Master Informatics Eng.

2020/21

A.J.Proença

Background to this course

The specialization website:

http://gec.di.uminho.pt/miei/cpd/

Perfil: Computação Paralela Distribuída

Mestrado Eng.ª Informática
2020/2021
Docente responsável: A. Proença

Direitos de Autor & Copyright
Avisos | *Equipa docente* | *Objetivos e Organização* | *Resultados de Aprendizagem* | *UCs: AA PCP AP ESC LI*

(Ano anterior: 2019/2020)

Ultima Modificação: 02 Out 2020

departamento de informática

Equipa docente

- **Docentes responsáveis pela lecionação das UCs**

Arquiteturas Avançadas (AA)
Alberto José Proença
email: aproenca<at>di.uminho.pt

Paradigmas de Computação Paralela (PCP)
João Luís Sobral
email: jls<at>di.uminho.pt

Sessões práticas de AA & PCP
André Martins Pereira
email: ampereira<at>di.uminho.pt

Algoritmos Paralelos (AP)
Equipa liderada por Rui Ralha e João Luís Sobral
email: r_ralha<at>math.uminho.pt e jls<at>di.uminho.pt

Engenharia dos Sistemas de Computação (ESC)
António Manuel Pina
email: pina<at>di.uminho.pt

The course website:
http://gec.di.uminho.pt/miei/cpd/aa/

CPD - Arquiteturas Avançadas

Mestrado Eng.ª Informática
2020/2021
Docente responsável: A. Proença

Sumários
de
Arquiteturas Avançadas

Semana: 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

(Ano anterior: 2019/2020)

Ultima Modificação: 01 Out 2020

departamento de informática

Avisos:

1. **Avaliação (proposta):** elementos de avaliação: **(i)** participação em todas as sessões laboratoriais (10% a 15%), **(ii)** trabalho prático com relatório jan-21 (30% a 50%), e **(iii)** teste escrito em data a definir, podendo ser na 1ª semana de jan-21 (40% a 60%). *(01-out-20)*

Display a menu

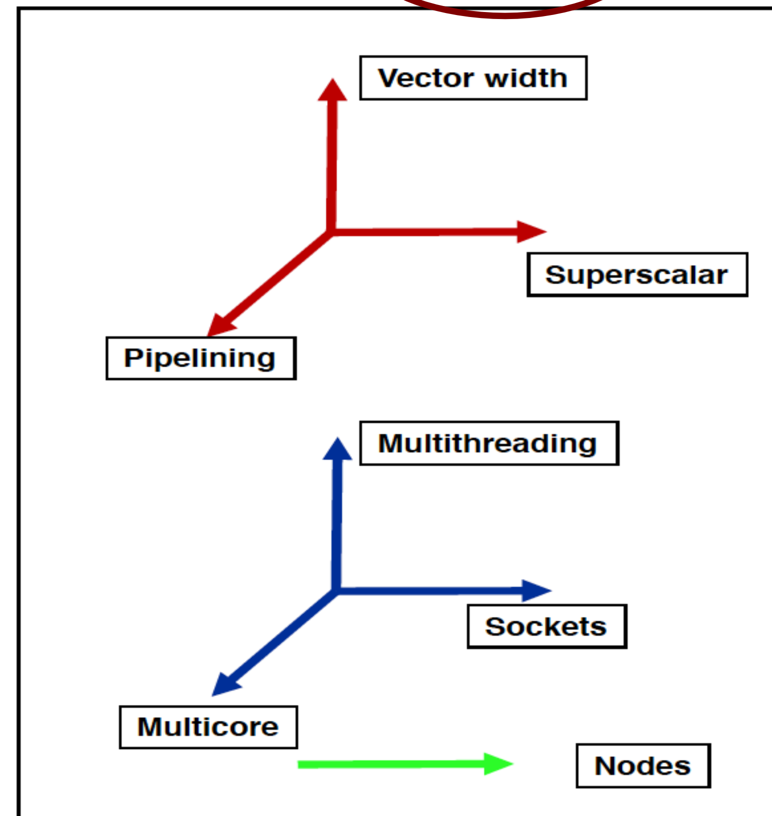
How:

- understanding the organization of computer system (its architecture) to develop efficient algorithms and data structures
- profiling & measuring the execution efficiency

the AA course

Where:

- in sequential code with ILP
  - *pipelining*
  - *superscalar w/ out-of-order exec*
  - vector processing
- in code with thread parallelism
  - multithreading in-core (SMT)
  - multithreading in multicore
  - multithreading in multiple devices
- in code with process parallelism
  - multiprocessing in interconnected servers (clusters)



Roles & expectations in a Master course

The AA lecturer:

- to motivate the students to the course contents
- to coach & guide the students, providing links to course materials
- to emphasize & discuss the key topics with the students
- to supply lab guides to experimental work & help the students to overcome difficulties
- to expect mature students that prepare the lectures & the experimental work on a weekly basis

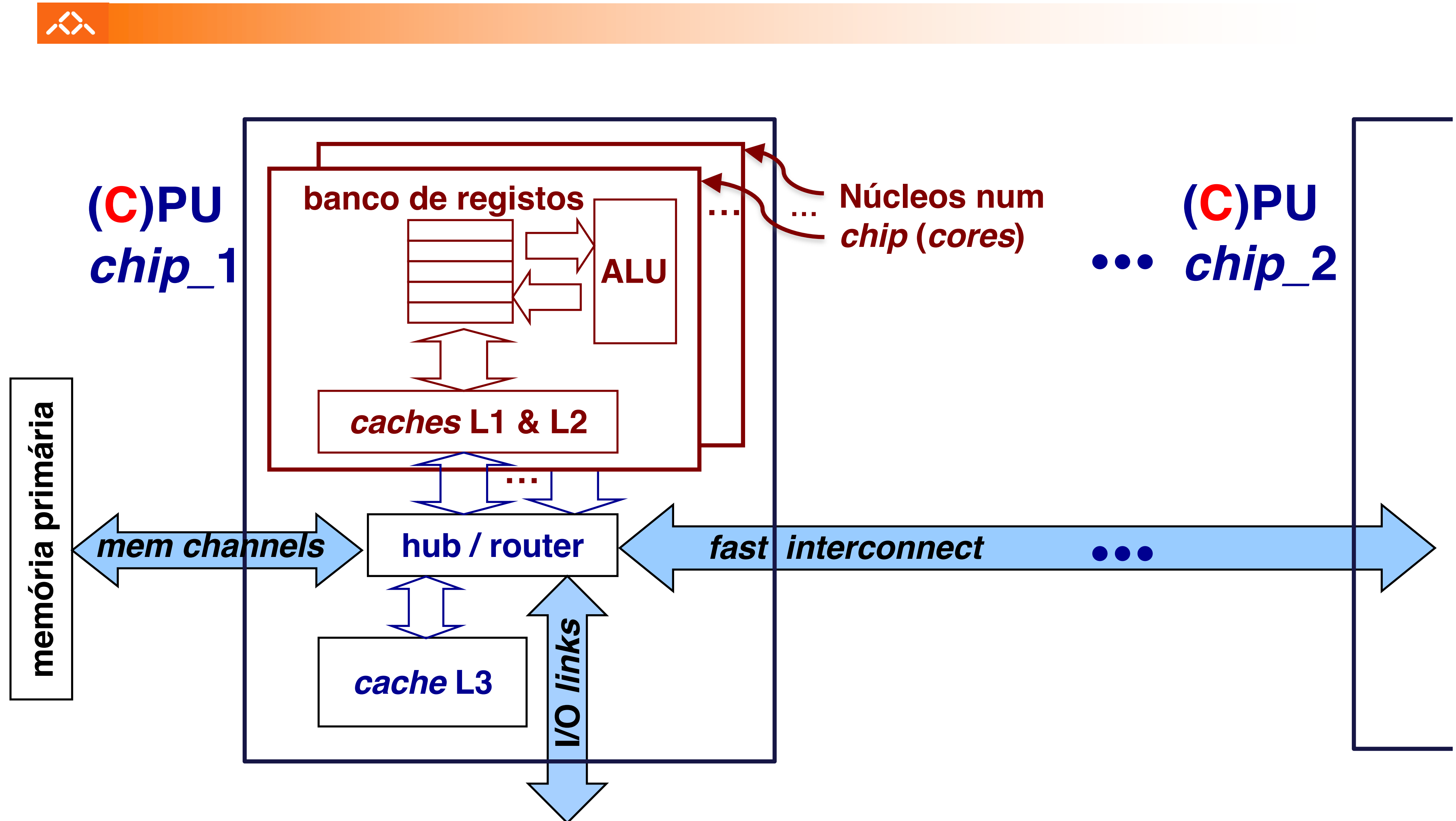
The students:

