PERFORMANCE ENGINEERING

Lecture 4: Analytical models calibration and validation

April 25th, 2022

Ana Lucia Varbanescu
a.l.varbanescu@uva.nl

Recap: modeling

- Combine problem and machine models into a function which defines performance.

- All models are simplifications of reality
  - Application is simplified
  - Machine is simplified
  - Mapping of application on machine is simplified
  - …

Recap: modeling

- Many different modeling strategies:
  - Benchmarking (time consuming & hard to analyse)
  - Statistical (time consuming, little insight)
  - Simulation (time consuming)
    - Cycle-accurate
    - Event-based
  - Analytical (difficult to build)

- In reality, we often end up with a mix of these models.
  - Use analytical models for the full application
  - Simulation models for memory hierarchies
  - Benchmarking for calibration

Recap: analytical modeling

- Analytical performance modeling is* based on *decomposing* the problem in *fine-grained operations* that can be *composed* and *benchmarked*.

- For example:

```
Model 0: T = T_compute + T_memory + T_comm
```

```
Model 1: T = N_op1 * t_compute1 +
             N_op2/p * t_compute2 + …
             N_op_L1 * t_L1 + N_op_L2 * t_L2…
```

**other ways to construct analytical models also exist…*

An analytical model

$$T = T_{comp} + T_{mem} + T_{comm} + T_{par}$$

- Serial : Ts
  - T_comp = N_alu_ops * t_op
  - T_mem = N_mem_ops * t_mem
  - T_comm = 0
  - T_par = 0
- Parallel : Tp
  - T_comp = N_alu_ops/**p** * t_op
  - T_mem = N_mem_ops/**p** * t_mem
  - T_comm = N_comm * t_comm
  - T_par = T_overhead

Overhead due to parallelism (mix of comp & mem)

Communication/synchronization (e.g., atomics, barriers, inter-node comm, etc.)

Refers to memory operations.

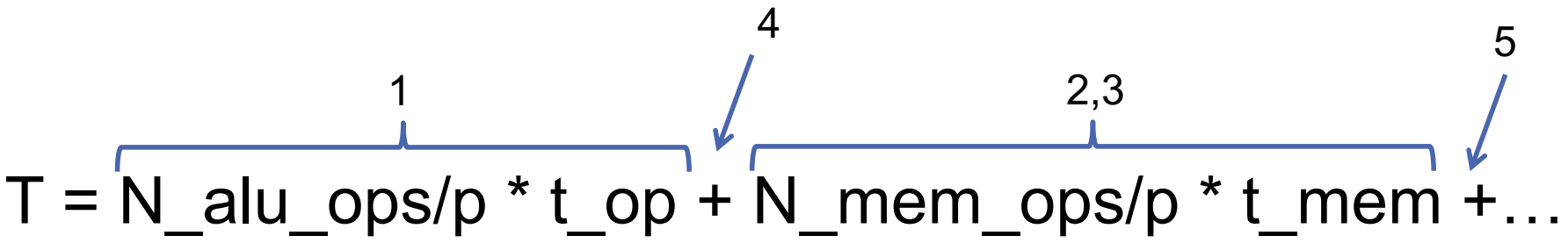
Another (analytical) model here …

Not the only possible model!

$$Tp = N\_alu\_ops/p * t\_op + N\_mem\_ops/p * t\_mem + \ldots$$

Improving the model ?

1. Take into account different arithmetic operations
2. Take into account different memory operations
3. Take into account the memory hierarchy
4. Take into account the overlapping of compute and load//store operations (memory operations)
5. Take into account the overlapping of computation and communication



Model different operations (1,2,3)

- Arithmetic operations of different kinds have different *latency* and *throughput*
  - Add, shift, …
  - Multiply, divide
  - SIMD or not
  - Int, float, double
  - …
- Memory operations have different *latency* and *throughput*
  - Load
  - Store
  - Cache hierarchy & policies

Average memory access time

T_access = hit_rate * hit_time + (1-hit_rate)*miss_time

Parameters?

- Measurements
  - Use LIKWID, perf, other tools
- Simulators
  - Valgrind/cachegrind, pyCacheSim, Dinero, …
- Analytical model ☺
  - Knowledge of the architecture
    - Cache model, policies, line sizes …
  - Knowledge of the application/compiler
    - What goes into cache & when

Back to our performance model …

T_comp, on p processors, assume perfect parallelism

```
Tp = N_alu_ops1/p * t_op1 + N_alu_ops2/p * t_op2 + … +
     + N_mem_ops/p * t_mem + …
```

T_mem, on p processors

Other terms if needed (comm, overhead)

No comp/mem overlap

```
t_mem = hit_rate * t_h + (1-hit_rate)* t_miss
```

Might change with different p values!

Increasing code complexity

- For loops:
  - T_for = sum all iterations => N x T_iteration
- While/do loops
  - T_while = sum all iterations => X x T_iteration
    - Key challenge: estimate X
- If statements
  - T_if = max (T_then, T_else)
  - T_if = P(if, true)*T_then + P(if, false)*T_else

Increasing code complexity

- Sequential vs. Parallel
  - Sequential = sum
  - Parallel = max
- Reduction ?
  - Depending on algorithm …
- Atomics ?
  - Best case: T_op
  - Worst case: |Proc| x T_op
  - Best estimate: Prob(contention) x T_op

So far …

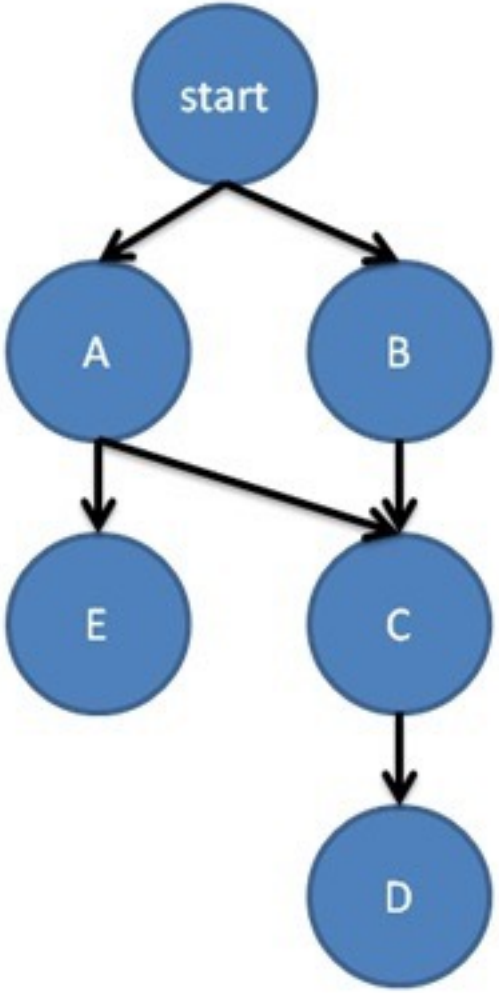
- Single-kernel applications
  - Matrix multiplication
  - Histogram
- Parallelism = data-parallelism
  - Each processor = same computation + part of the data
- What about larger applications and their parallelism?

Larger applications?

