Master Informatics Eng.

2018/19

A.J.Proença

Instruction-Level Parallelism & Data Parallelism

(some slides are borrowed, mod's in green)

The BIG Picture

$$\text{PU Time} = \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{IC}} \times \underbrace{\frac{\text{Clock cycles}}{\text{Instruction}}}_{\text{CPI}} \times \underbrace{\frac{\text{Seconds}}{\text{Clock cycle}}}_{T_c}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c
  - Processor design: ILP, vectorization, mem-hierar, ...

Key concepts to revise:

- *numerical data representation (for error analysis)*
- *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, …***
  - ***thread-level parallelism***
  - ***data parallelism: SIMD/vector processing, ...***
  - ***memory hierarchy: cache levels, ...***

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

Processor arch: beyond Instruction-Level Parallelism

- When exploiting ILP, goal is to minimize CPI
  - Pipeline CPI (*efficient to exploit loop-level parallelism*) =>
    - Ideal pipeline CPI + ✔
    - Structural stalls + ✔
    - Data hazard stalls + ✔
    - Control stalls + ✔
    - Memory stalls ... *cache techniques ...*
  - Multiple issue =>
    - find enough parallelism to keep pipeline(s) occupied

Does Multiple Issue Work?

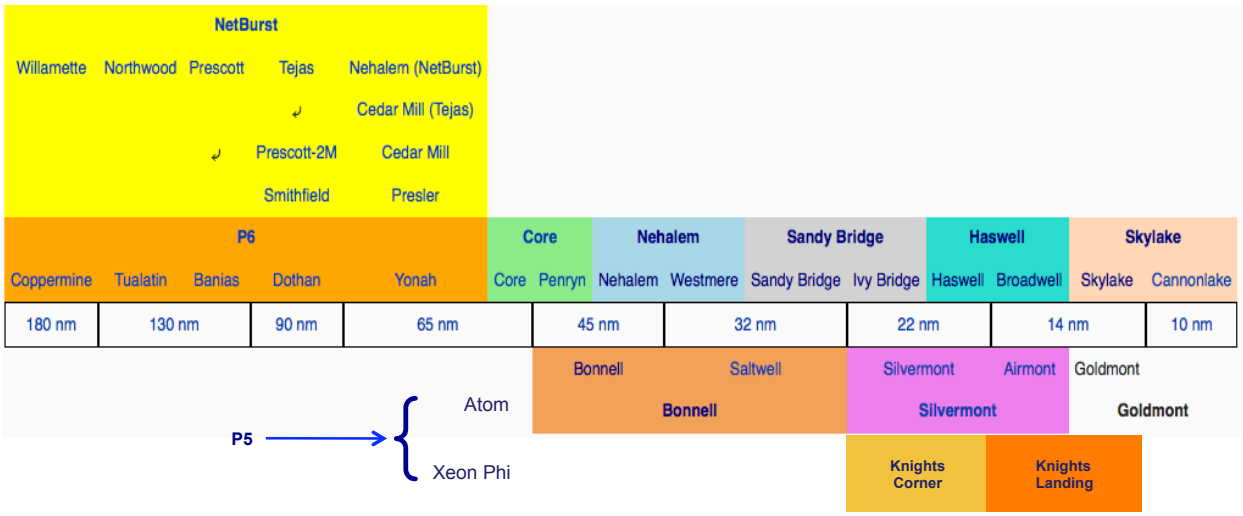
The BIG Picture

- 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

Multiple Issue and Static Scheduling

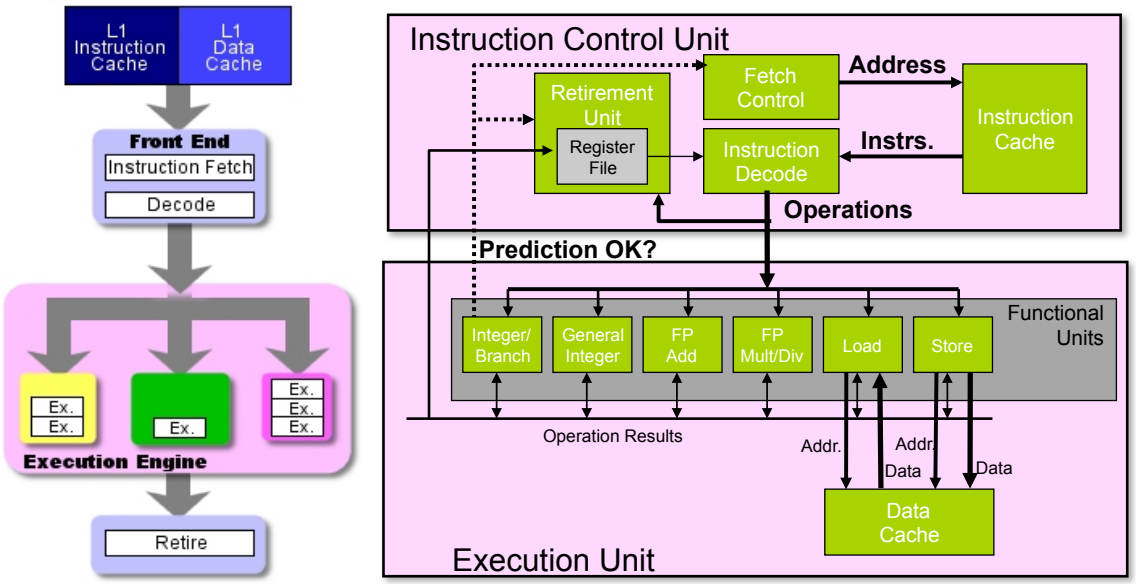
- To achieve CPI < 1, need to complete multiple instructions per clock cycle
- Solutions:
  - statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors

Internal x86 roadmap



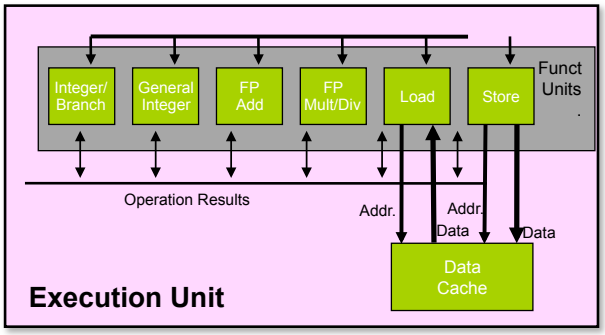
Internal architecture of Intel P6 processors

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

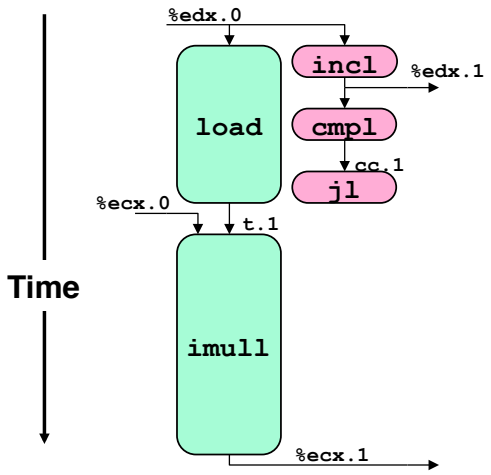
- **Translating 1st iteration**

```
.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
```