- to weekly follow the provided course materials *(read/view)*
- to actively participate on the online and lab classes *(using laptops)*
- to learn and acquire the relevant skills through guided self-study, to help their future professional activities for a long time

Key concepts to revise:

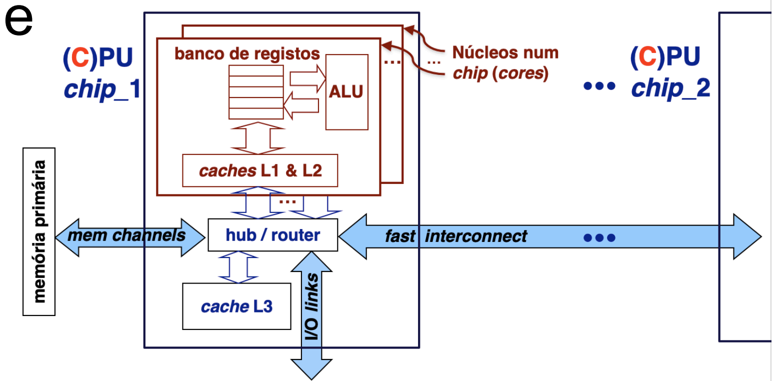
- *numerical data representation (integers & FP)*
- *ISA (Instruction Set Architecture)*
- *how C compilers generate code (a look into assembly code)*
  - *how scalar and structured data are allocated*
  - *how control structures are implemented*
  - *how to call/return from function/procedures*
  - *what architecture features impact performance*
- *improvements to enhance performance in a single PU*
  - *ILP: pipeline, multiple issue, …*
  - *data parallelism: SIMD/vector processing, ...*
  - *thread-level parallelism*
  - *memory hierarchy: cache levels, ...*

Keyword: **paralellism** (covers ILP, data parallelism, thread-level parallelism)


(C)PU
chip_1
banco de registos
ALU
caches L1 & L2
Núcleos num
chip (cores)
(C)PU
chip_2
memória primária
mem channels
hub / router
fast interconnect
cache L3
I/O links

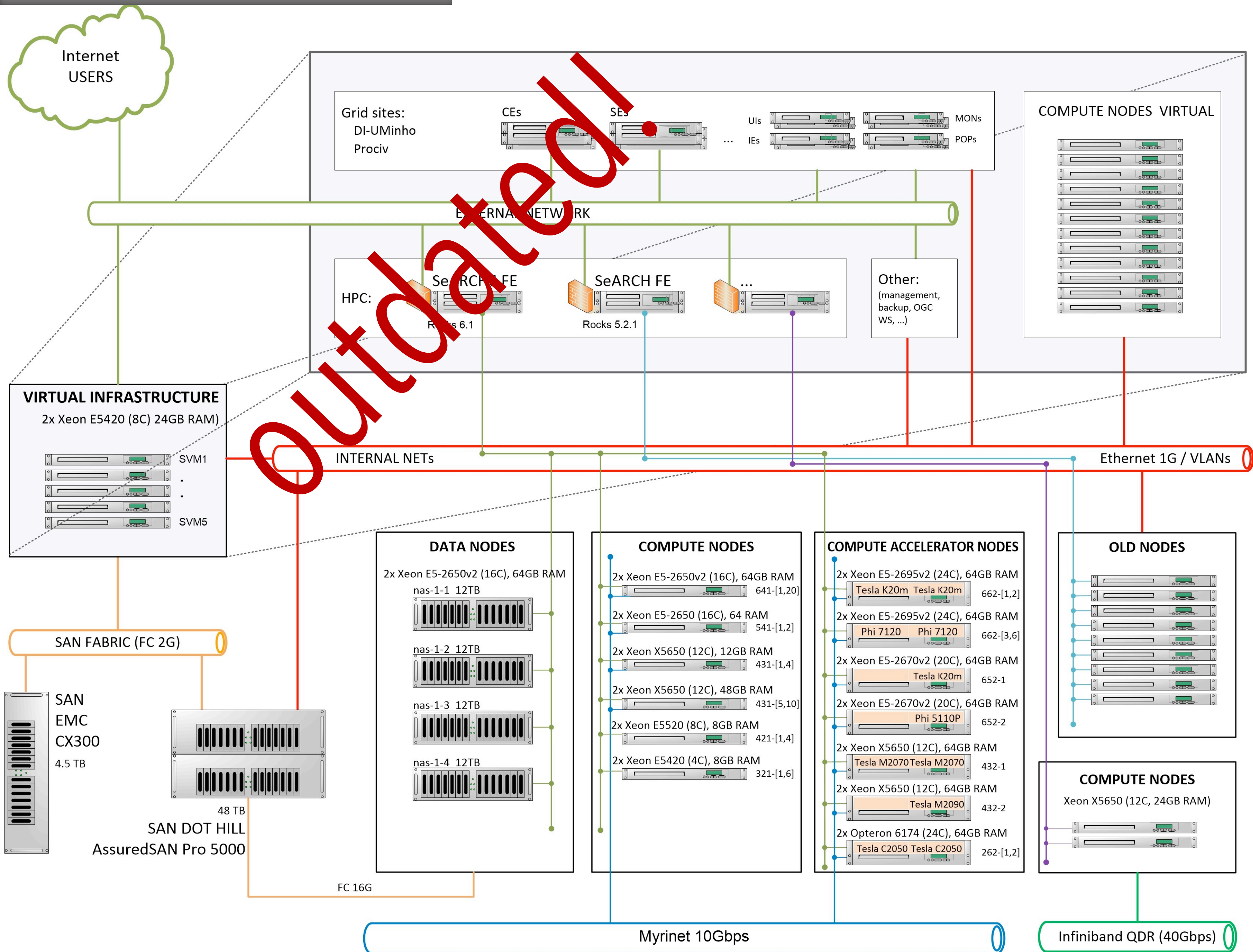
Questions/homework:

1. Identify the current available devices with the largest #cores; state how many in the device/package & show an image
   a) Designed by Intel
   b) Designed by AMD
   c) Designed by ARM
   d) Designed by a japanese company
   e) Designed by chinese company
   f) Worldwide



