` (1 can be `branch`)
  - 1 `FP Add`
  - 1 `FP Multiply or Divide`
  - 1 `load`
  - 1 `store`



- Some instructions require > 1 cycle, but can be pipelined:

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

A detailed example: generic & abstract form of `combine`

```
void abstract_combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP data[i];
  *dest = t;
}
```

- **Procedure to perform addition** (w/ some improvements)
  - compute the sum of all vector elements
  - store the result in a given memory location
  - structure and operations on the vector defined by ADT
- **Metrics**
  - Clock-cycles Per Element, **CPE**

Converting instructions with registers into operations with tags

- **Assembly version for `combine4`**
  - data type: *integer* ; operation: *multiplication*

```
.L24:                           # Loop:
  imull  (%eax,%edx,4),%ecx     # t *= data[i]
  incl   %edx                   # i++
  cmpl   %esi,%edx              # i:length
  jl     .L24                   # if < goto Loop
```

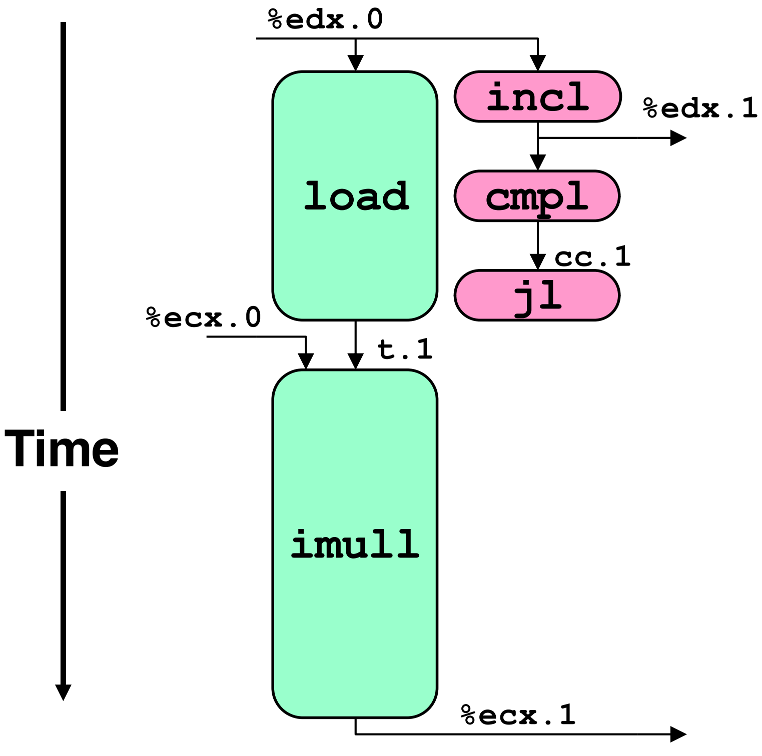
- **Translating 1st iteration**

```
.L24:
  imull  (%eax,%edx,4),%ecx

  incl   %edx
  cmpl   %esi,%edx
  jl     .L24
```

```
load   (%eax,%edx.0,4)  ➔ t.1
imull  t.1, %ecx.0      ➔ %ecx.1
incl   %edx.0           ➔ %edx.1
cmpl   %esi, %edx.1     ➔ cc.1
jl     -taken cc.1
```

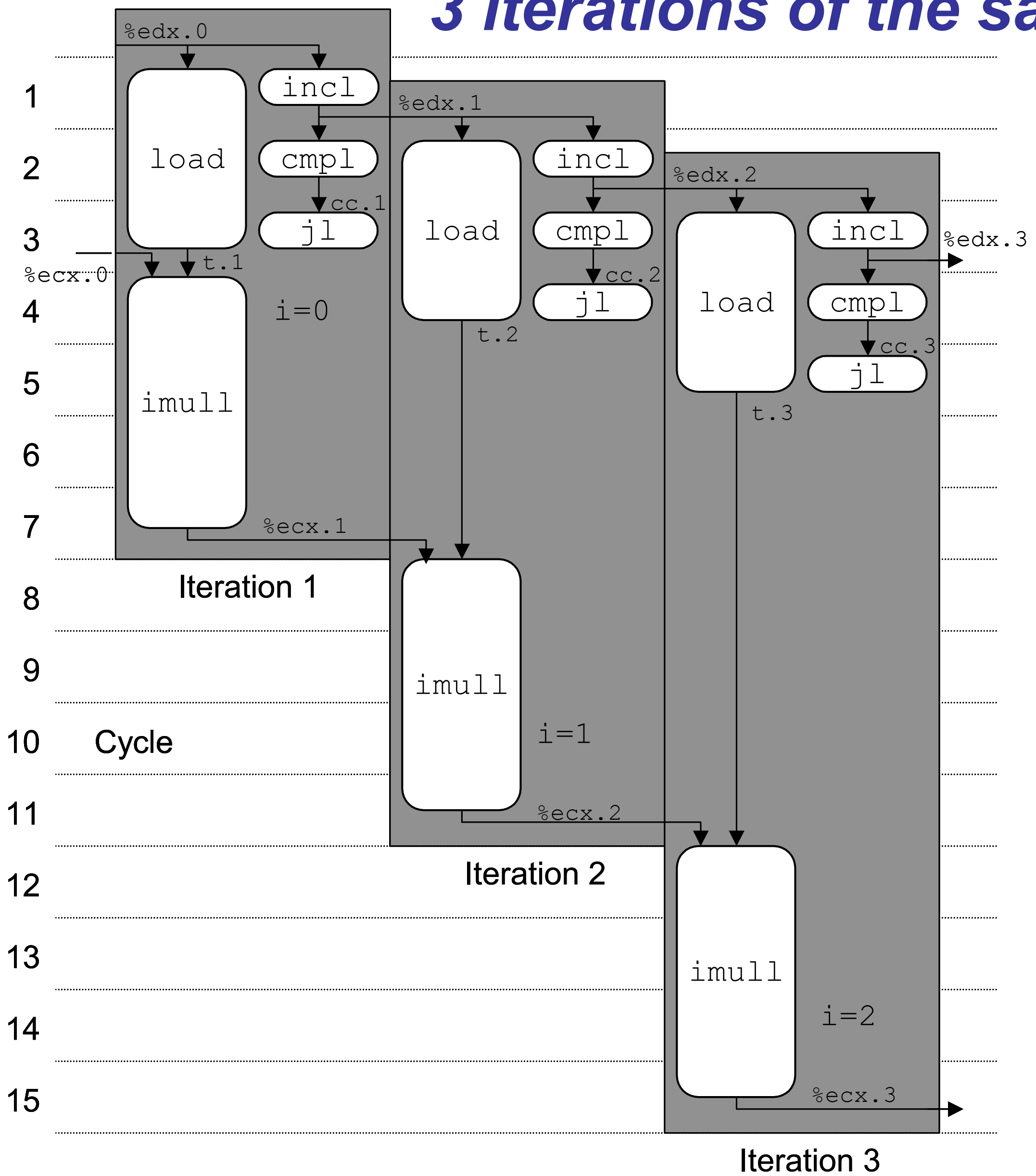
Visualizing instruction execution in P6: 1 iteration of the multiplication cycle on `combine`




```
load   (%eax,%edx.0,4)  ➔  t.1
imull  t.1, %ecx.0      ➔  %ecx.1
incl   %edx.0           ➔  %edx.1
cmpl   %esi, %edx.1     ➔  cc.1
jl     -taken cc.1
```