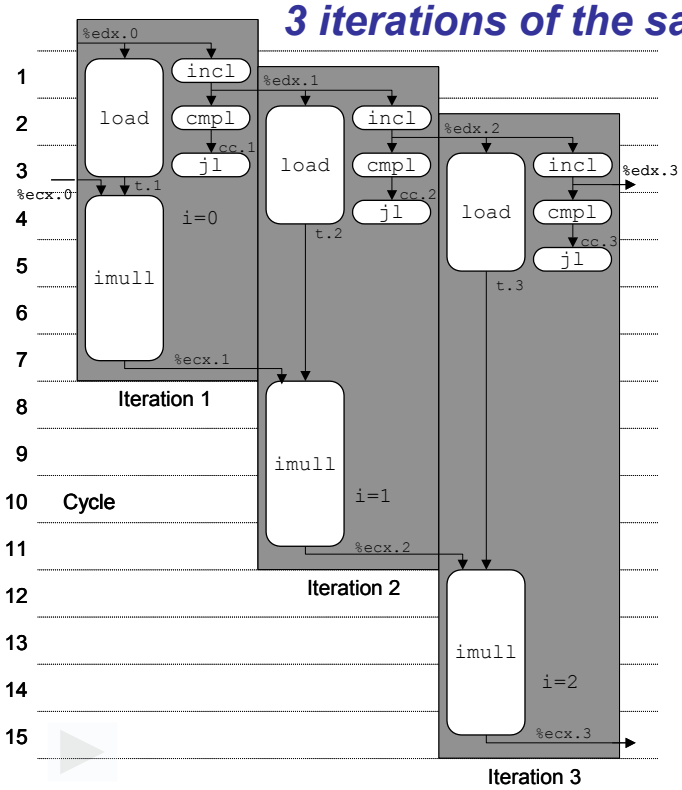
Visualizing instruction execution in P6: 1 iteration of the multiplication cycle on `combine`




```
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
```

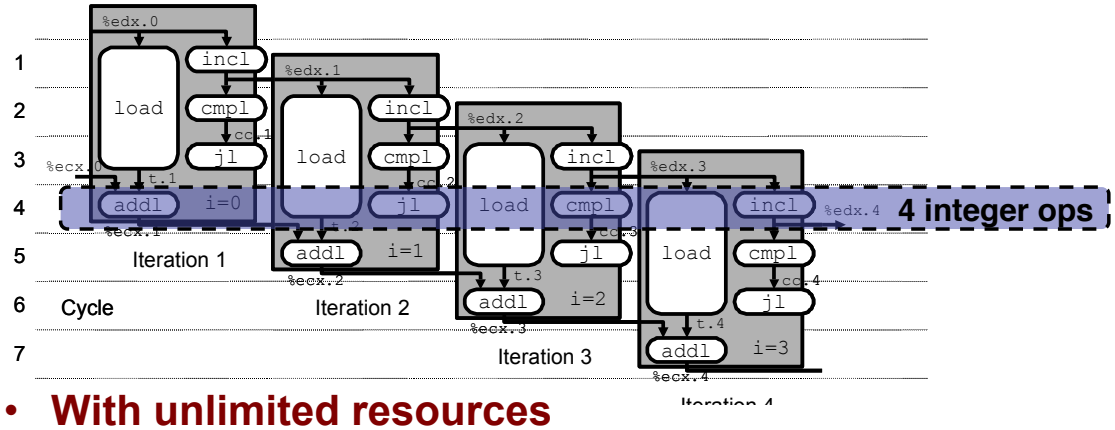
- **Operations**
  - vertical axis shows the time the instruction is executed
    - an operation cannot start with its operands
  - time length measures latency
- **Operands**
  - arcs are only showed for operands that are used in the context of the *execution unit*

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



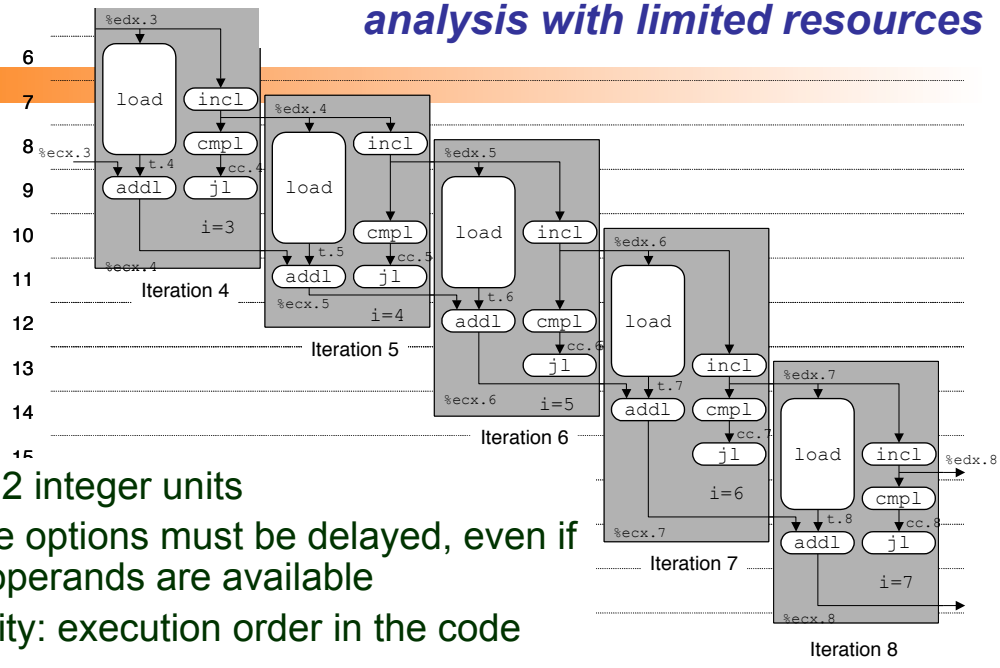
- **With unlimited resources**
  - parallel and pipelined execution of operations at the EU
  - out-of-order and speculative execution
- **Performance**
  - limitative factor: latency of integer multiplication
  - CPE: 4.0

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)