2. What are the key chalenges to design a chip with a very large large number of cores?

The cluster at DI



Key textbook for AA

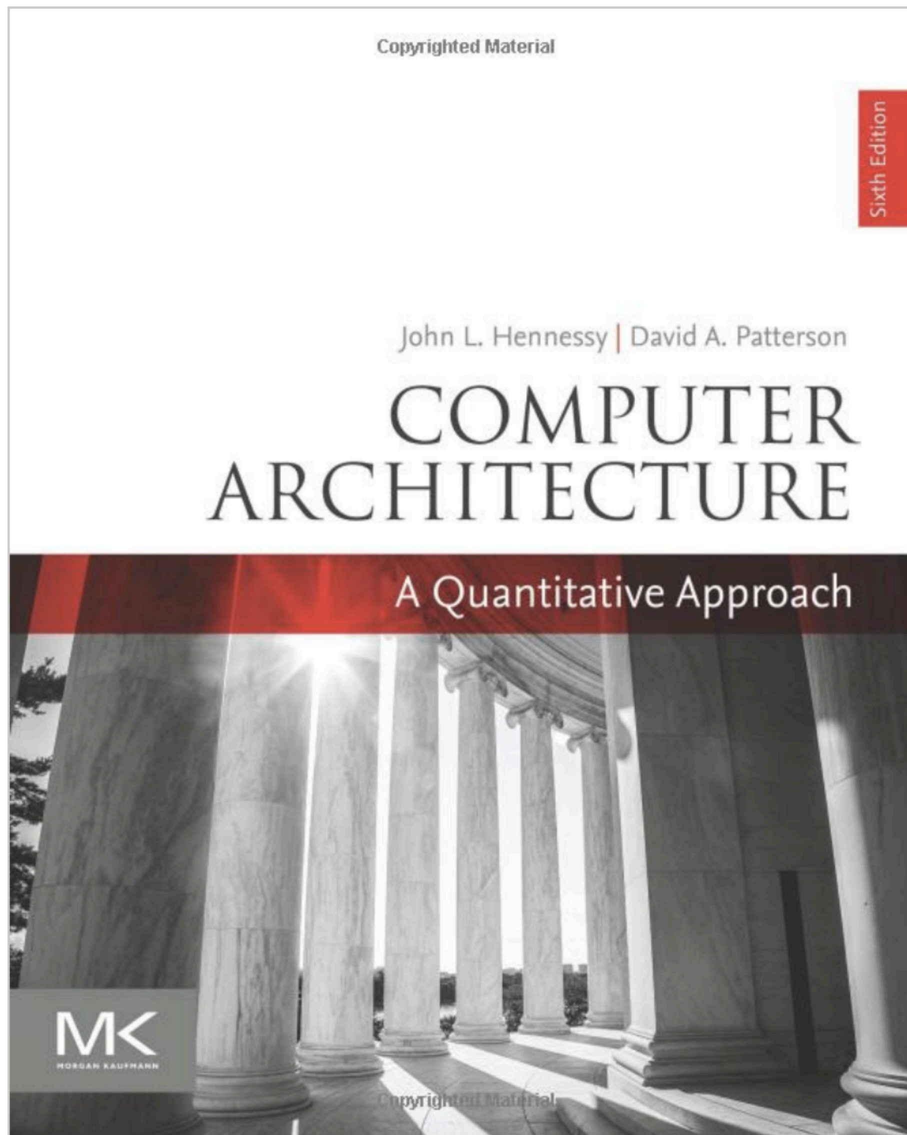


Table of Contents

Printed Text

1. Fundamentals of Quantitative Design and Analysis
2. Memory Hierarchy Design
3. Instruction-Level Parallelism and Its Exploitation
4. Data-Level Parallelism in Vector, SIMD, and GPU Architectures
5. Multiprocessors and Thread-Level Parallelism
6. The Warehouse-Scale Computer
7. Domain Specific Architectures

A. Instruction Set Principles
B. Review of Memory Hierarchy
C. Pipelining: Basic and Intermediate Concepts

Online

D. Storage Systems
E. Embedded Systems
F. Interconnection Networks
G. Vector Processors
H. Hardware and Software for VLIW and EPIC
I. Large-Scale Multiprocessors and Scientific Applications
J. Computer Arithmetic
K. Survey of Instruction Set Architectures
L. Advanced Concepts on Address Translation
M. Historical Perspectives and References

MK®

Recommended textbook (1)

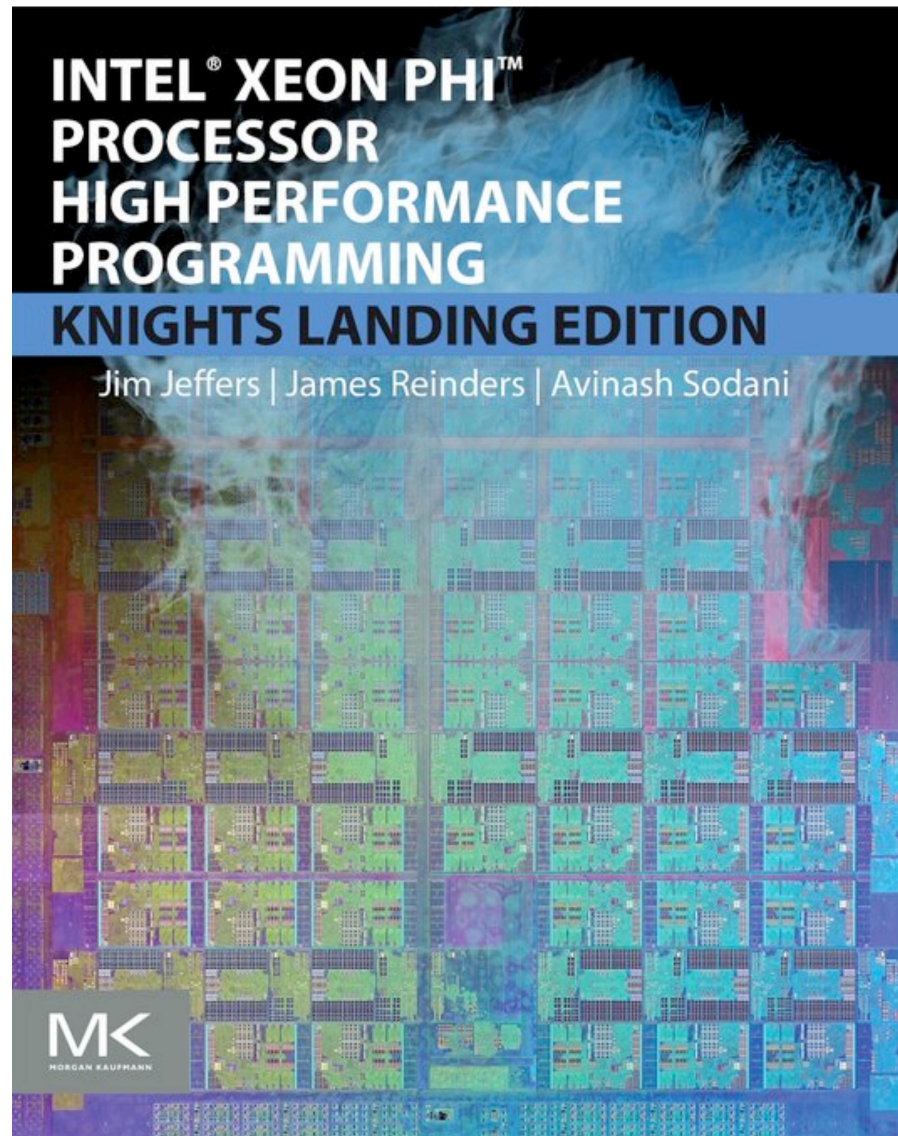


Table of Contents

Section I: Knights Landing.

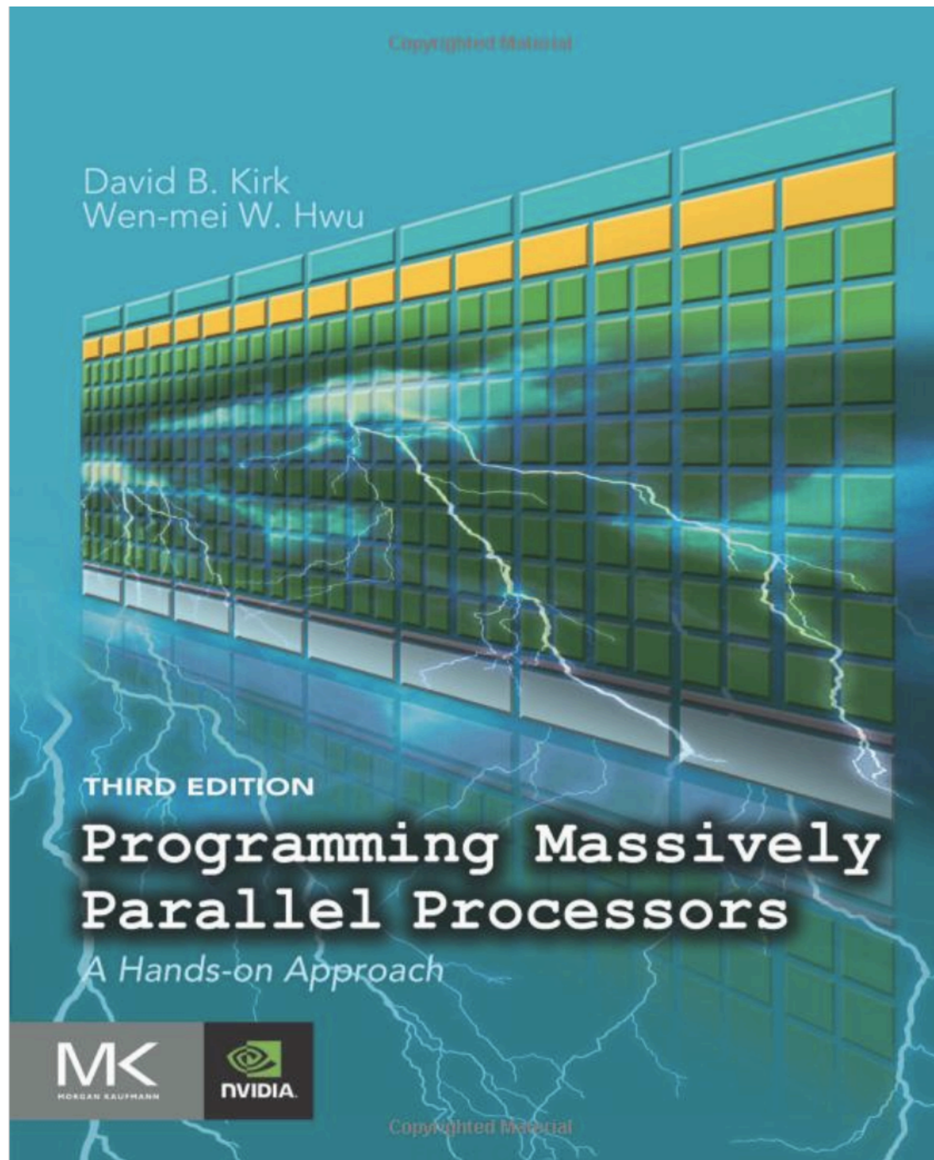
Chapter 1: Introduction
Chapter 2: Knights Landing Overview
Chapter 3: Programming MCDRAM and Cluster Modes
Chapter 4: Knights Landing Architecture
Chapter 5: Intel Omni-Path Fabric
Chapter 6: µarch Optimization Advice

