OoO Instruction Benchmarking Framework on the Back of Dragons

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Regionales Rechenzentrum Erlangen (RRZE)
Julian Hammer <julian.hammer@fau.de>, Georg Hager (advisor) <georg.hager@fau.de>, Gerhard Wellein (advisor) <gerhard.wellein@fau.de>

Artifact Available
git.io/fANPW

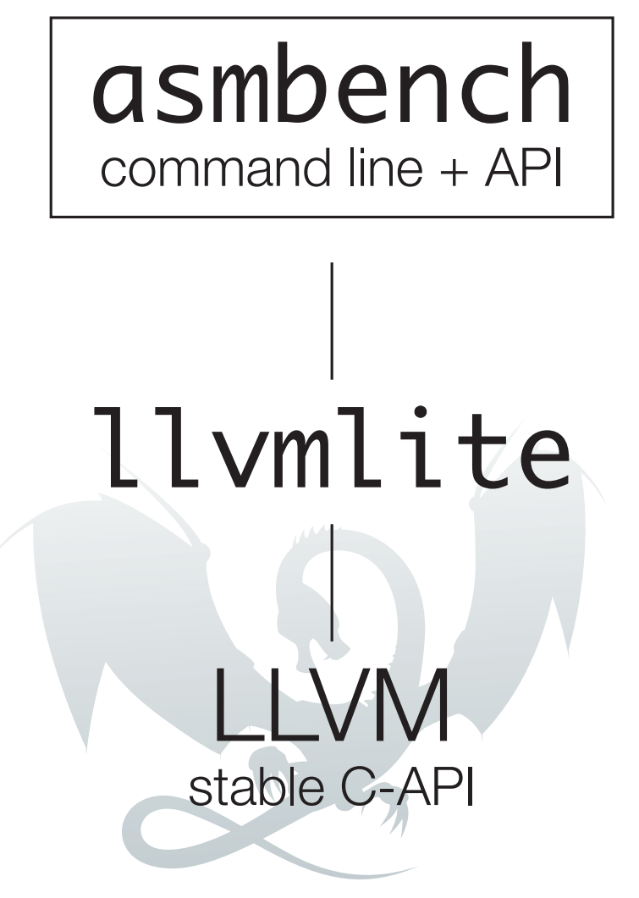
Goal

Construction of a framework for automatic throughput, latency and concurrency measurements, to generate input for an out-of-order (OoO) runtime prediction model, as a more versitile open-source replacement to Intel IACA, with support for non-Intel architectures.

Reciprocal throughput:	How often can an instruction be scheduled?
Latency:	How long until result becomes available?
Resource conflicts:	Which other instructions can not run in parallel?

Overview

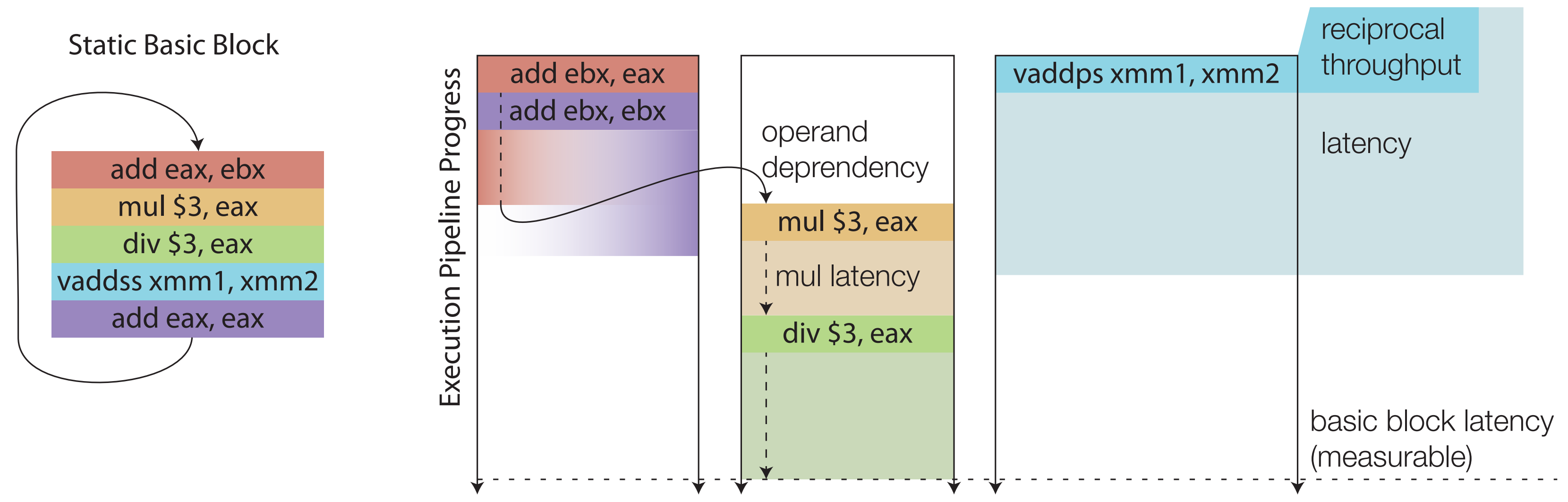
`asmbench` abstracts instructions, registers, immediates and memory operands, as well as dependency chains. It is based on the `llvmlite` library, which in turn is based on the stable LLVM C-API. `asmbench` offers a command line interface and a programming interface.



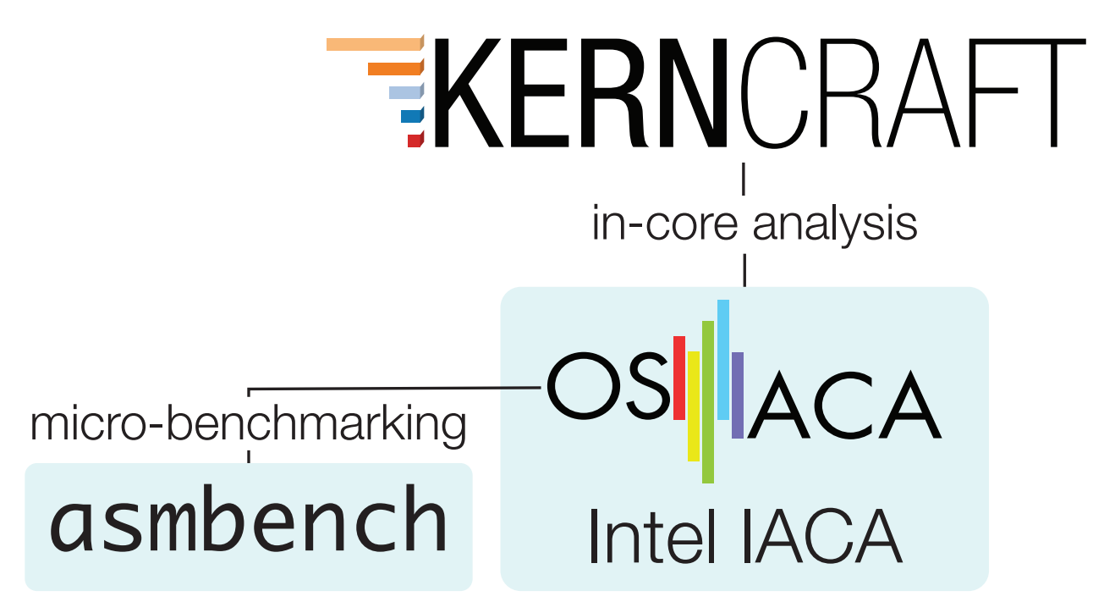
Underlying Hardware

`asmbench` assumes that

1. Instruction's maximum reciprocal throughput and latency is defined by executing hardware resources (e.g., integer ALU pipeline).
2. Data dependencies enforce serialization and lead to latency delays.
3. Perfect out-of-order scheduling.



Related Work

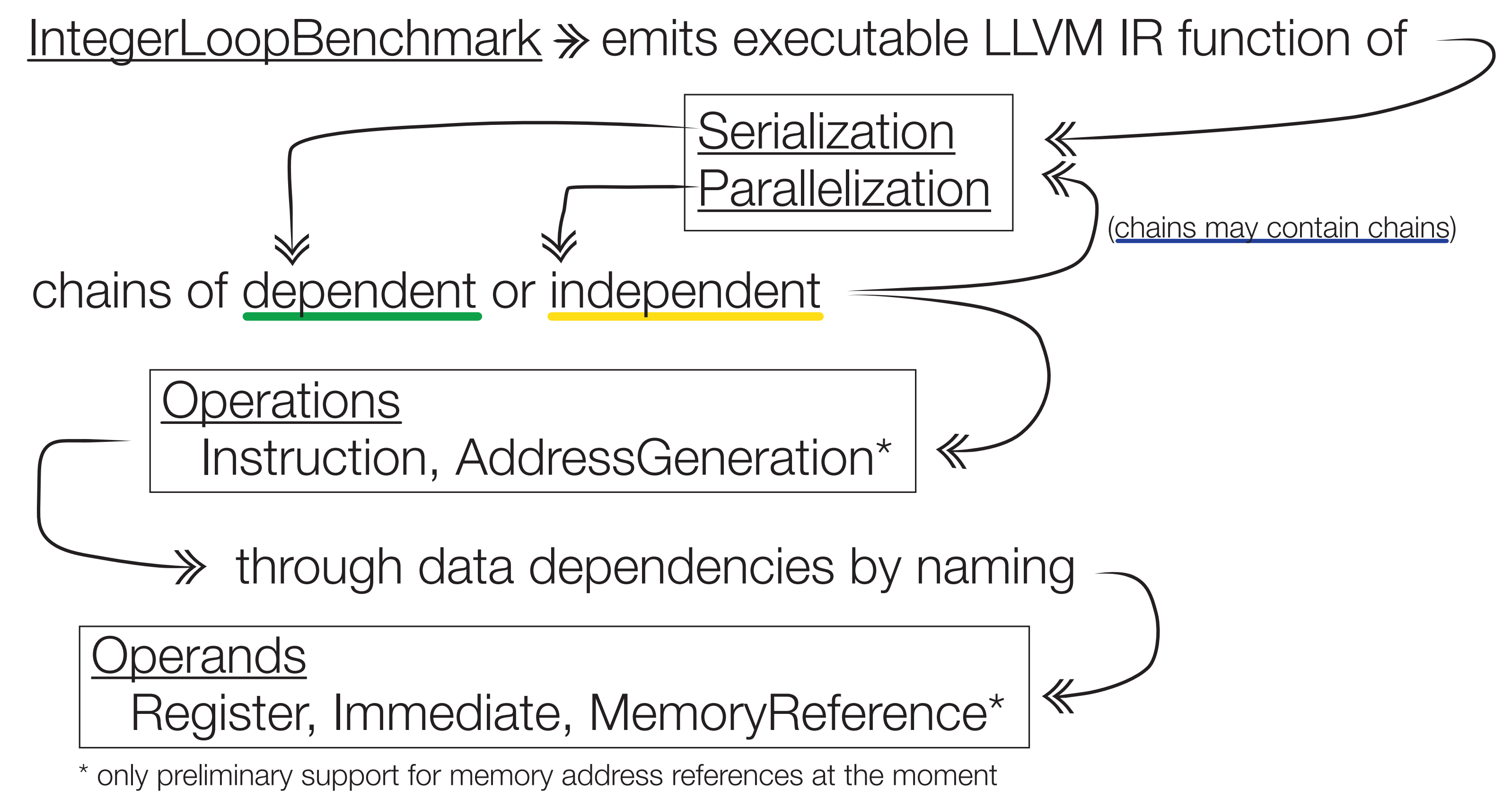


- To model non-Intel architectures *Kerncraft* [KC] will use *OSACA* [OS], which in-turn will use `asmbench` to construct micro-benchmarks for model creation.
- *Agner Fog's Instruction Tables* [AF] for throughput, latency, port mapping or many x86 models are not complete nor machine readable.
- *LLVM Exegesis* [EX] validates scheduling information of individual instructions, relies on Intel specific performance counter.
- *mubench* [MU] is an open source project for timing based instruction benchmarking. Development has seized in 2006, focused on x86.
- *likwid-bench* [LB] allows manually written assembly blocks to be executed in an enviornment suited for performance measurements.
- *Vendor documentation* [Intel], incomplete and occasionally incorrect.

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
Supported by
Federal Ministry of Education and Research
METACCA

ASM Representation Model

`asmbench` uses the following model to describe and construct benchmarks, and synthesise LLVM IR code:


IntegerLoopBenchmark ≫ emits executable LLVM IR function of
Serialization
Parallelization
(chains may contain chains)
chains of dependent or independent
Operations
Instruction, AddressGeneration*
≫ through data dependencies by naming
Operands
Register, Immediate, MemoryReference*
* only preliminary support for memory address references at the moment

Latency Benchmarks

To ensure latency bound execution, an instruction chain is used where the input of each instruction depends on the output of the previous one.

Throughput Benchmarks

We ensure throughput bound execution with at least eight non-depend instructions (at least as many as execution pipelines and stages).

Example: Latency and Throughput

To illustrate the potential, we chose 24 instructions and analyzed them on Intel Skylake (I7-6700HQ) and AMD Zen (EPYC 7451):

		Intel Skylake / I7-6700HQ						AMD Zen / EPYC 7451			
LLVM Naming	**pyasmjit String**	**Latency**	**[Intel]**	**[AF]**	**Throughput**	**[Intel]**	**[AF]**	**Latency**	**[AF]**	**Throughput**	**[AF]**
ADD32ri	add {src:i32:1}, {srcdst:i32:r}	1.00	1	1	0.30	0.25	0.25	1.00	1.00	0.28	0.25
ADD64ri32	add {src:i32:1}, {srcdst:i64:r}	1.00	1	1	0.30	0.25	0.25	1.00	1.00	0.29	0.25
INC64r	inc {srcdst:i64:r}	1.00	1	1	0.30	0.25	0.25	1.00	n/a	0.29	n/a
SUB32ri	sub {src:i32:1}, {srcdst:i64:r}	1.00	1	1	0.30	0.25	0.25	1.00	1.00	0.29	0.25
MOV64ri32	mov {src:i32:1}, {srcdst:i64:r}	0.41	n/a	0-1	0.27	n/a	0.25	0.38	n/a	0.26	n/a
VINSERTF128rr	vinsertf128 {src:i8:0}, {src:<2 x double>:x}, {src:<4 x double>:x}, {dst:<4 x double>:x}	3.00	3	3	1.00	1	1	1.00	1.00	0.74	0.5
VCVTSI642SSrr	vcvtsi2ss {src:i64:r}, {src:float:x}, {dst:float:x}	2.00	6	n/a	2.00	1	n/a	1.00	n/a	1.00	n/a
VADDPDYrr	vaddpd {src:<4 x double>:x}, {src:<4 x double>:x}, {dst:<4 x double>:x}	4.00	4	4	0.52	0.5	.5-1	3.00	3.00	1.00	1
VADDSDrr	vaddsd {src:double:x}, {src:double:x}, {dst:double:x}	4.00	n/a	4	0.52	n/a	0.5	3.00	3.00	0.50	0.5
VADDSSrr	vaddss {src:float:x}, {src:float:x}, {dst:float:x}	4.00	n/a	4	0.52	n/a	0.5	3.00	3.00	0.50	0.5
VFMADD213PDYr	vfmadd213pd {src:<4 x double>:x}, {src:<4 x double>:x}, {srcdst:<4 x double>:x}	4.00	n/a	n/a	0.52	n/a	0.5-1	5.00	5.00	1.00	1
VFMADD213PDr	vfmadd213pd {src:<2 x double>:x}, {src:<2 x double>:x}, {srcdst:<2 x double>:x}	4.00	n/a	n/a	0.52	n/a	0.5-1	5.00	5.00	0.61	1
VFMADD213PSYr	vfmadd213ps {src:<4 x double>:x}, {src:<4 x double>:x}, {srcdst:<4 x double>:x}	4.00	n/a	n/a	0.52	n/a	0.5-1	5.00	5.00	1.00	1
VFMADD213PSr	vfmadd213ps {src:<2 x double>:x}, {src:<2 x double>:x}, {srcdst:<2 x double>:x}	4.00	n/a	n/a	0.52	n/a	0.5-1	5.00	5.00	0.61	1
VFMADD213SDr	vfmadd213sd {src:double:x}, {src:double:x}, {srcdst:double:x}	4.00	n/a	n/a	0.52	n/a	0.5-1	5.00	5.00	0.61	1
VFMADD213SSr	vfmadd213ss {src:float:x}, {src:float:x}, {srcdst:float:x}	4.00	n/a	n/a	0.52	n/a	0.5-1	5.00	5.00	0.61	1
VMULPDYrr	vmulpd {src:<4 x double>:x}, {src:<4 x double>:x}, {dst:<4 x double>:x}	4.00	4	4	0.52	0.5	.5-1	4.00	4.00	1.00	1
VMULSDrr	vmulsd {src:double:x}, {src:double:x}, {dst:double:x}	4.00	n/a	n/a	0.52	n/a	0.5-1	4.00	4.00	0.55	0.5
VMULSSrr	vmulss {src:float:x}, {src:float:x}, {dst:float:x}	4.00	n/a	n/a	0.52	n/a	0.5-1	3.00	3.00	0.51	0.5
VSUBSDrr	vsubsd {src:double:x}, {src:double:x}, {dst:double:x}	4.00	n/a	4	0.52	n/a	0.5	3.00	3.00	0.50	0.5
VSUBSSrr	vsubss {src:float:x}, {src:float:x}, {dst:float:x}	4.00	n/a	n/a	0.52	n/a	.5-1	3.00	3.00	0.50	0.5
VDIVPDYrr	vdivpd {src:<4 x double>:x}, {src:<4 x double>:x}, {dst:<4 x double>:x}	13.01	14	13-14	8.00	8	8	8.00	8-13	8.00	8-9
VDIVSDrr	vdivsd {src:double:x}, {src:double:x}, {dst:double:x}	13.01	n/a	13-15	4.00	n/a	4	8.00	8-13	4.00	4-5
VDIVSSrr	vdivss {src:float:x}, {src:float:x}, {dst:float:x}	11.01	n/a	n/a	3.00	n/a	3-5	10.01	10.00	3.02	3

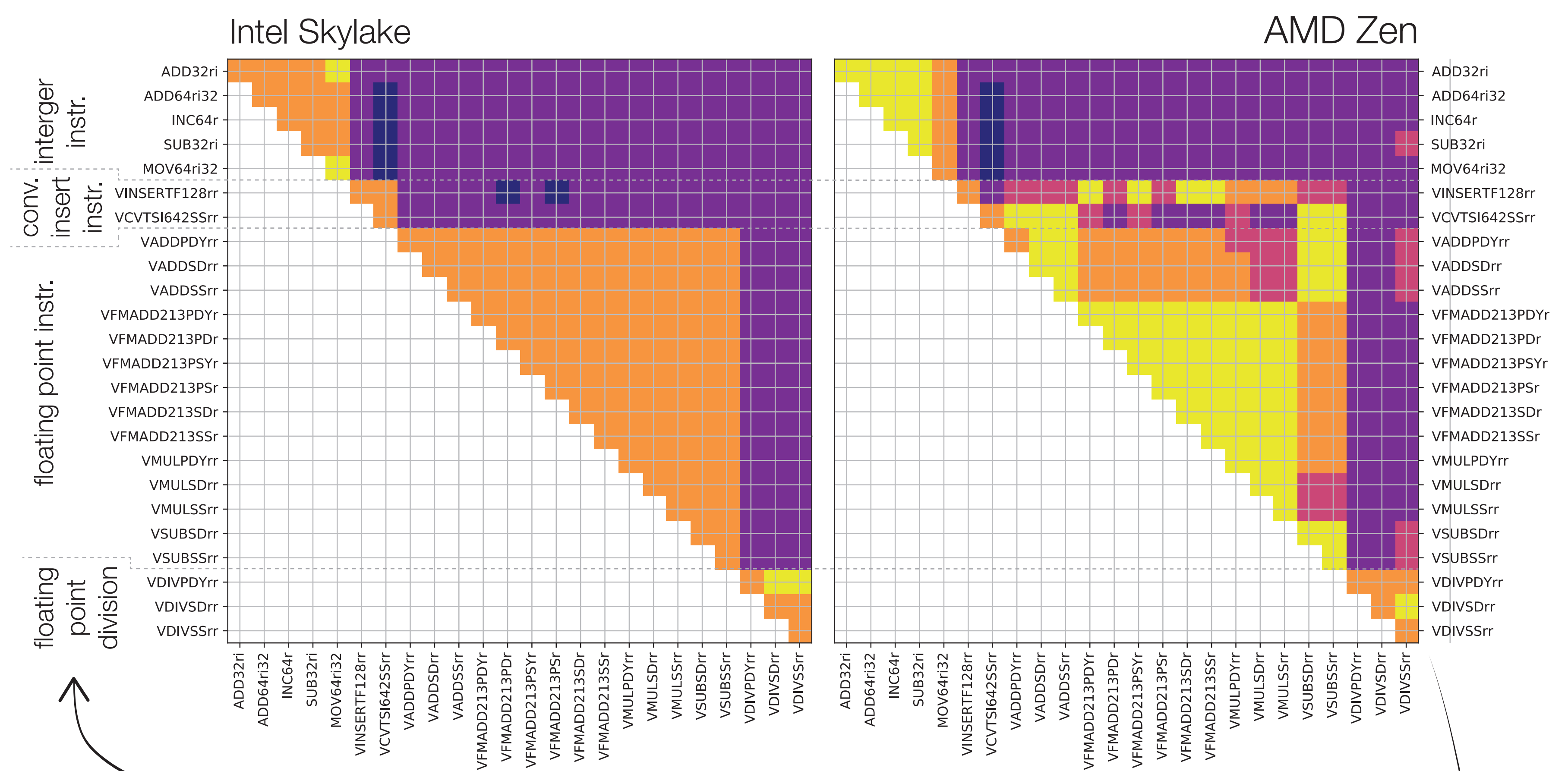
No information in / Faster measurement than / Slower measurement than } currently available resources [AF, Intel].

References
[AF] Fog; Instruction Tables, 2018. https://www.agner.org/optimize/instruction_tables.pdf
[EX] LLVM Exegesis. https://llvm.org/docs/CommandGuide/llvm-exegesis.html
[MU] mubench, last resease 2006: http://mubench.sourceforge.net/
[LB] Treibig et al.; likwid-bench: An extensible microbenchmarking platform for x86 multicore compute nodes; 2012
[Intel] Intel 64 and IA-32 Architectures Optimization Reference Manual; June 2016
[KC] Hammer et al.; Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels; 2016
[OS] Laukemann et al.; Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures; 2018

Resource Conflicts

To quantify the overlap of two distinct instructions, we use the following metric, with the reciprocal throughput of A as $\mathrm{TP}^{-1}(A)$:

$$\frac{\mathrm{TP}^{-1}(A+B) - \max(\mathrm{TP}^{-1}(A), \mathrm{TP}^{-1}(B))}{\min(\mathrm{TP}^{-1}(A), \mathrm{TP}^{-1}(B))} = \begin{cases} \gg 1 & \text{additional overhead} \\ \approx 1 & \text{no overlap conflict} \\ \approx 0 & \text{complete overlap} \\ \ll 0 & \text{elimination} \end{cases}$$



Four instruction groups form: integer, convert and insert, floating point (FP) and FP divisions. Each group overlaps with the others, but shows no overlap within. This is the basis for a concurrency model.

Load and Store in L1

Load latency and throughput behaviour is studied with pointer chasing. Both AMD Zen and Intel Skylake show a latency of 2 and reciprocal throughput of 0.5 cycles.

Stores are work-in-progress, since they are mostly "fire-and-forget", but occupy shared resources in the address generation units.

ISA Extraction

We parse LLVM's TableGen database to extract available instructions of an instruction set architecture (ISA). This works well for x86, but will require some adaptations for other architectures.

Future Work

- Support for other instruction set architectures (i.e., ARM, Power8)
- More flexible instruction serialization
- Combined load and instruction benchmarking
- Store benchmarks
- Parallel benchmarking for higher throughput

Try It Yourself

Disable frequency scaling and turbo mode, then run:

```
$ pip3 install --user asmbench[sc18src]==0.1.2
$ python3 -m asmbench.sc18src
```