- Application = DAG of tasks/operations
  - Operations = nodes
  - Dependencies = edges
    - Dependency = child-after-parent
- Two metrics:
  - The amount of **work** to be executed

    T1 = A+B+C+D+E
  - The **span** of the application
    - Also called critical-path length or depth

    T∞ = B+C+D
- Parallelism: T1/T∞
  - the average amount of work per step.



Practice with work & span

- Sequential vs. Parallel (A -> B vs. A || B)
  - Work?
  - Span?
- Reduction ?
  - Work?
  - Span?
- Atomic update
  - Work?
  - Span?

Practice with work & span

- Sequential vs. Parallel
  - Work?
    - Seq: A+B Par: A+B
  - Span?
    - Seq: A+B Par: max(A,B)
- Reduction ?
  - Work: O (N)
  - Span: O(N) or O(log N)
- Atomic update ?
  - Work: P x T_op
  - Span:
    - Worst case: P x T_op
    - Best estimate: #contention x T_op

DAGs and analytical modeling

- Sequential execution (A,B) => T = A+B
  - Typical for dependencies!
- Parallel execution (A || B) => T = max(A,B)

- How do we obtain the DAG?
  - Source-code analysis (read: manually, assumed static)
  - Models of computation (read: based on patterns/programming models)
  - Automated (read: through schedulers)

Models of parallel computation

- Data-parallel
- Fork-and-join
  - “Extension” of data parallelism (OpenMP’s model)
    - Careful with synchronization!
- Task-parallel
  - DAG required
- Master-workers, Divide-and-conquer
  - Bounds-analysis (BCET, WCET, ACET)
    - Execution can be data-dependent!
- BSP (Bulk-synchronous parallel)
  - Specialized computation and communication stages
  - Enforced synchronization

DAGs and analytical modeling

- Sequential execution (A,B) => T = A+B
  - Typical for dependencies!
- Parallel execution (A || B) => T = max(A,B)

- How do we obtain the DAG?
  - Source-code analysis (read: manually, assumed static)
  - Models of computation (read: based on patterns/programming models)
  - Automated (read: through schedulers)

- How do we use the DAG?
  - Combine the tasks into a realistic analytical model
    - Critical path determines execution time
    - … but must include scheduling dependencies!

The impact of computational models

- Work = always the same* (± overhead)
- Scheduling = depending on the model
  - Master/worker ?
  - Divide and conquer?
  - Task parallelism?

- What is the impact of scheduling on the DAG?
  - Might reduce parallelism => more sequential execution
  - Adds more dependencies in the graph => span is “longer”
    - These are “scheduling dependencies”
      - Application and processor specific!

Model calibration

So far & still to do …

- Application performance is a symbolic expression (typically using “+” and “max” operators) composed of operations and costs for different parts of the application

- Model calibration:
  - Estimate the number of operations
  - Estimate the cost of each operation

- Model validation!

Back to our performance model …

T_comp, on p processors, assume perfect parallelism

```
Tp = N_alu_ops1/p * t_op1 + N_alu_ops2/p * t_op2 + … +
     + N_mem_ops/p * t_mem + …
```

T_mem, on p processors

Other terms if needed (comm, overhead)

No comp/mem overlap

```
t_mem = hit_rate * t_h + (1-hit_rate)* t_miss
```

Might change with different p values!

- This is a symbolic performance model
  - May allow for performance ranking!
- For an actual prediction … ?
  - Determine N_op's
  - Determine t's

Discussion later today!

Determining number of operations

- By hand:
  - At algorithmic level => number of useful operations
  - Exclude “overhead” => loop index computation, if statements, …
  - Pro’s: easy to separate
  - Con’s: if overhead is high, accuracy drops

- Profilers:
  - Determine number of operations of each kind
  - Pro’s: accurate
  - Con’s: difficult to separate core from overhead

Determining the latency per operation

- Theoretical latency
  - From catalogues, in cycles

Integer instructions

Instruction	Operands	µops fused domain	µops unfused domain	µops each port	Latency	Recipro-cal through put	Comments
Move instruc-tions							
MOV	r,i	1	1	p0156		0.25	
MOV	r8/16,r8/16	1	1	p0156	1	0.25	
MOV	r32/64,r32/64	1	1	p0156	0-1	0.25	may be elim.
MOV	r8l,m	1	2	p23 p0156		0.5	
MOV	r8h,m	1	1	p23		0.5	
MOV	r16,m	1	2	p23 p0156		0.5	
MOV	r32/64,m	1	1	p23	2	0.5	all addressing modes

- Measured latency
  - Benchmarking and microbenchmarking

Measuring hardware performance

“real” hardware performance

- Microbenchmark = synthetic benchmark = a small program that tests/stresses a specific component of the machine, aiming to determine its actual limits in practice.
  - Synthetic => still under lab conditions!
  - Measure close-to-peak hardware performance
  - Most of the current ones: focus on memory!

- Benchmark (suite) = application (suite) that test the abilities of processors to behave in “real” conditions.
  - Based on real-life applications
  - Extrapolate findings based on similarity

“real” Hardware performance

- Microbenchmarking*
  - Evaluates hardware features in isolation
  - Goal: find out the true limits of the hardware components
  - Platform-specific results
  - **Compared** with the theoretical peak, **per platform**.

- Benchmarking
  - Evaluates the FULL platform
  - Application-specific performance
    - Top500 – computation capability
    - Graph500 – graph processing capability
    - Green500 – energy consumption
  - **Compares platforms**

*Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, Andreas Moshovos. ”Demystifying GPU Microarchitecture through Microbenchmarking”, ISPASS 2010

Microbenchmarking

- How would you build a microbenchmark for an arithmetic operation?
  - ADD
  - DIV

- How would you build a microbenchmark for the memory system?
  - L2 cache bandwidth?
  - Main memory latency?

Main techniques:

- Pointer chasing (for latency)
- Stride- and size- variation (for throughput)

Microbenchmarking examples

- Compute operations
  - CPU: nanoBench, likwid-bench, …
  - GPU: MIPP, various papers, …
- Memory operations
  - ”memory mountain”
    - see Computer Systems: A Programmer’s Perspective
  - CPU: nanoBench, lmbench3, …
  - GPU: various papers
- Different compute and memory mixes
  - STREAM / BabelStream / …

Try it out …

- Run this benchmark on your system, collect and plot the data, and report performance:

http://csapp.cs.cmu.edu/3e/code.html

  - Download code
  - Find the different versions of the benchmark (mem)
  - Try the basic
  - Try the most optimized