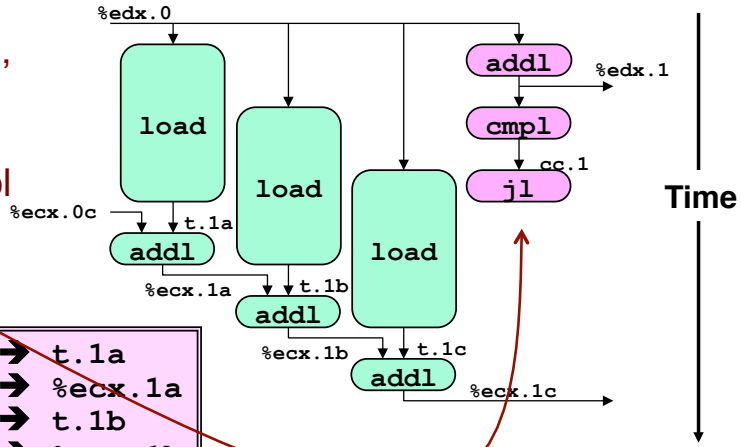
```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* junta 3 elem's no mesmo ciclo */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
           + data[i+2];
  }
  /* completa os restantes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

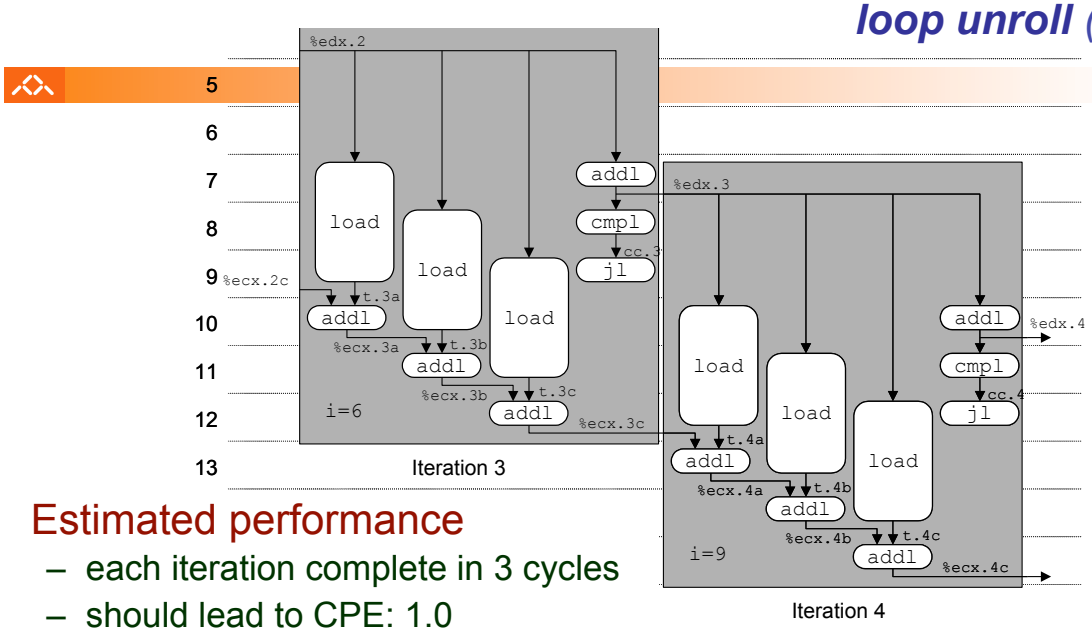
Machine dependent optimization techniques: loop unroll (2)

- **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

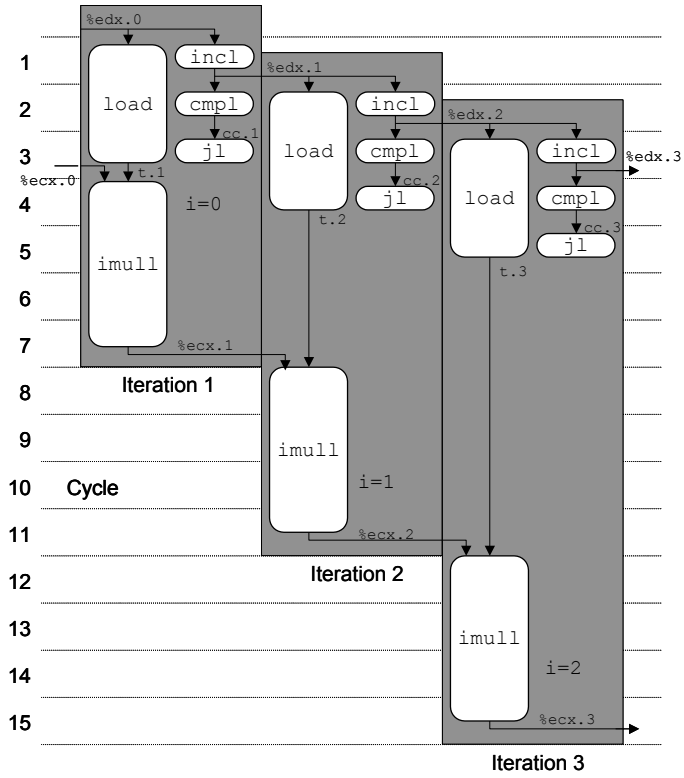
Machine dependent optimization techniques: loop unroll (4)

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:

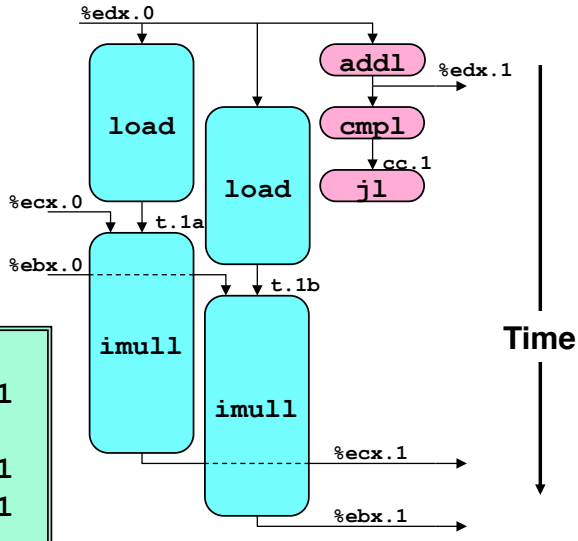
- accumulate in 2 different products
  - can be in parallel, if OP is associative!
- merge at the end

– Performance

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

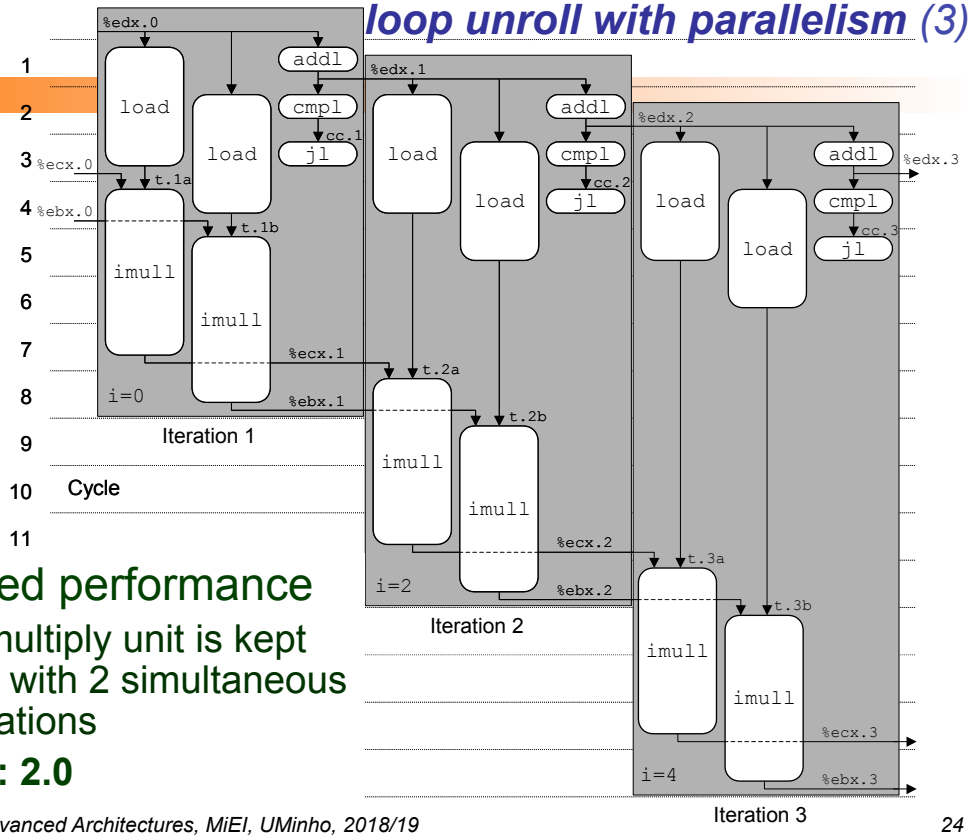
Machine dependent optimization techniques: loop unroll with parallelism (2)

- 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*

Processor arch: beyond Instruction-Level Parallelism

- When exploiting ILP, goal is to minimize CPI
  - Pipeline CPI (*efficient to exploit loop-level parallelism*) =>
    - Ideal pipeline CPI + ✔
    - Structural stalls + ✔
    - Data hazard stalls + ✔
    - Control stalls + ✔
    - Memory stalls ... *cache techniques ...*
  - Multiple issue =>
    - find enough parallelism to keep pipeline(s) occupied
- Multithreading =>
  - find additional ways to keep pipeline(s) occupied

