Master Informatics Eng.

2022/23

A.J.Proença

Optimizing sequential code

(revision: most slides from an undergrad course)

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

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

The following slides are from CAQA followed by a set adapted from a selection given in CS at UM.

The originals (of CS, in Portuguese) are in:

- http://gec.di.uminho.pt/mei/cp2122/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/cp2122/videos_sc.zip

Appendix C
Pipelining: Basic and Intermediate Concepts

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i*+1		IF	ID	EX	MEM	WB			
Instruction *i*+2			IF	ID	EX	MEM	WB		
Instruction *i*+3				IF	ID	EX	MEM	WB	
Instruction *i*+4					IF	ID	EX	MEM	WB

Figure C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

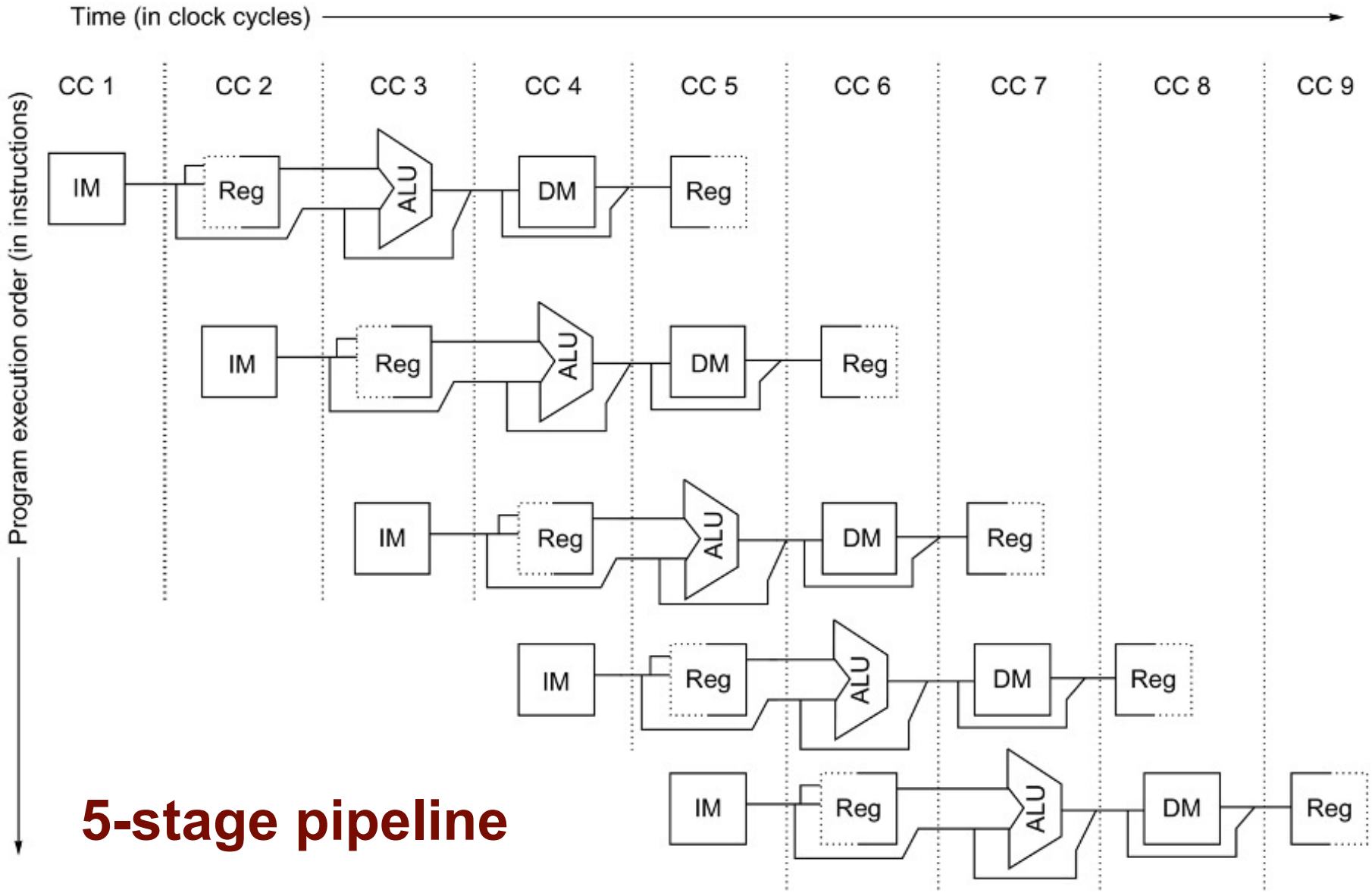
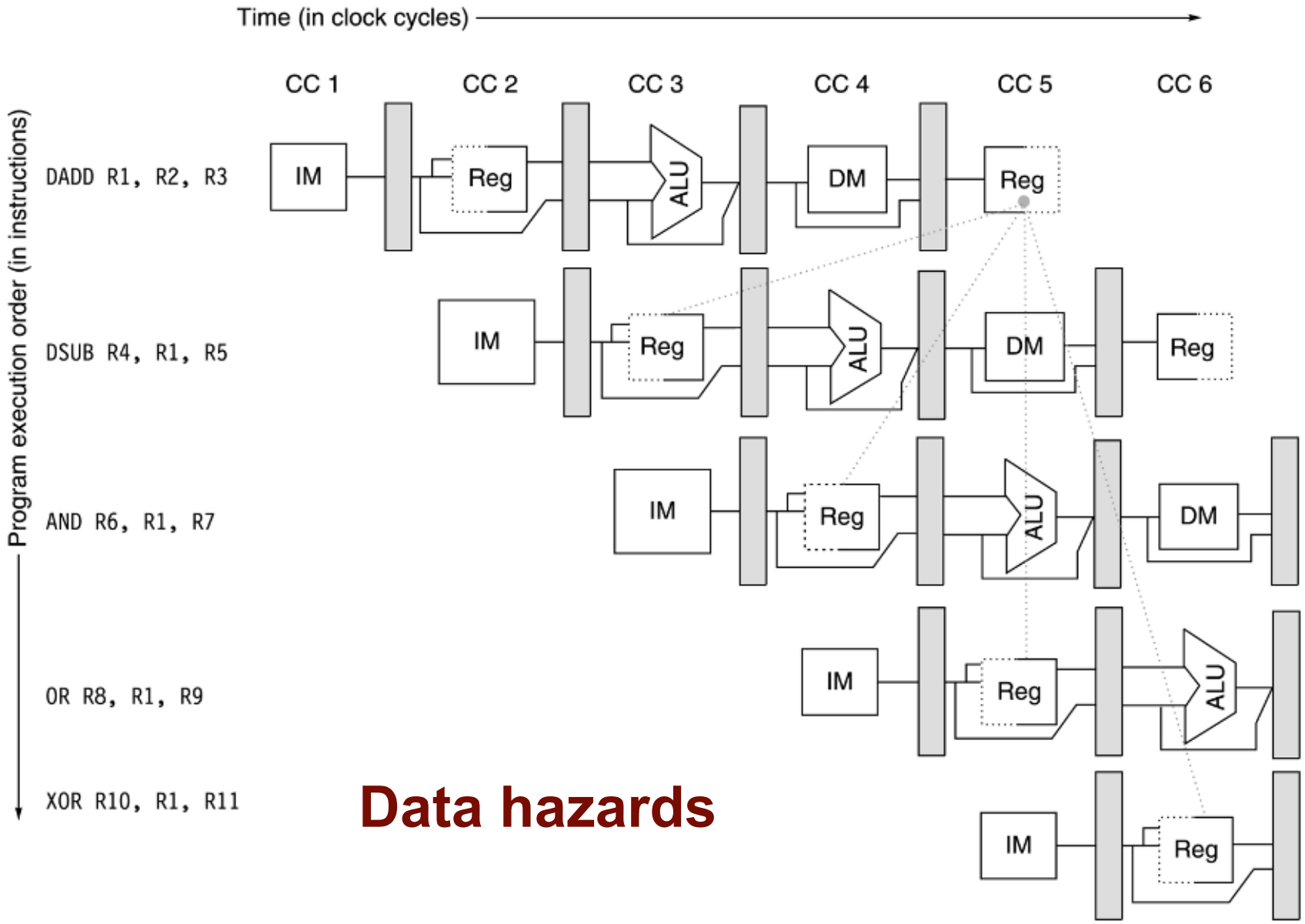
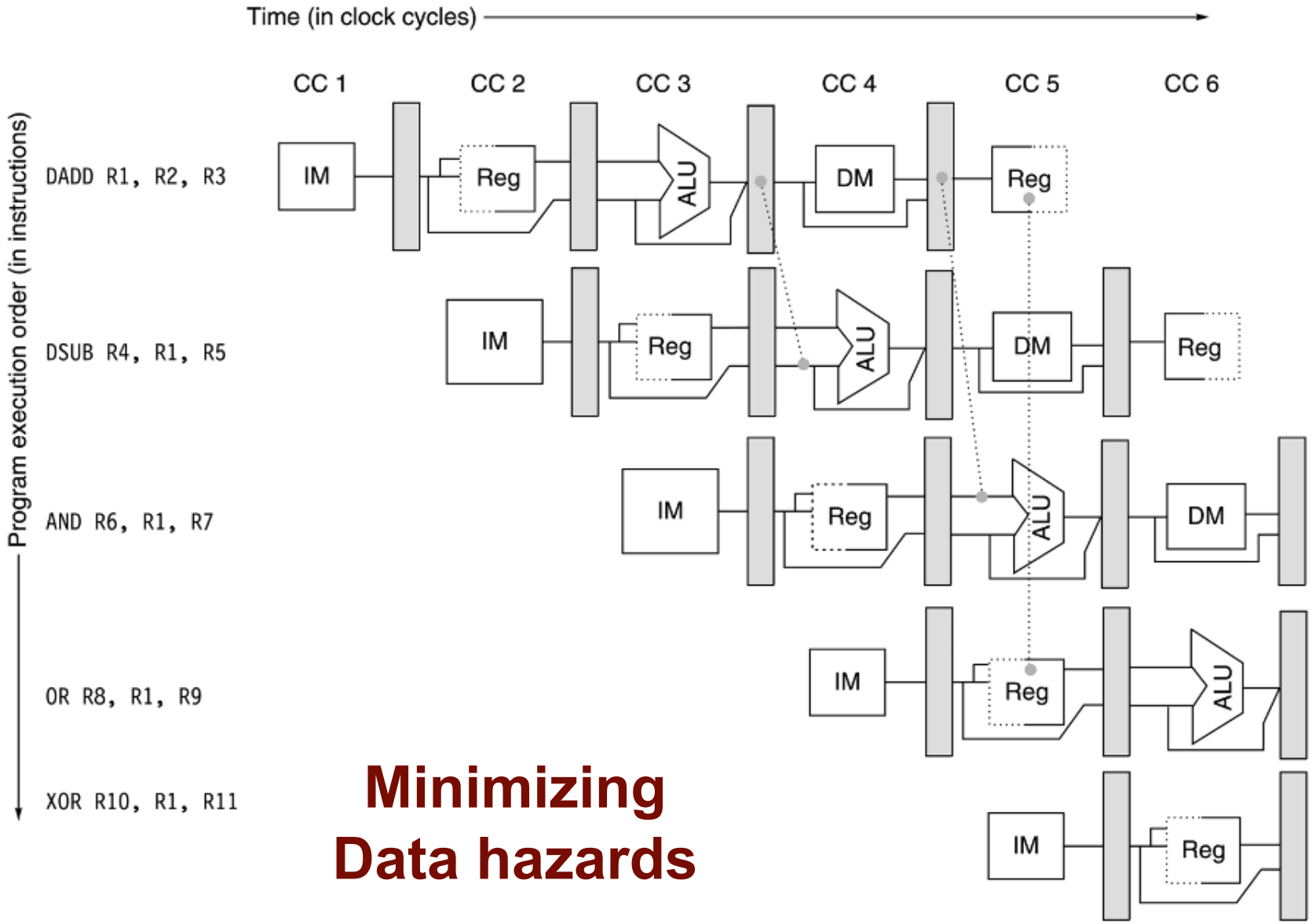