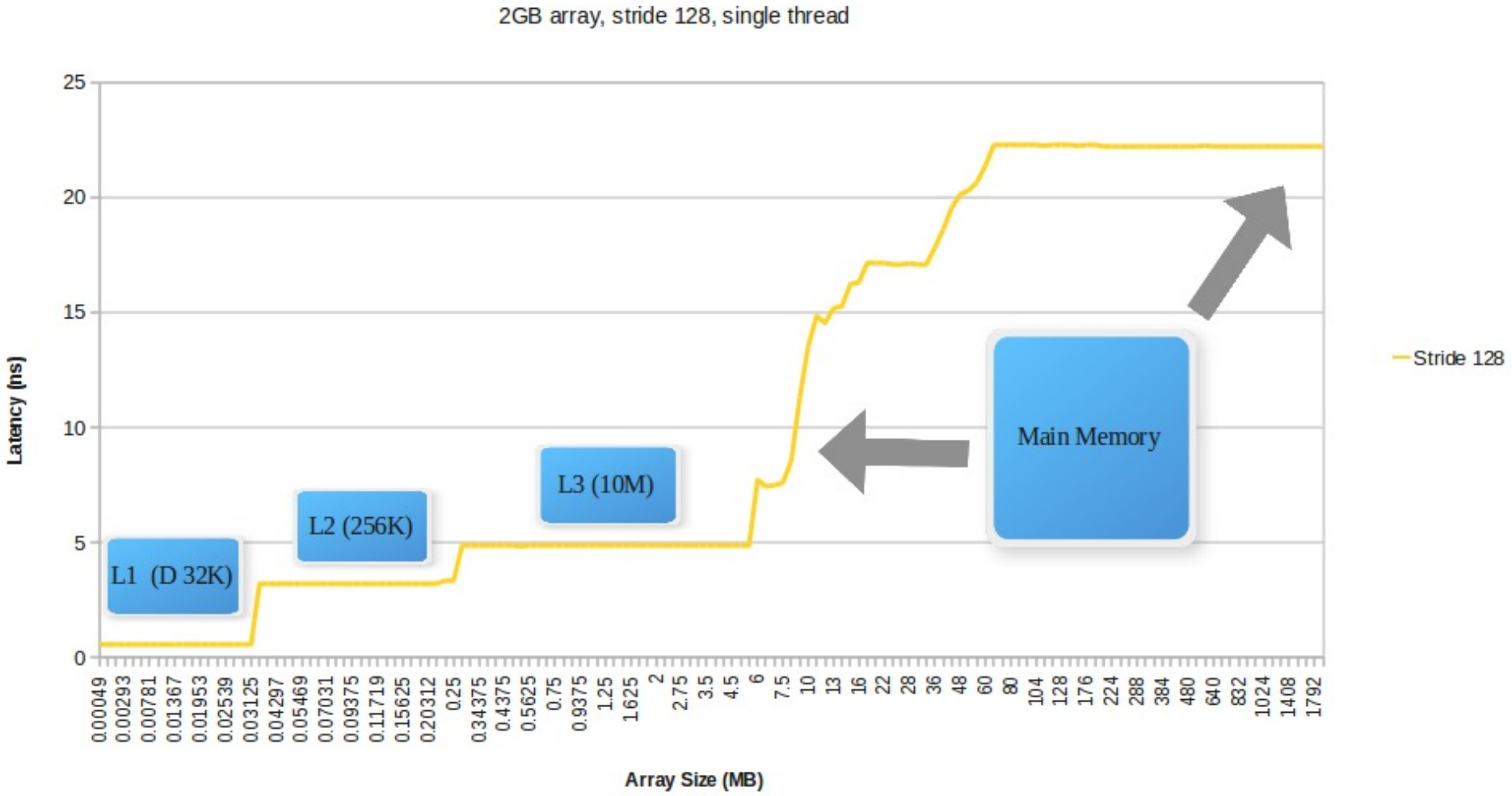
Pointer chasing*

- Mechanism to determine memory latency (and bandwidth)
  - Takes into account cache lines, memory, and NUMA domains
    - NUMA = non-uniform memory architecture
- Two-steps:
  - Fill buffer with pointers with desired structure
    - Random, stride, …
  - Chase pointers:

```
void** p = (void**) memory[0];
while (count-- > 0)
        p = (void**) *p;
```

*https://github.com/maleadt/pChase

An example: lmbench3*



*http://www.bitmover.com/lmbench/

Example: STREAM*

- (Micro)benchmarking the memory in parallel systems
  - It measures the bandwidth for (long) vector operations.

Name	Kernel op.	Bytes/iter.	FLOPs/iter.
COPY	A(i) = B(i)	16	0
SCALE	A(i) = q* B(i)	16	1
SUM	A(i) = B(i) + C(i)	24	1
TRIAD	A(i) = B(i) + q*C(i)		

  - Copy : measures transfer rates in the absence of arithmetic.
  - Scale : adds a simple arithmetic operation.
  - Sum : adds a third operand to allow multiple load/store ports on vector machines to be tested.
  - Triad : allows chained/overlapped/fused multiply/add operations.

*https://www.cs.virginia.edu/stream/ref.html#what

Benchmarking suites

- Collections of “representative” applications
- Allow testing processors in real-life conditions and compare them

- Application-specific benchmarking suites
  - Top500
  - Graph500
  - Green500

- Scientific benchmarking suites:
  - SPEC benchmarks
  - NAS parallel benchmarks
  - SPLASH-2
  - PARSEC
  - …

Example: SPEC benchmarks

- SPEC CPU2017*
  - Evaluates different features of the CPUs
  - Looks at throughput and latency

Short Tag	Suite	Contents	Metrics
intspeed	SPECspeed 2017 Integer	10 integer benchmarks	SPECspeed2017_int_base SPECspeed2017_int_peak
fpspeed	SPECspeed 2017 Floating Point	10 floating point benchmarks	SPECspeed2017_fp_base SPECspeed2017_fp_peak
intrate	SPECrate 2017 Integer	10 integer benchmarks	SPECrate2017_int_base SPECrate2017_int_peak
fprate	SPECrate 2017 Floating Point	13 floating point benchmarks	SPECrate2017_fp_base SPECrate2017_fp_peak

- SPEC ACCEL
  - Evaluate acceleration potential
    - 19 OpenCL, 15 OpenACC, 15 OpenMP-4 (target offloading)

*interesting analysis: Ankur Limaye et al. – “A Workload Characterization of the SPEC CPU2017 Benchmark Suite”

How useful are benchmarks

- Drive computer architecture research
  - Good benchmarks => advance
  - Bad benchmarks => optimize the wrong elements
- Compare architecture capabilities
- Expensive to run exhaustively
  - A lot of research into selecting representative applications* and inputs
    - One of the leading groups: Ghent University, Belgium, in the Department of Electronics and Information Systems (ELIS)

http://users.elis.ugent.be/~leeckhou/papers/TC06.pdf
http://users.elis.ugent.be/~leeckhou/papers/taco14-breughe.pdf

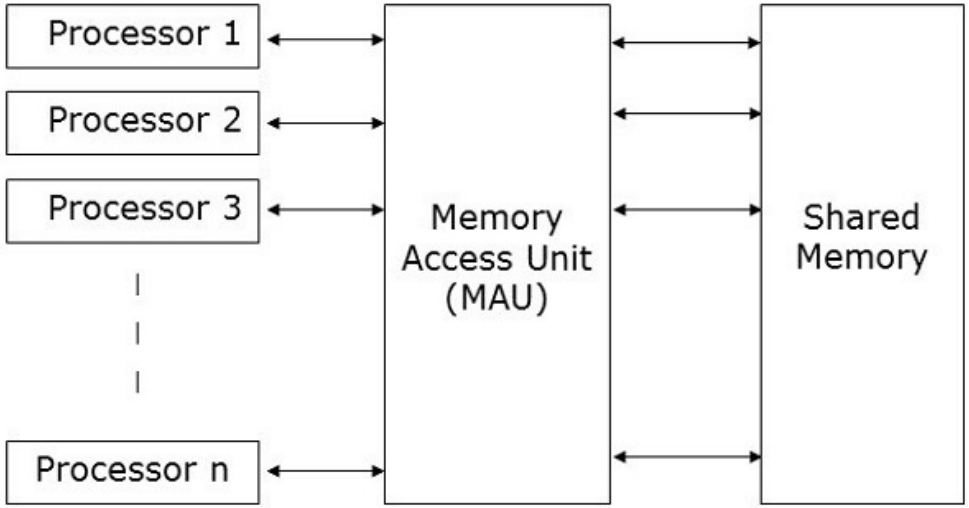
In summary …

- (micro)benchmarking is useful to determine operations costs
  - Usually in cycles
- Determining the operation cost per operation calibrates the model for a specific system.
- Operations costs combined with number of operations help transform a model from a symbolic one in a numerical one.

Revisiting the machine model

Simplistic machine models so far …

- PRAM – Parallel Random-Access Machines



- Then …
  - CPU + L1-cache + Mem
  - ALU + L1-cache + L2-cache + Mem …
  - ….

Are we missing anything?

Instruction-level parallelism (ILP)

- Multiple instructions are executed at the same time
  - Property of the application
  - Enabled by hardware
- CPUs provision hardware for this!

```
int a = b + c;
int x = y * z;
int p = q - r;
int result = a/x;
int result += p;

for (i=0; i<10; i++) {
  myArray[i] = result * i;
}
```

Accounting for application ILP

- Hardware **can execute** multiple instructions in parallel
- Performance **limited** by data dependencies
- **Simple transformations** can yield dramatic performance improvement
  - Compilers can often make these transformations
  - Limitations appear depending on data types and memory ops:
    - Lack of associativity and distributivity in floating-point arithmetic
    - Memory aliasing
    - Procedure calls
    - Library calls
    - …

CPU features for ILP

- Pipelining
  - Multiple instructions “in-flight” in different stages
- Operations are split in (micro)operations => more parallelism
- Superscalar execution
  - Multiple execution units
  - Multiple instructions “in-flight” in the same stage
- Out-of-order execution
  - Any order that does not violate data dependencies
  - Useful for keeping execution units busy
- Branch prediction
  - May avoid penalty for jumps
- Speculative execution
  - Keeps the pipeline full

Instructions or μops?