Multithreading

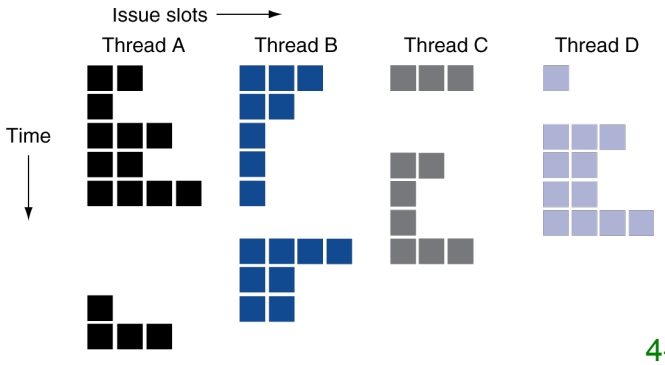
- Performing multiple threads of execution in parallel
  - Replicate registers, PC/IP, etc.
  - Fast switching between threads
- Fine-grain multithreading / time-multiplexed MT
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

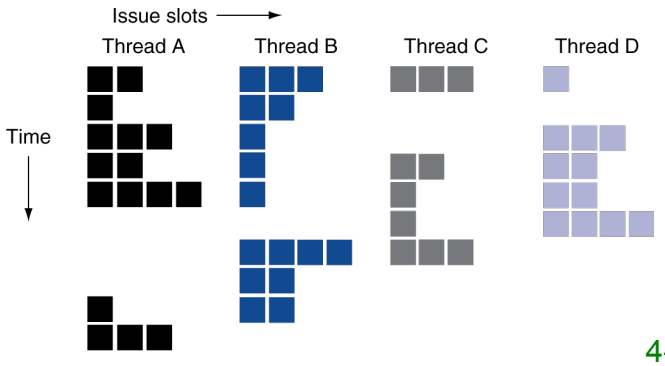
***HT**: Hyper-Threading, Intel trade mark for their SMT implementation*
*MT in Xeon Phi KNC: 4-way SMT with time-mux MT, **not HT**!*

Multithreading Example

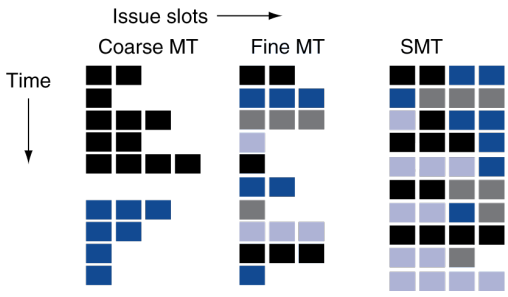


4-way superscalar

Multithreading Example



4-way superscalar



WIKIPEDIA
The Free Encyclopedia

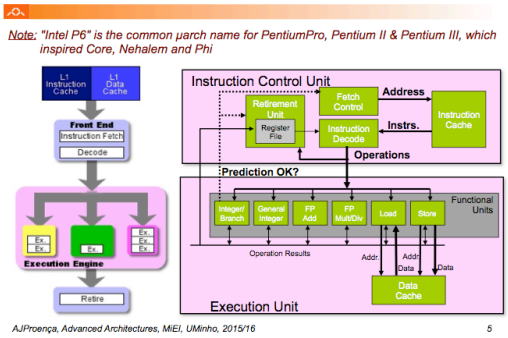
Article Talk

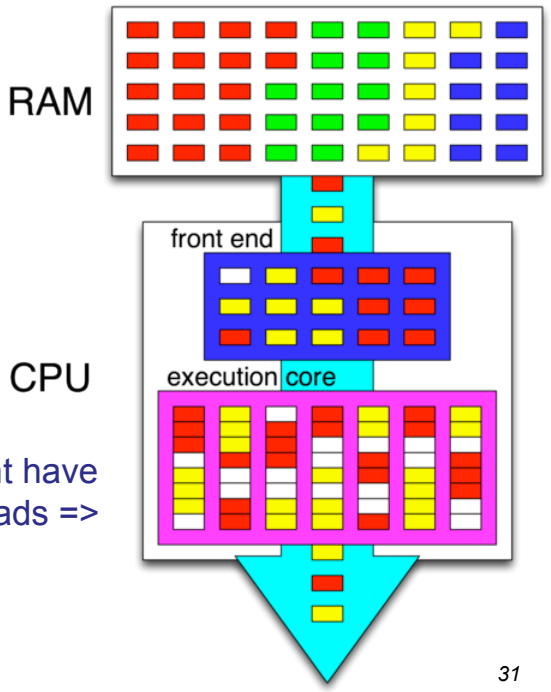
Hyper-threading

From Wikipedia, the free encyclopedia

As seen before...

Internal architecture of Intel P6 processors





The pipelined functional units might have better use if shared among more threads =>

Note: white boxes are bubbles...

Processor arch: beyond Instruction-Level Parallelism

- When exploiting ILP, goal is to minimize CPI
  - Pipeline CPI (*efficient to exploit loop-level parallelism*) =>
    - Ideal pipeline CPI + ✔
    - Structural stalls + ✔
    - Data hazard stalls + ✔
    - Control stalls + ✔
    - Memory stalls ... *cache techniques ...*
  - Multiple issue =>
    - find enough parallelism to keep pipeline(s) occupied
- Multithreading =>
  - find additional ways to keep pipeline(s) occupied