Section II: Parallel Programming
Chapter 7: Programming Overview for Knights Landing
Chapter 8: Tasks and Threads
Chapter 9: Vectorization
Chapter 10: Vectorization Advisor
Chapter 11: Vectorization with SDLT
Chapter 12: Vectorization with AVX-512 Intrinsics
Chapter 13: Performance Libraries
Chapter 14: Profiling and Timing
Chapter 15: MPI
Chapter 16: PGAS Programming Models
Chapter 17: Software Defined Visualization
Chapter 18: Offload to Knights Landing
Chapter 19: Power Analysis

Section III: Pearls
Chapters 20-26: Results on LAMMPS, SeisSol, WRF, N-Body Simulations, Machine Learning, Trinity mini-applications and QCD are discussed.

MK®

Recommended textbook (2)



Contents

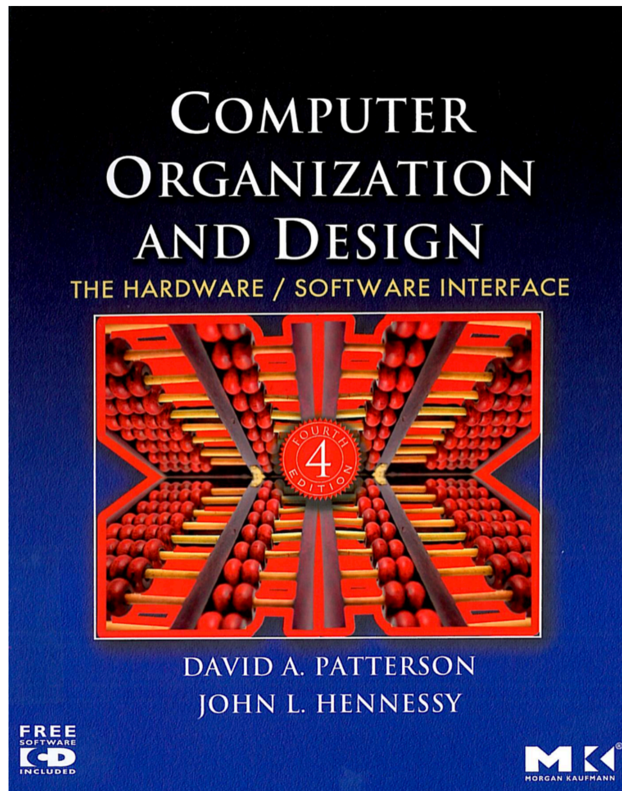
1. Introduction
2. Data parallel computing
3. Scalable parallel execution
4. Memory and data locality
5. Performance considerations
6. Numerical considerations
7. Parallel patterns: Convolution
8. Parallel patterns: Prefix Sum
9. Parallel patterns : Parallel Histogram Computation
10. Parallel patterns: Sparse Matrix Computation
11. Parallel patterns: Merge Sort
12. Parallel patterns: Graph Searches
13. CUDA dynamic parallelism
14. Application case study—non-Cartesian magnetic …
15. Application case study—molecular visualization …
16. Application case study—machine learning
17. Parallel programming and computational thinking
18. Programming a heterogeneous computing cluster
19. Parallel programming with OpenACC
20. More on CUDA and graphics processing computing
21. Conclusion and outlook

Appendix A. An introduction to OpenCL
Appendix B. THRUST: a productivity-oriented library for CUDA

Concepts from undegrad Computer Systems

Some notes/comments:

– *most slides are borrowed from*

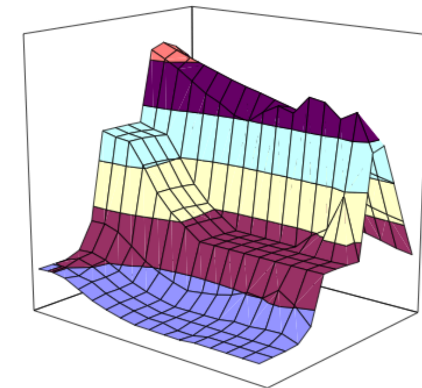


and some from

Computer Systems

A Programmer's Perspective [1]

(*Beta Draft*)



Randal E. Bryant
David R. O'Hallaron

August 1, 2001

more details at
http://gec.di.uminho.pt/miei/sc1920/

The BIG Picture

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Paralellism helps to significantly reduce CPI

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

The BIG Picture

- Yes, but not as much as we' d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

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

The following slides are a selection from CS.

The originals are in:

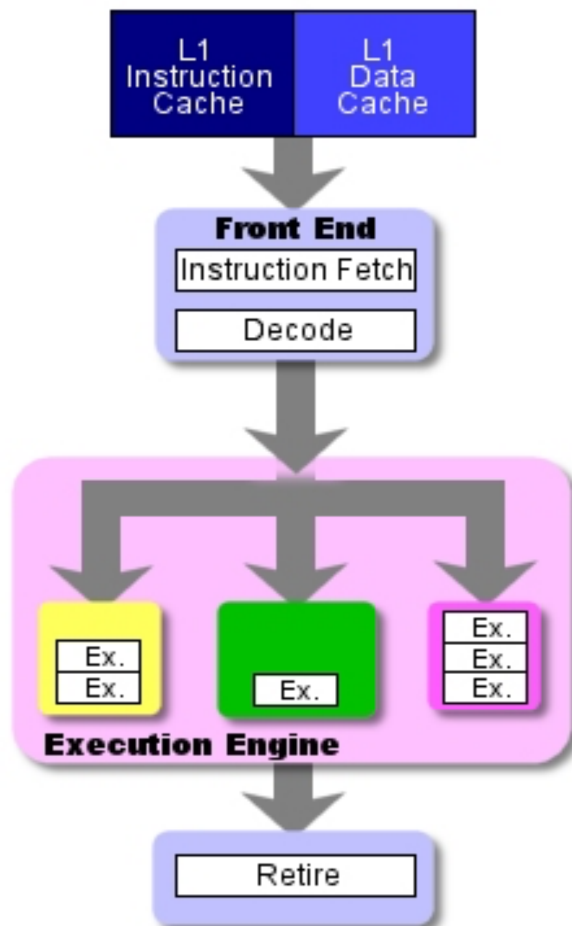
- http://gec.di.uminho.pt/miei/cpd/aa/slides_sc.zip

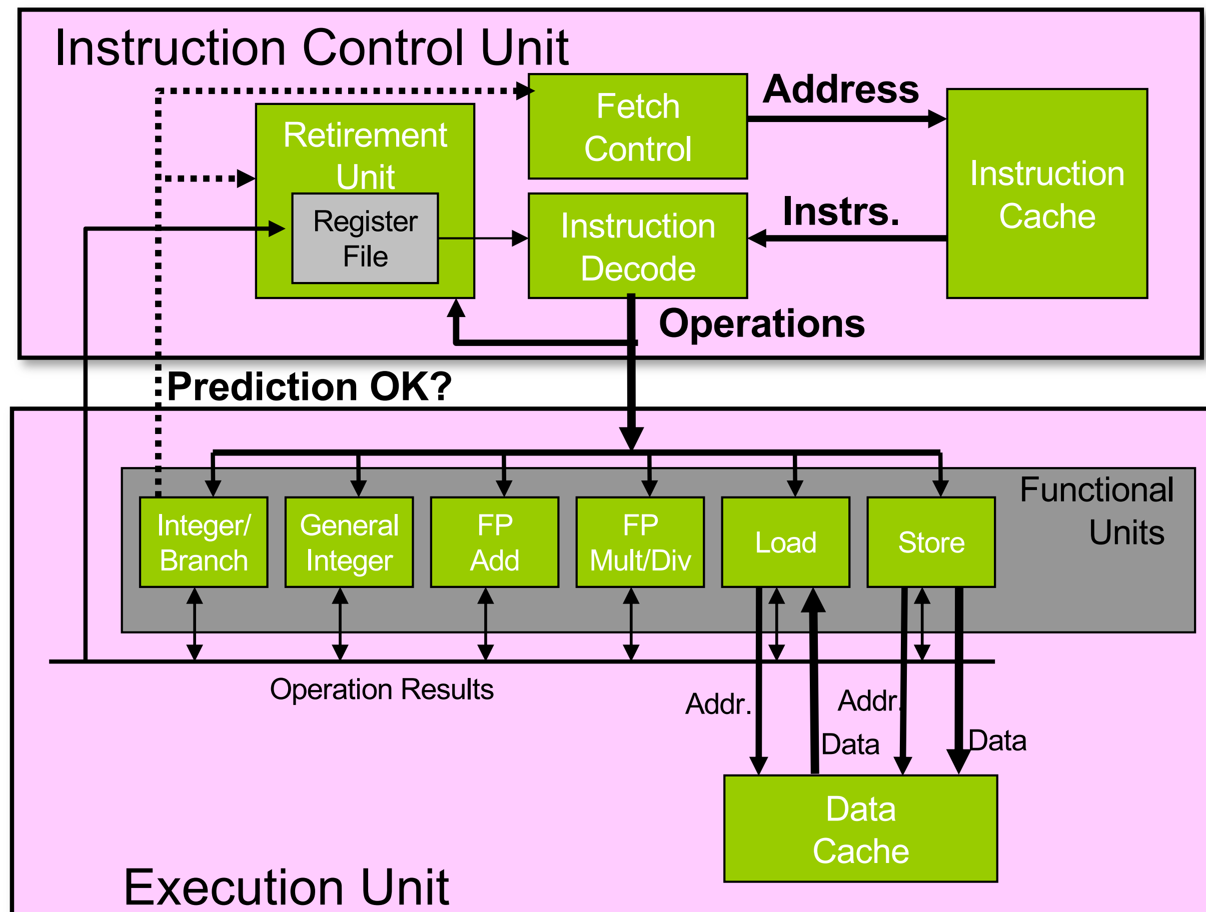
Last year lectures were recorded and the videos were placed on the e-platform; they are available here:

- http://gec.di.uminho.pt/miei/cpd/aa/videos_sc.zip

Internal architecture of Intel P6 processors

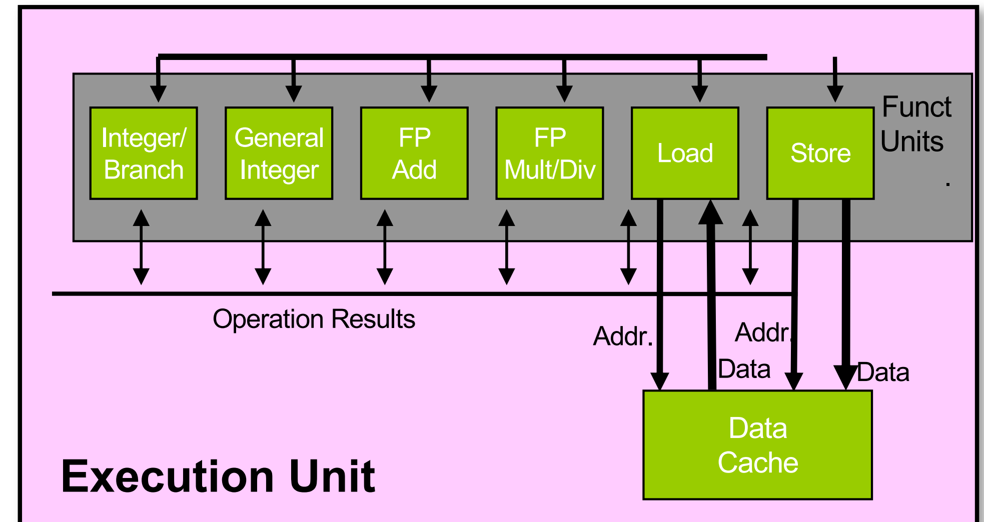
<u>Note</u>: "Intel P6" is the common μarch name for PentiumPro, Pentium II & Pentium III, which inspired Core, Nehalem and later generations





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
```

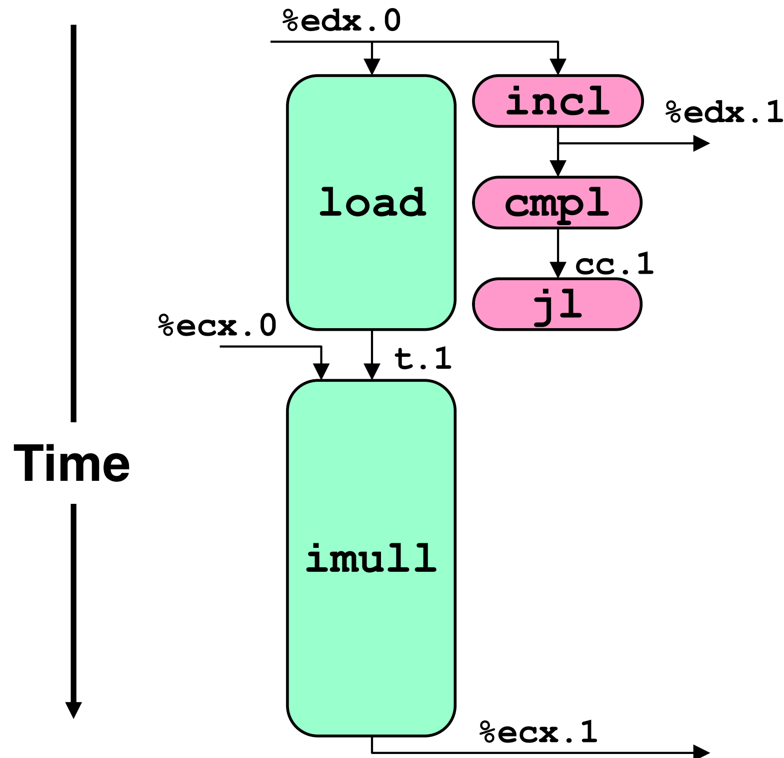
- **Translating 1st iteration**