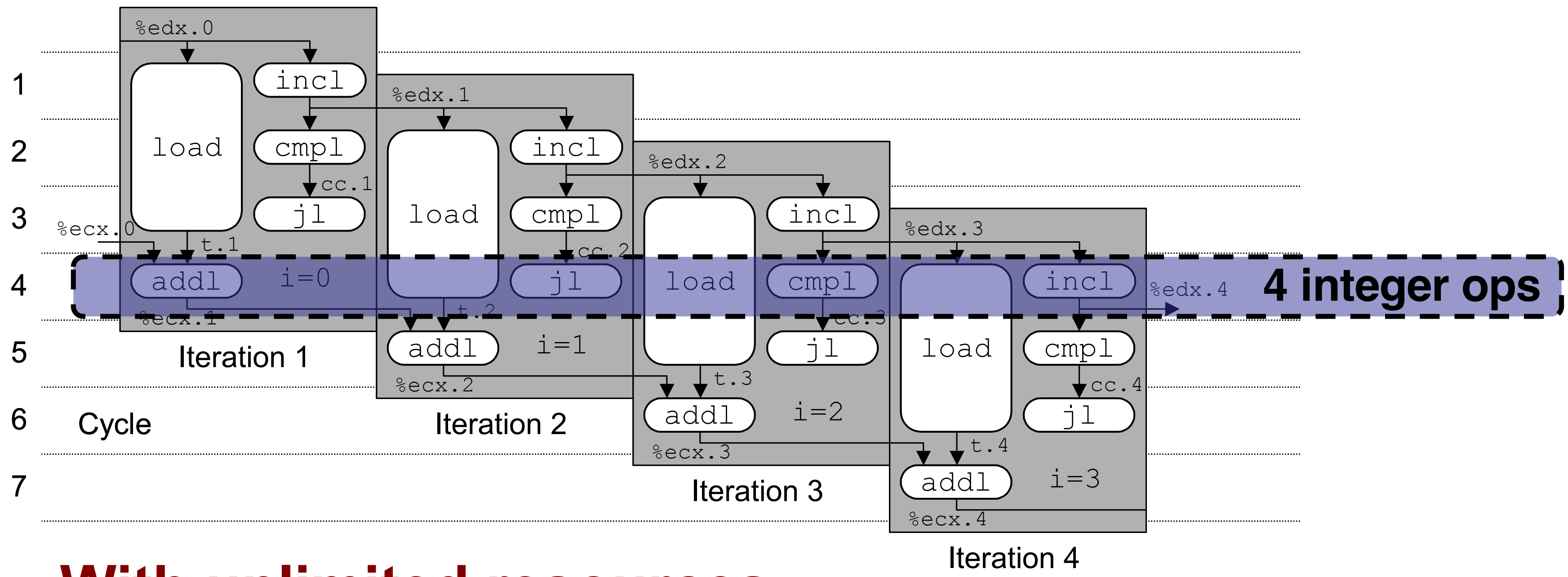
- **Operations**
  - vertical axis shows the time the instruction is executed
    - an operation cannot start with its operands
  - time length measures latency
- **Operands**
  - arcs are only showed for operands that are used in the context of the *execution unit*

Visualizing instruction execution in P6: 3 iterations of the same cycle on `combine`



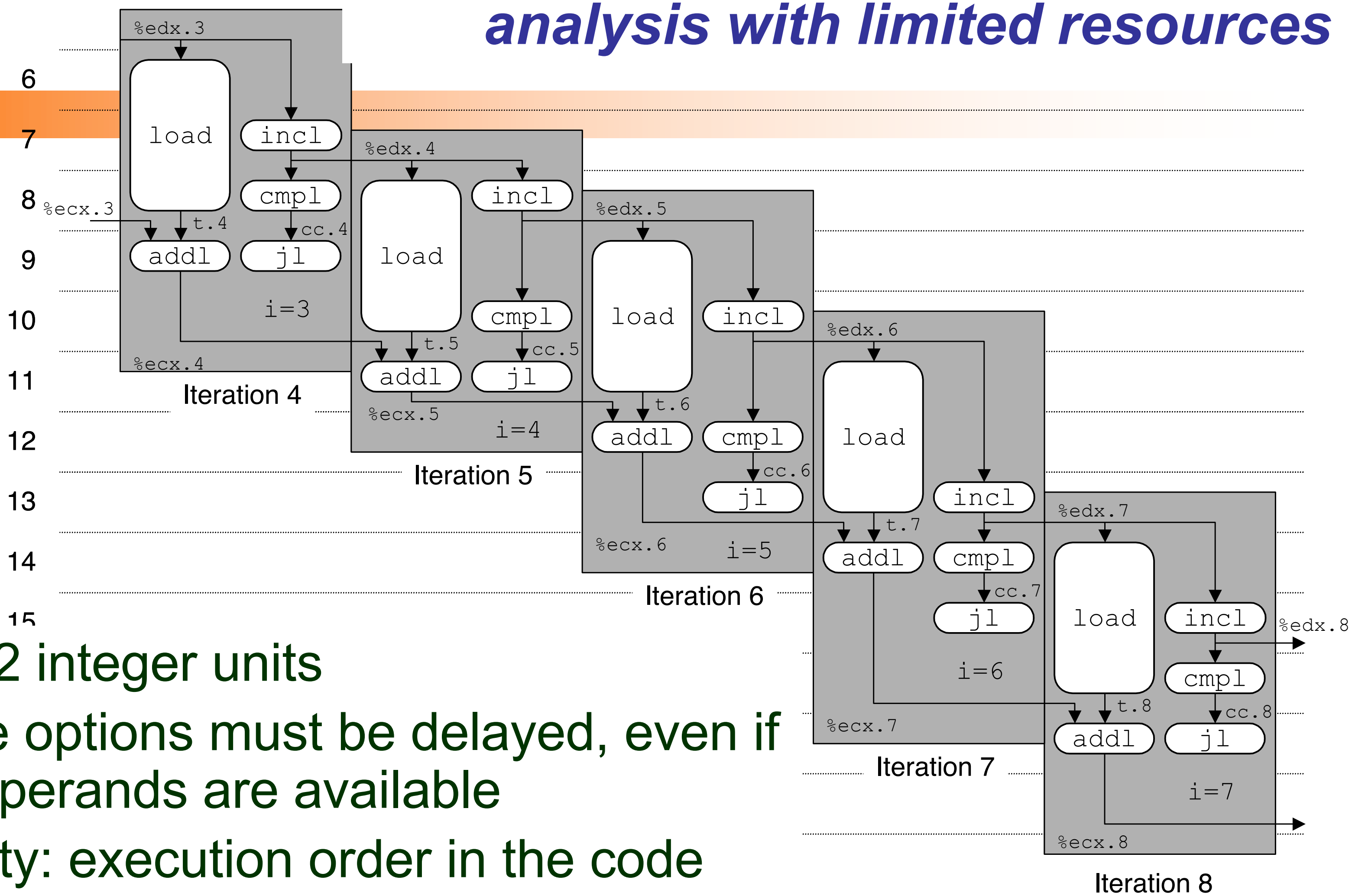
- **With unlimited resources**
  - –parallel and pipelined execution of operations at the EU
  - –out-of-order and speculative execution
- **Performance**
  - –limitative factor: latency of integer multiplication
  - –CPE: 4.0