- Insert data parallelism features

Instruction and Data Streams

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

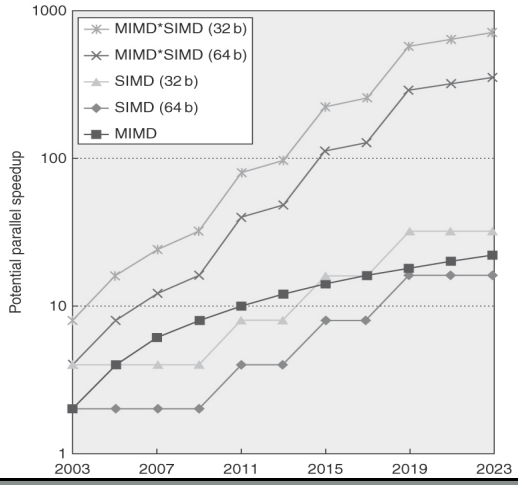
Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processing
- SIMD is more energy efficient than MIMD
  - only needs to fetch one instruction per data operation
  - makes SIMD attractive for personal mobile devices
- SIMD allows programmers to continue to think sequentially

SIMD Parallelism

- Vector architectures *(slides 5 to 18)*
- SIMD & extensions *(slides 19 to 24)*
- Graphics Processor Units (GPUs) *(next set)*

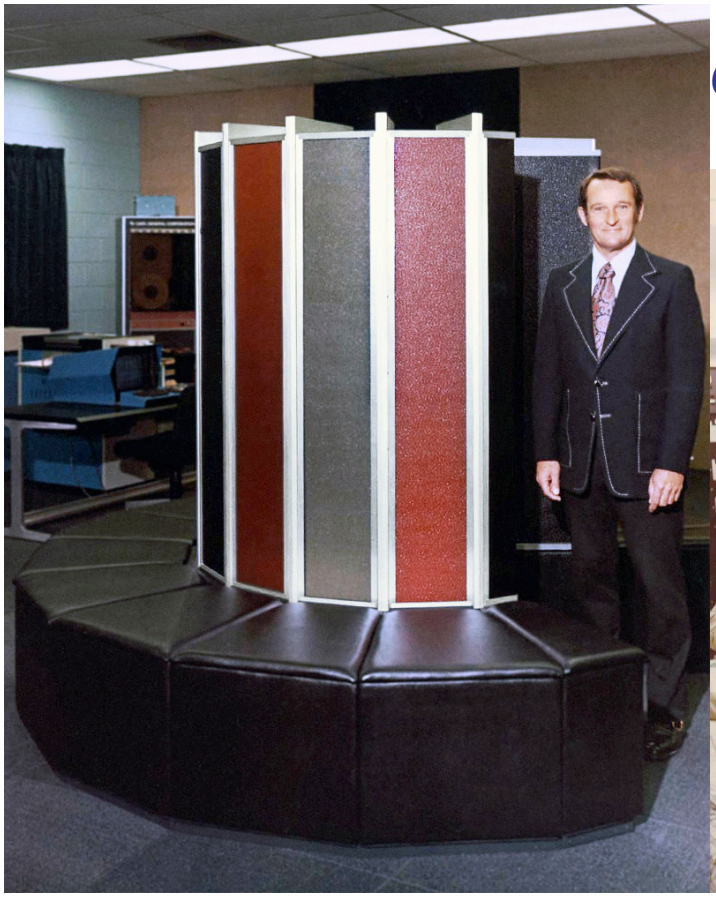
- For x86 processors:
  - Expected grow:
    2 more cores/chip/year
  - SIMD width:
    2x every 4 years
  - Potential speedup:
    SIMD 2x that from MIMD!



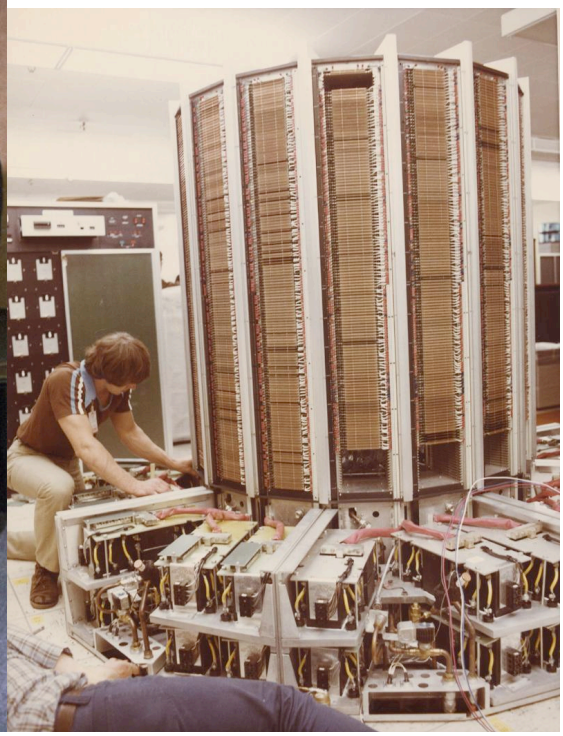
Vector Architectures

- Basic idea:
  - Read sets of data elements *(gather from memory)* into "vector registers"
  - Operate on those registers
  - Store/scatter the results back into memory

- Registers are controlled by the compiler
  - Used to hide memory latency
  - Leverage memory bandwidth



Cray-1 Supercomputer (1976)



Challenges

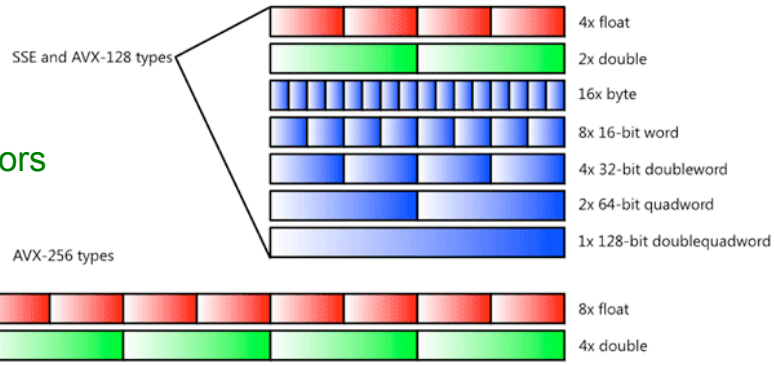
- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Improvements:
  - > 1 element per clock cycle *(1)*
  - Non-64 wide vectors *(2)*
  - IF statements in vector code *(3)*
  - Memory system optimizations to support vector processors *(4)*