Figure C.2 The pipeline can be thought of as a series of data paths shifted in time.



Data hazards

Figure C.4 The use of the result of the `add` instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.



Minimizing Data hazards

Figure C.5 A set of instructions that depends on the `add` result uses forwarding paths to avoid the data hazard

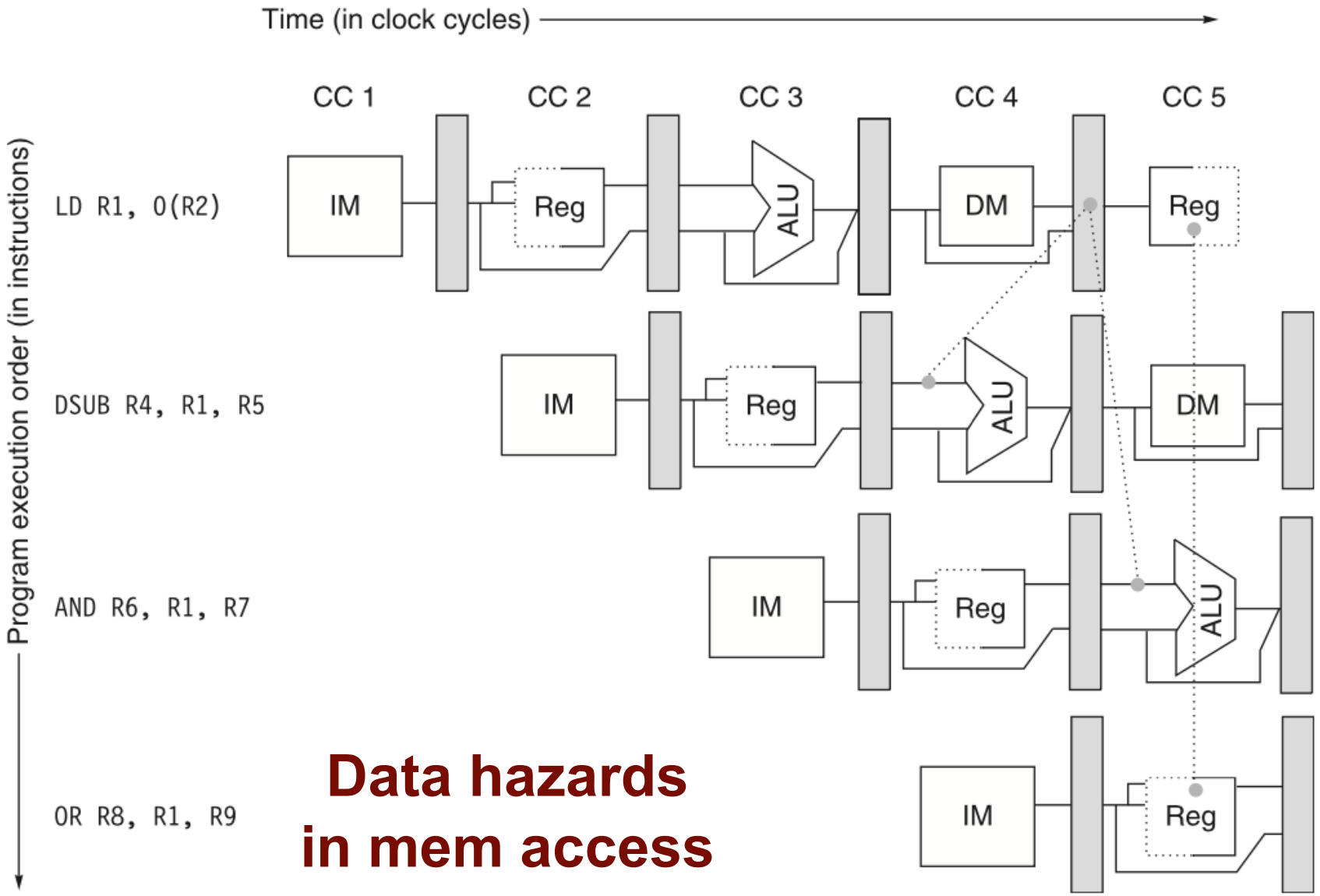


Figure C.7 The load instruction can bypass its results to the `and` and `or` instructions, but not to the `sub`, because that would mean forwarding the result in "negative time."

Minimizing control hazards *(1)* :

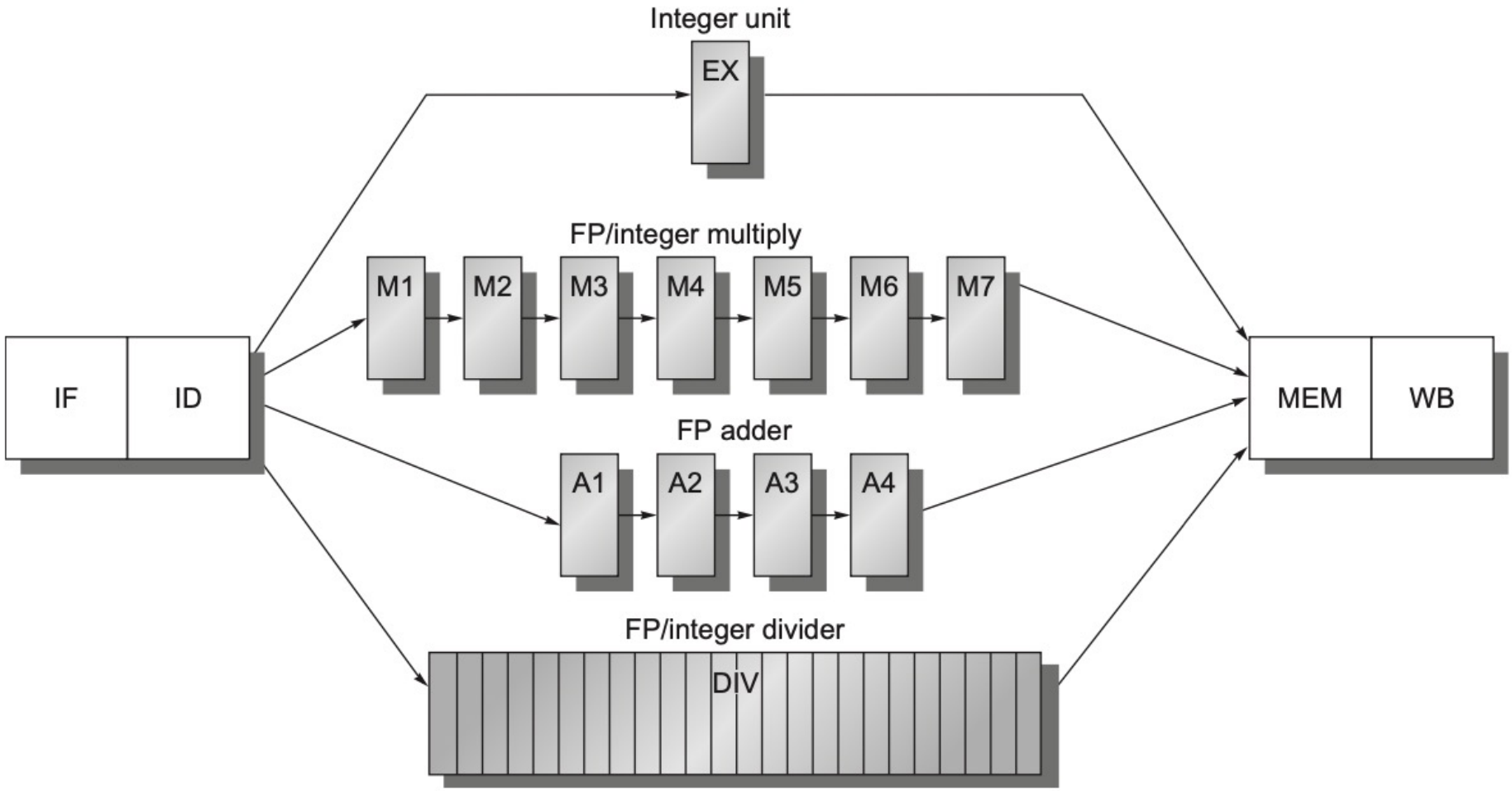
1. Add a branch delay slot
2. Predict not-taken / taken

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

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

3. 1- or 2-bit branch-prediction buffer
4. Especulation: follow both paths

An architecture with multiple pipelined execution units

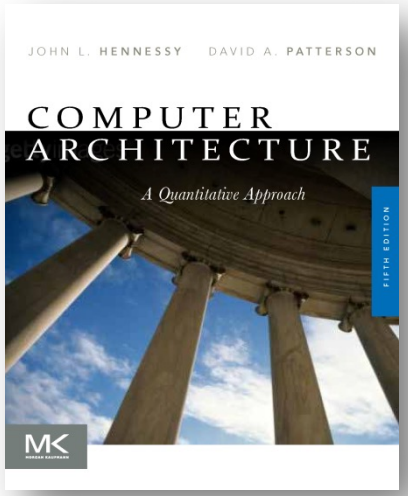


A typical FP code sequence with RAW hazards

	Clock cycle number																
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
`fld f4,0(x2)`	IF	ID	EX	MEM	WB												
`fmul.d f0,f4,f6`		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
`fadd.d f2,f0,f8`			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
`fsd f2,0(x2)`					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

A structural hazard: access to Reg in WB

	Clock cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
`...`		IF	ID	EX	MEM	WB					
`...`			IF	ID	EX	MEM	WB				
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	MEM	WB
`...`					IF	ID	EX	MEM	WB		
`...`						IF	ID	EX	MEM	WB	
`fld f2,0(x2)`							IF	ID	EX	MEM	WB



Chapter 3

Instruction-Level Parallelism and Its Exploitation

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls

$$CPU_{time} = \#Instr * CPI * Clk_{cycle}$$

- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)
- Data Hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)
- Control Dependence
  - Ordering of instruction `i` with respect to a branch instruction

Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
- Dynamic scheduling:
  - Rearrange order of instructions to reduce stalls while maintaining data flow
- Hardware-Based Speculation
  - Execute instructions along predicted execution paths but only commit the results if prediction was correct

Multiple Issue

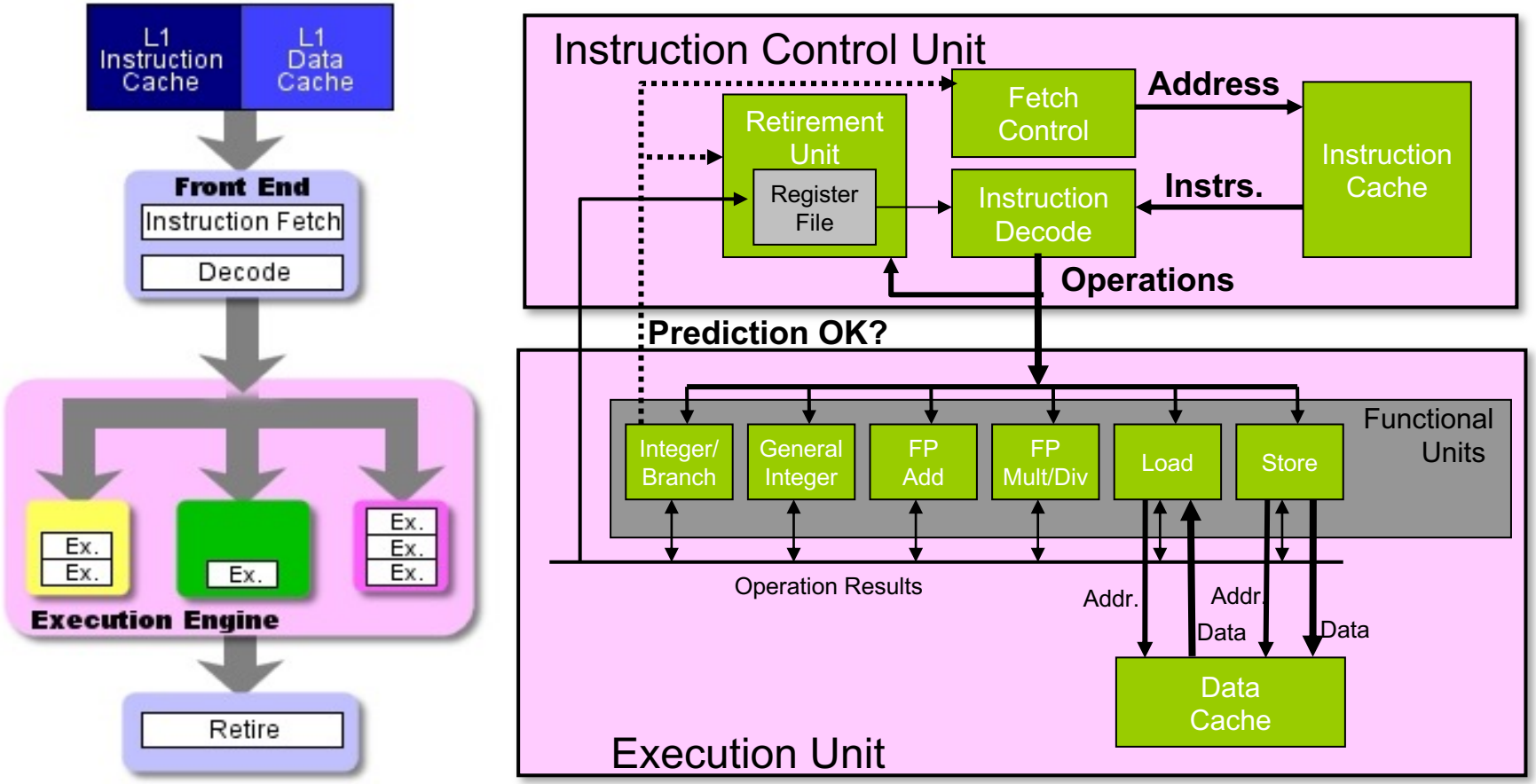
Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Case study:

- perform an arithmetic operation on all elements of a vector
- ILP techniques in the processor architecture:
  - instr-fetch & decode unit separate from operand-fetch & operation-execute & save-result ← **bottleneck!**
  - superscalar n-way and OoO execution
  - basic 2-bit branch prediction
  - latency of L1 cache: 1 clock cycle
  - no support for vector computing
- metrics to use: time to operate on each element, in clock cycles per element (CPE)
- improvement methodology: iterative, measure the new CPE for each applied technique

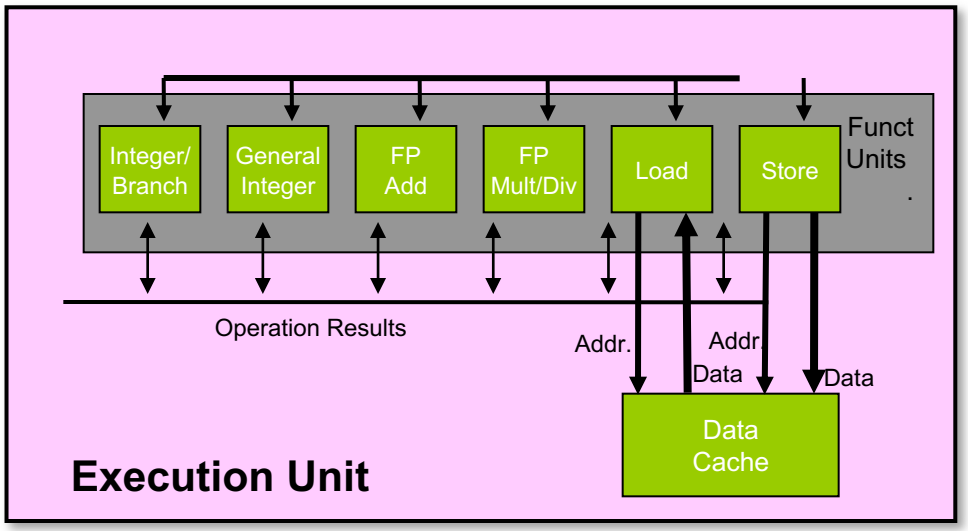
Internal architecture of Intel P6 processors

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

Code: a 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
```

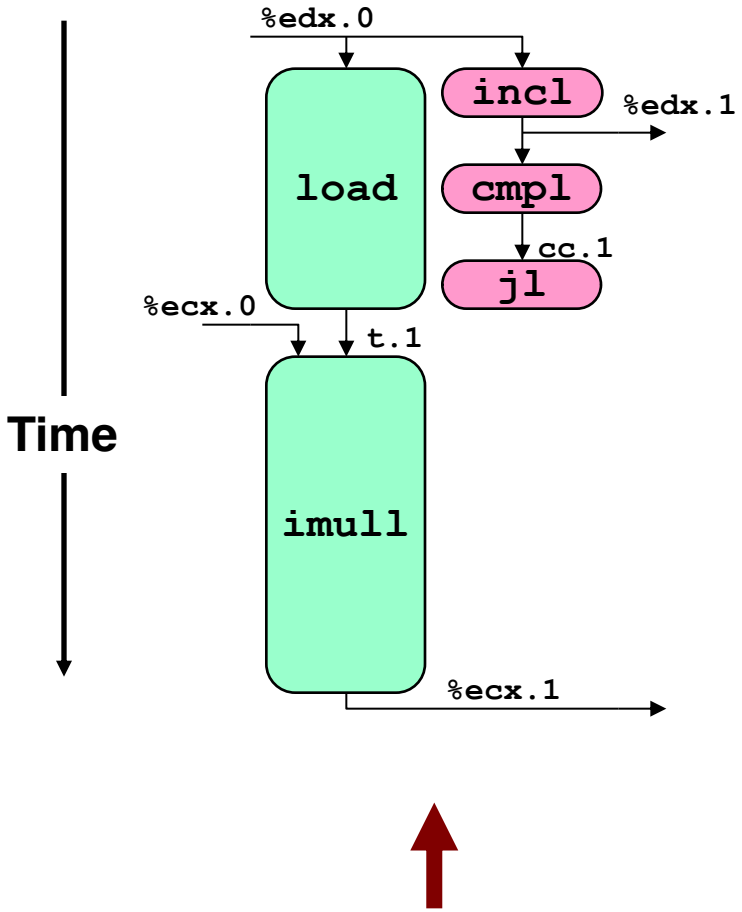
- **Translating 1st iteration**

```
.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`

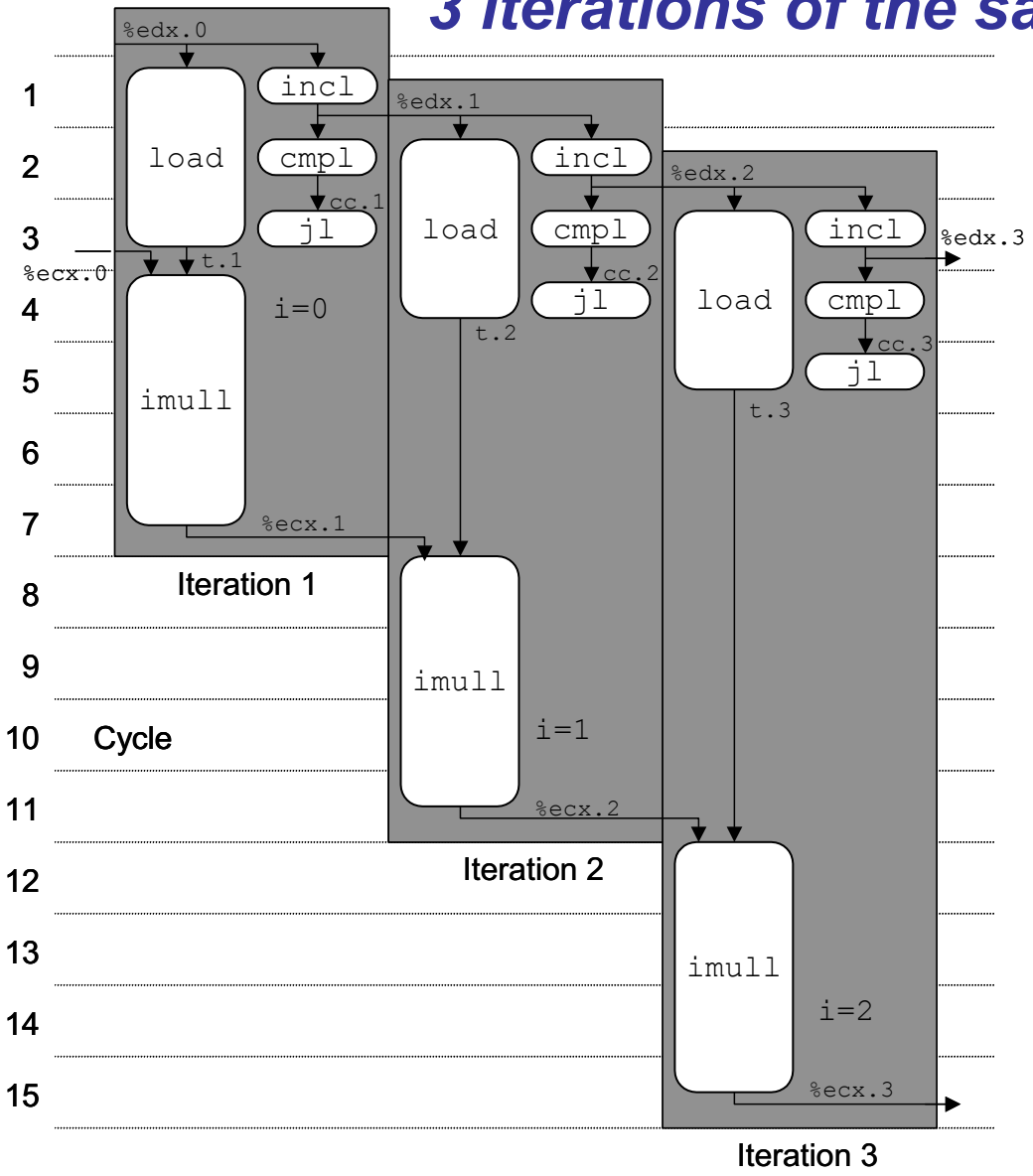



```
load   (%eax,%edx.0,4) ➔ t.1
imull  t.1, %ecx.0     ➔ %ecx.1
incl   %edx.0          ➔ %edx.1
cmpl   %esi, %edx.1    ➔ cc.1
jl     -taken cc.1
```

