Advanced Architectures

Master Informatics Eng.

2015/16

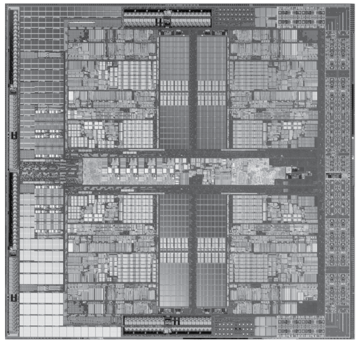
A.J.Proença

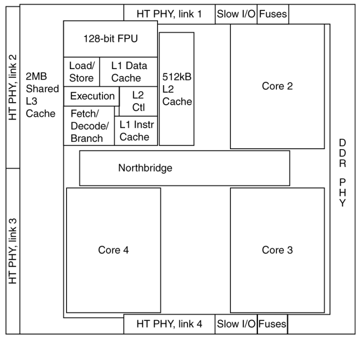
Concepts from undegrad Computer Systems (2)

(most slides are borrowed)

Inside the Processor

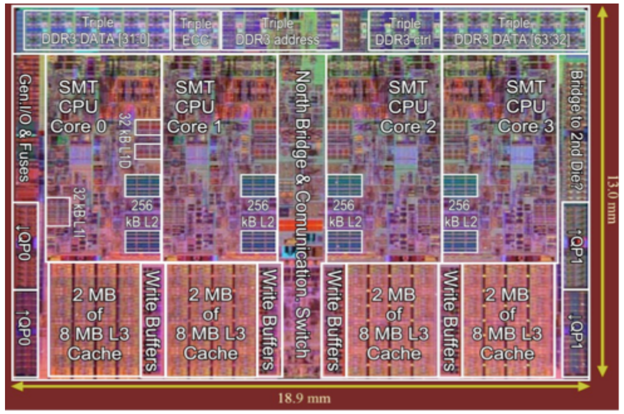
- AMD Barcelona: 4 processor cores





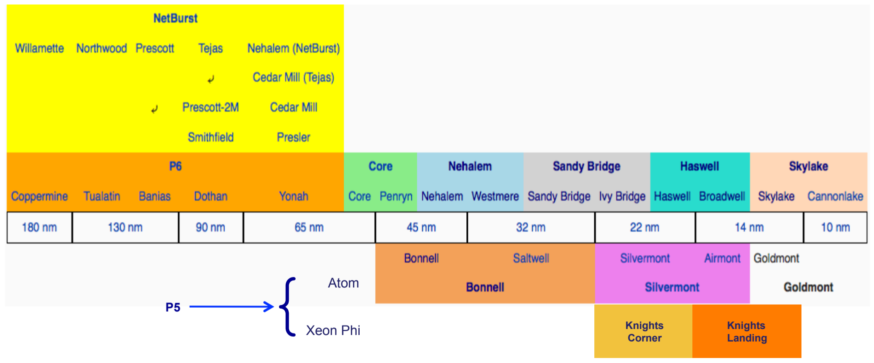
Multilevel On-Chip Caches

Intel Nehalem 4-core processor



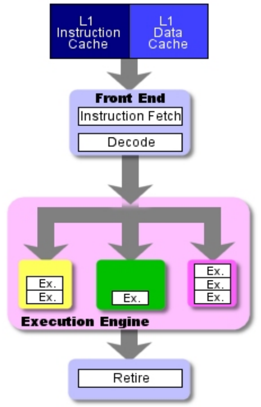
Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