  - Multiple dimensional matrices *(5)*
  - Sparse matrices *(6)*
  - Programming a vector computer *(7)*

Vector Programming *(7)*

- Compilers are a key element to give hints on whether a code section will vectorize or not
- Check if loop iterations have data dependencies, otherwise vectorization is compromised
- Vector Architectures have a too high cost, but simpler variants are currently available on off-the-shelf devices; however:
  - most do not support non-unit stride => care must be taken in the design of data structures
  - same applies for gather-scatter...

SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Intel SIMD Ext started with 64-bit wide vectors and grew to wider vectors and more facilit
  - Current AVX generation is 512-bit wide

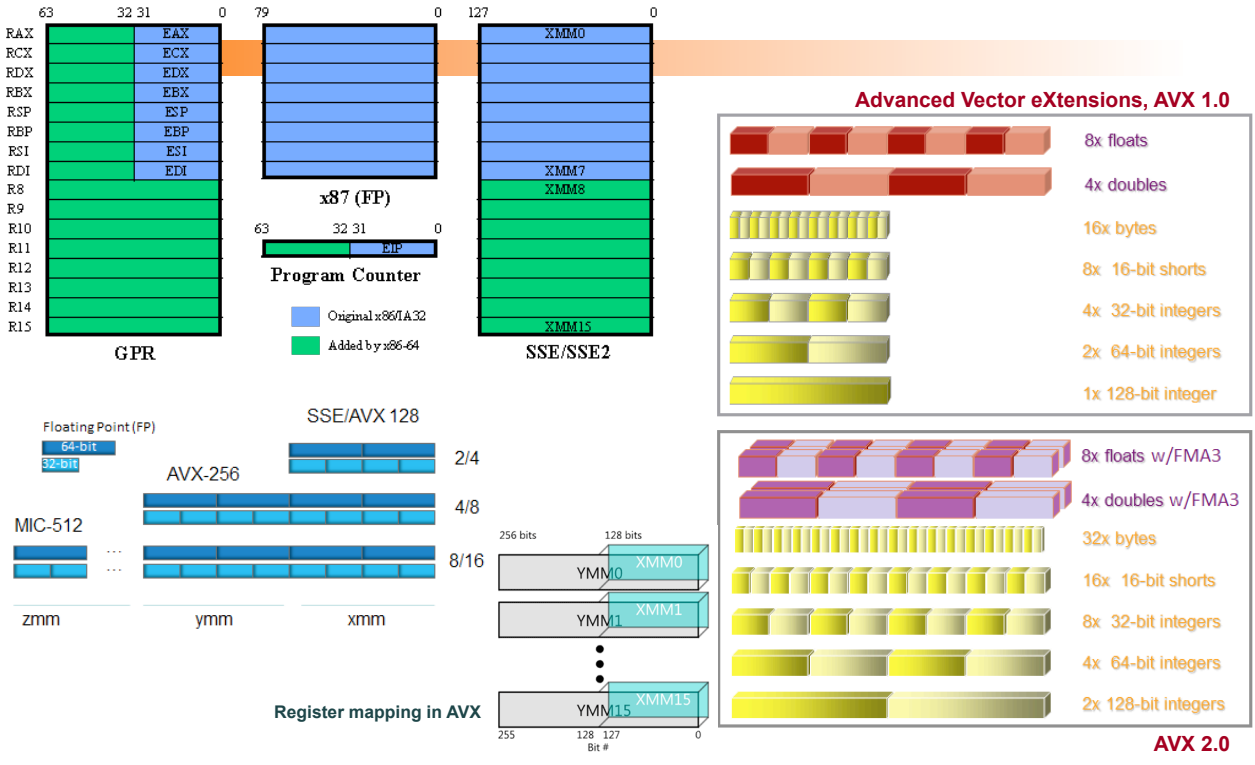


- Limitations, compared to vector architectures:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather, but…)
  - No mask registers

SIMD Implementations

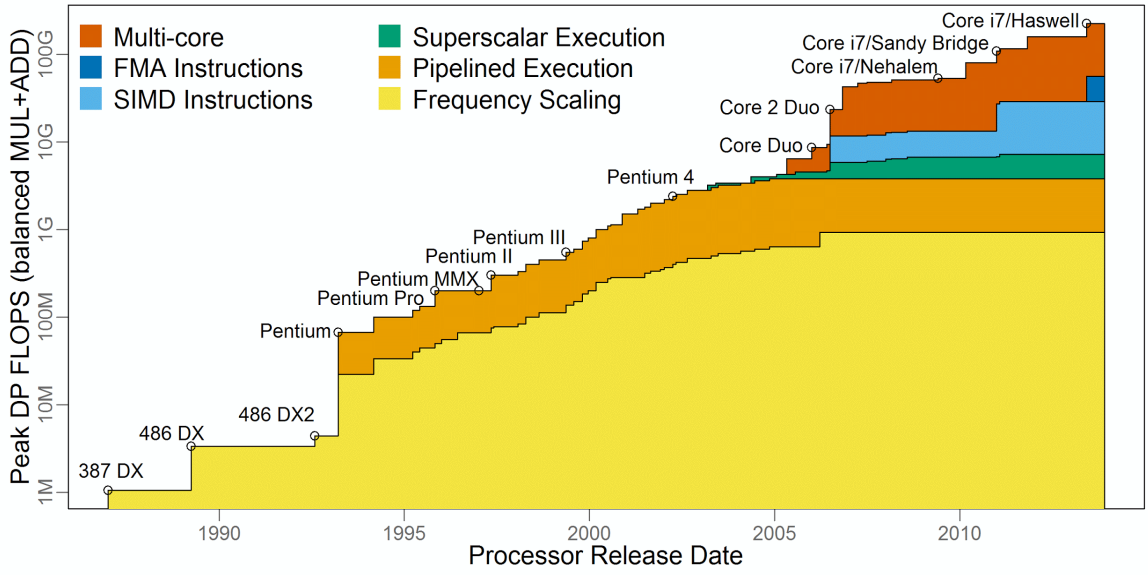
- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector eXtensions (AVX) (2010)
    - Eight 32-bit fp ops or Four 64-bit fp ops (integer in AVX-2)
    - 512-bits wide in AVX-512 (and also in Larrabee & Phi-KC)
  - Operands **must be in consecutive and aligned** memory locations

Extensões de processamento vetorial no Intel 64



Next: AVX-512

Additional features in Intel x86



from Marat Dukhan, 2014

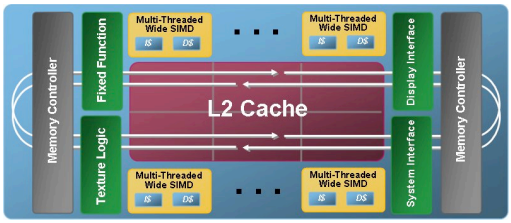
Beyond Vector/SIMD architectures