- **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*

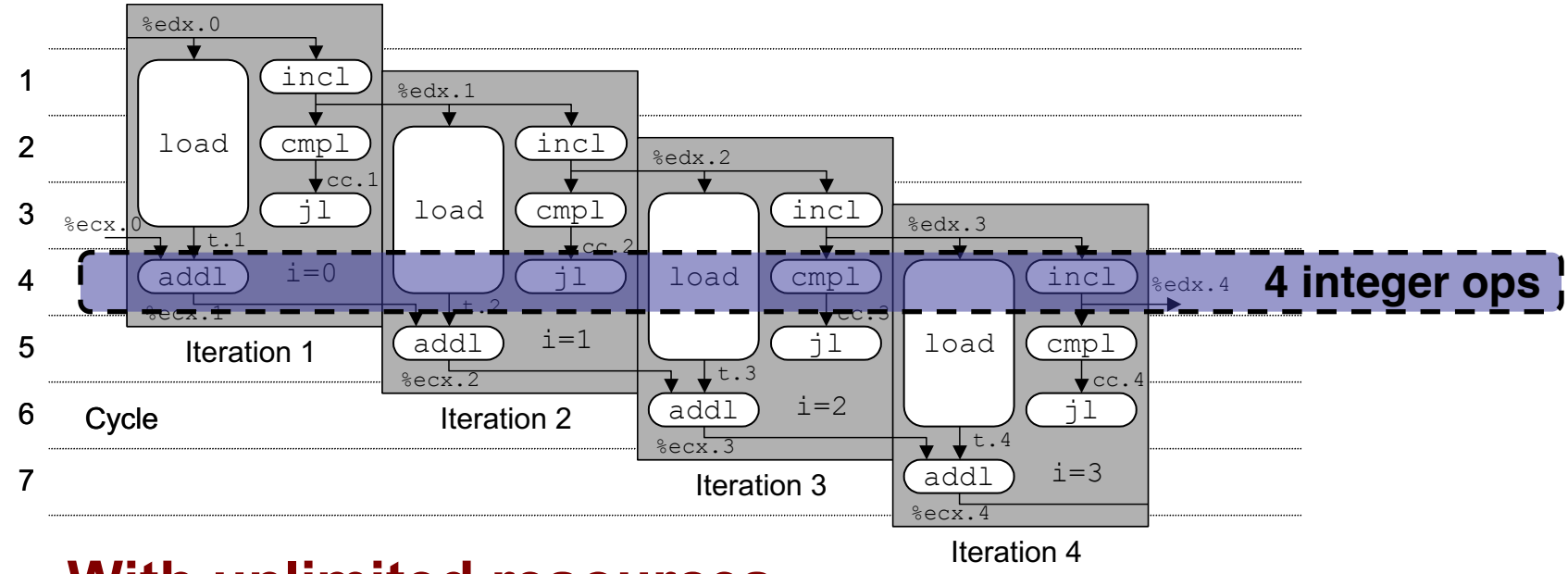
2-way superscalar w/ OoO execution

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



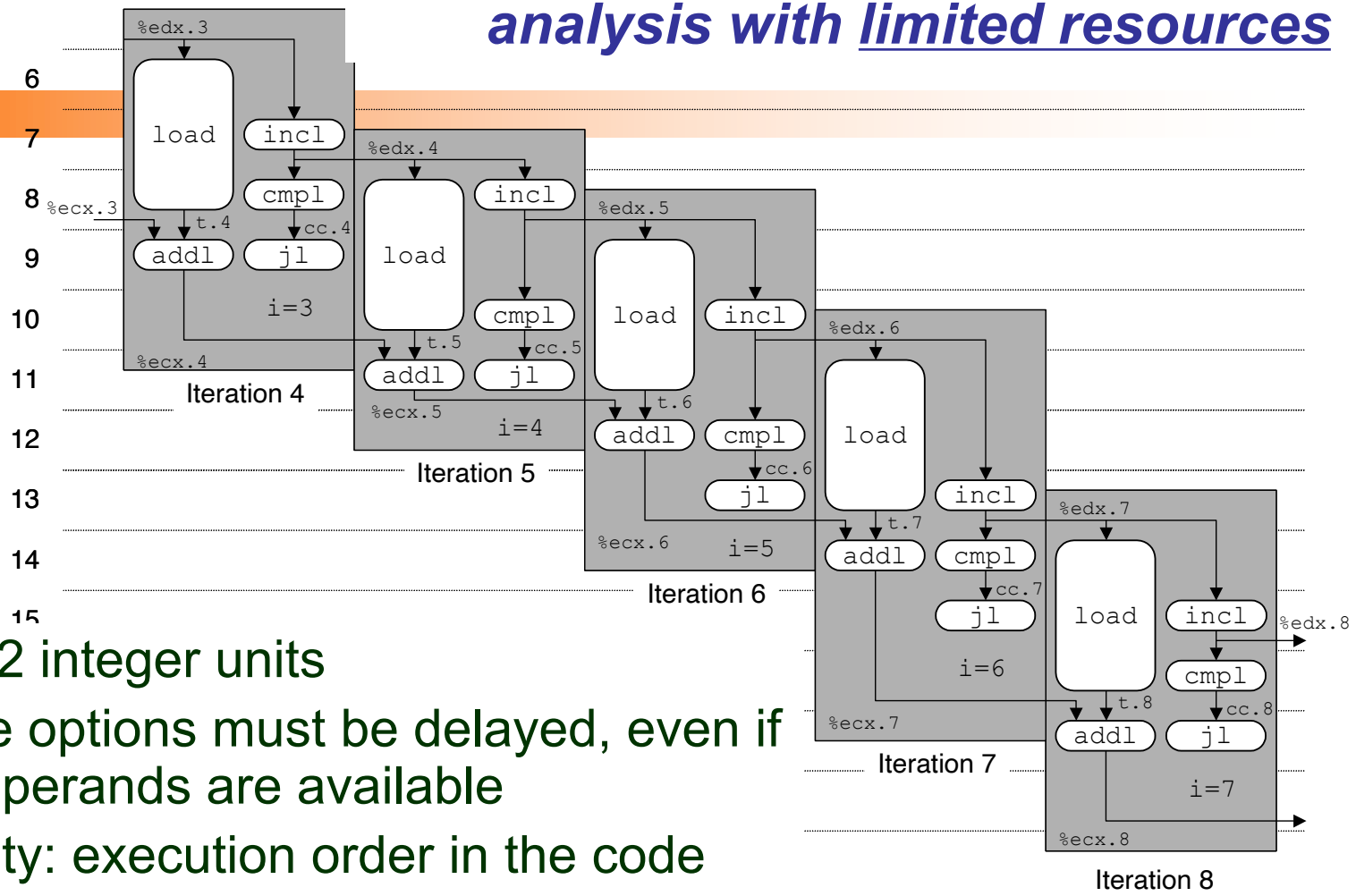
- **With unlimited resources**
  - parallel and pipelined execution of operations at the EU
  - out-of-order and speculative execution
- **Performance**
  - limitative factor: latency of integer multiplication
  - CPE: 4.0

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)