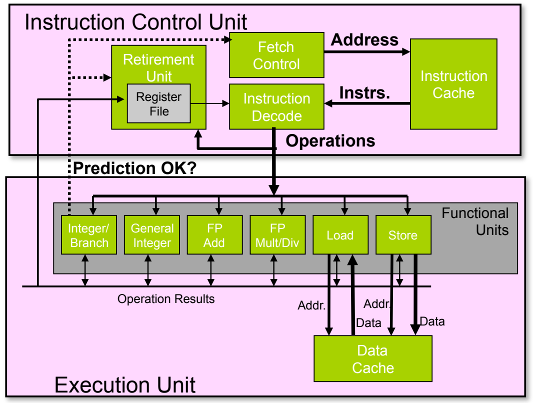
Internal x86 roadmap



Internal architecture of Intel P6 processors

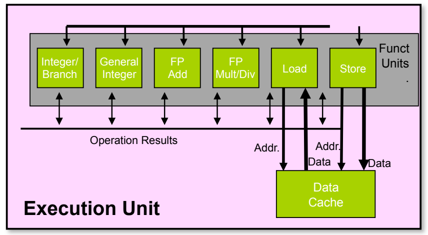
Note: "Intel P6" is the common μarch name for PentiumPro, Pentium II & Pentium III, which inspired Core, Nehalem and Phi





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
```

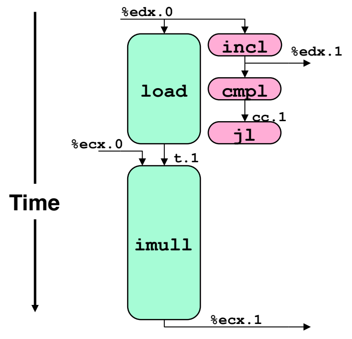
- **Translating 1st iteration**

```
.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
```

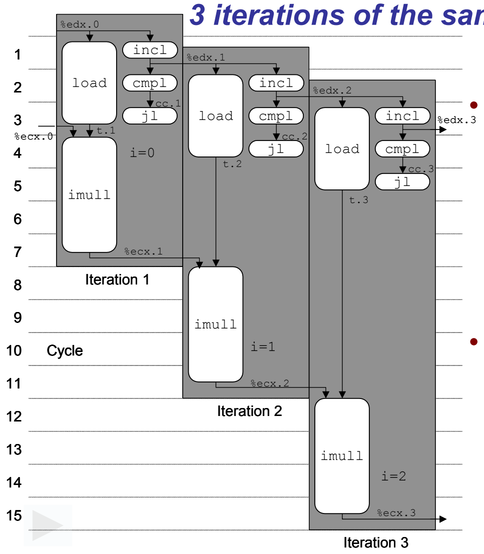
Visualizing instruction execution in P6:
1 iteration of the multiplication cycle on `combine`



```
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
```

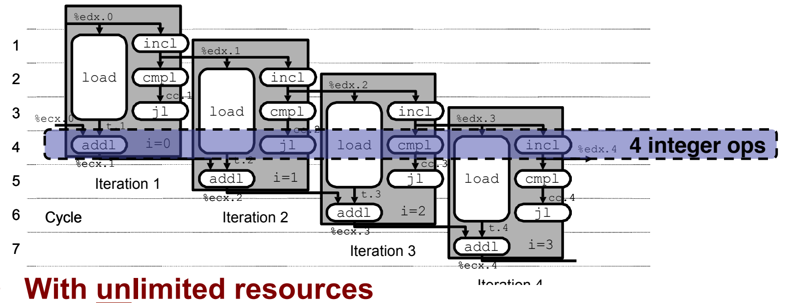
- **Operations**
  - vertical axis shows the time the instruction is executed
    - an operation cannot start with its operands
  - time length measures latency
- **Operands**
  - arcs are only showed for operands that are used in the context of the *execution unit*

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



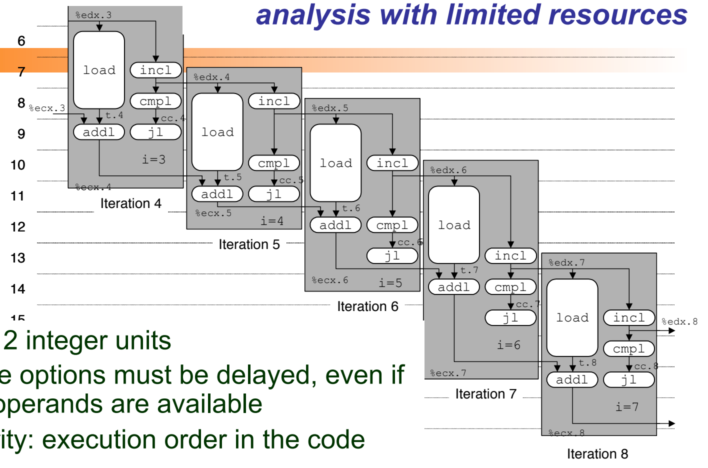
- **With unlimited resources**
  - parallel and pipelined execution of operations at the EU
  - out-of-order and speculative execution
- **Performance**
  - limitative factor: latency of integer multiplication
  - CPE: 4.0

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)*

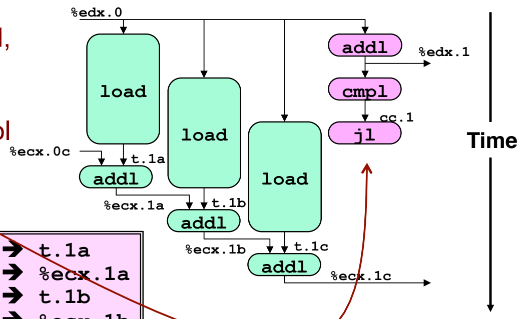
```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* junta 3 elem's no mesmo ciclo */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
               + data[i+2];
  }
  /* completa os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

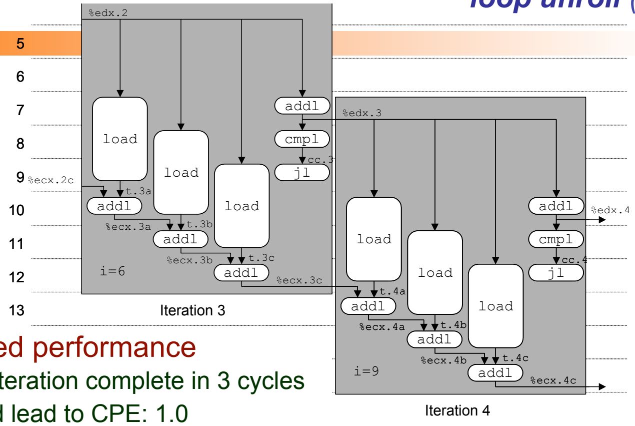
Machine dependent optimization techniques: loop unroll *(2)*

- **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

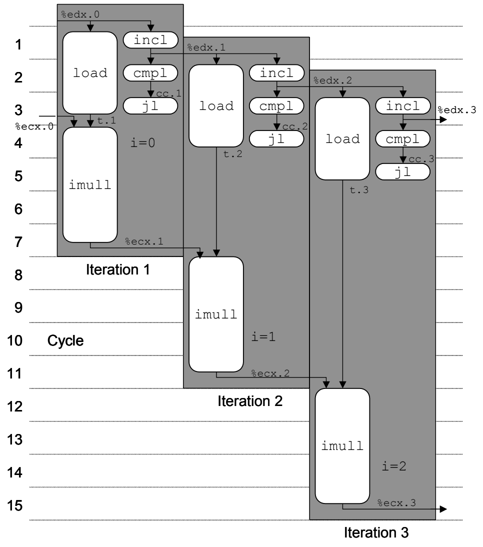
Machine dependent optimization techniques: loop unroll *(4)*

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)

```
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;
}
```

... *versus* parallel!