```
.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
```

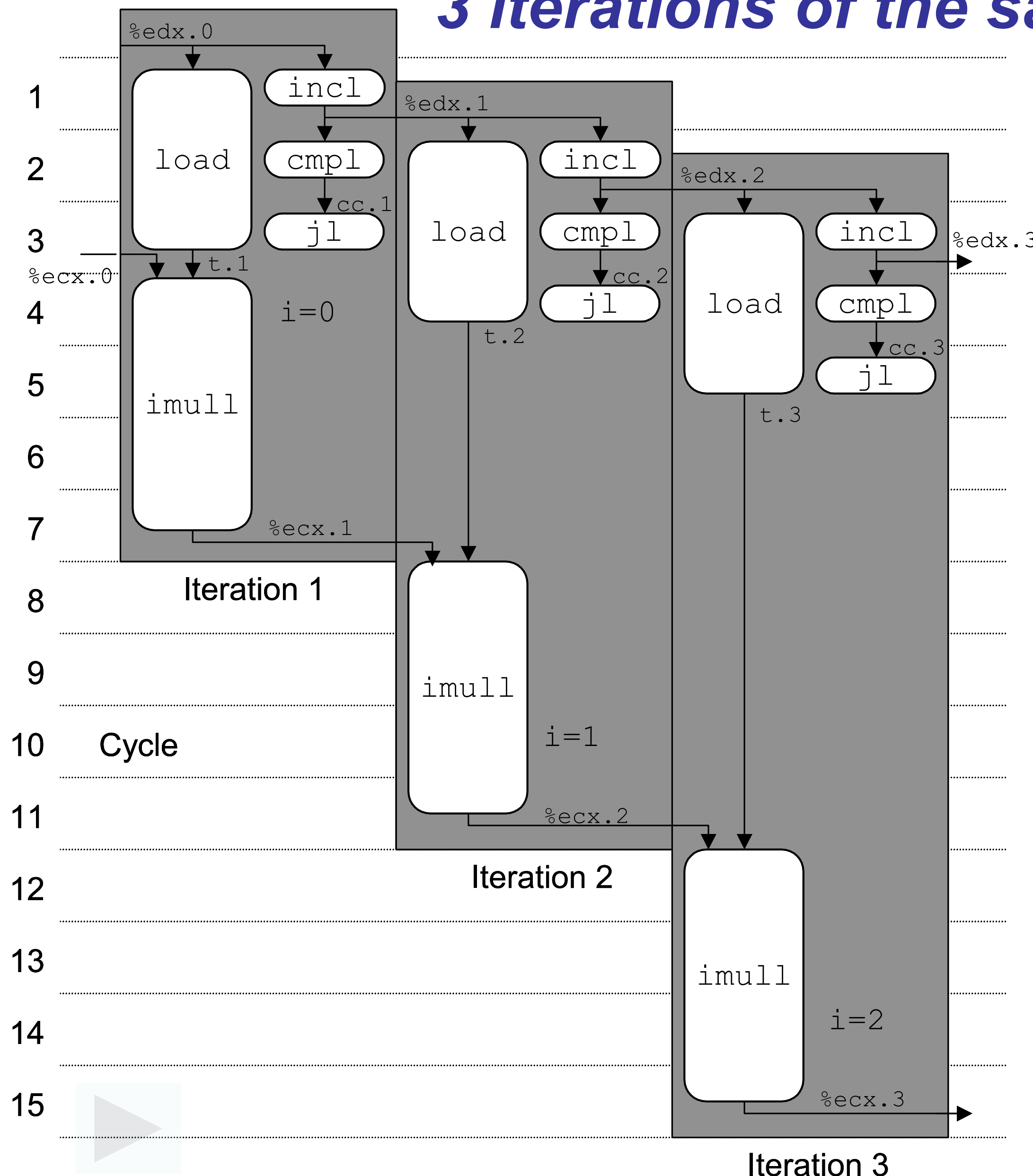
Visualizing instruction execution in P6: 1 iteration of the multiplication cycle on `combine`




```
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
```

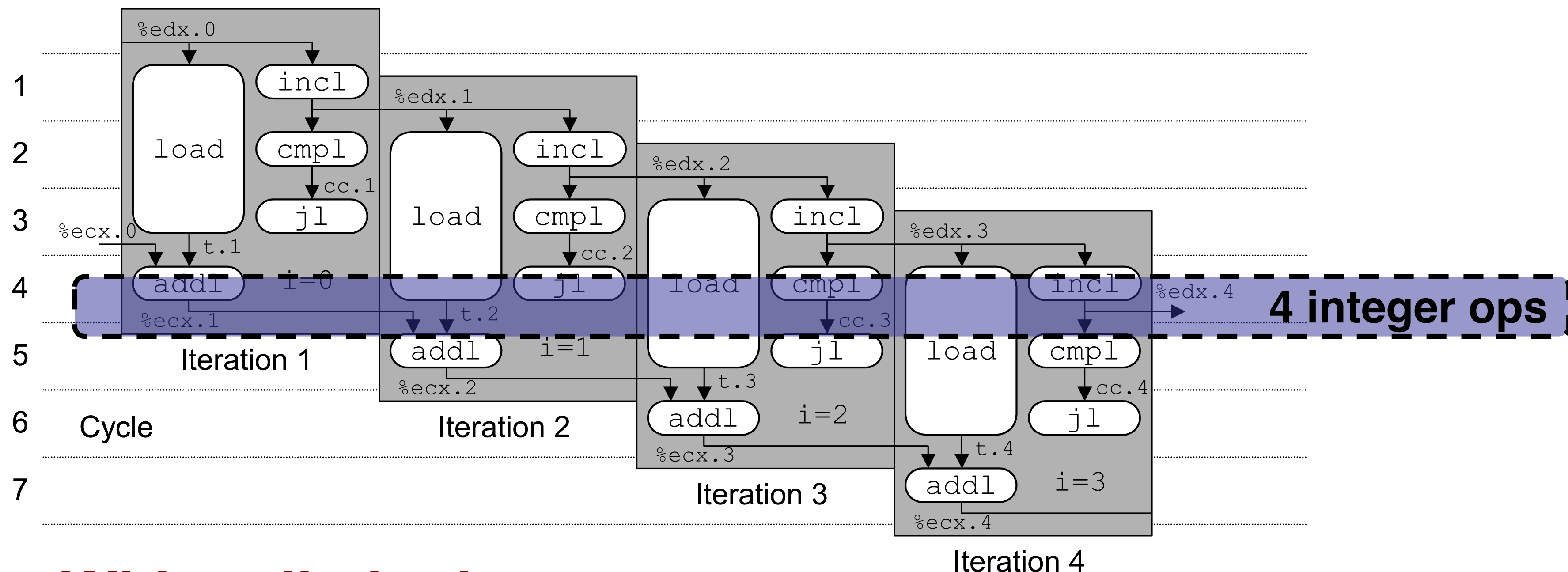
- **Operations**
  - vertical axis shows the time the instruction is executed
    - an operation cannot start with its operands
  - time length measures latency
- **Operands**
  - arcs are only showed for operands that are used in the context of the *execution unit*

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



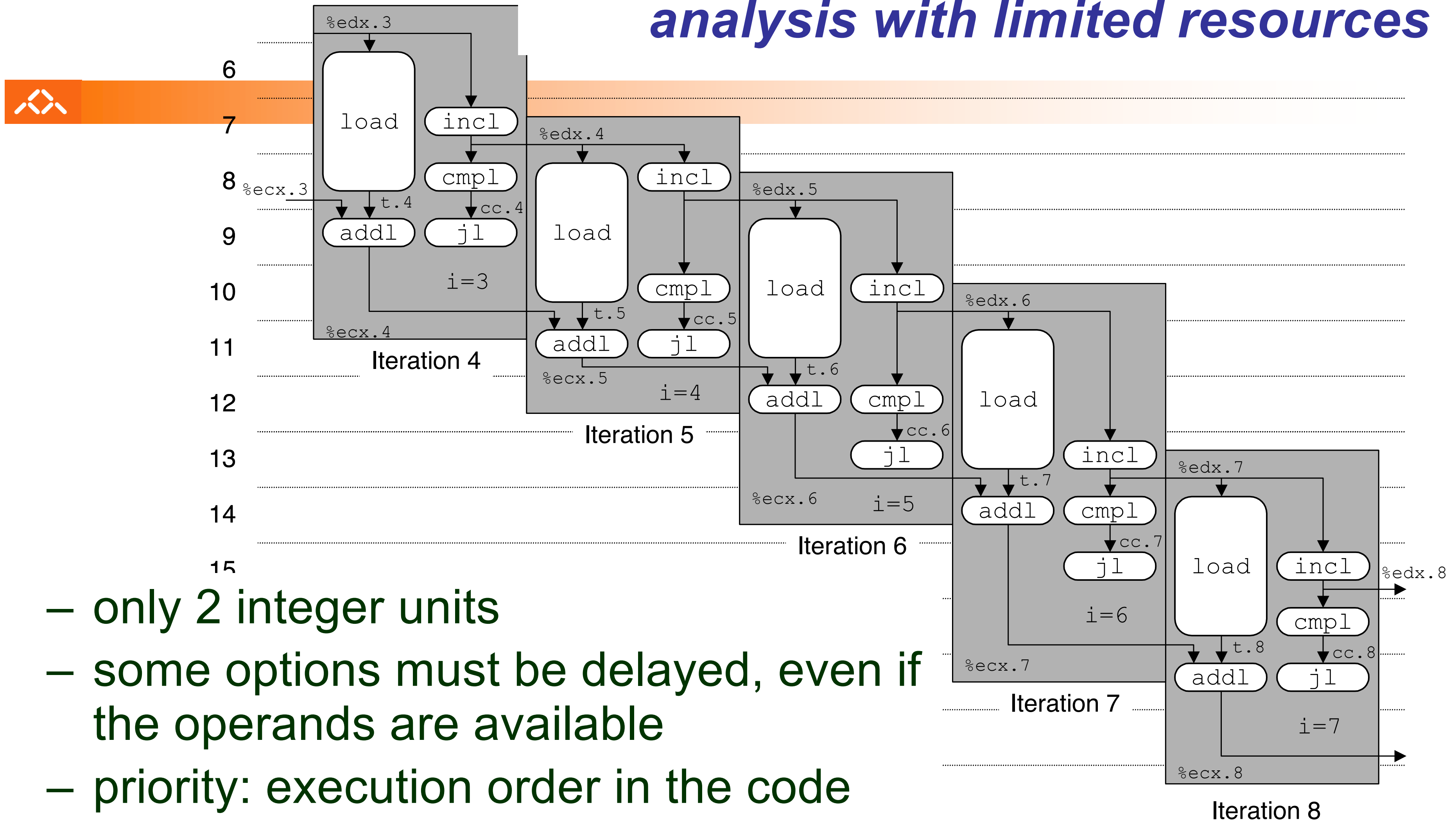
- **With unlimited resources**
  - parallel and pipelined execution of operations at the EU
  - out-of-order and speculative execution
- **Performance**
  - limitative factor: latency of integer multiplication
  - CPE: 4.0

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



- **With <u>unlimited</u> 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

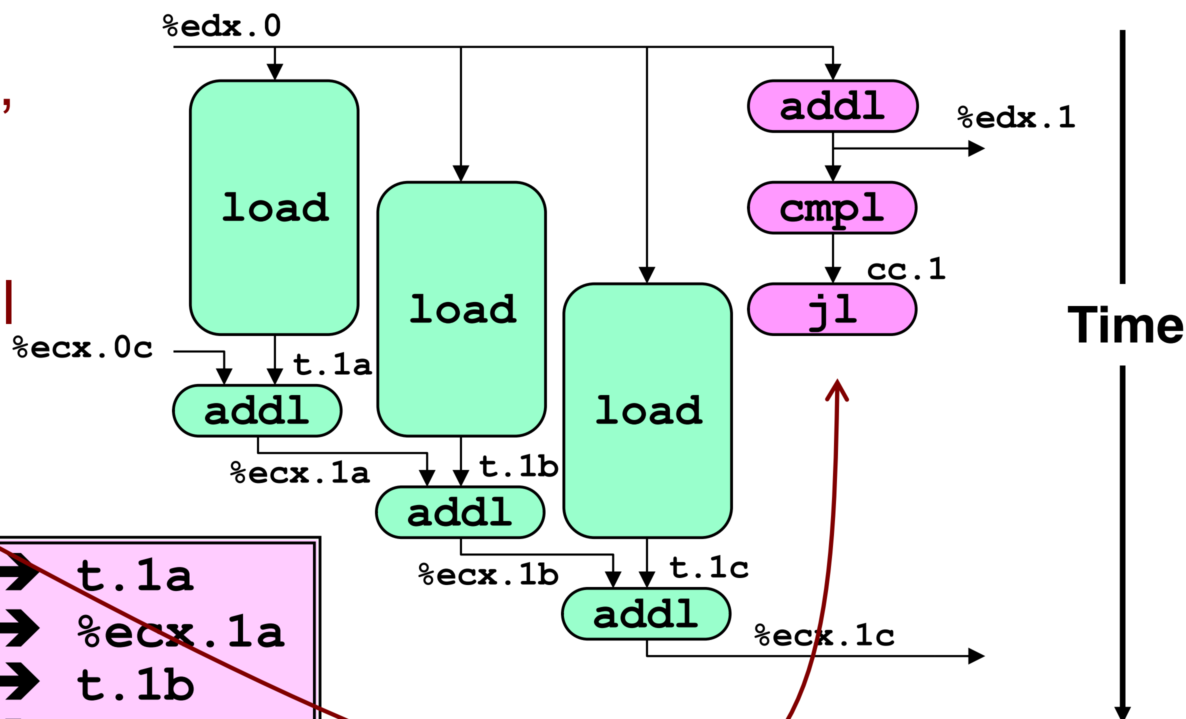
```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* junta 3 elem's no mesmo ciclo */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
           + data[i+2];
  }
  /* completa os restantes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

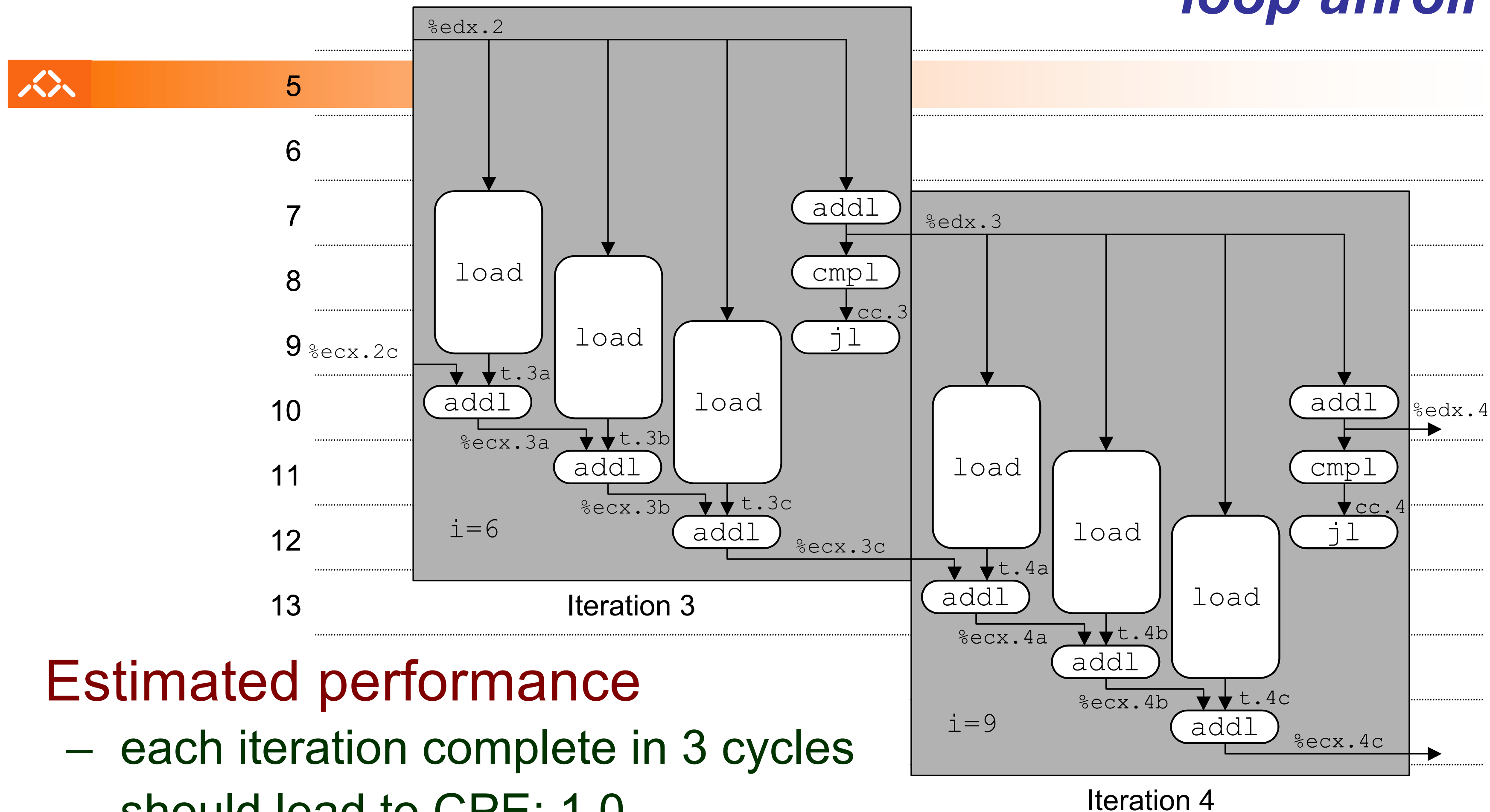
Machine dependent optimization techniques: loop unroll (2)

- **`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

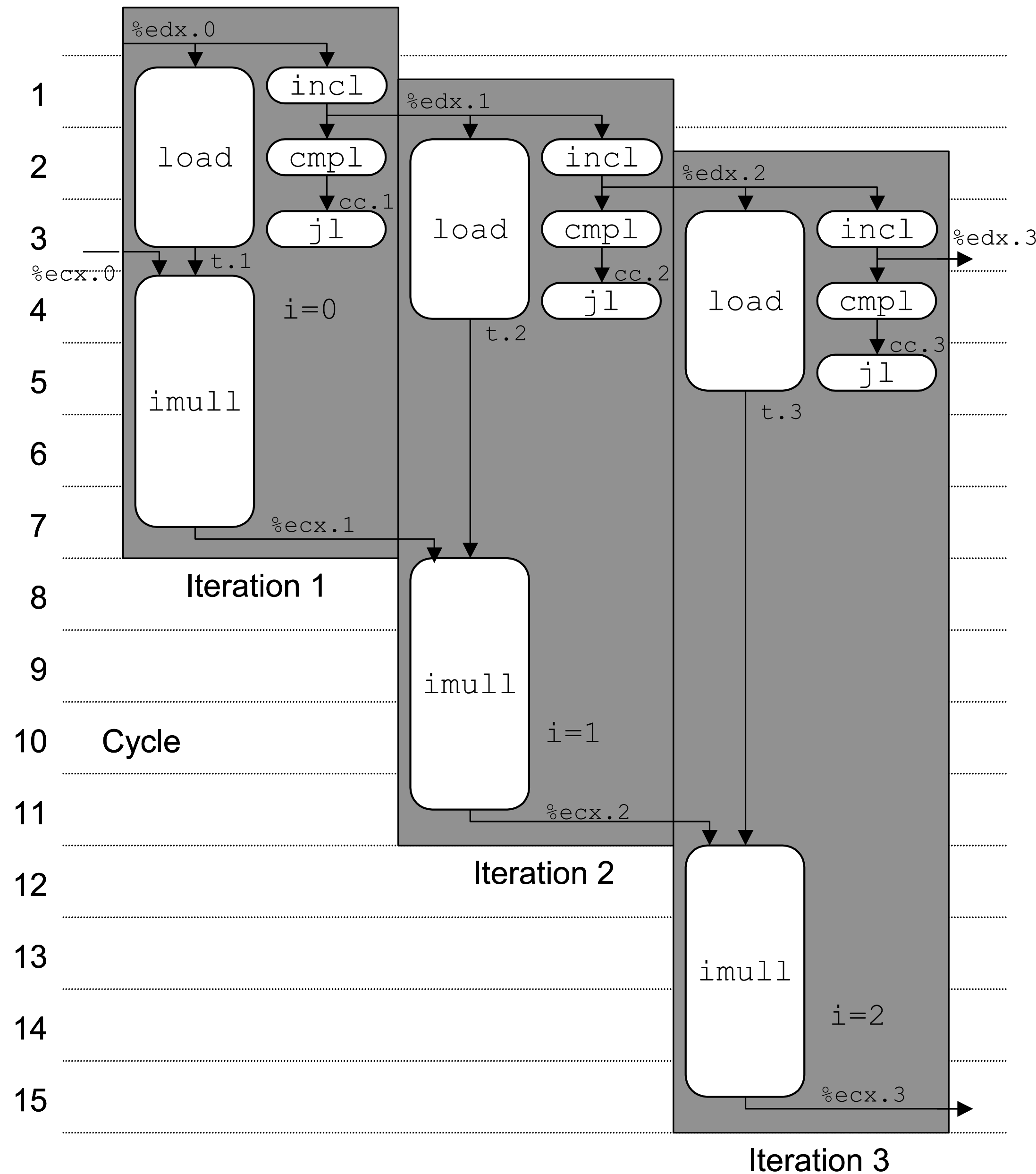
Machine dependent optimization techniques: loop unroll (4)

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?



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:

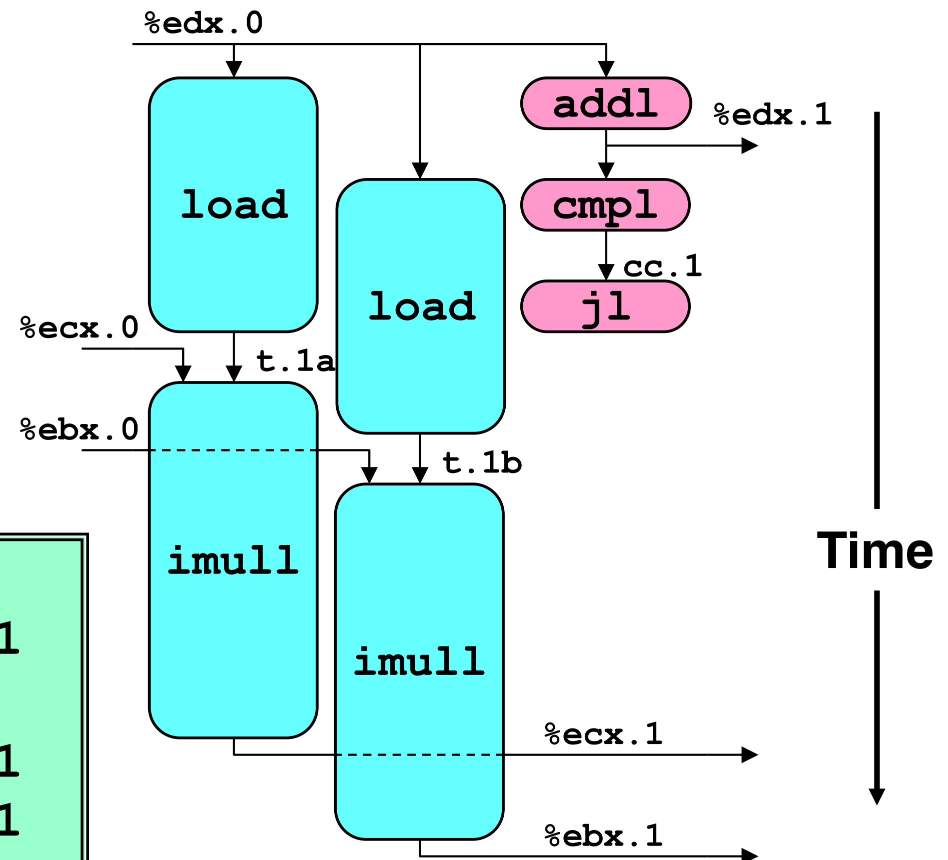
- accumulate in 2 different products
  - can be in parallel, if OP is associative!
- merge at the end

- 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)

Cycle: 1 2 3 4 5 6 7 8 9 10 11

Iteration 1 (i=0): %edx.0, load, load, addl, cmpl, cc.1, jl, %ecx.0, %ebx.0, t.1a, t.1b, imull, imull, %ecx.1, %ebx.1

Iteration 2 (i=2): %edx.1, load, load, addl, cmpl, cc.2, jl, t.2a, t.2b, imull, imull, %ecx.2, %ebx.2

Iteration 3 (i=4): %edx.2, load, load, addl, cmpl, cc.3, jl, %edx.3, t.3a, t.3b, imull, imull, %ecx.3, %ebx.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*