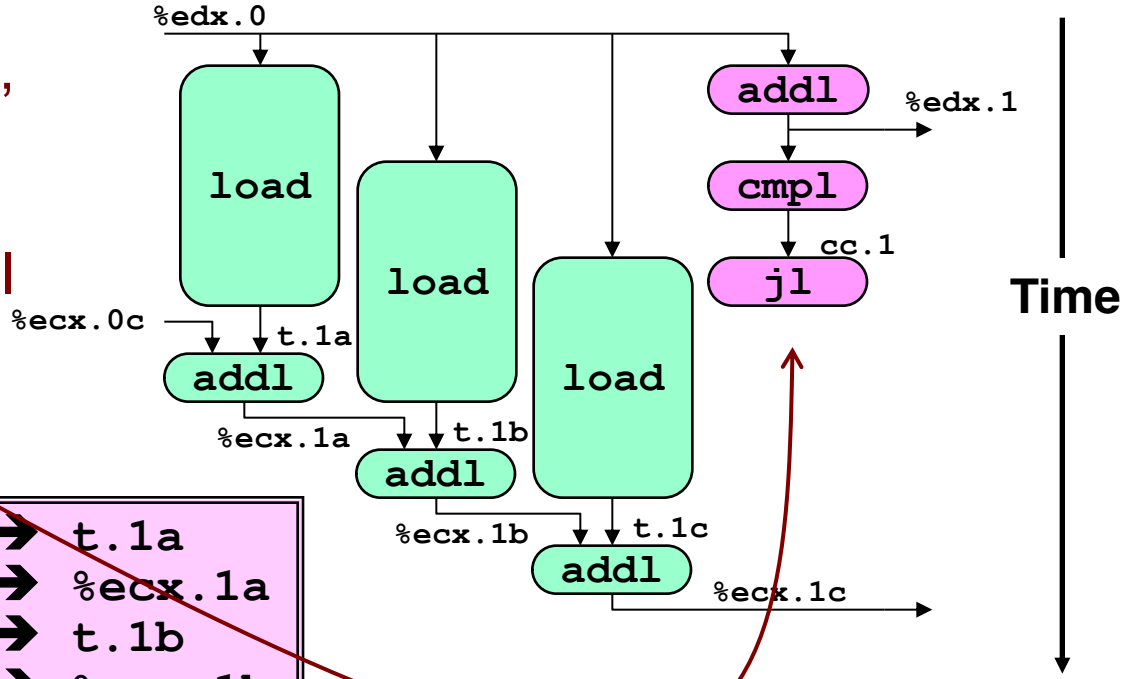
```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* junta 3 elem's no mesmo ciclo */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
         + data[i+2];
  }
  /* completa os restantes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

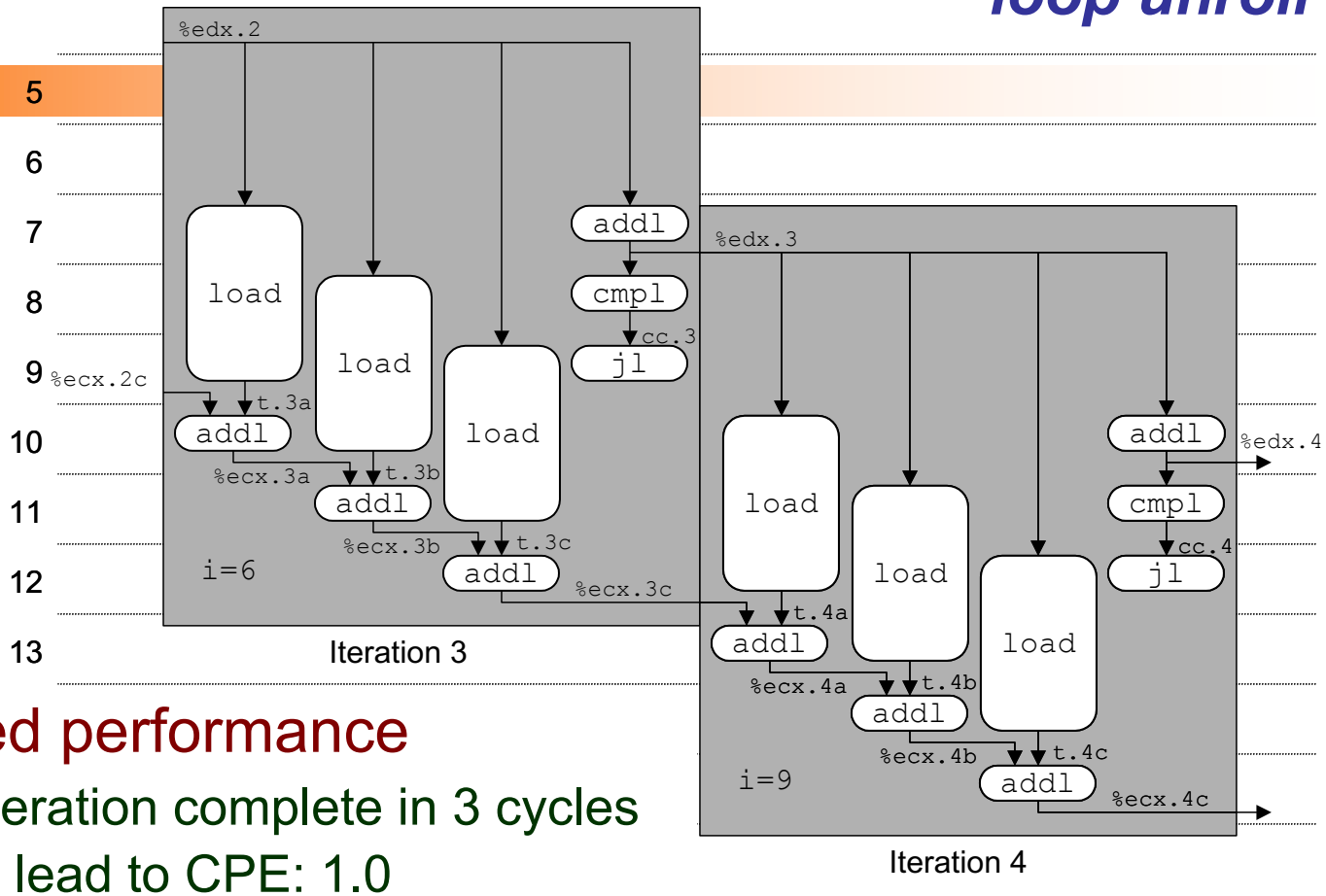
Machine dependent optimization techniques: loop unroll (2)

- **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

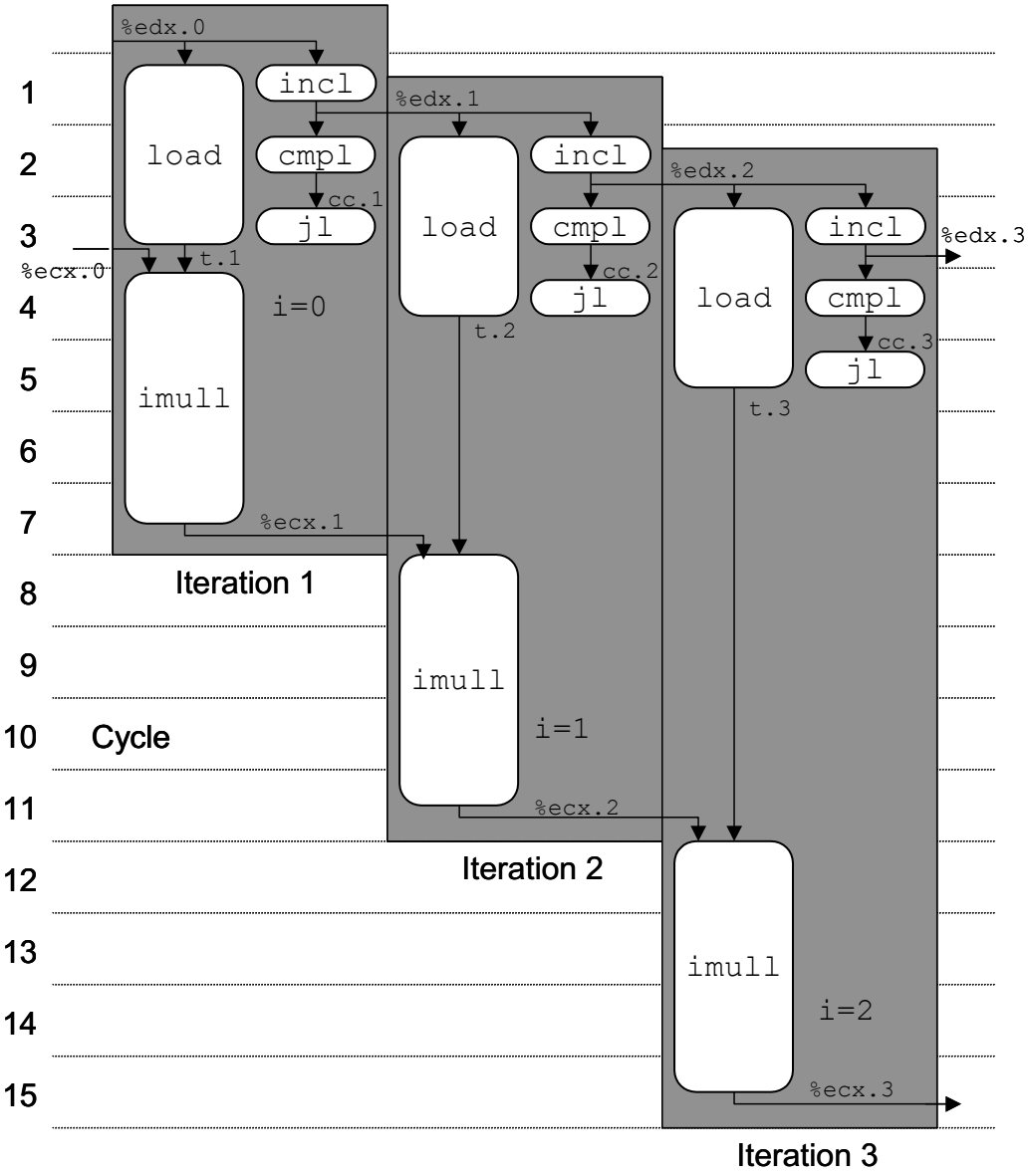
Machine dependent optimization techniques: loop unroll (4)

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;
}
```

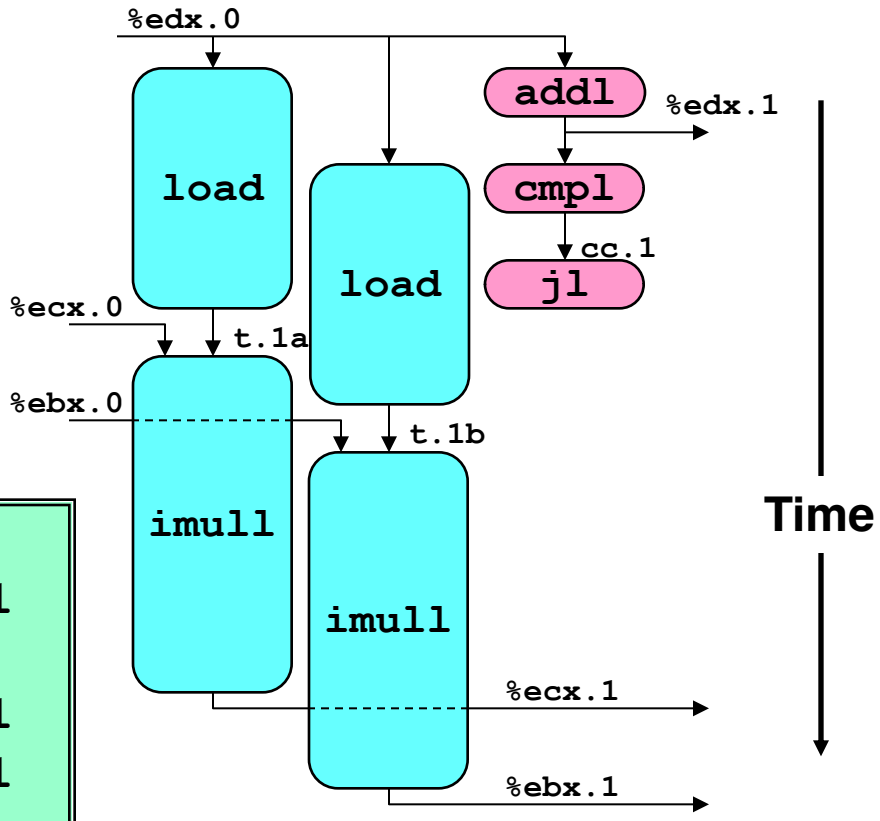
Optimization 5:

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

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

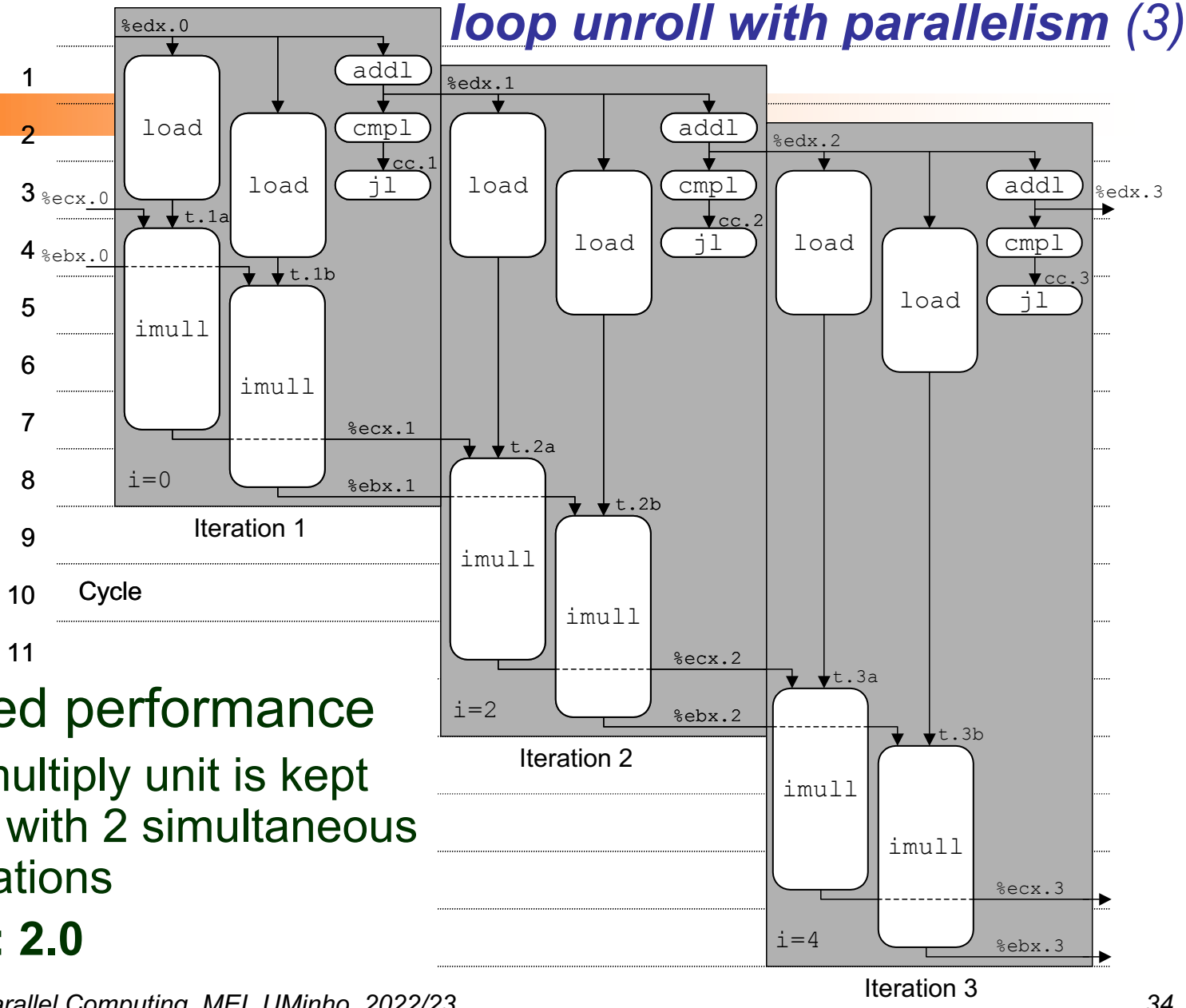
Machine dependent optimization techniques: loop unroll with parallelism (2)

- 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)

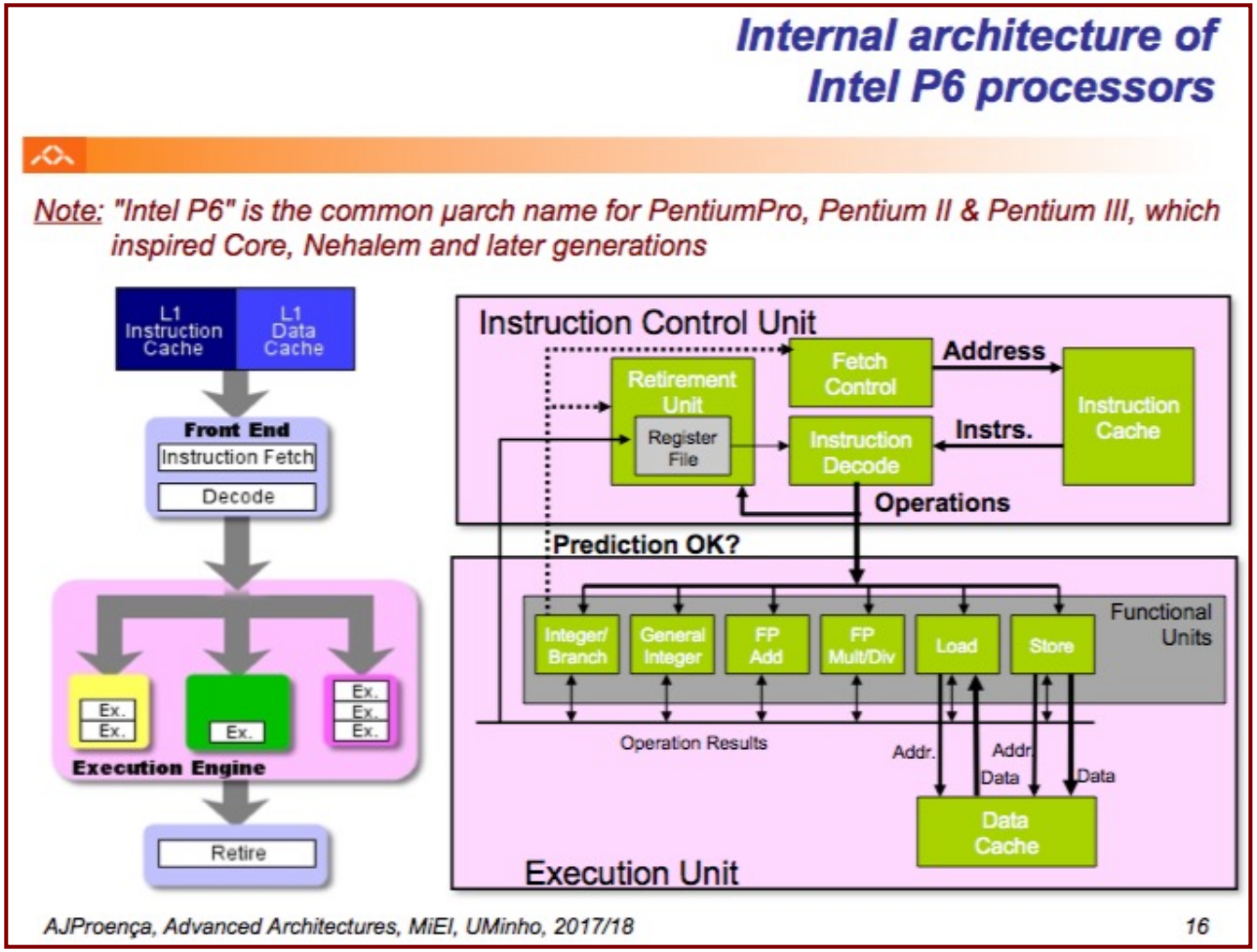


Estimated performance

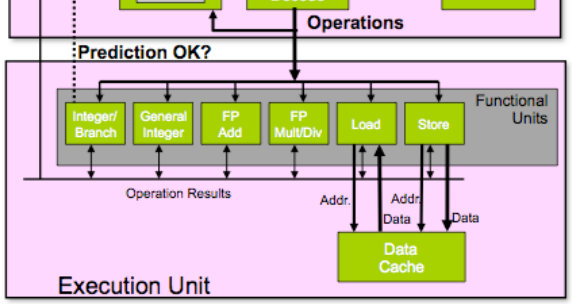
- the multiply unit is kept busy with 2 simultaneous operations
- **CPE: 2.0**

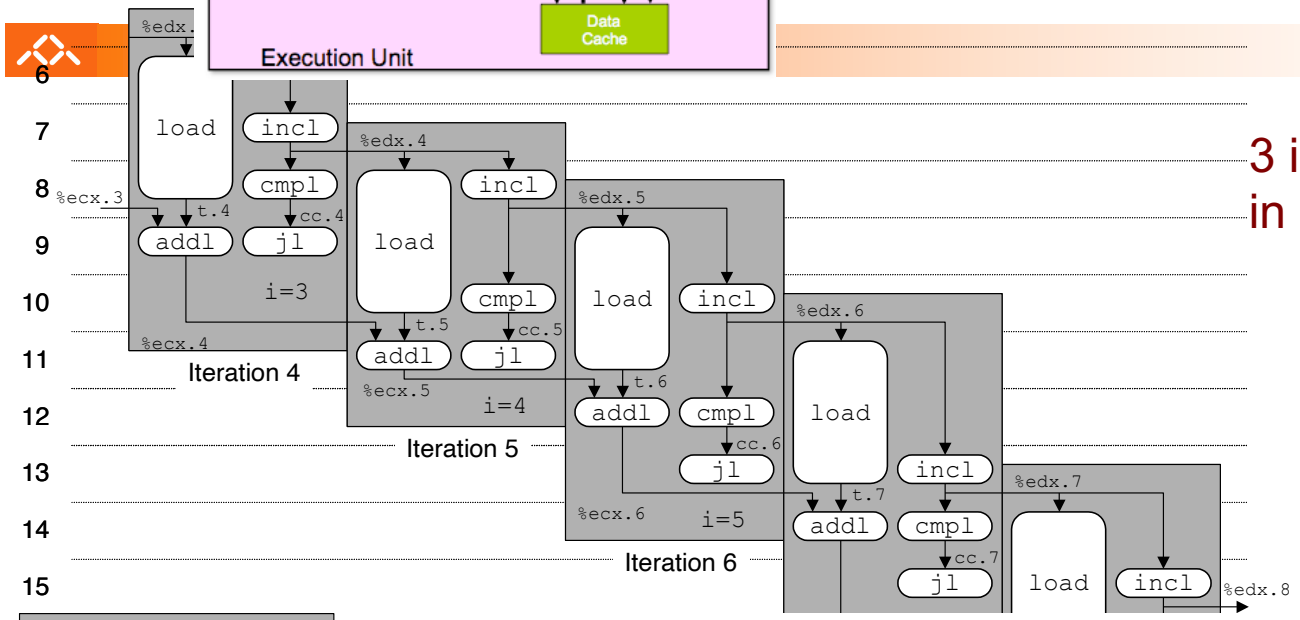
The n-way superscalar P6: how this architecture supports n-way multiple issue

As seen before...

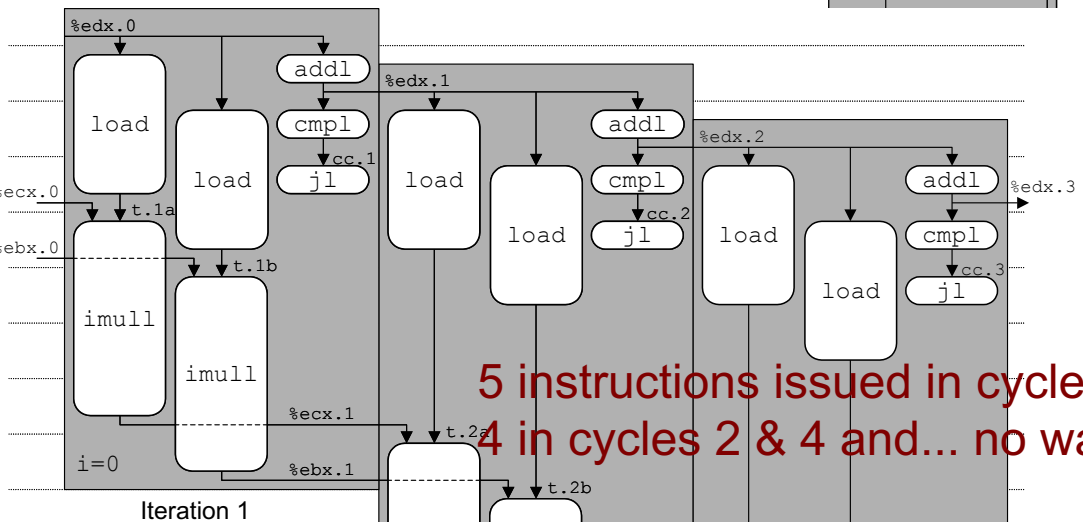


Executions views of `combine`: can this architecture be 3-way?



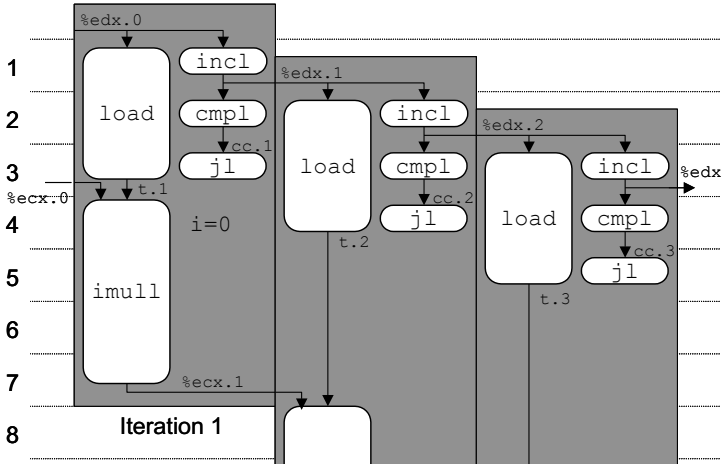


3 instructions issued in cycles 8, 9... ok!



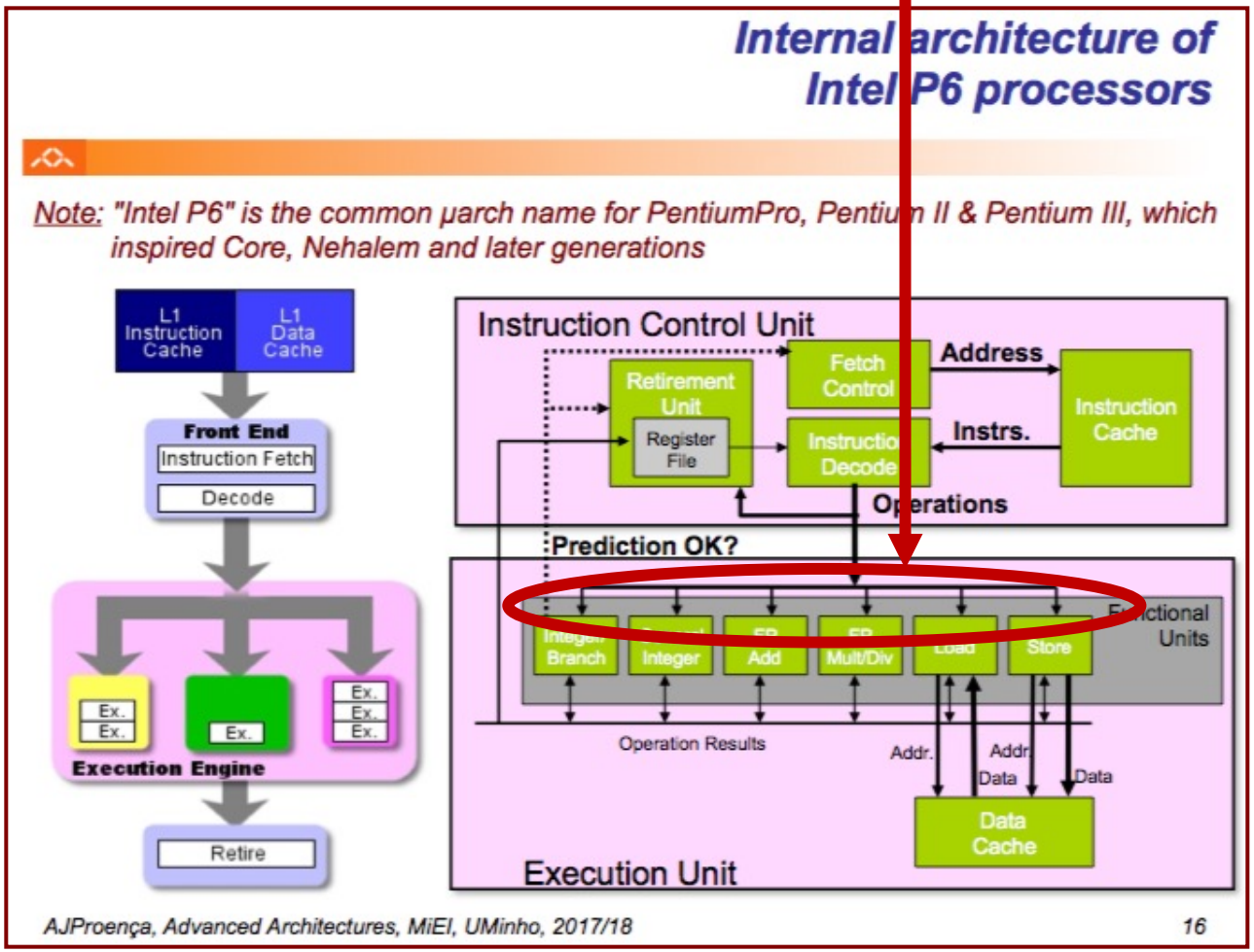
5 instructions issued in cycle 3, 4 in cycles 2 & 4 and... no way!

4 instructions issued in cycle 3... no way!



The n-way superscalar P6: how this architecture supports 6-way multiple issue

As seen before...



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*

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

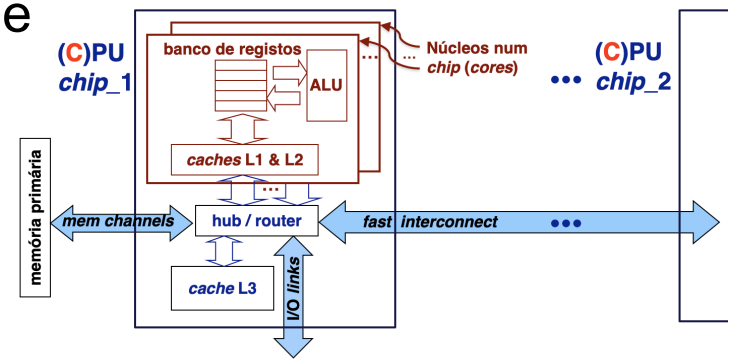
- Code generation
  - **Register allocation** - most frequently used variables are kept in processor registers
  - **Instruction selection** – selects 1 of several different ways to perform an operation
  - **Instruction scheduling** – avoid pipeline stalls
  - **Re-materialization** - recalculates a 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

???

Multicore architectures: homework T1...

Questions/homework T1:

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 number of cores?