Optimization 5:

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

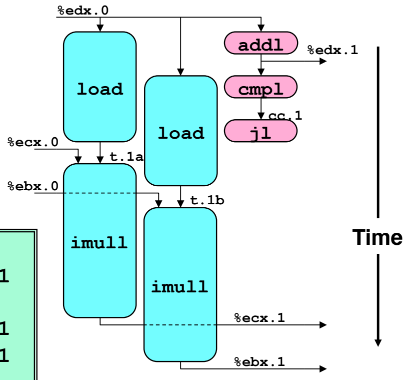
– Performance

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

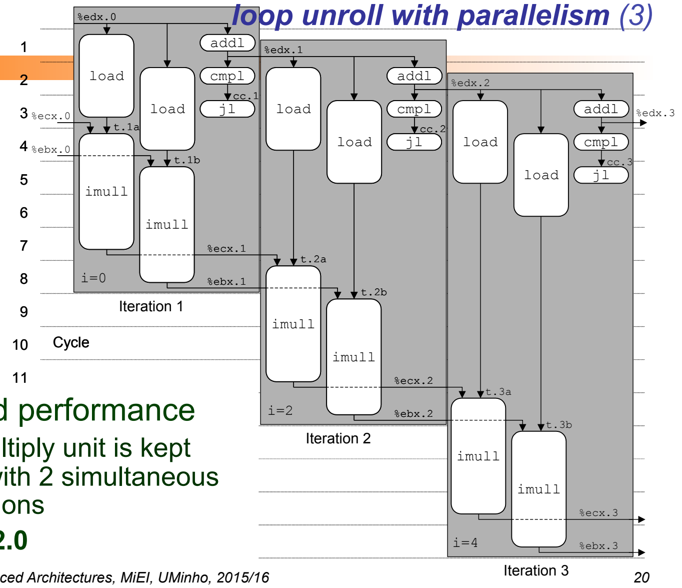
Machine dependent optimization techniques: loop unroll with parallelism (2)

- 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*

Code optimization: ILP limitations

- It requires a lot of registers!
  - to save results from add/multip
  - only 6 integer registers in IA32
    - also used as pointers, loop control, …
  - 8 fp registers
  - when registers aren't enough, temp's are pushed to the stack
    - cuts performance gains (see assembly in integer product with 8x unroll & 8x parallelism)
  - re-naming registers is not enough
    - it is not possible to reference more operands than those at the instruction set
    - … main drawback at the IA32 instruction set
- Operations to parallelize must be associative!
  - fp add & multipl in a computer is not associative!
    - (3.14+1e20)-1e20 not always the same as 3.14+(1e20-1e20)…

Limitation of parallelism: not enough registers

- **`combine`**
  - integer multiplication
  - 8x unroll & 8x parallelism
  - 7 local variables share 1 register (`%edi`)
    - note the stack accesses
    - performance improvement is compromised...
    - consequence: register spilling

```
.L165:
        imull  (%eax),%ecx
        movl  -4(%ebp),%edi
        imull  4(%eax),%edi
        movl  %edi,-4(%ebp)
        movl  -8(%ebp),%edi
        imull  8(%eax),%edi
        movl  %edi,-8(%ebp)
        movl  -12(%ebp),%edi
        imull  12(%eax),%edi
        movl  %edi,-12(%ebp)
        movl  -16(%ebp),%edi
        imull  16(%eax),%edi
        movl  %edi,-16(%ebp)
      …
        addl  $32,%eax
        addl  $8,%edx
        cmpl  -32(%ebp),%edx
        jl  .L165
```

Last week homework

The problem:

- To identify all Intel Xeon processor microarchitectures from Hammer and Core till the latest releases, and build a table with:
  - # pipeline stages, # simultaneous threads, degree of superscalarity, vector support , # cores, type/speed of interconnectors,…
- To identify the Xeon generations at the SeARCH cluster

Expected table headings:

µArch	Name	Year	Pipeline Stages	Issue Width	Vector Support	Cores	Interface

Example of a CPU generation at the cluster:

Core/Clovertown E5345, 65nm, 2.33GHz, 14 pipe stages, w/o HT, 4c, 4 issues wide, 128-bit SIMD, 4 cores, ... (32KB-Intrs + 32KB Data)L1, 2x4MB L2)