Visualizing instruction execution in P6: 4 iterations of the addition cycle on `combine`



- **With unlimited resources**
- **Performance**
  - it can start a new iteration at each clock cycle
  - theoretical CPE: 1.0
  - it requires parallel execution of 4 integer operations

Iterations of the addition cycles: analysis with limited resources



– only 2 integer units
– some options must be delayed, even if the operands are available
– priority: execution order in the code

• **Performance**
– expected CPE: 2.0

Machine dependent optimization techniques: loop unroll (1)

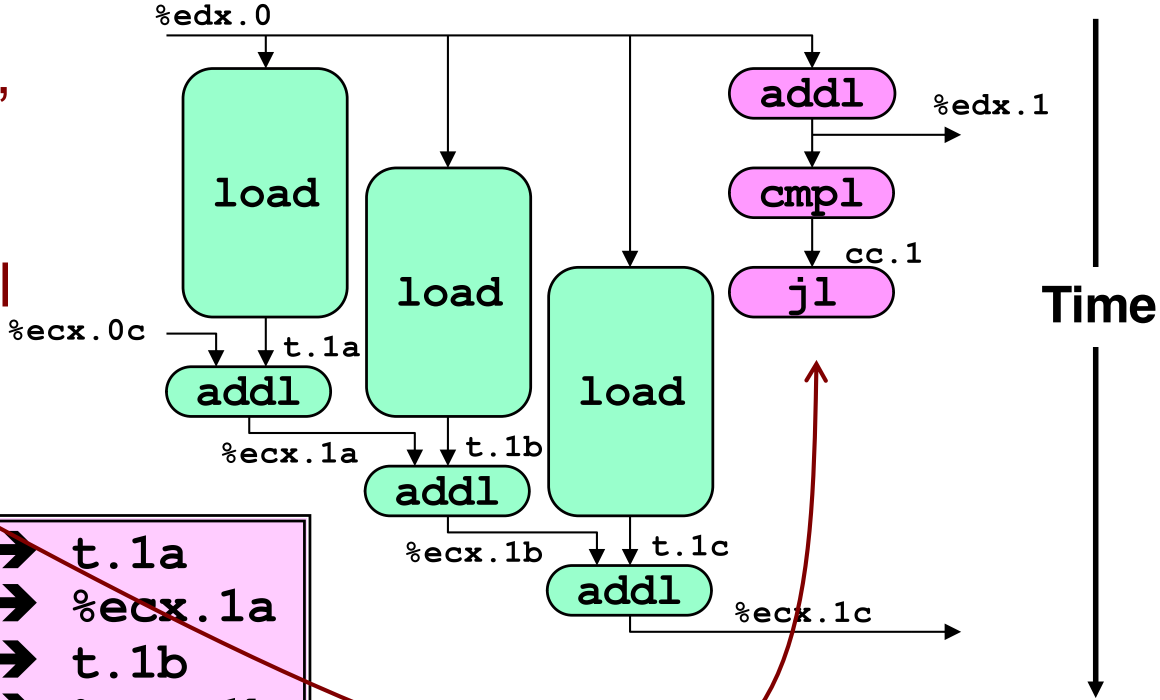
```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* junta 3 elem's no mesmo ciclo */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
         + data[i+2];
  }
  /* completa os restantes elem's */
  for (; i < length; i++) {
    sum += data[i];
  }
  *dest = sum;
}
```

Optimization 4:

- merges several (**3**) iterations in a single loop cycle
- reduces cycle overhead in loop iterations
- runs the extra work at the end
- **CPE: 1.33**

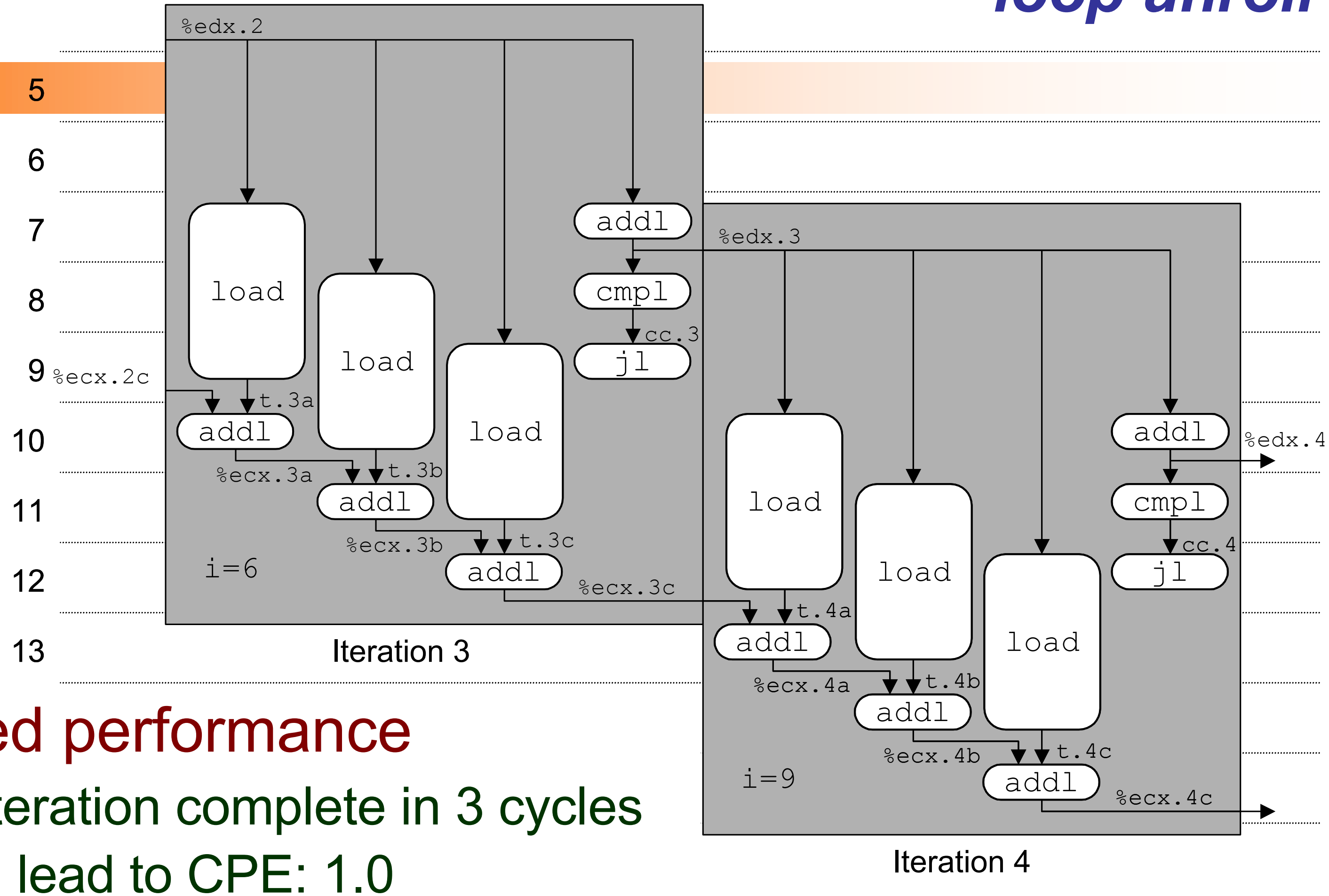
Machine dependent optimization techniques: loop unroll (2)