- All instructions are “decomposed” into micro-operations

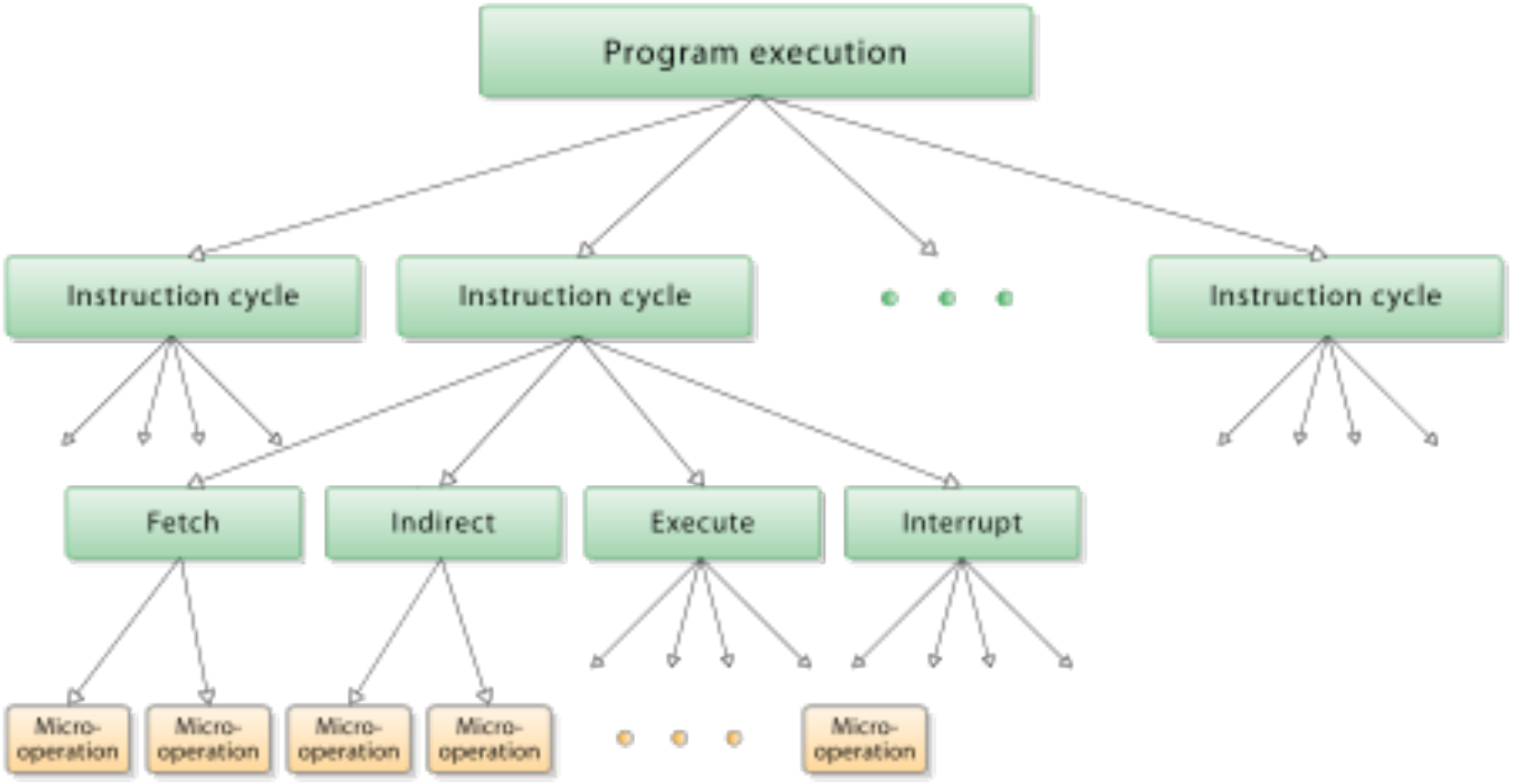
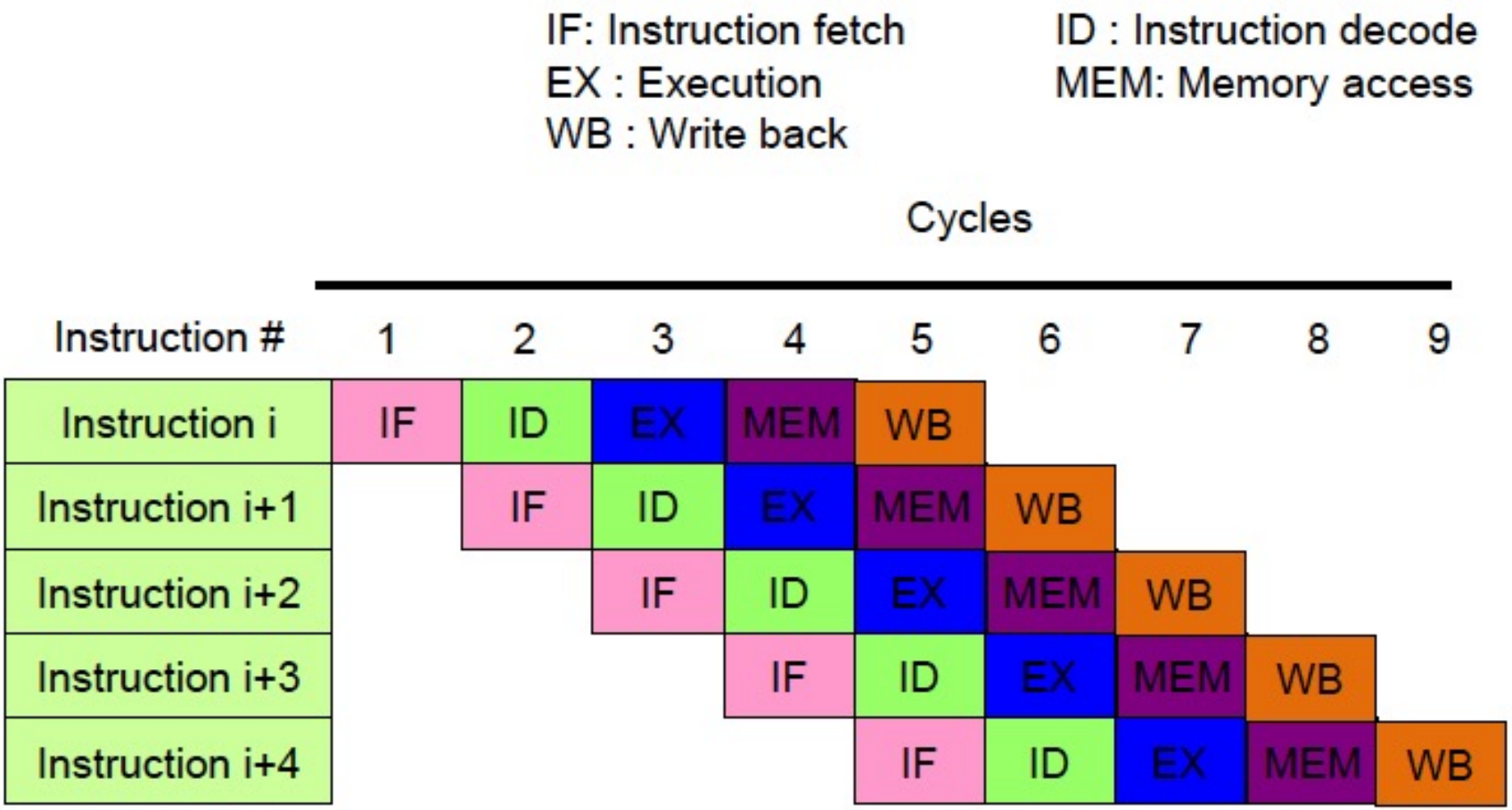
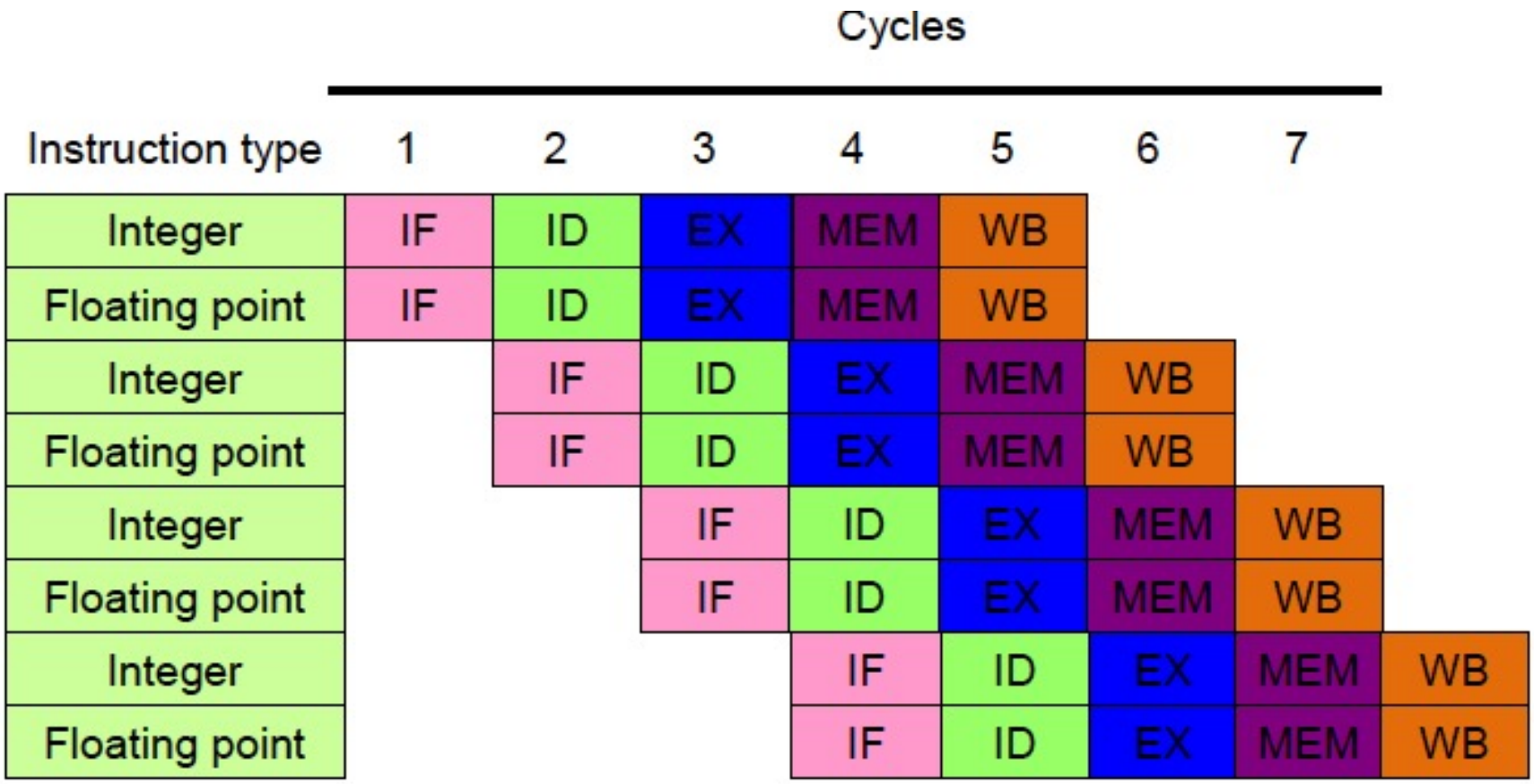


Image courtesy of Wikipedia

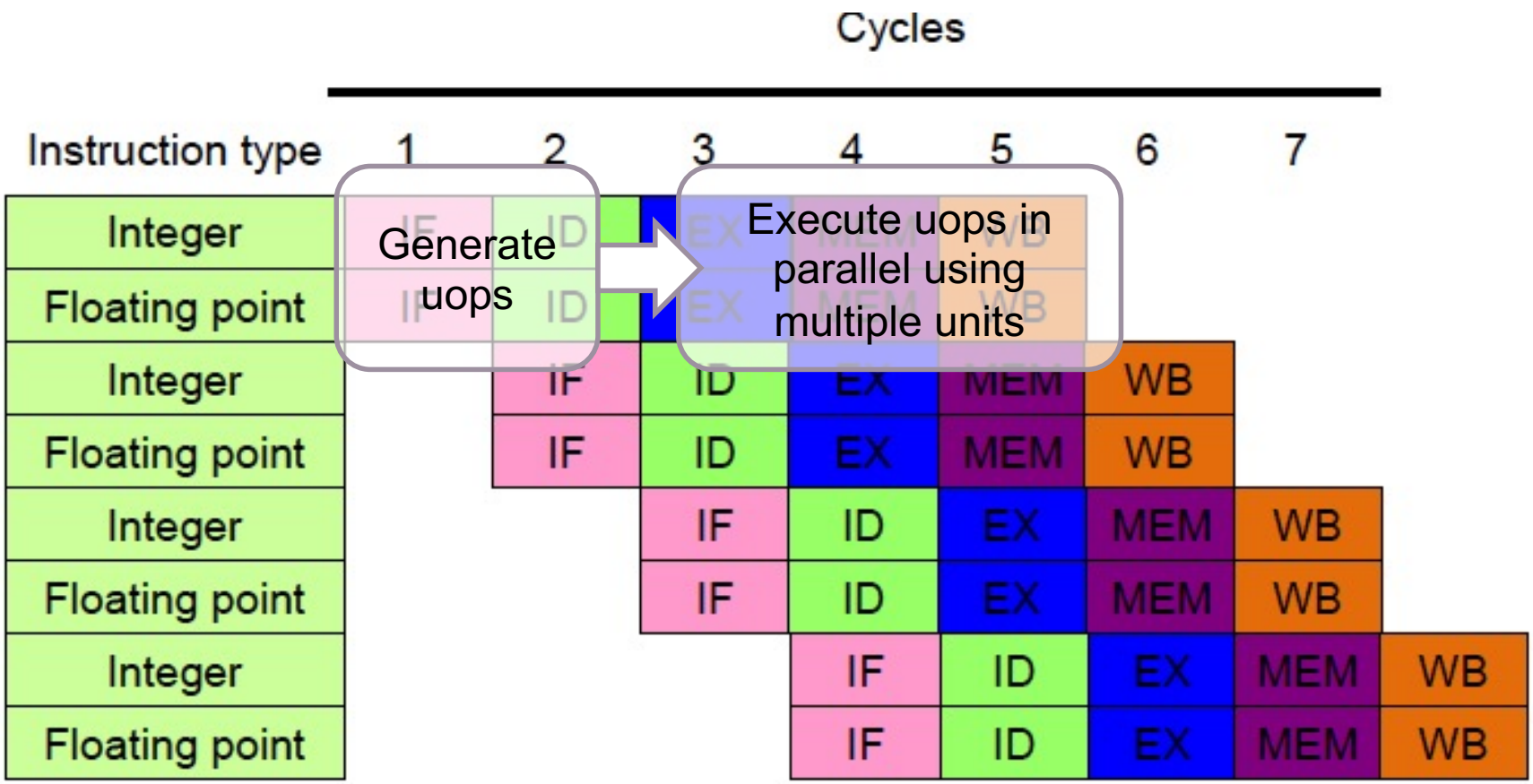
BTW … pipelining?



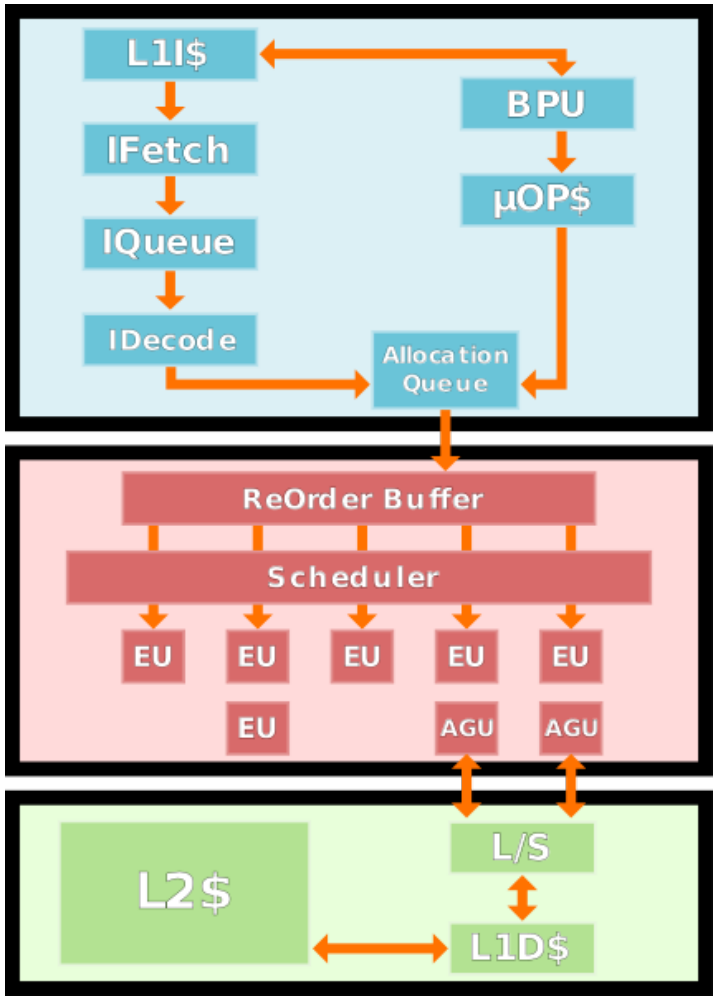
BTW … superscalar?



BTW … µops ?


Cycles
Instruction type
1
2
3
4
5
6
7
Integer
Floating point
Integer
Floating point
Integer
Floating point
Integer
Floating point
IF
ID
EX
MEM
WB
Generate uops
Execute uops in parallel using multiple units

A super-scalar CPU



Front-end:
- Reads instructions
- Decodes to uOPs
- Pushes uOPs to the “Allocation queue”

Back-end:
- Optimizes uOPs and reorders them
- Schedules uOPs
- Executes uOPs on functional units

Memory system
- Load/store buffers
- L1 data cache
- Unified L2 cache

In a nutshell …

- Front-end feeds the the back-end with sufficient operations
  - By decoding instructions coming from memory.
- All instructions arrive in the decode queue.
  - Interfaces with the backend
- Back-end:
  - IDQ to reorder buffer: micro-operations visit the reorder buffer
    - register allocation, renaming, and retiring
  - µOPs are sent to the unified scheduler
  - Scheduler assigns work to the execution units
    - queuing the µOPs on the appropriate ports
  - Once a uOP has been executed, it is retired
- Memory system
  - dedicated 32 KiB L1 data cache and 32 KiB L1 instruction cache.
  - Core-private 256 KiB L2 cache that is shared by both of the L1 caches.
  - Slice of the shared L3 cache (LLC)

Adapted from Wikichip:
https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)#Overview_2

A more realistic picture … [1]

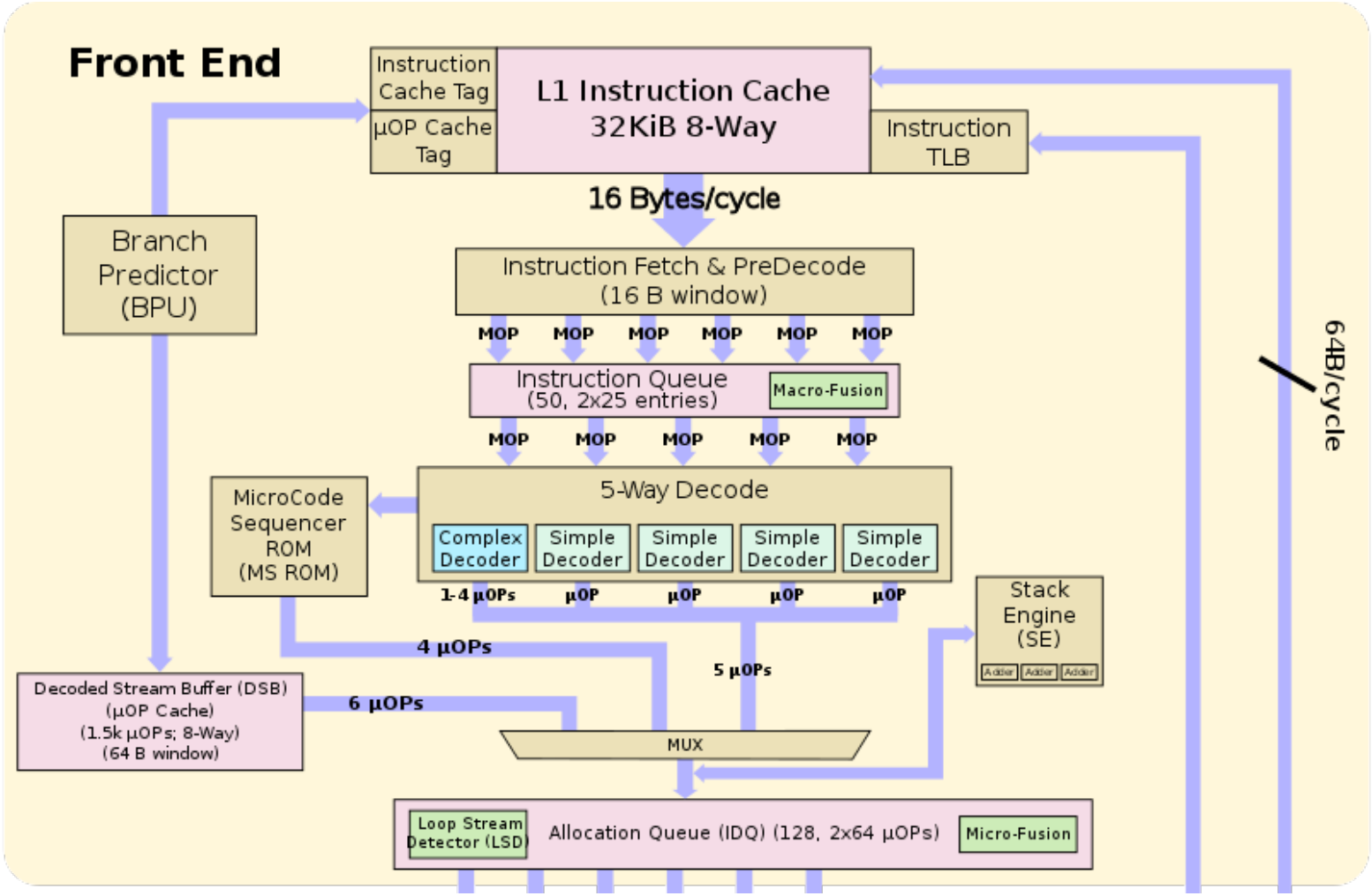


Image courtesy of Wikichip

A more realistic picture … [2]

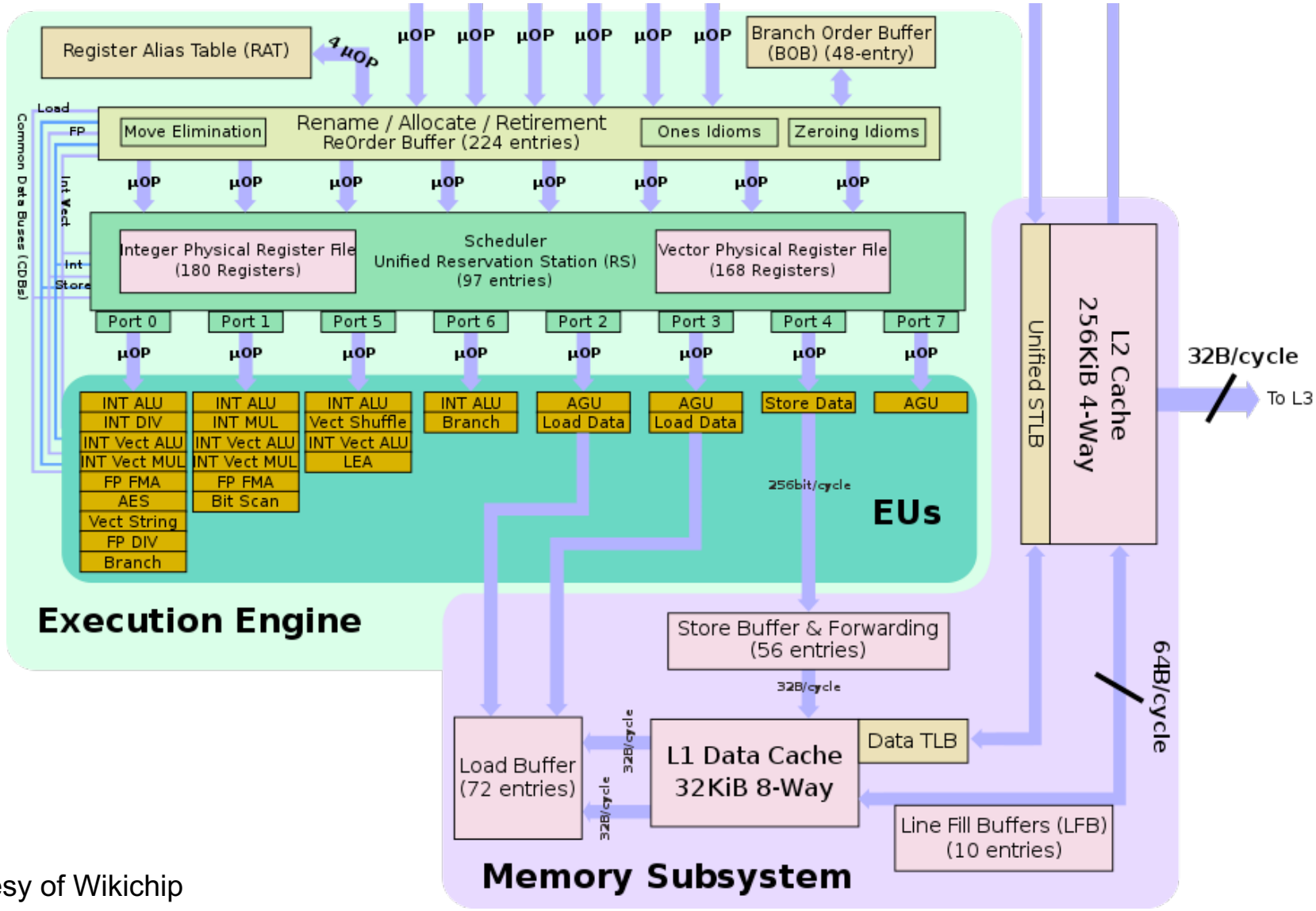


Image courtesy of Wikichip

Execution units & ports

Execution Units

Execution Unit	# of Units
ALU	4
DIV	1
Shift	2
Shuffle	1
Slow Int	1
Bit Manipulation	2
FP Mov	1
SIMD Misc	1
Vec ALU	3
Vec Shift	2
Vec Add	2
Vec Mul	2

Scheduler Ports Designation

Port 0	Integer/Vector Arithmetic, Multiplication, Logic, Shift, and String ops
	FP Add, Multiply, FMA
	Integer/FP Division and Square Root
	AES Encryption
	Branch2
Port 1	Integer/Vector Arithmetic, Multiplication, Logic, Shift, and Bit Scanning
	FP Add, Multiply, FMA
Port 5	Integer/Vector Arithmetic, Logic
	Vector Permute
	x87 FP Add, Composite Int, CLMUL
Port 6	Integer Arithmetic, Logic, Shift
	Branch
Port 2	Load, AGU
Port 3	Load, AGU
Port 4	Store, AGU
Port 7	AGU

Execution unit performance

- Latency: how many cycles an operation takes
- Throughput: maximum number of independent instructions of the same type executed per cycle
- Reciprocal throughput: 1/throughput
- Issue latency: the minimum interval (in cycles) between two consecutive instructions
  - Equivalent to reciprocal throughput (see Agner Fog's lists)

Operation	Integer		Single precision		Double precision	
	Latency	Issue	Latency	Issue	Latency	Issue
Addition	1	0.3	3	1.0	3	1.0
Multiplication	3	1.0	4	1.0	5	1.0
Division	11-21	5-13	10-15	6-11	10-23	6-19

Fully pipelined unit.

So … the impact of ILP?

- Can easily make errors when counting operations.

- Instructions are sequences of microoperations (uops)
  - Generated by the compiler
  - Reordered by the compiler AND the hardware
- Mapping:
  - Which instruction is mapped on which unit?
    - i.e., where does it execute?
- Scheduling
  - How are the instructions scheduled for execution?
    - i.e., when do they execute?

- Why do we care?!

An example

- Example:

  Compute: $y = a_0 + a_1 * x + a_2 * x * x + a_3 * x * x * x$

  Compute: $y = a_0 + a_1 * x + a_2 * x * x + a_3 * (x * x^2)$

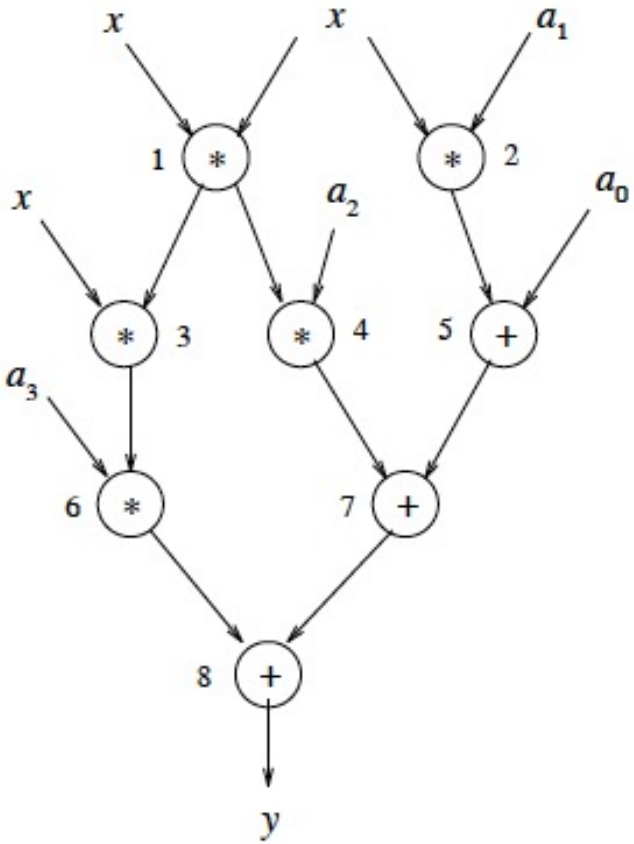
- Analytical performance model:

  Work: T = 5 x t_mul + 3 x t_add

  Span: 3 * + 1 +

- Task graph
  - Parallelism: 2, 3, 2, 1

- Scheduling for 2 execution units: *,+
  - Parallelism in practice: 1, 1, 2, 2, 1



Our simple models revisited

- What is the impact of DAGs, scheduling, and contention for our previous models?
  - Assumption?
  - Reality?
- Refine the model!
  - Scheduling of operations on cores / processing units …
    - ILP (instruction level parallelism) constraints
    - Memory and operations overlap
- … but VERY architecture dependent!
  - Understand the architecture in detail
  - Use a simulator
  - Use performance counters & microbenchmarking

https://software.intel.com/en-us/node/544393

Must do that by hand ?!

- Different tools to help out :
  - Static analysis of the code (with strong assumptions)
  - Mapping & scheduling simulators
  - More in-depth models (ECM – Execution-Cache-Model)
- Three examples:
  - IACA – from Intel
    - Check versions 2.1, 2.2, and 3.0
  - LLVM-MCA
  - OSACA – from FAU Erlangen*

*https://github.com/RRZE-HPC/OSACA

IACA - output

```
Throughput Analysis Report
--------------------------
Block Throughput: 3.00 Cycles       Throughput Bottleneck: Dependency chains
Loop Count:  28
Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------------------
|  Port  |   0   - DV   |   1   |   2   -  D   |   3   -  D   |   4   |   5   |   6   |   7   |
---------------------------------------------------------------------------------------
| Cycles |  1.5     0.0  |  1.5  |  1.0     1.0 |  1.0     1.0 |  0.0  |  1.5  |  1.5  |  0.0  |
---------------------------------------------------------------------------------------

DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3)
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion occurred
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of   |                    Ports pressure in cycles                     |      |
|  Uops    |  0  - DV    |  1   |  2  -  D    |  3  -  D    |  4   |  5   |  6   |  7   |
---------------------------------------------------------------------------------------
|   1      |             |      | 1.0     1.0 |             |      |      |      |      | movss xmm0, dword ptr [r11+rax*4]
|   1      |             |      |             |             |      | 1.0  |      |      | add rax, 0x1
|   2^     | 1.0         |      |             | 1.0     1.0 |      |      |      |      | mulss xmm0, dword ptr [r9]
|   1      |             |      |             |             |      | 0.5  | 0.5  |      | add r9, r10
|   1      |             | 0.5  |             |             |      |      | 0.5  |      | cmp esi, eax
|   1      |             | 1.0  |             |             |      |      |      |      | addss xmm1, xmm0
|   1      | 0.5         |      |             |             |      |      | 0.5  |      | jnle 0xfffffffffffffe0
Total Num Of Uops: 8
```