- Vector/SIMD-extended architectures are hybrid approaches
  - mix (super)**scalar + vector** op capabilities on a single device
  - **highly pipelined** approach to reduce memory access penalty
  - **tightly-closed access to shared memory**: lower latency
- Evolution of Vector/SIMD-extended architectures
  - **CPU cores with wider vectors and/or SIMD cores**:
    - DSP VLIW cores with vector capabilities: **Texas Instruments** (...?)
    - PPC cores coupled with SIMD cores: **Cell** (past...) , **IBM Power BQC...**
    - ARM64 cores coupled with SIMD cores: from Tegra to Parker (**NVidia**) (...?)
    - x86 many-core: **Intel** MIC / Xeon KNL, **AMD** FirePro**...**
    - other many-core: **ShenWay** 260, **Adapteva** Epiphany-V**...**
  - **coprocessors (require a host scalar processor): accelerator devices**
    - on disjoint physical memories (e.g., **Xeon KNC** with PCI-Expr, **PEZY-SC**)
    - focus on SIMT/SIMD to hide memory latency: **GPU**-type approach
    - ISA-free architectures, code compiled to silica: **FPGA**

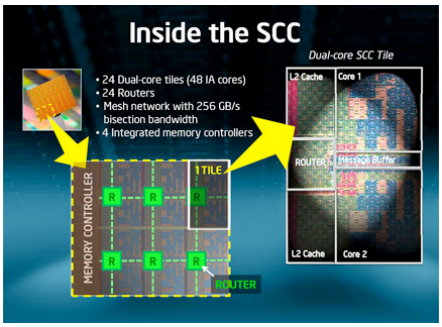
Intel MIC: Many Integrated Core

Intel evolution, from:

- Larrabee (80-core GPU) & SCC

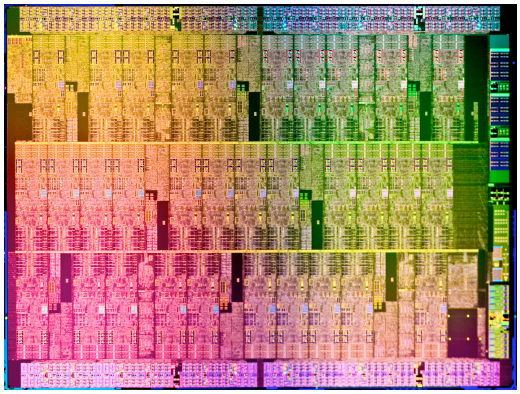


Single-chip Cloud Computer, 24x dual-core tiles

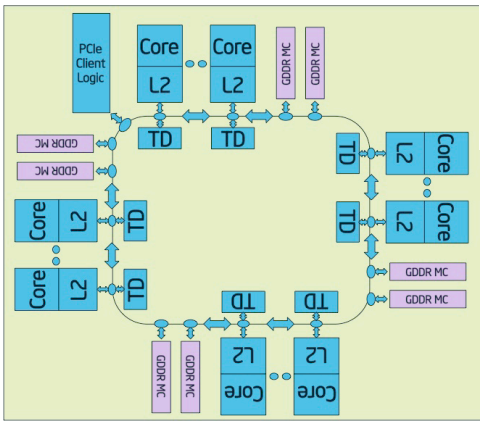


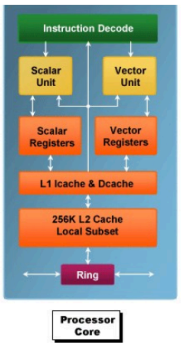
to MIC:

- Knights Ferry (pre-production, Stampede)
- Knights Corner → Xeon Phi co-processor up to 61 Pentium cores