- **`loads`** can be pipelined, there are no dependencies
- only a set of loop control instructions




```
load  (%eax,%edx.0,4)    ➔ t.1a
iaddl t.1a, %ecx.0c      ➔ %ecx.1a
load 4(%eax,%edx.0,4)    ➔ t.1b
iaddl t.1b, %ecx.1a      ➔ %ecx.1b
load 8(%eax,%edx.0,4)    ➔ t.1c
iaddl t.1c, %ecx.1b      ➔ %ecx.1c
iaddl $3,%edx.0          ➔ %edx.1
cmpl %esi, %edx.1        ➔ cc.1
jl-taken cc.1
```

Machine dependent optimization techniques: loop unroll (3)



- Estimated performance
  - each iteration complete in 3 cycles
  - should lead to CPE: 1.0
- Measured performance
  - CPE: 1.33
  - 1 iteration for each 4 cycles

Machine dependent optimization techniques: loop unroll (4)

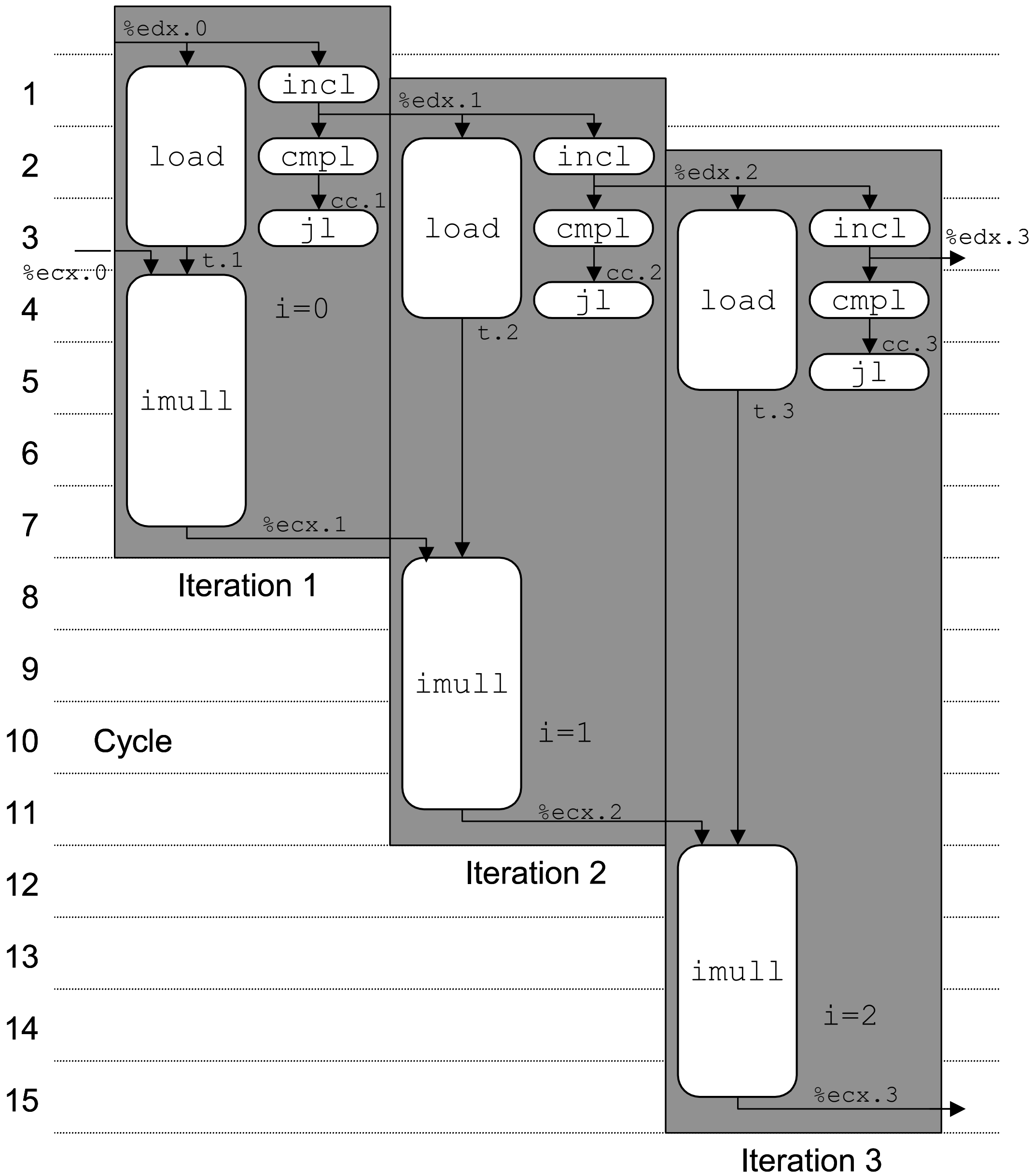
CPE value for several cases of loop unroll:

Degree of Unroll		1	2	3	4	8	16
Integer	Addition	2.00	1.50	1.33	1.50	1.25	1.06
Integer	Product	4.00					
fp	Addition	3.00					
fp	Product	5.00					

– only improves the integer addition
  - remaining cases are limited to the unit latency

– result does not linearly improve with the degree of unroll
  - subtle effects determine the exact allocation of operations

What else can be done?



Machine dependent optimization techniques: loop unroll with parallelism (1)

Sequential ... *versus* parallel!

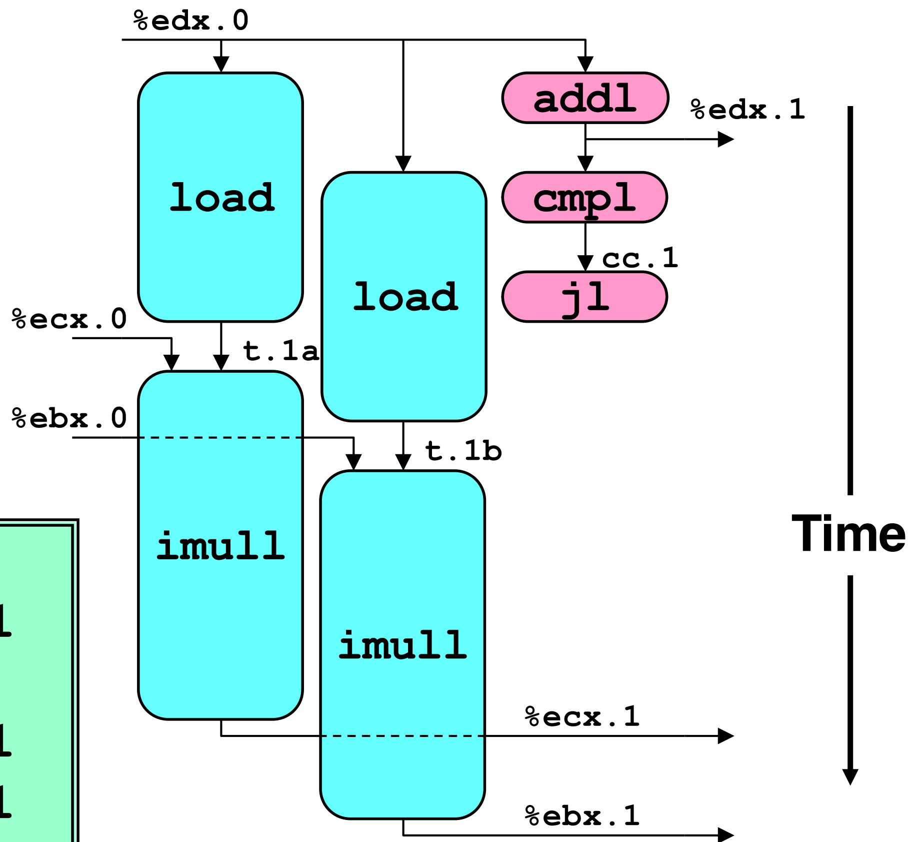
```
void combine6(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x0 = 1;
  int x1 = 1;
  int i;
  /* junta 2 elem's de cada vez */
  for (i = 0; i < limit; i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
  }
  /* completa os restantes elem's */
  for (; i < length; i++) {
    x0 *= data[i];
  }
  *dest = x0 * x1;
}
```

Optimization 5:
- accumulate in 2 different products
  - can be in parallel, if OP is associative!
- merge at the end

- Performance
  - **CPE: 2.0**
  - improvement 2x

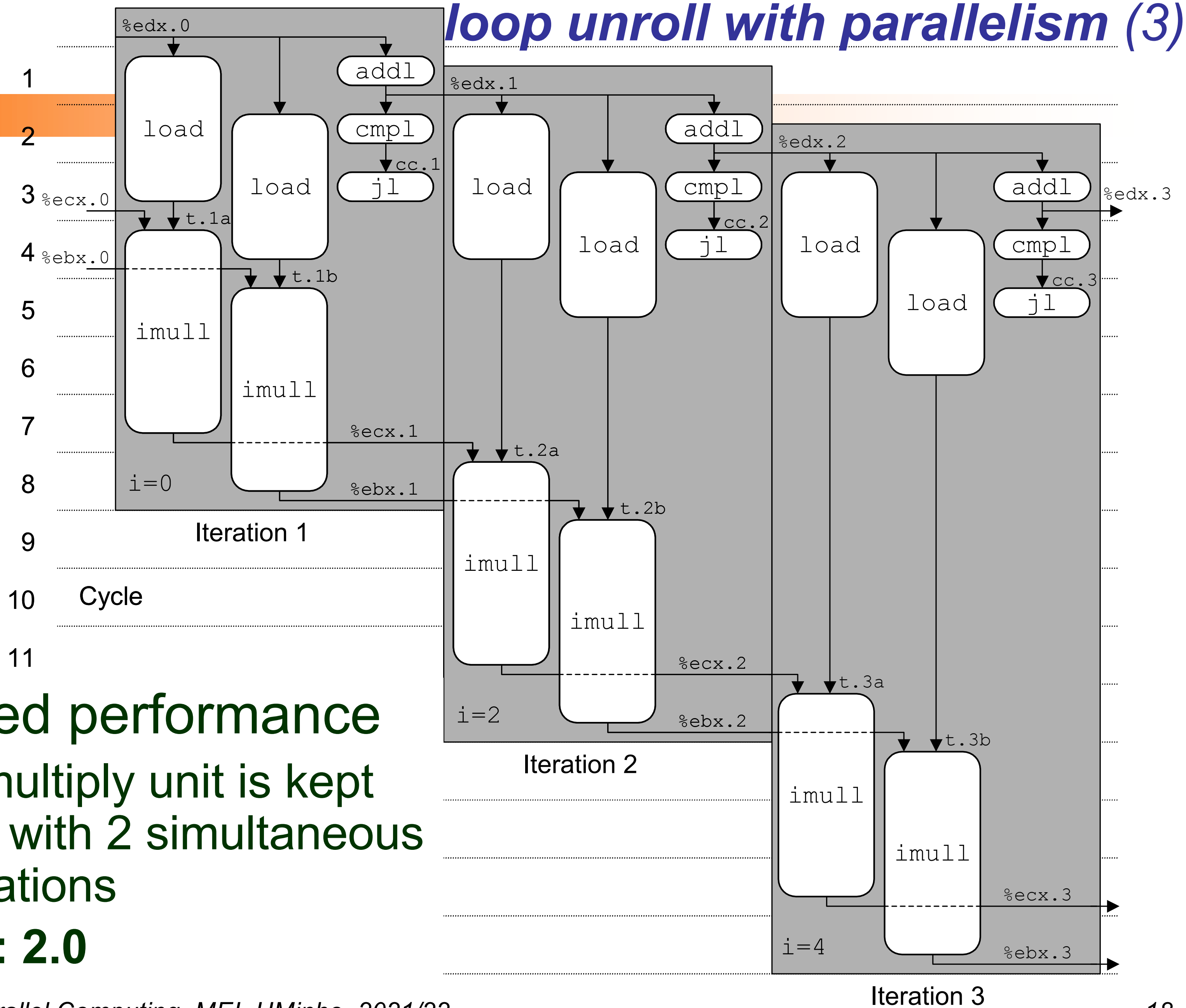
Machine dependent optimization techniques: loop unroll with parallelism (2)

– each product at the inner cycle does not depend from the other one…
– so, they can be pipelined
– known as iteration splitting




```
load (%eax,%edx.0,4)    ➔ t.1a
imull t.1a, %ecx.0      ➔ %ecx.1
load 4(%eax,%edx.0,4)   ➔ t.1b
imull t.1b, %ebx.0      ➔ %ebx.1
iaddl $2,%edx.0         ➔ %edx.1
cmpl %esi, %edx.1       ➔ cc.1
jl-taken cc.1
```

Machine dependent optimization techniques: loop unroll with parallelism (3)



Estimated performance

- the multiply unit is kept busy with 2 simultaneous operations
- **CPE: 2.0**

Code optimization techniques: comparative analyses of `combine`

Method	**Integer**		**Real** (single precision)	
	+	*****	**+**	*****
***Abstract* -g**	**42.06**	**41.86**	**41.44**	**160.00**
***Abstract* -O2**	**31.25**	**33.25**	**31.25**	**143.00**
Move `vec_length`	**20.66**	**21.25**	**21.15**	**135.00**
Access to data	**6.00**	**9.00**	**8.00**	**117.00**
Accum. in temp	**2.00**	**4.00**	**3.00**	**5.00**
***Unroll* 4x**	**1.50**	**4.00**	**3.00**	**5.00**
***Unroll* 16x**	1.06	**4.00**	**3.00**	**5.00**
***Unroll* 2x, paral. 2x**	**1.50**	**2.00**	**2.00**	**2.50**
***Unroll* 4x, paral. 4x**	**1.50**	**2.00**	1.50	**2.50**
***Unroll* 8x, paral. 4x**	**1.25**	1.25	**1.50**	2.00
Theoretical Optimiz	**1.00**	**1.00**	**1.00**	**2.00**
Worst : Best	***39.7***	***33.5***	***27.6***	***80.0***

Otimização de Desempenho

Resumo

– Fases de desenvolvimento

1. Selecionar o melhor algoritmo
   – Utilizar a análise de complexidade para comparar algoritmos
2. Escrever código legível e fácil de gerir
3. Eliminar bloqueadores de otimizações
4. Medir o perfil de execução
   – Otimizar as partes críticas para o desempenho
      » Operações repetidas muitas vezes (e.g., ciclos interiores)

– Código com otimizações é mais complexo de ler, manter e de garantir a correção

Common compiler optimizations

- Loops
  - Identify **induction variables** that are increased or decreased by a fixed amount on every iteration of a loop (e.g., j = i*4 +1 => j+= 5)
  - **Fission** - break a loop into multiple loops, each taking only a part of the loop's body
  - **Fusion** – combine loops to reduce loop overhead
  - **Inversion** - changes a standard *while* loop into a *do/while*
  - **Interchange** - exchange inner loops with outer loops
  - **Loop-invariant** code motion
  - **Loop unrolling** - duplicates the body of the loop multiple times
  - **Loop splitting** - breaks into multiple loops which have the same bodies but iterate over different contiguous portions of the index range
- Data flow
  - **Common sub-expression elimination/sharing**
  - **Reduction in strength -** expensive op's replaced with less expensive op's
  - **Constant folding** - replaces expressions of constants (e.g., 3 + 5) with their final value (8)
  - **Dead store elimination** - removal of assignments to variables that are not read

Common compiler optimizations

- Code generation
  - **Register allocation** - most frequently used variables are kept in processor registers
  - **Instruction selection** – selects 1 of several different ways to perform an operation
  - **Instruction scheduling** – avoid pipeline stalls
  - **Re-materialization** - recalculates a value instead of loading it from memory
- Other optimizations
  - **Bounds-checking elimination**
  - **Code-block reordering** – alters the order of basic blocks
  - **Dead code elimination**
  - **Inline expansion** - insert the body of a procedure inside the calling code
- Limitations
  - Memory aliasing & side effects of functions
  - Compilers do not typically improve the algorithmic complexity
  - A compiler typically only deals with a part of a program at a time
  - Time overhead of compiler optimizations

Resolution

Homework: ex 1 on mem hierarchy

Consider the following case study:
- … code in the SeARCH node with the Xeon Skylake …
- … same 2 instructions … in all cores of a single chip…
- … cores 6-way superscalar … 2 load units/core … cold data cache.

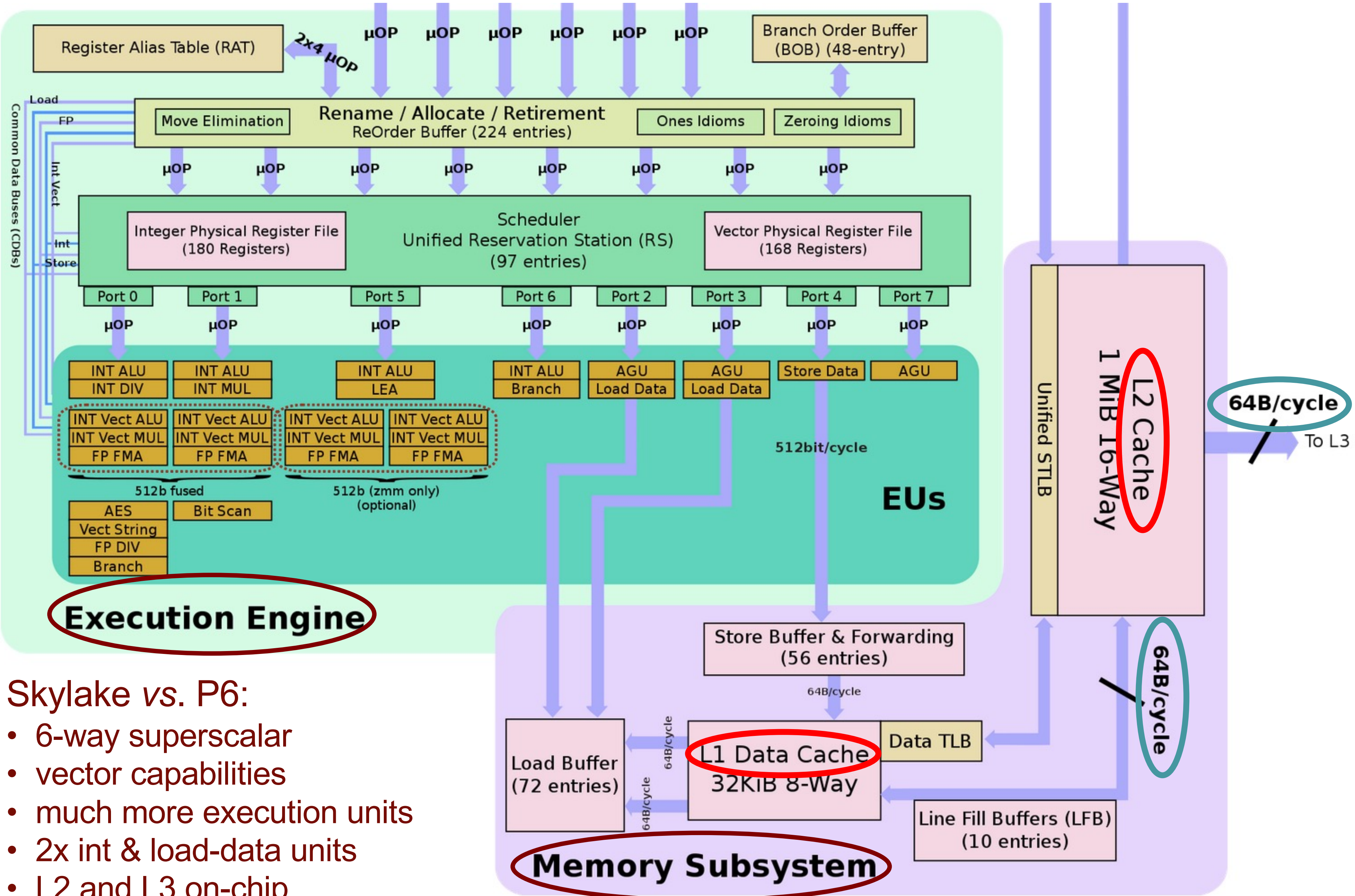
Compute:

a) **The** max required bandwidth to access the external RAM …

b) **The** aggregate peak bandwidth … DRAM-4 (w/ all memory channels).

- each clock cycle needs 2 mem accesses to fetch 2 doubles
- **max required bandwidth** to fetch a cache line for each double (cache is cold & doubles are far away):
  ???? GB/s
  note: the following 7 pairs of doubles are already in cache
- RAM in each Skylake Gold 6130: 6x DDR4-2666 (6x8 GiB)
- **peak bandwidth** of 6x DDR4-2666 in 6 memory channels:
  ???? GB/s

Partial view of a Skylake core (server)

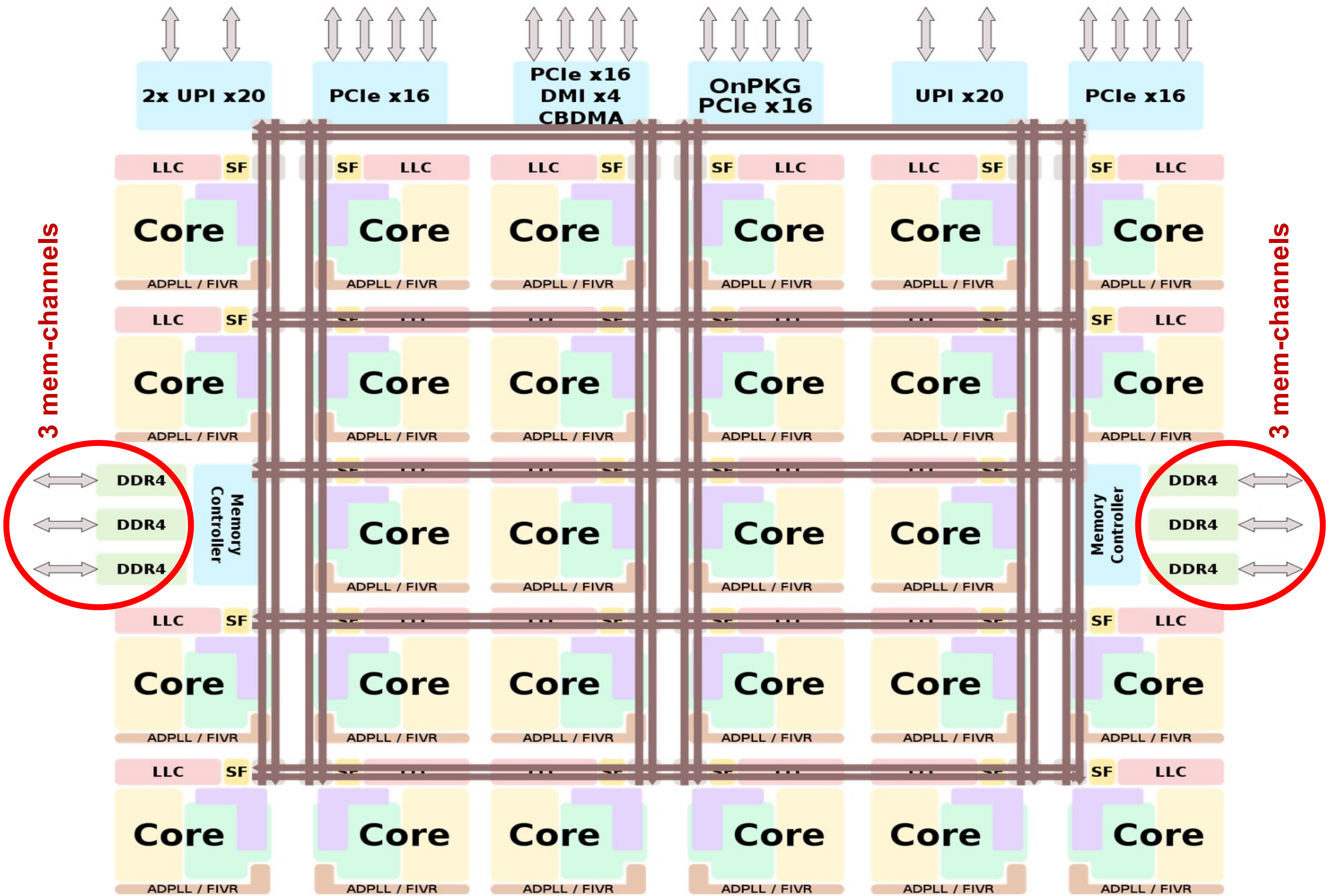


Skylake *vs.* P6:

- 6-way superscalar
- vector capabilities
- much more execution units
- 2x int & load-data units
- L2 and L3 on-chip

https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

Architecture of a 28-core Skylake (server)



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

Resolution

Homework: ex 1 on mem hierarchy

- each clock cycle needs 2 mem accesses to fetch 2 doubles
- **max required bandwidth** to fetch a cache line for **each** double (cache is cold & doubles are far away):
  (16 cores x 2 lines x 64 B/line) x clock_frequency =
  2048 B x 2 GHz = **4096 GB/s**
  Note: the following 7 pairs of doubles are already in cache
- RAM in each Skylake Gold 6130: 6x DDR4-2666 (6x8 GiB)
- **peak bandwidth** of 6x DDR4-2666 in 6 memory channels:
  6 mem_chan x 2.666 GT/s x 64 b/chan = **128 GB/s**

Homework: ex 2 on mem hierarchy

Similar to problem 1 (same node/chip in the cluster), but consider now:

- execution of code taking advantage of the AVX-512 facilities;
- execution of the same 2 vector instructions (that are already in the instruction cache) in all cores: load in register a vector of doubles followed by a multiplication by another vector of doubles in memory;
- the Skylake cores are 6-way superscalar and 2-way MT, and each core supports 2 simultaneous vector loads;
- the Skylake 6130 base clock rate with **AVX-512** code is **1.3 GHz**;
- these instructions are executed with a cold data cache.

Compute/estimate:

- **The** max required bandwidth to access the external RAM when executing these 2 vector instructions.
  Compare with the peak bandwidth computed before.

* https://en.wikichip.org/wiki/intel/xeon_gold/6130

Resolution

Exercise 2 on memory hierarchy

- each clock cycle needs 2 mem accesses to fetch 2 vectors with 8 doubles each (512 bits)
- **max required bandwidth** to fetch a cache line for each vector with 8 doubles (cache is cold):

  ???? GB/s

- note: same max required bandwidth as exercise 1, but this **mem access is required at each clock cycle**
- RAM in each Skylake Gold 6130: 6x DDR4-2666 (6x8 GiB)
- **peak bandwidth** of 6x DDR4-2666 in 6 memory channels: **??? GB/s**

Resolution

Exercise 2 on memory hierarchy

- each clock cycle needs 2 mem accesses to fetch 2 vectors with 8 doubles each (512 bits)
- **max required bandwidth** to fetch a cache line for each vector with 8 doubles (cache cold, AVX-512 clock rate lower*):
(16 cores x 2 lines x 64 B/line) x clock_rate =
2048 B x **1.3** GHz = **2662.4 GB/s**
- note: same max required bandwidth as exercise 1,
but this **mem access is required at each clock cycle**
- RAM in each Skylake Gold 6130: 6x DDR4-2666 (6x8 GiB)
- **peak bandwidth** of 6x DDR4-2666 in 6 memory channels:
6 mem_chan x 2.666 GT/s x 64 b/chan = **128 GB/s**

* https://en.wikichip.org/wiki/intel/xeon_gold/6130

Homework: ex 3 on cache performance

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: ?? x ?? = ??
  - D-cache: ?? x ?? x ?? = ??
- Actual CPI = 2 + ?? + ?? = **??**

Resolution

Exercise 3 on cache performance

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = **2**
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 0.02 x 100 = 2
  - D-cache: 0.36 x 0.04 x 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = **5.44**

Homework: ex 4 on multilevel cache

- Given
  - CPU base **CPI = 1**, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective **CPI = 9** (= 1 + 0.02 × 400)
- Now add L-2 cache …
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- CPI = 1 + ?? × ?? + ?? × ?? = **??**
- Performance ratio = 9 / ?? = **??**

Resolution

Exercise 4 on multilevel cache

- **CPU**: base CPI = 1, clock rate = 4GHz
- **L1 cache**: L1 miss rate/instruction = 2%
- **L2 cache**: access time = **5ns**, L2 miss rate/instruction = 25%, global miss rate = 2% x 25% = **0.5%**
- **Main memory**: access time = **100ns**
- **With just primary cache**
  - Miss penalty = 100ns / 0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = **9**
- **With L1 & L2 cache**
  - L1 miss penalty, L2 hit = **?? cycles**
  - L2 miss penalty = **?? cycles**
- **CPI** = 1 + **2%** × **??** cycles + **0.5%** × **???** cycles = **???**
- **Performance ratio** = 9 / **???** = **???**

Resolution

Exercise 4 on multilevel cache

- **CPU**: base CPI = 1, clock rate = 4GHz
- **L1 cache**: L1 miss rate/instruction = 2%
- **L2 cache**: access time = **5ns**, L2 miss rate/instruction = 25%, global miss rate = 2% x 25% = **0.5%**
- **Main memory**: access time = **100ns**
- **With just primary cache**
  - Miss penalty = 100ns / 0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = **9**
- **With L1 & L2 cache**
  - L1 miss penalty, L2 hit = 99.5% × 5ns / 0.25ns ≈ **20 cycles**
  - L2 miss penalty = 100ns / 0.25ns = **400 cycles**
- **CPI** = 1 + **2%** × **20** cycles + **0.5%** × **400** cycles = **3.4**
- **Performance ratio** = 9 / **3.4** = **2.6**

Homework: ex 5 on multilevel performance

Characterize the memory system of Xeon Skylake Gold 6130:

1. **L1 I-cache**
   - size **?** KiB/core, **?**-way set associative, **?** sets, line size **?** B, hit time **?** cycles, **?** B/cycle on transfer bandwidth L1 to the instruction fetch unit

   L1 D-cache
   - size **?** KiB/core, **?**-way set associative, **?** sets, line size **?** B, hit time **?** cycles, **?** B/cycle on load bandwidth L1 to load buffer unit
2. **L2 cache**
   - size **?** KiB/core, **?**-way set associative, **?** sets, line size **?** B, hit time **?** cycles, **?** B/cycle on load bandwidth L2 to L1
3. **L3 cache**
   - size **?** KiB/core, **?**-way set associative, **?** sets, line size **?** B, hit time **?** cycles, **?** B/cycle on load bandwidth L3 to L2
4. **DRAM**, DDR4-2666
   - up to **?** GT/s, bandwidth **?** GB/s per channel, **?** mem channels, aggregate bandwidth **?** GB/s, **?** B/cycle on peak load bandwidth DRAM to L3, NUMA-local latency **?** ns, NUMA-remote latency **?** ns

Resolution

Exercise 5 on multilevel performance

Characterize the memory system of Xeon Skylake Gold 6130:

1. **L1 I-cache**
   - size **32** KiB/core, **8**-way set associative, **64** sets, line size **64** B, hit time **?** cycles, **16** B/cycle on transfer bandwidth L1 to the instruction fetch unit

   L1 D-cache
   - size **32** KiB/core, **8**-way set associative, **64** sets, line size **64** B, hit time **4** cycles, **2x64** B/cycle on load bandwidth L1 to load buffer unit
2. **L2 cache**
   - size **1** MiB/core, **16**-way set associative, **1024** sets, line size **64** B, hit time **14** cycles, **64** B/cycle on load bandwidth L2 to L1
3. **L3 cache**
   - size **1.375** MiB/core, **11**-way set associative, **2048** sets, line size **64** B, hit time **50-70** cycles, **64** B/cycle on load bandwidth L3 to L2
4. **DRAM**, DDR4-2666
   - up to **2.666** GT/s, bandwidth **21.33** GB/s per channel, **6** mem channels, aggregate on peak load bandwidth DRAM to L3 **128** GB/s, NUMA-local latency **80** ns, NUMA-remote latency **120-140** ns

Homework: ex 6 on multilevel performance

Similar to problem 1 (same node/chip in the cluster, code), but consider now:

- execution of scalar code in a **2 GHz** single-core (already in L1 I-cache);
- code already takes advantage of all data cache levels (**L1, L2 & L3**), where 50% of data is placed on the RAM modules in the memory channels of the other PU chip (**NUMA architecture**);
- remember: the Skylake cores are **6-way superscalar** and **2-way MT**, and each core supports **2 simultaneous loads**;
- **cache latency time on hit**: take the average of the specified values;
- **memory latency**: 80 nsec (**NUMA local**), 120 nsec (**NUMA remote**);
- **miss rate per instruction** *(load* or *store):*
  - at **L1: 2%;** at **L2: 50%;** at **L3: 80%** (these are not global values!).

Compute/estimate:

1. **The miss penalty** per instruction at each cache level.
2. **The average memory stall cycles** per instruction that degrades CPI.

Resolution

Exercise 6 on memory hierarchy

- **PU**: base CPI = 1, clock rate = 2 GHz
- **L1 cache**: L1 miss rate/instruction = 2%;
- **L2 cache**: access time = **14** cycles, global miss rate = 2% x 50% = **1%**
- **L3 cache**: access time = **60** cycles, L3 miss rate = 80%,
  global miss rate = 1% x 80% = **0.8%**
- **Main memory**: NUMA local access time = **80ns**, NUMA remote = **120ns**
  average memory access = ((80ns+120ns)/2) / 0.5ns = **200 cycles**

- **CPI?**