Electrical & Computer ENGINEERING

Know Your Machine - Benchmarking

18-645: How to write fast code I

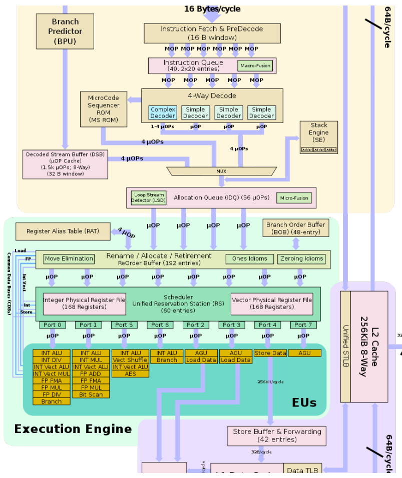
Tze Meng Low

1

Electrical & Computer ENGINEERING

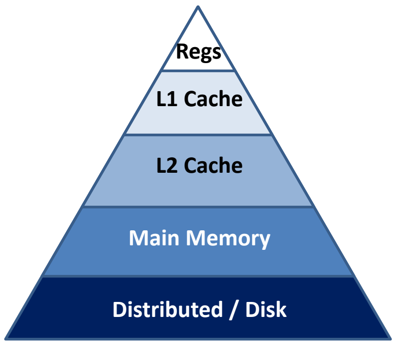
Recap: Two main parts – Execution & Data

Computation


16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
Instruction Queue (40, 2x20 entries)
MicroCode Sequencer ROM (MS ROM)
4-Way Decode
Complex Decoder
Simple Decoder
Stack Engine (SE)
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (32 B window)
MUX
Allocation Queue (IDQ) (56 µOPs)
Loop Stream Detector (LSD)
Register Alias Table (RAT)
Branch Order Buffer (BOB) (48-entry)
Rename / Allocate / Retirement ReOrder Buffer (192 entries)
Move Elimination
Ones Idioms
Zeroing Idioms
Integer Physical Register File (168 Registers)
Scheduler Unified Reservation Station (RS) (60 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
EUs
Execution Engine
Store Buffer & Forwarding (42 entries)
Data TLB
L2 Cache 256KiB 8-Way
Unified STLB
64B/cycle
To L3

https://en.wikichip.org/wiki/intel/microarchitectures/haswell

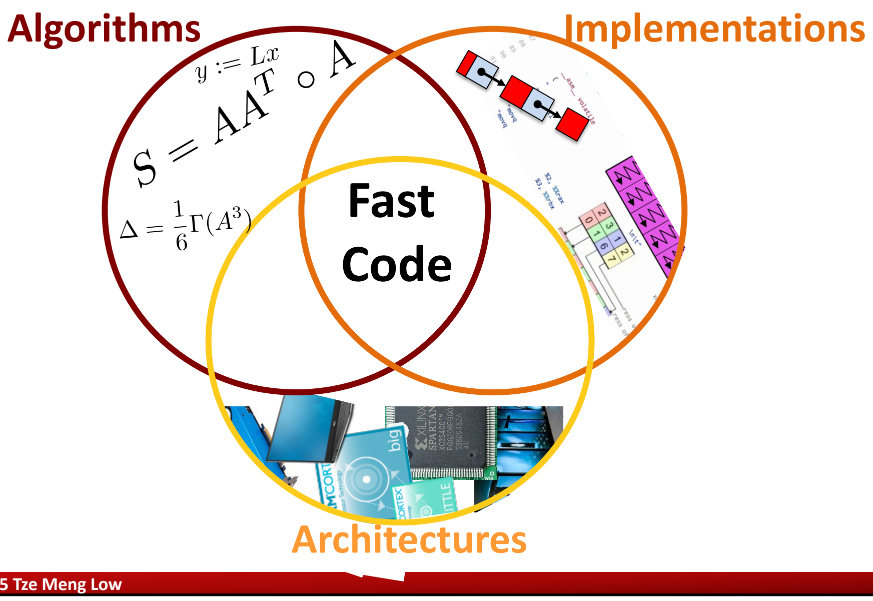
Data Movement



2

Electrical & Computer ENGINEERING

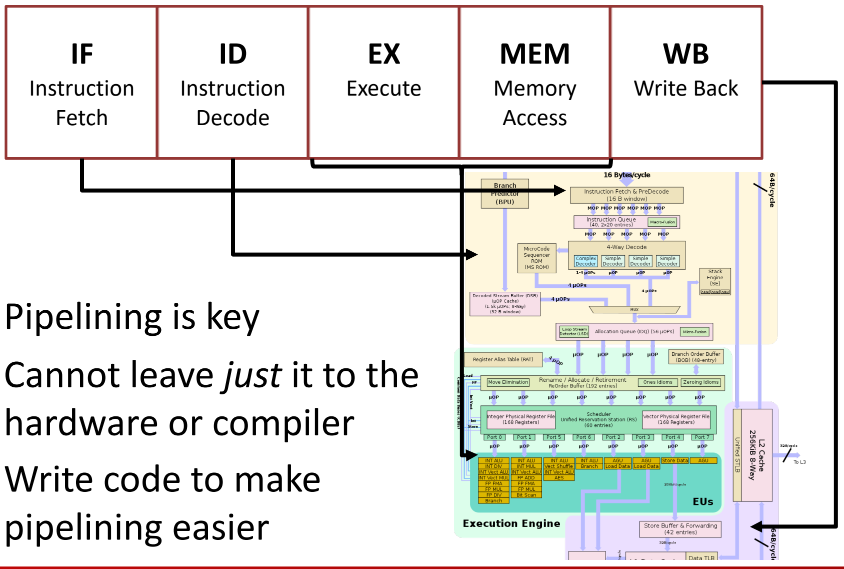
Recap: Same idea at different level



3

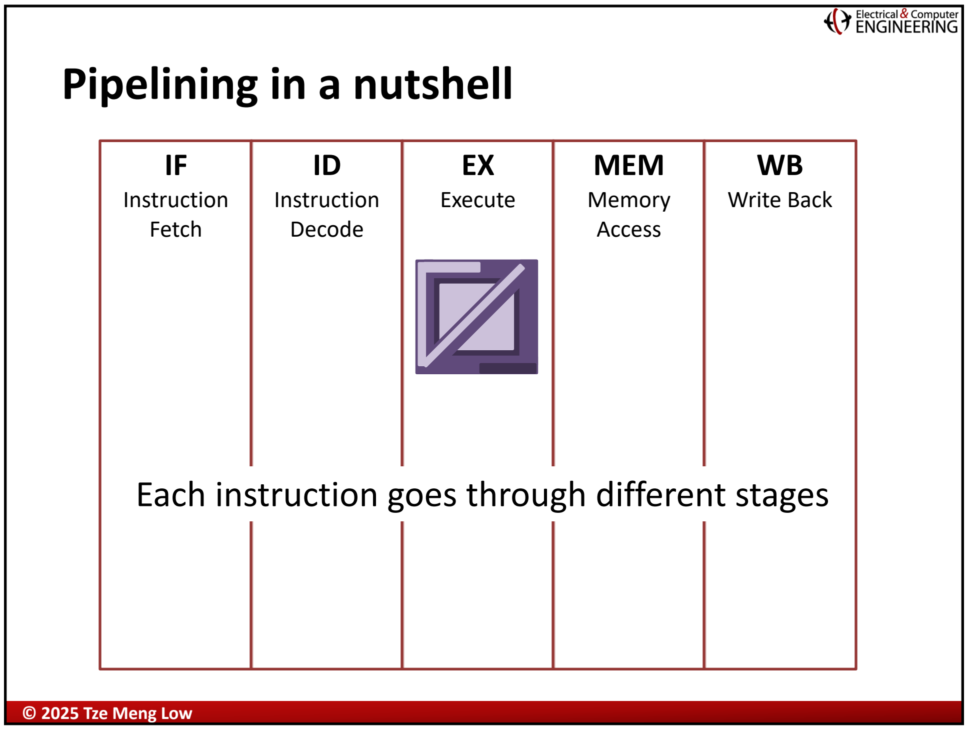
Electrical & Computer ENGINEERING

Model Instruction Pipeline



- Pipelining is key
- Cannot leave *just* it to the hardware or compiler
- Write code to make pipelining easier

4



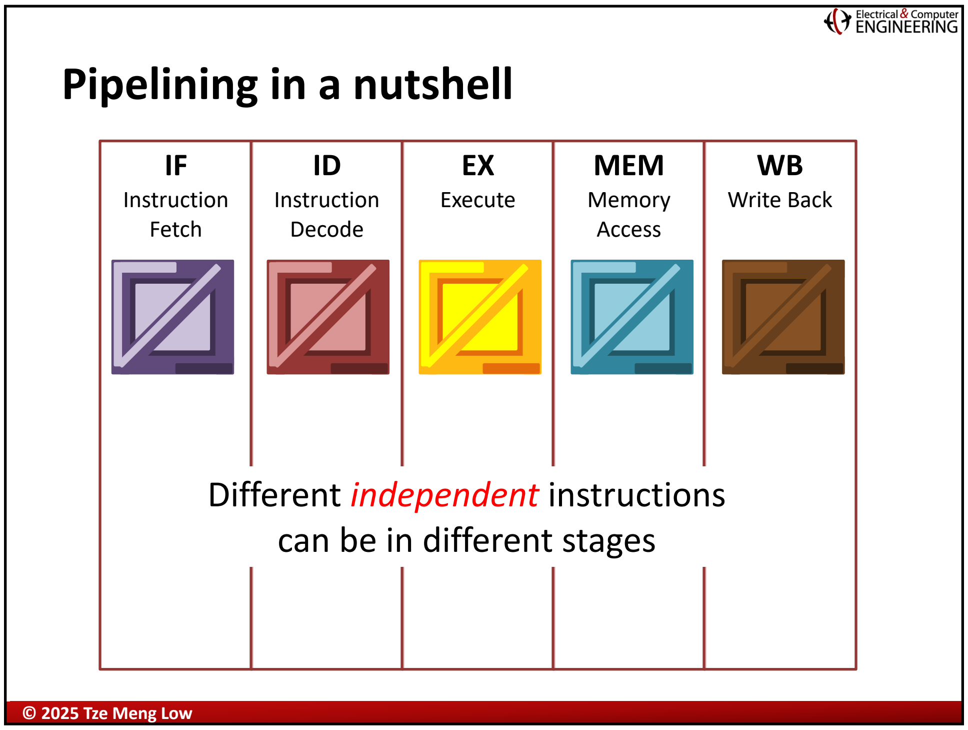
Electrical & Computer ENGINEERING

Pipelining in a nutshell

IF	ID	EX	MEM	WB
Instruction Fetch	Instruction Decode	Execute	Memory Access	Write Back

Each instruction goes through different stages

7



Electrical & Computer ENGINEERING

Pipelining in a nutshell

IF	ID	EX	MEM	WB
Instruction Fetch	Instruction Decode	Execute	Memory Access	Write Back

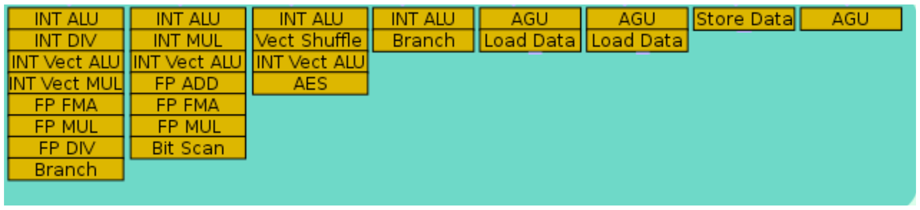
Different *independent* instructions can be in different stages

10

Electrical & Computer ENGINEERING

Many functional units

- Many specialized queues/pipelines in HW



- Goal of "Fast code" developer
  - Use as many pipelines as possible
  - Avoid stalling any of the pipelines
    - Find independent instructions

How many instructions do we need?

12

Electrical & Computer ENGINEERING

Benchmarking

- Want to know determine machine capability
  - Latency of instructions
    - Minimum # cycles before next *dependent* instruction
  - Instruction throughput
    - Instructions processed per clock cycle
- Benchmarking is empirical
  - It will be messy and time-consuming (Start HW early)
  - Do not expect perfect answers
  - Getting accurate timings is difficult
- ***Key assumption: Instruction is fully pipelined***

13

Timing

- Methods for timing
  - x86 (single-core performance)
    - Wall clock time `gettimeofday`
    - rdtsc (returns time stamp counter) `__rdtsc`

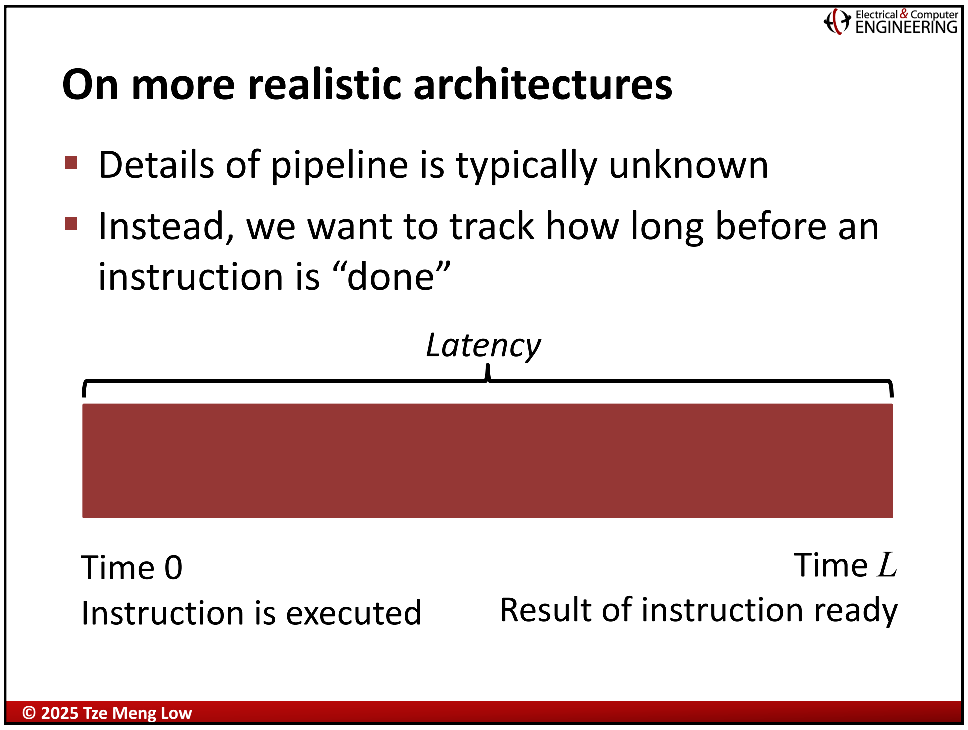
```
unsigned long long rdtsc()
{
  unsigned long long int x;
  unsigned a, d;

  __asm__ volatile("rdtsc" : "=a" (a), "=d" (d));

  return ((unsigned long long)a) | (((unsigned long long)d) << 32);
}
```

    - returns clock cycles at nominal frequency
  - Other architectures
    - Clock events on GPU (milliseconds)

Previously

IF Instruction Fetch	**ID** Instruction Decode	**EX** Execute	**MEM** Memory Access	**WB** Write Back

- Assumed that each stage take 1 time unit
  - 1 cycle or 1 clock tick
- Total of 5 independent instructions can be "in flight" or executed



On more realistic architectures

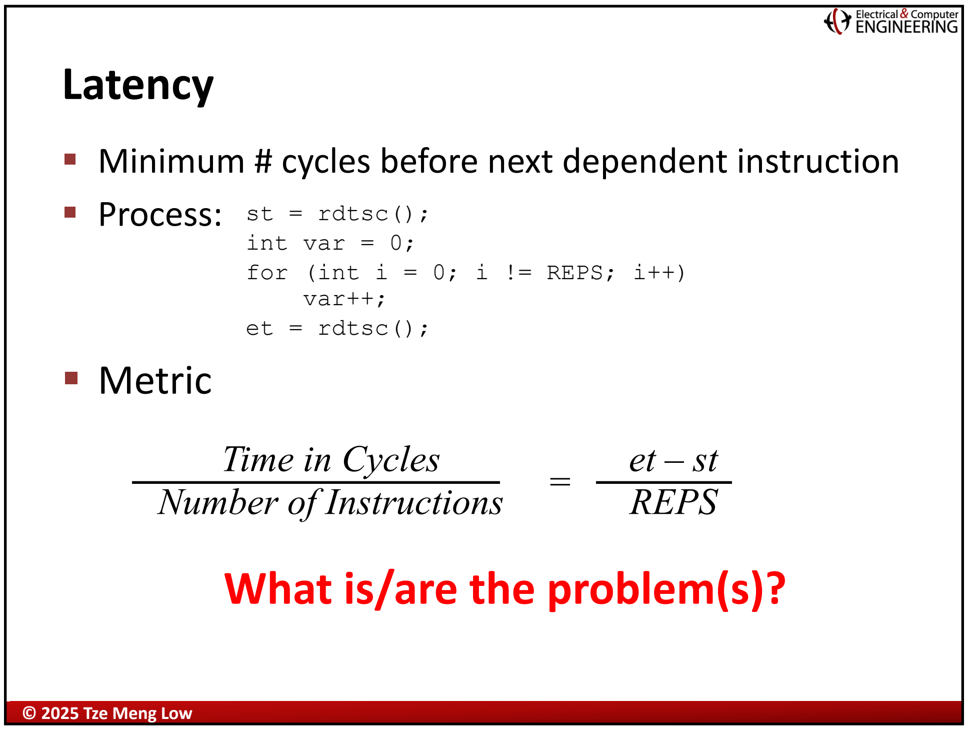
- Details of pipeline is typically unknown
- Instead, we want to track how long before an instruction is "done"

Latency

Time 0
Instruction is executed

Time L
Result of instruction ready

17



Latency

- Minimum # cycles before next dependent instruction
- Process:

```
st = rdtsc();
int var = 0;
for (int i = 0; i != REPS; i++)
    var++;
et = rdtsc();
```

- Metric

$$\frac{\text{Time in Cycles}}{\text{Number of Instructions}} = \frac{et - st}{REPS}$$

What is/are the problem(s)?

20

Electrical & Computer ENGINEERING

Multiple instructions for the same code

- Difficulties:
  - Compiler may not select desired instructions

```
float b = 1.0;
float a = 0.0;
a += b;
a += b;
…
```

addsd

fadd

Both are floating point addition.

CHECK YOUR ASSEMBLY!!

  - Compiler may optimize instructions out

```
float b = 1.0;
float a = 0.0;
a += b;
a += b;
…
a += b;
```

Compiler optimization

```
float b = 1.0;
float a = 100.0;
```

Mac OS

```
otool -tV demo.x
```

Linux

```
objdump -d demo.x
```

Electrical & Computer ENGINEERING

Macro Intrinsics

- Use C macros that hide inline assembly
- Write in macros instead

```
#define MULTIPLY(dest, src)
__asm__ __volatile__(
"imul %[rsrc], %[rdest]\n"
: [rdest] "+r"(dest)
: [rsrc] "r"(src)
);
```

Try not move code

Assembly to test

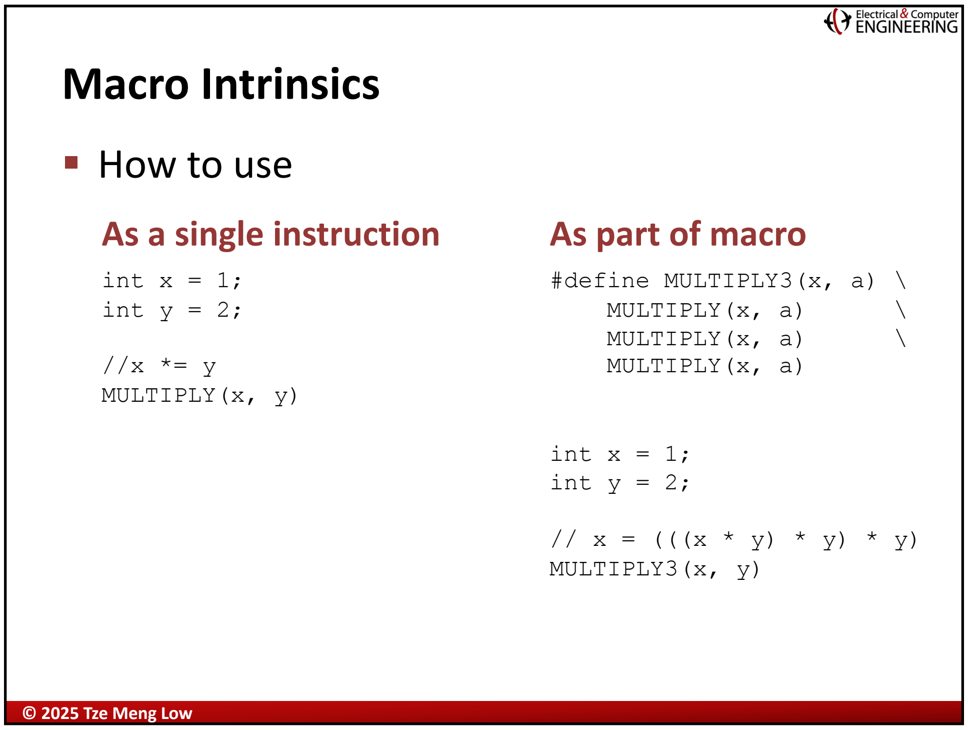
Destination is modified (+)
Destination is a register (r)

Input is read from register

Other operand descriptors

m	- memory
x	- SIMD register

Veras et al, Compilers, hands-off my hands-on optimizations, Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing, March 2016



Macro Intrinsics

- How to use

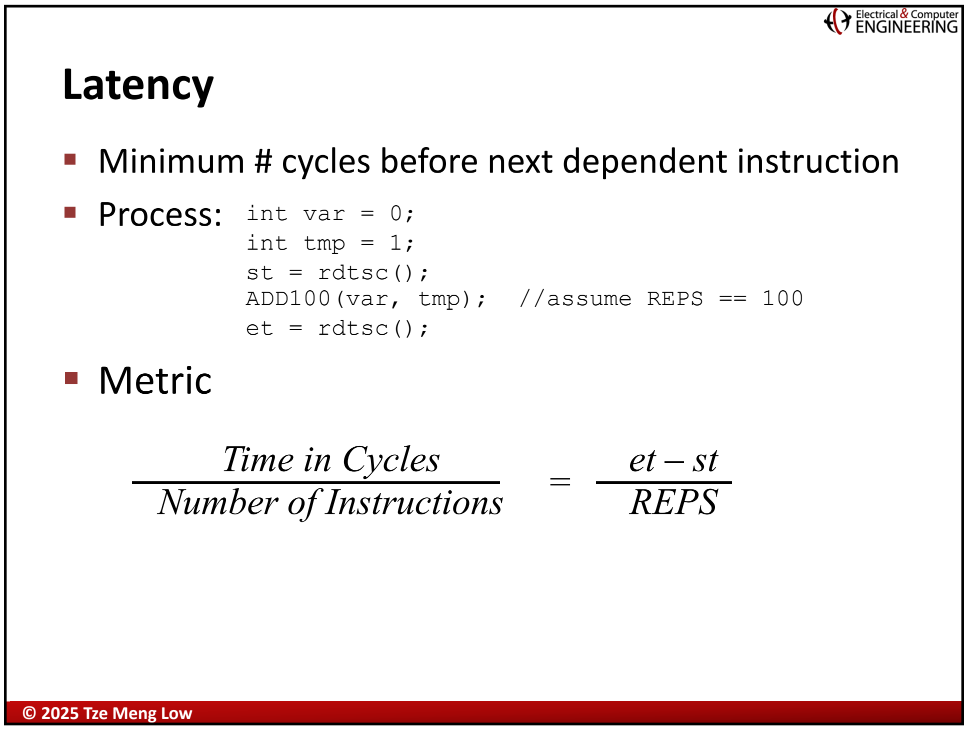
As a single instruction

```
int x = 1;
int y = 2;

//x *= y
MULTIPLY(x, y)
```

As part of macro

```
#define MULTIPLY3(x, a)  \
    MULTIPLY(x, a)       \
    MULTIPLY(x, a)       \
    MULTIPLY(x, a)


int x = 1;
int y = 2;

// x = (((x * y) * y) * y)
MULTIPLY3(x, y)
```

29



Latency

- Minimum # cycles before next dependent instruction
- Process:

```
int var = 0;
int tmp = 1;
st = rdtsc();
ADD100(var, tmp);   //assume REPS == 100
et = rdtsc();
```

- Metric

$$\frac{\textit{Time in Cycles}}{\textit{Number of Instructions}} = \frac{et - st}{REPS}$$

30

Electrical & Computer ENGINEERING

Latency

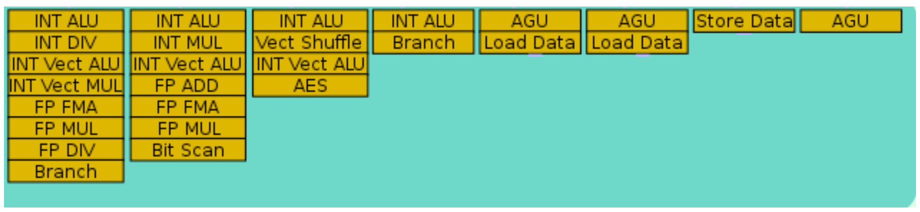
- Minimum # cycles before next dependent instruction
- Process:
  - Time a chain of dependent instructions
  - Divide time by number of instructions in chain
- Notes:
  - `rdtsc` may not be accurate for small cycle counts
    - Create long (> 100s – 1000s) chains
  - ***This is empirical. Don't expect exact numbers. If in doubt, round up***

31

Electrical & Computer ENGINEERING

Throughput

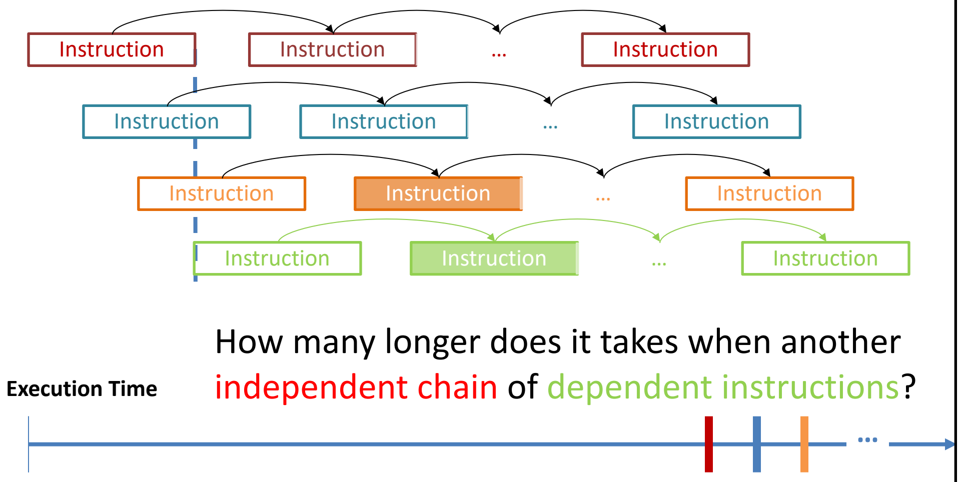
- Instructions processed per unit time
- Notice that some of the instructions can be processed by multiple functional units



32

Throughput

- Assume only 1 functional unit, 3 cycles latency



How many longer does it takes when another independent chain of dependent instructions?

47

Throughput benchmarking

- Process:
  - Time 1 chain of dependent instructions
  - Time increasing *interleaved* chains of dependent instructions
  - Find largest number of chains *before* execution time increases significantly
  - Compute throughput

$$Throughput = \frac{\#Instr.\ Per\ Chain \times \#Chains}{Time\ in\ Cycles}$$

48

Interleaving code is necessary

Don't assume that the compiler does it for you!!!

- HW has limited capability to interleave code
- Manual interleaving (use Macro intrinsics)

```
c += a[0] * b[0]
c += a[1] * b[1]
c += a[2] * b[2]
      ⋮
c += a[998] * b[998]
c += a[ 999] * b[999]
c += a[1000] * b[1000]
d += a[0] * b[0]
d += a[1] * b[1]
d += a[2] * b[2]
      ⋮
d += a[998] * b[998]
d += a[999] * b[999]
d += a[1000] * b[1000]
```

```
c += a[0] * b[0]
d += a[1] * b[1]
c += a[2] * b[2]
d += a[2] * b[2]
      ⋮

c += a[998] * b[998]
d += a[998] * b[998]
c += a[ 999] * b[999]
d += a[ 999] * b[999]
c += a[1000] * b[1000]
d += a[1000] * b[1000]
      ⋮
```

Throughput

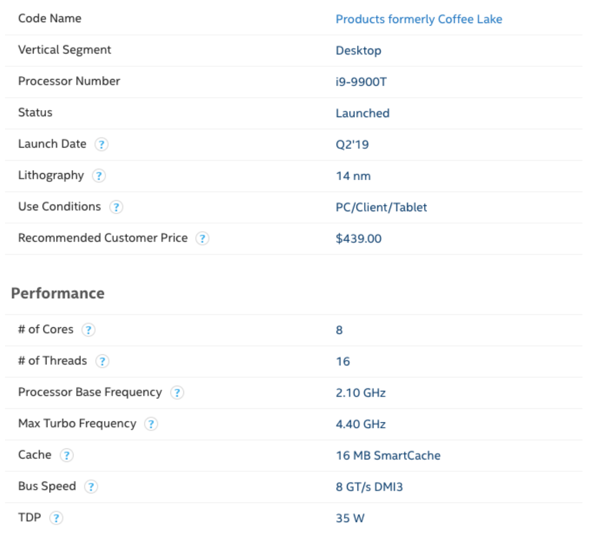
- Instructions processed per unit time

2 customers leave queues **every 6 minutes**

1 customer leave queue **every 3 minutes**

1 customers / 3 min

Hardware information on the web



Code Name	Products formerly Coffee Lake
Vertical Segment	Desktop
Processor Number	i9-9900T
Status	Launched
Launch Date ?	Q2'19
Lithography ?	14 nm
Use Conditions ?	PC/Client/Tablet
Recommended Customer Price ?	$439.00

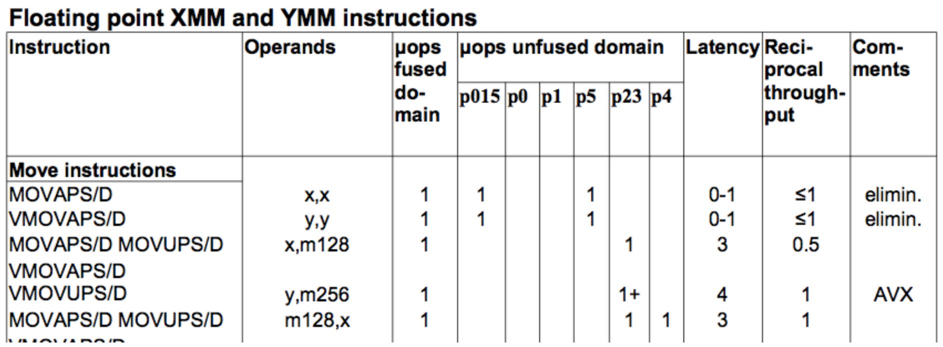
Performance

# of Cores ?	8
# of Threads ?	16
Processor Base Frequency ?	2.10 GHz
Max Turbo Frequency ?	4.40 GHz
Cache ?	16 MB SmartCache
Bus Speed ?	8 GT/s DMI3
TDP ?	35 W

https://ark.intel.com/content/www/us/en/ark/products/191044/intel-core-i9-9900t-processor-16m-cache-up-to-4-40-ghz.html

51

Hardware information on the web



Floating point XMM and YMM instructions

Instruction	Operands	μops fused domain	μops unfused domain						Latency	Reciprocal throughput	Comments
			p015	p0	p1	p5	p23	p4			
Move instructions											
MOVAPS/D	x,x	1	1			1			0-1	≤1	elimin.
VMOVAPS/D	y,y	1	1			1			0-1	≤1	elimin.
MOVAPS/D MOVUPS/D	x,m128	1					1		3	0.5	
VMOVAPS/D VMOVUPS/D	y,m256	1					1+		4	1	AVX
MOVAPS/D MOVUPS/D	m128,x	1					1	1	3	1	

WARNING! Many such sites, not all are accurate!
Same information, Reported differently
Similar but not identical architectures

https://uops.info/
https://www.agner.org/optimize/instruction_tables.pdf
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

52

Summary

- Benchmarking is important for knowing information we need for designing fast code
- Benchmarking is messy and is not precise
- These information can be found through the writing of small programs (micro-benchmark)