- Knights Landing
  Xeon Phi full processor,
  36x dual-core tiles with 64-bit Atoms

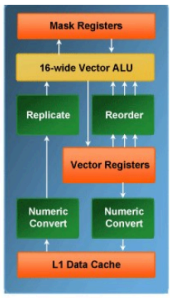


Intel Knights Corner architecture

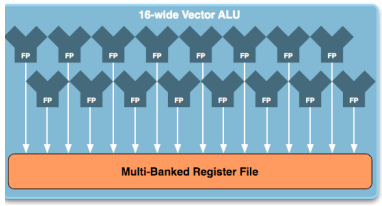


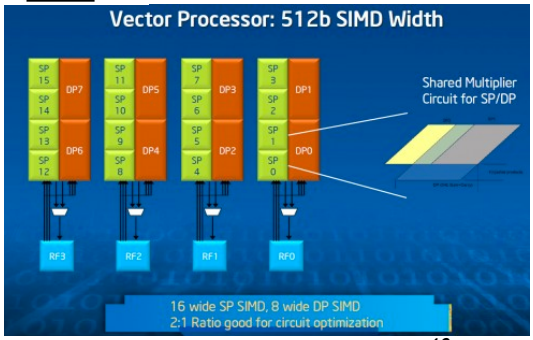


Processor Core

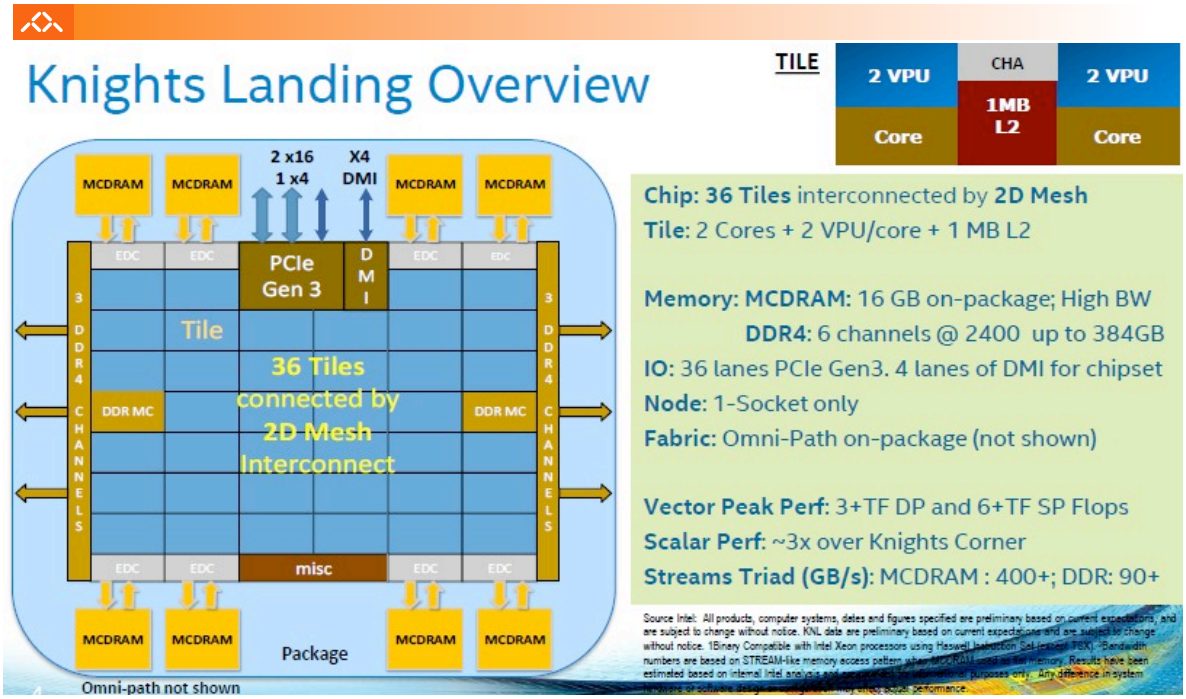


Vector Unit

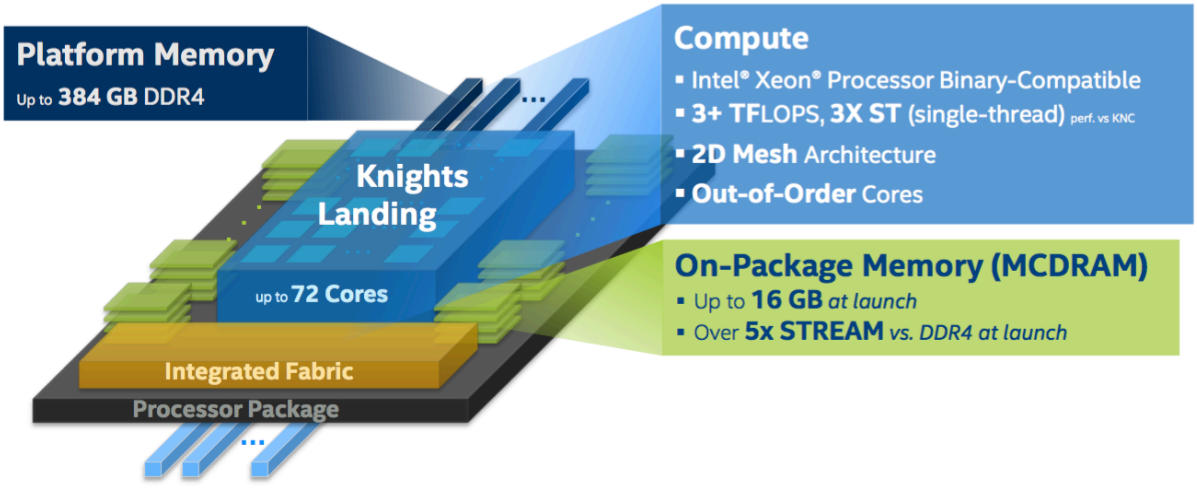




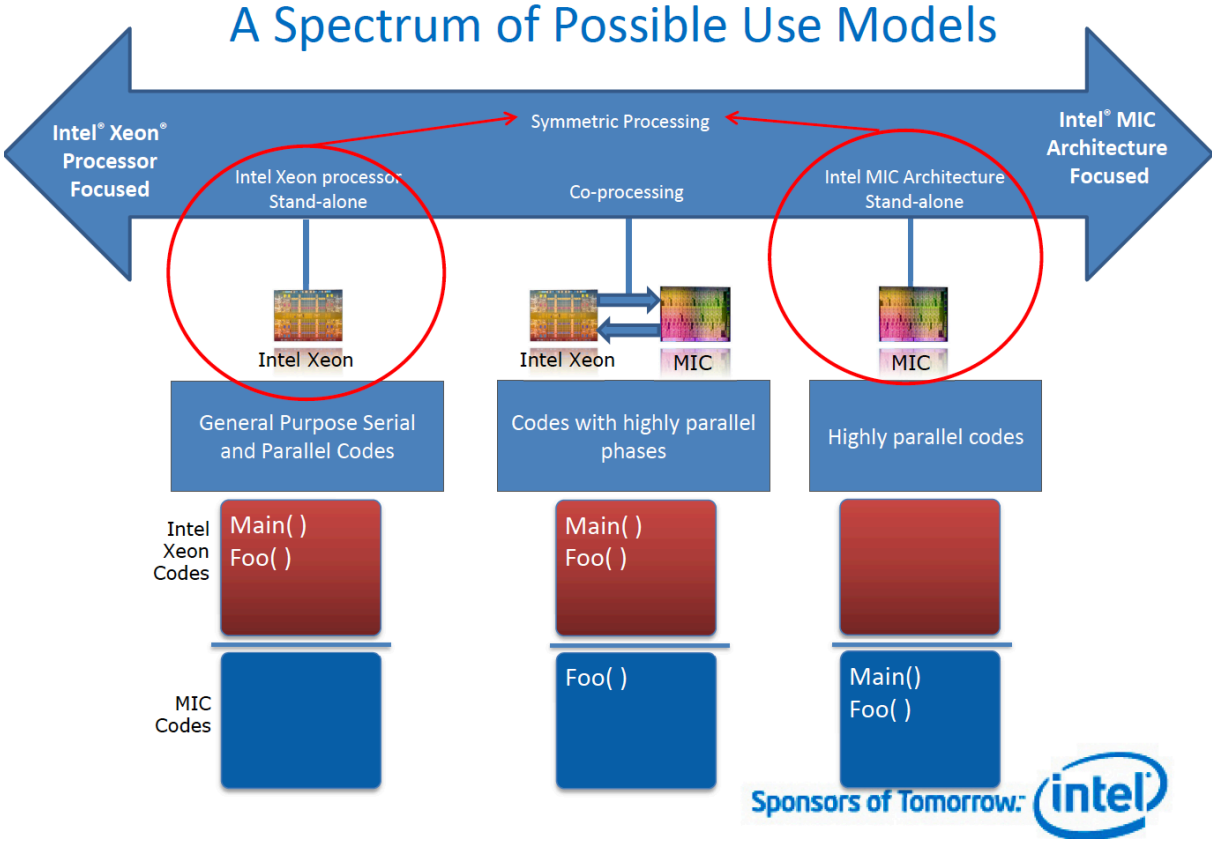
Intel Knights Landing architecture



INTEL® XEON PHI™ X200 PROCESSOR OVERVIEW

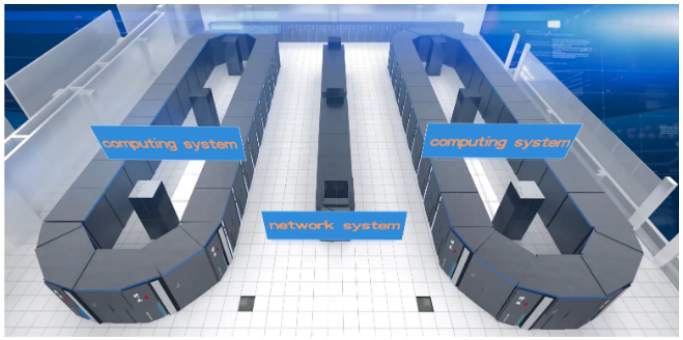


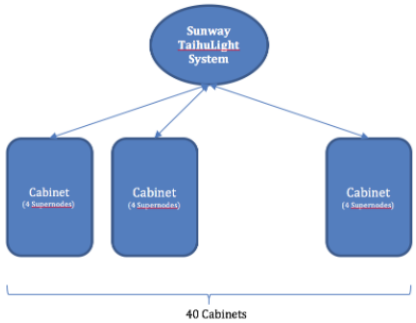
A Spectrum of Possible Use Models

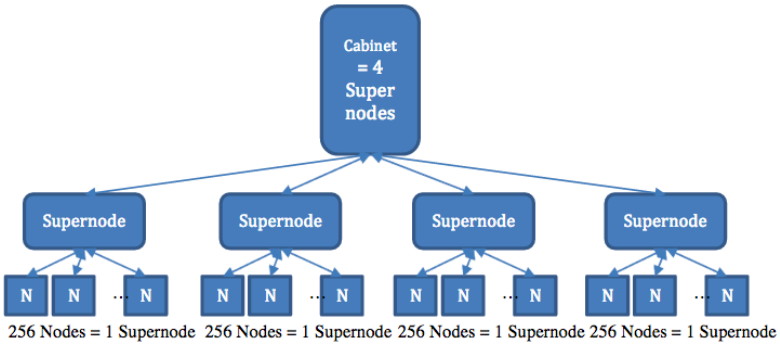


#1 from June'16 TOP500: Sunway TaihuLight

Overview of the Sunway TaihuLight System

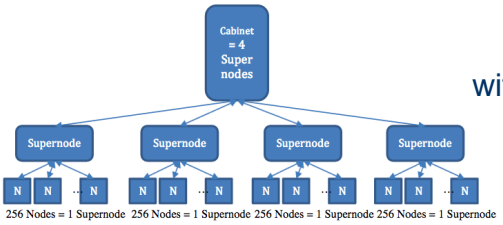




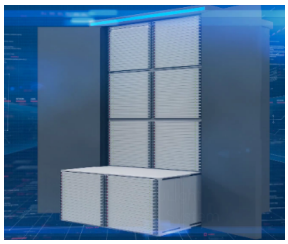


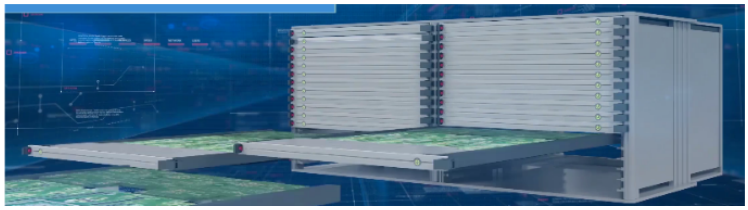
http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf

#1 from June'16 TOP500: Sunway TaihuLight

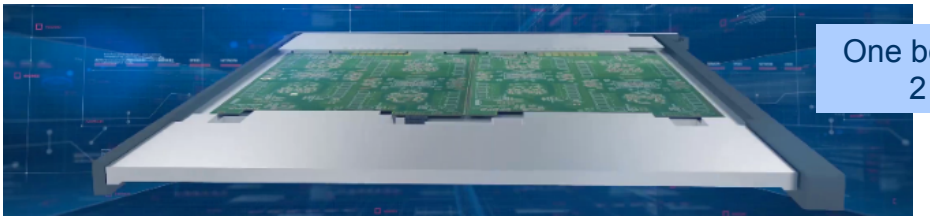


One cabinet with 4 Supernodes





One Supernode with 32 boards

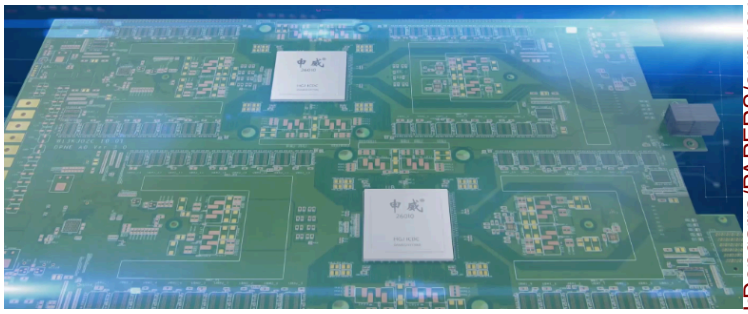


One board with 4 cards, 2 up & 2 down

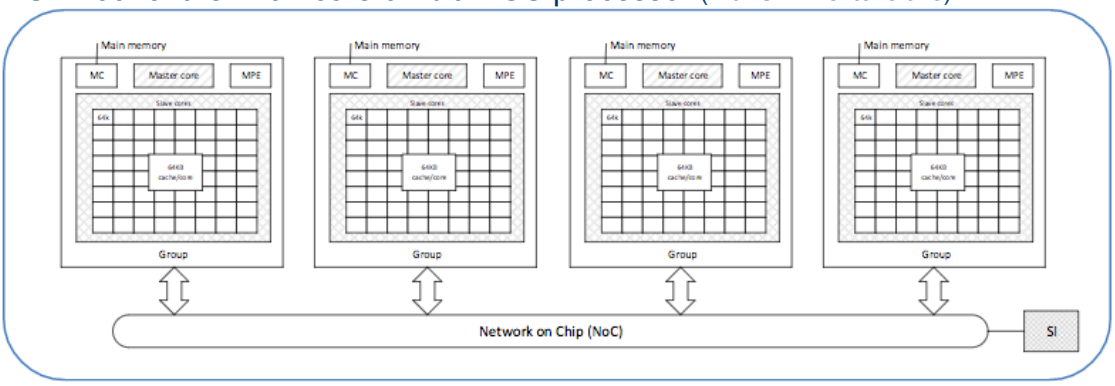
http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf

#1 from June'16 TOP500: Sunway TaihuLight

One card with two nodes *(two SW26010 chips)*



SW26010: the 4x64-core 64-bit RISC processor (with SIMD extensions)



http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf