/INFOMOV/

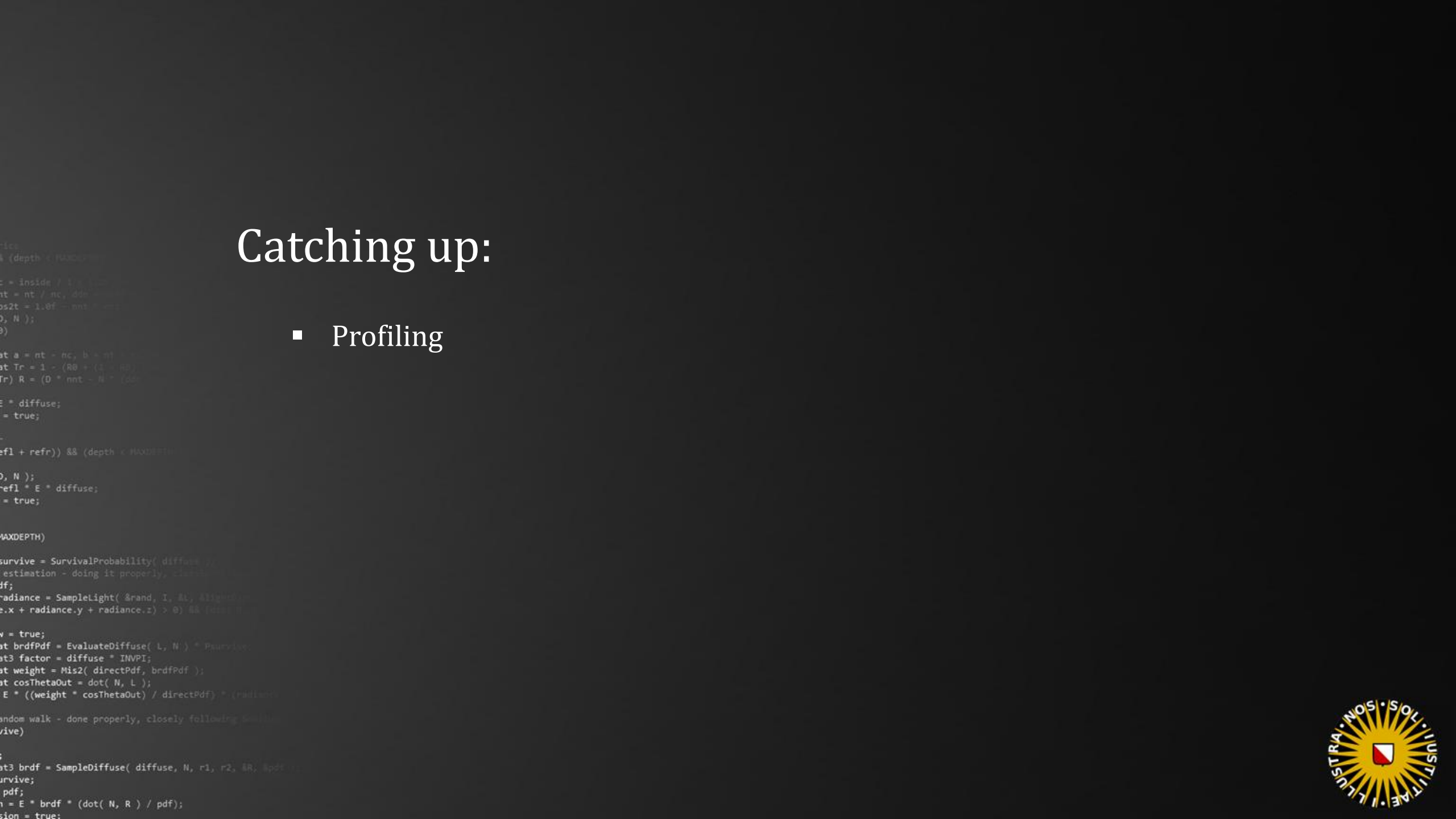
Optimization & Vectorization

J. Bikker - Sep-Nov 2015 - Lecture 2: “Low Level”

Welcome!

Catching up:

- Profiling



Never Assume

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
10. Report.

Do you actually need to speed it up?
By how much?

Things to consider:

- You have a finite amount of time for this
- You don’t want to break anything
- You don’t want to reduce maintainability

➔ Focus on ‘low hanging fruit’ – typically a small portion of the code.

Never Assume

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
10. Report.

Don’t trust your intuition

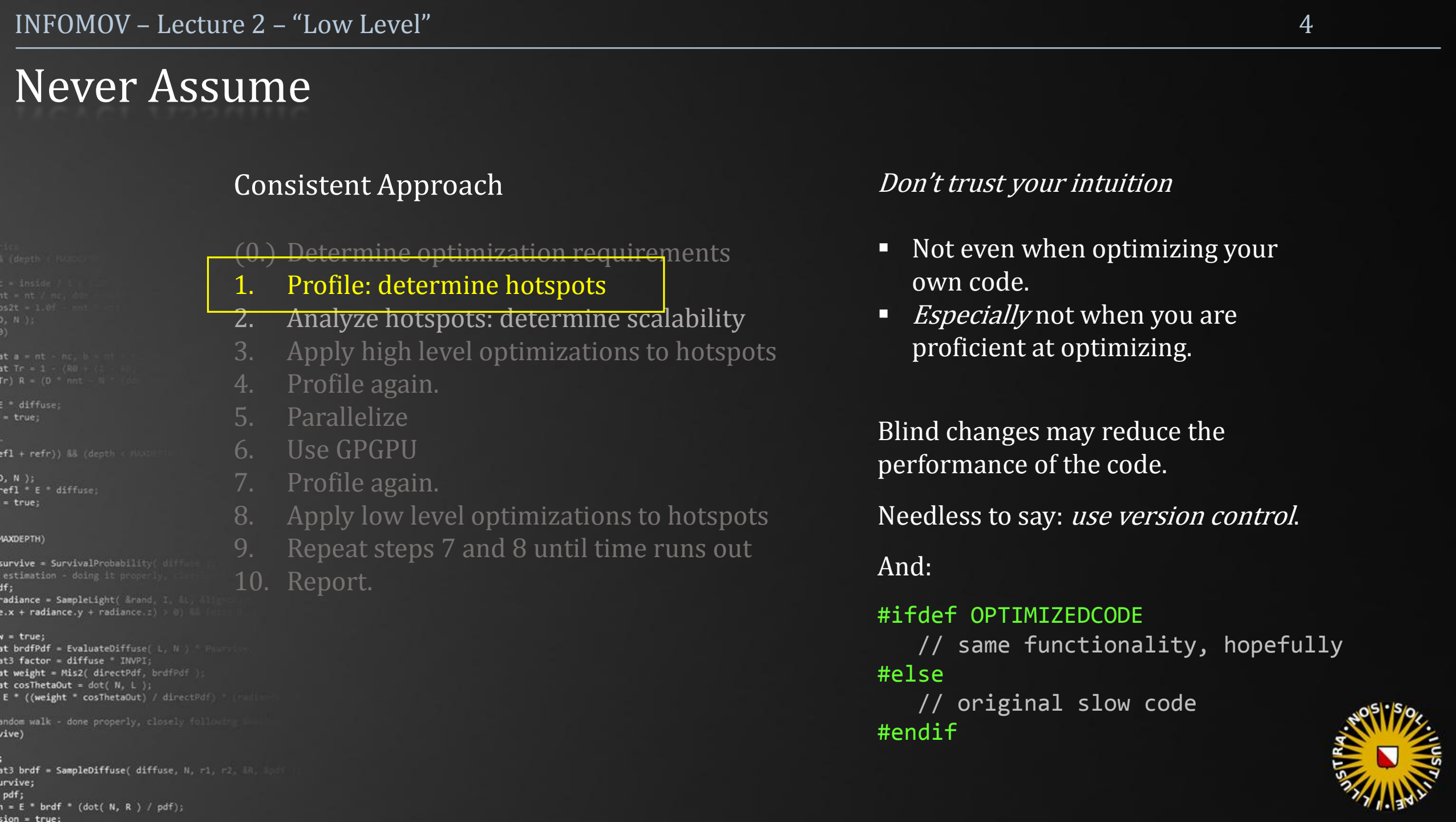
- Not even when optimizing your own code.
- *Especially* not when you are proficient at optimizing.

Blind changes may reduce the performance of the code.

Needless to say: *use version control*.

And:

```
#ifdef OPTIMIZEDCODE
   // same functionality, hopefully
#else
   // original slow code
#endif
```



Never Assume

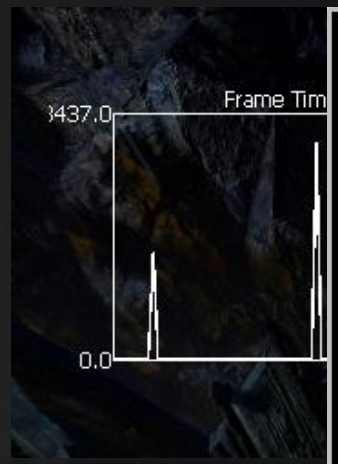
Profiling

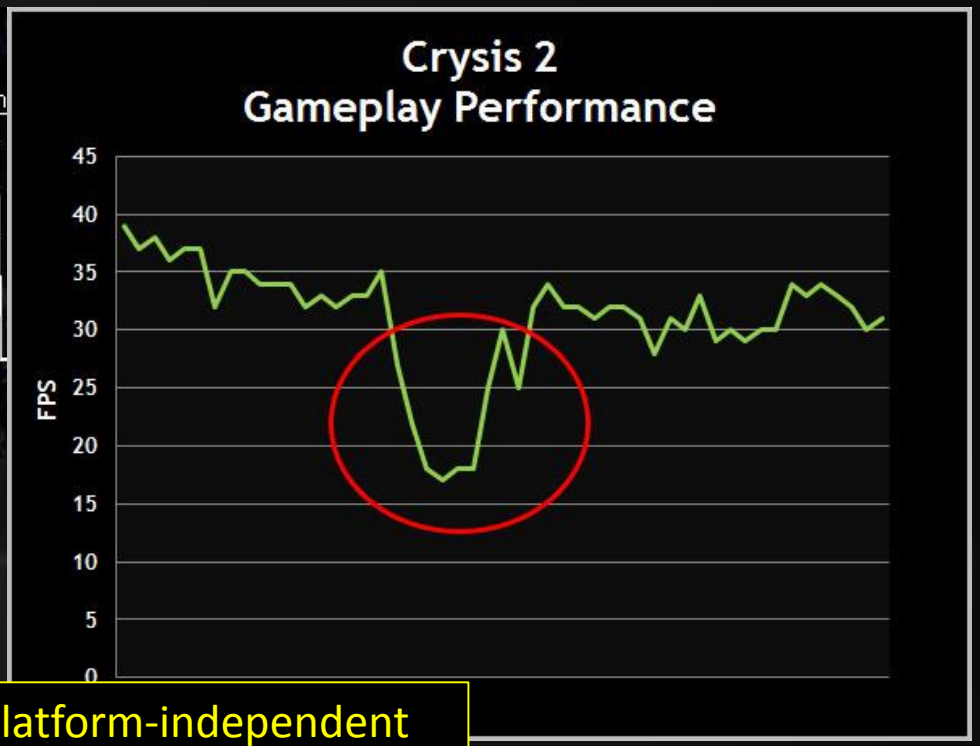
Measuring application performance

- Using external tools
- Using timers in the code

Measurements:

- How much time is spent were? (inclusive / exclusive, cycles, percentage)
- How often is each function called?
- Low level behavior: stalls / latencies, branch mispredictions, occupation, …
- Performance over time: lag, spikes, stutter





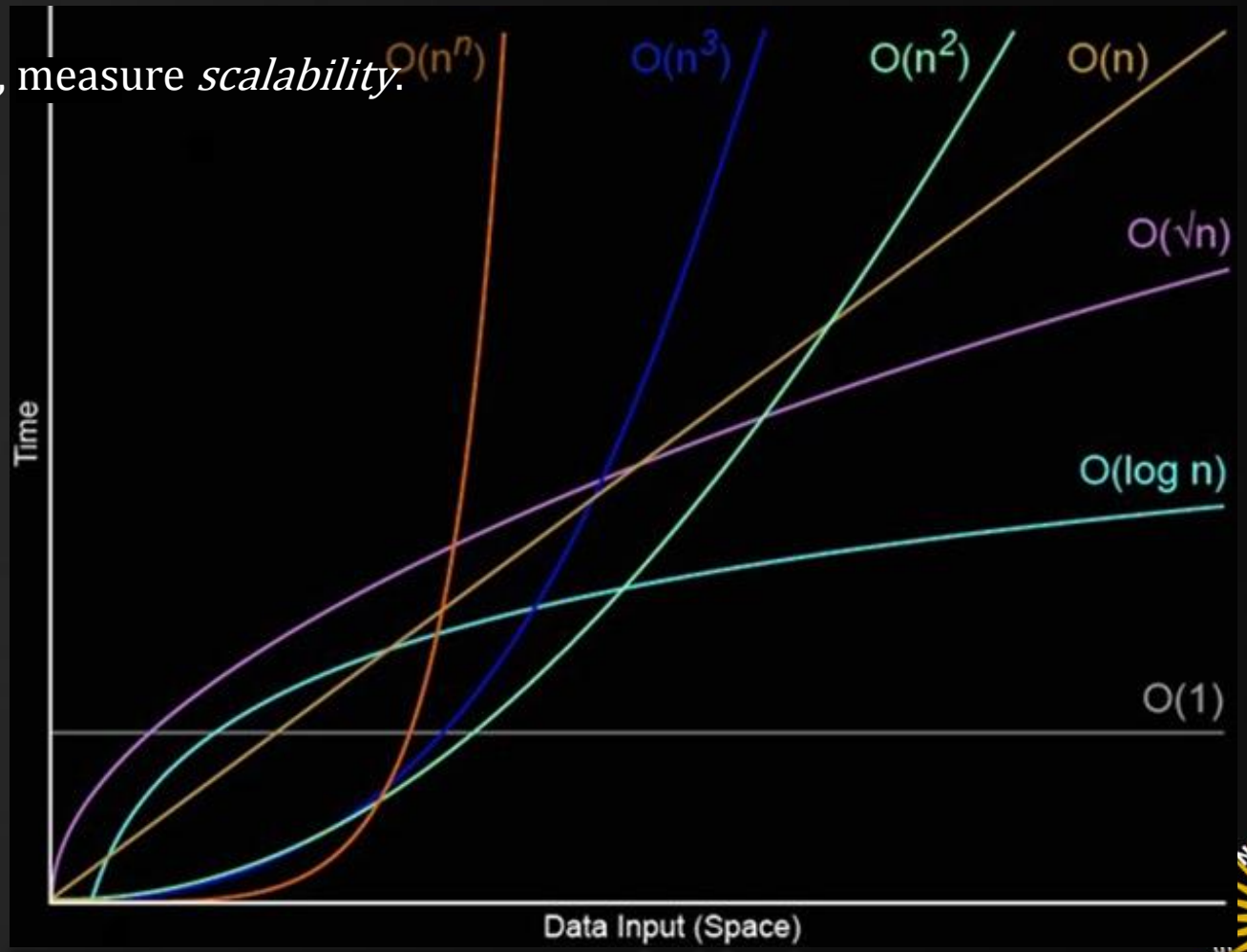
Platform-independent

Platform-dependent

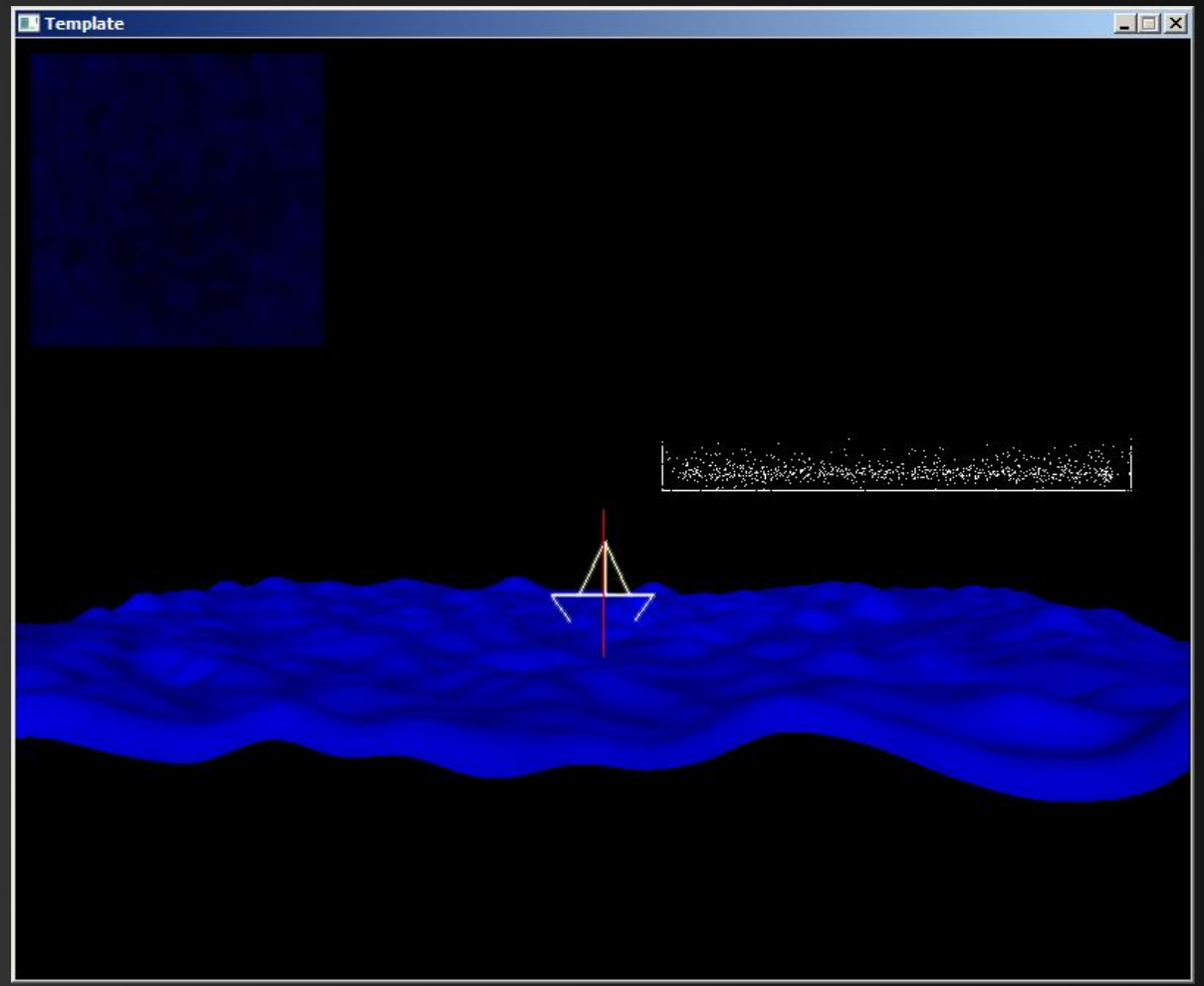
Never Assume

What if the goal is to have a 10x larger army in your RTS?

Don’t just measure performance, measure *scalability*.



Profiler Output



Never Assume

Profiling – getting accurate results

A profiler needs information about your code:
this is typically available in *debug* builds.

However:
Debug builds have very different performance characteristics, for many reasons. We need to profile in *release* mode.

Enabling debug information in release mode in Visual Studio:

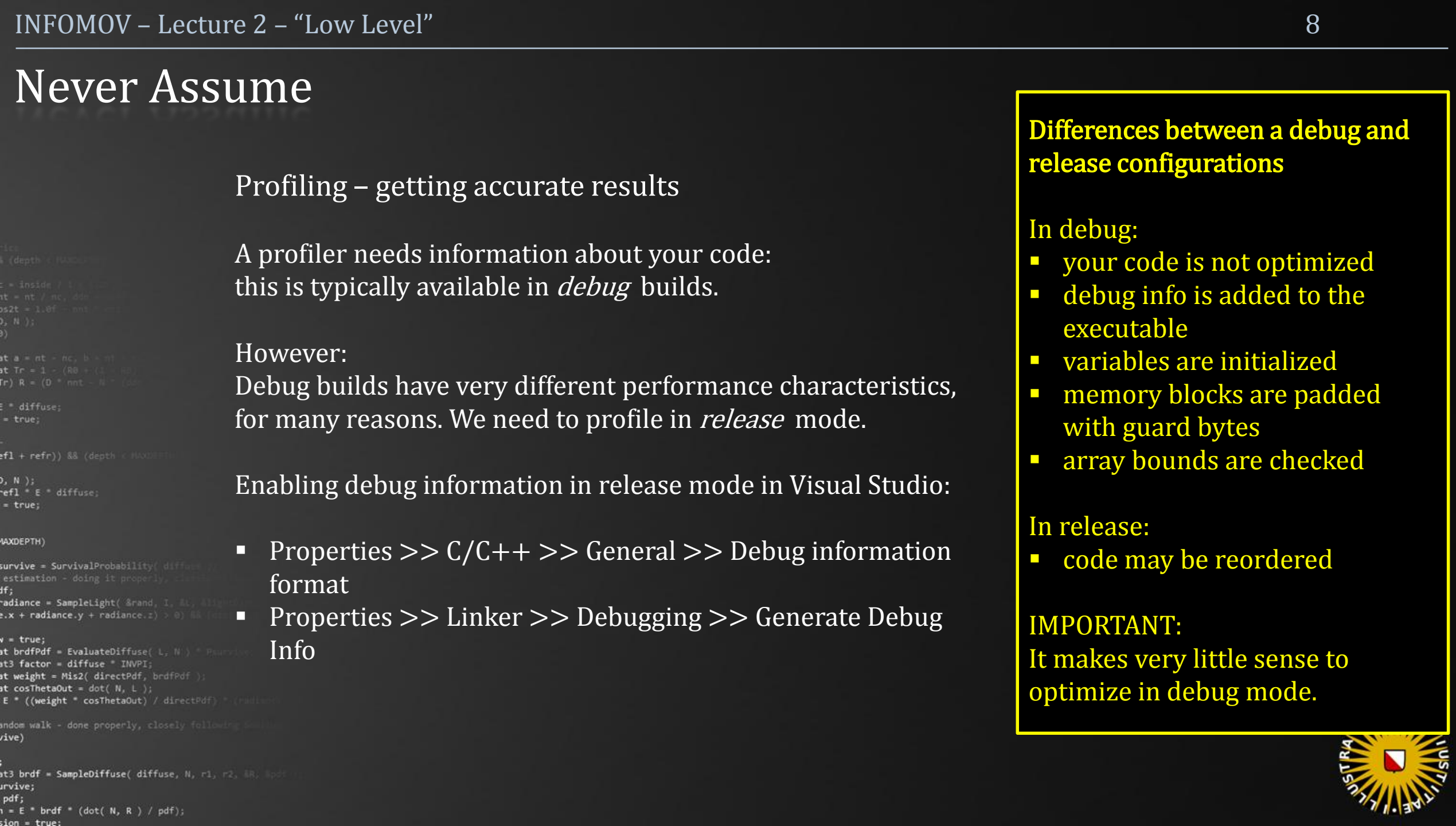
- Properties >> C/C++ >> General >> Debug information format
- Properties >> Linker >> Debugging >> Generate Debug Info

Differences between a debug and release configurations

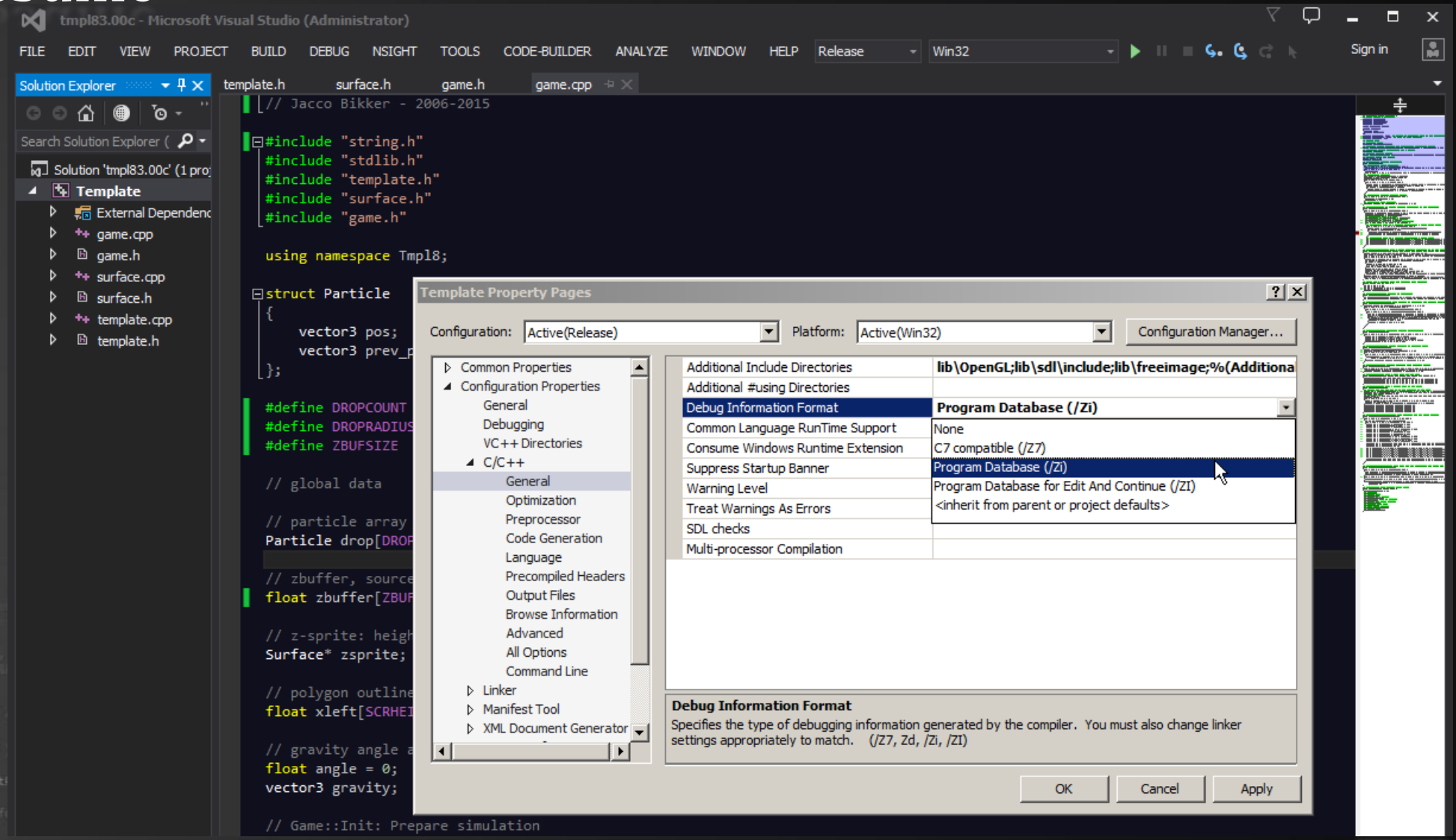
In debug:
- your code is not optimized
- debug info is added to the executable
- variables are initialized
- memory blocks are padded with guard bytes
- array bounds are checked

In release:
- code may be reordered

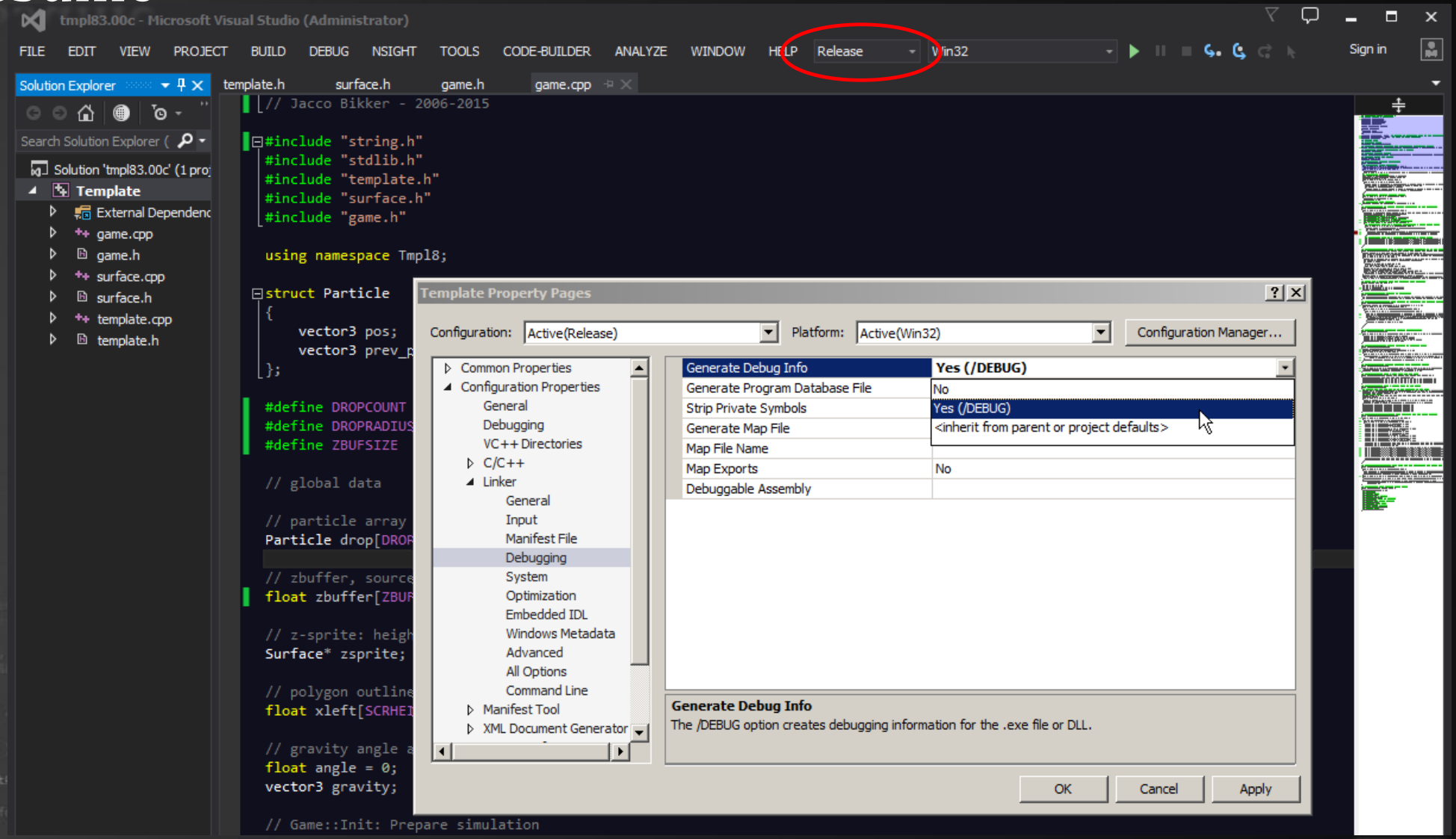
IMPORTANT:
It makes very little sense to optimize in debug mode.



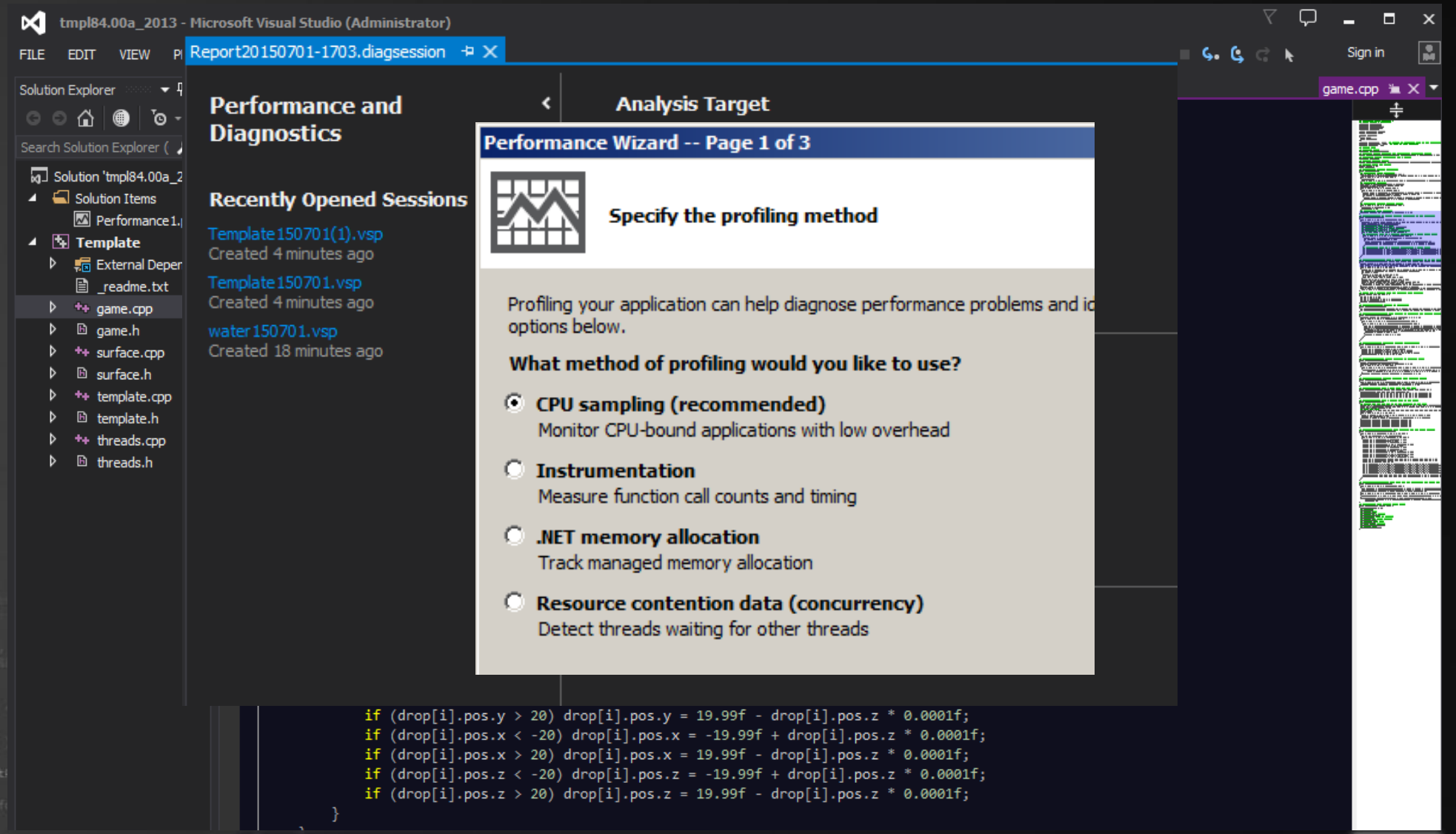
Never Assume



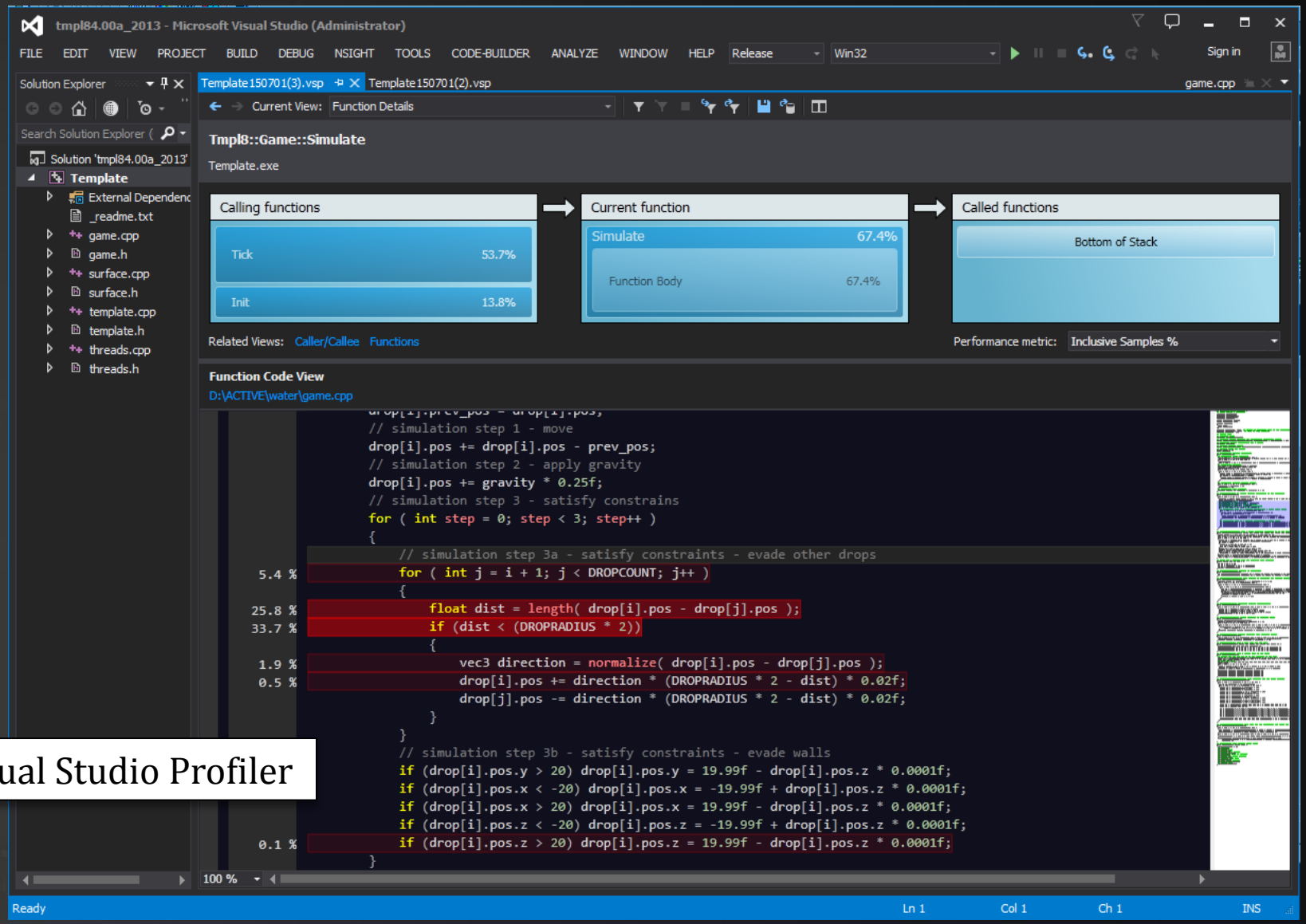
Never Assume



Tools

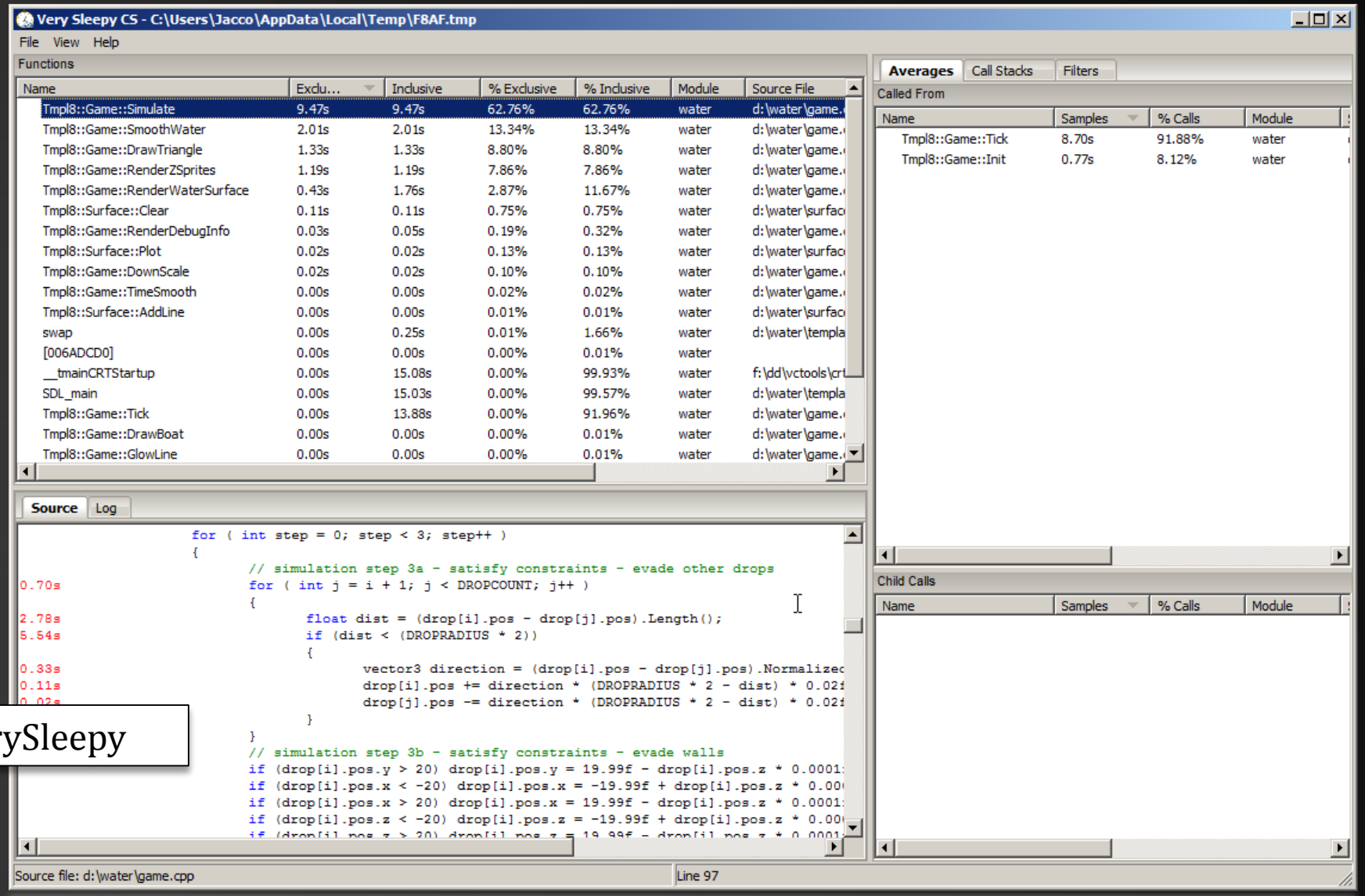

tmpl84.00a_2013 - Microsoft Visual Studio (Administrator)
FILE EDIT VIEW
Report20150701-1703.diagsession
Sign in
Solution Explorer
Search Solution Explorer
Solution 'tmpl84.00a_2
Solution Items
Performance1.
Template
External Deper
_readme.txt
game.cpp
game.h
surface.cpp
surface.h
template.cpp
template.h
threads.cpp
threads.h
Performance and Diagnostics
Recently Opened Sessions
Template150701(1).vsp
Created 4 minutes ago
Template150701.vsp
Created 4 minutes ago
water150701.vsp
Created 18 minutes ago
Analysis Target
Performance Wizard -- Page 1 of 3
Specify the profiling method
Profiling your application can help diagnose performance problems and ic
options below.
What method of profiling would you like to use?
CPU sampling (recommended)
Monitor CPU-bound applications with low overhead
Instrumentation
Measure function call counts and timing
.NET memory allocation
Track managed memory allocation
Resource contention data (concurrency)
Detect threads waiting for other threads
game.cpp
if (drop[i].pos.y > 20) drop[i].pos.y = 19.99f - drop[i].pos.z * 0.0001f;
if (drop[i].pos.x < -20) drop[i].pos.x = -19.99f + drop[i].pos.z * 0.0001f;
if (drop[i].pos.x > 20) drop[i].pos.x = 19.99f - drop[i].pos.z * 0.0001f;
if (drop[i].pos.z < -20) drop[i].pos.z = -19.99f + drop[i].pos.z * 0.0001f;
if (drop[i].pos.z > 20) drop[i].pos.z = 19.99f - drop[i].pos.z * 0.0001f;

Tools



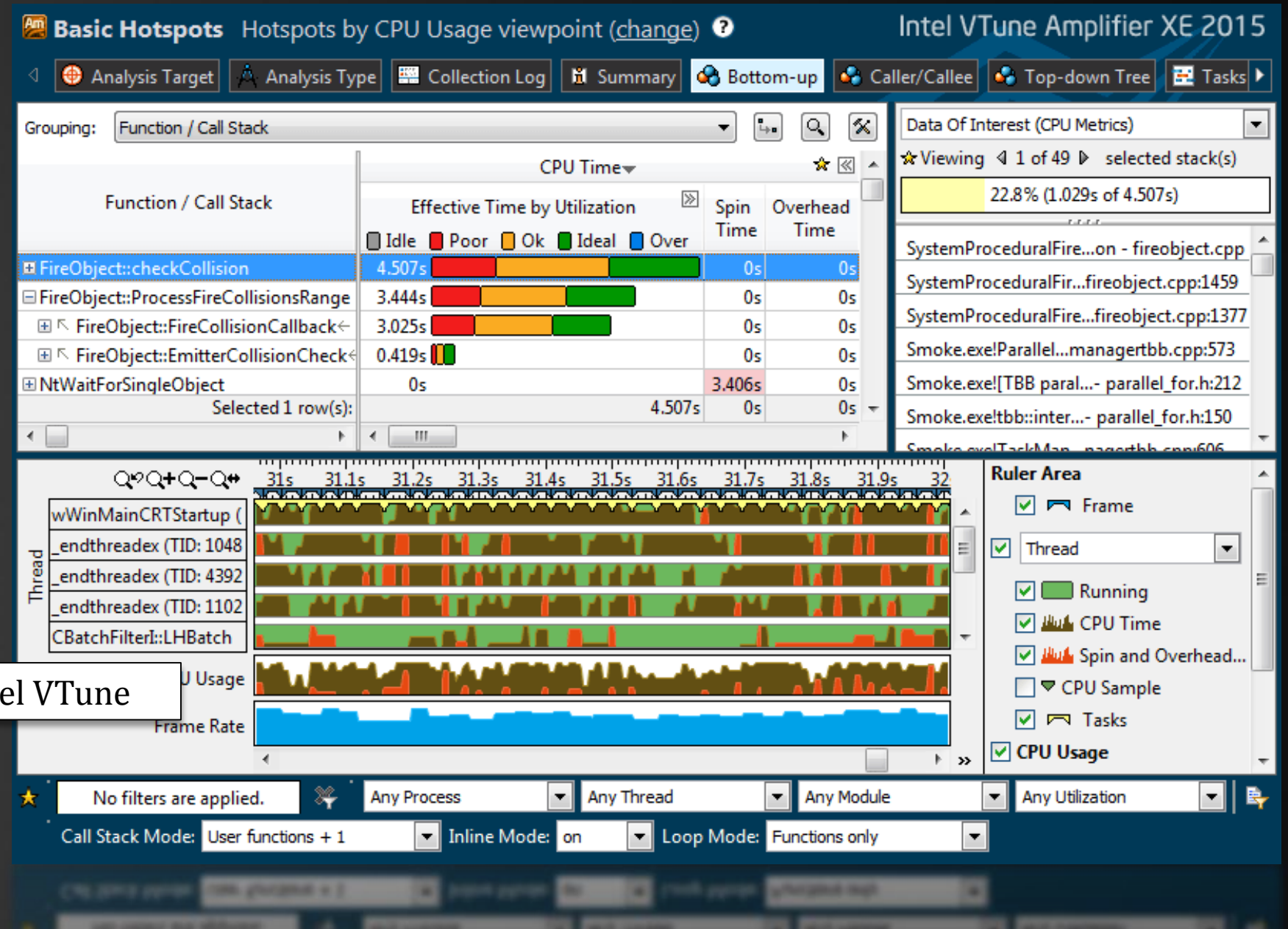
Visual Studio Profiler

Tools



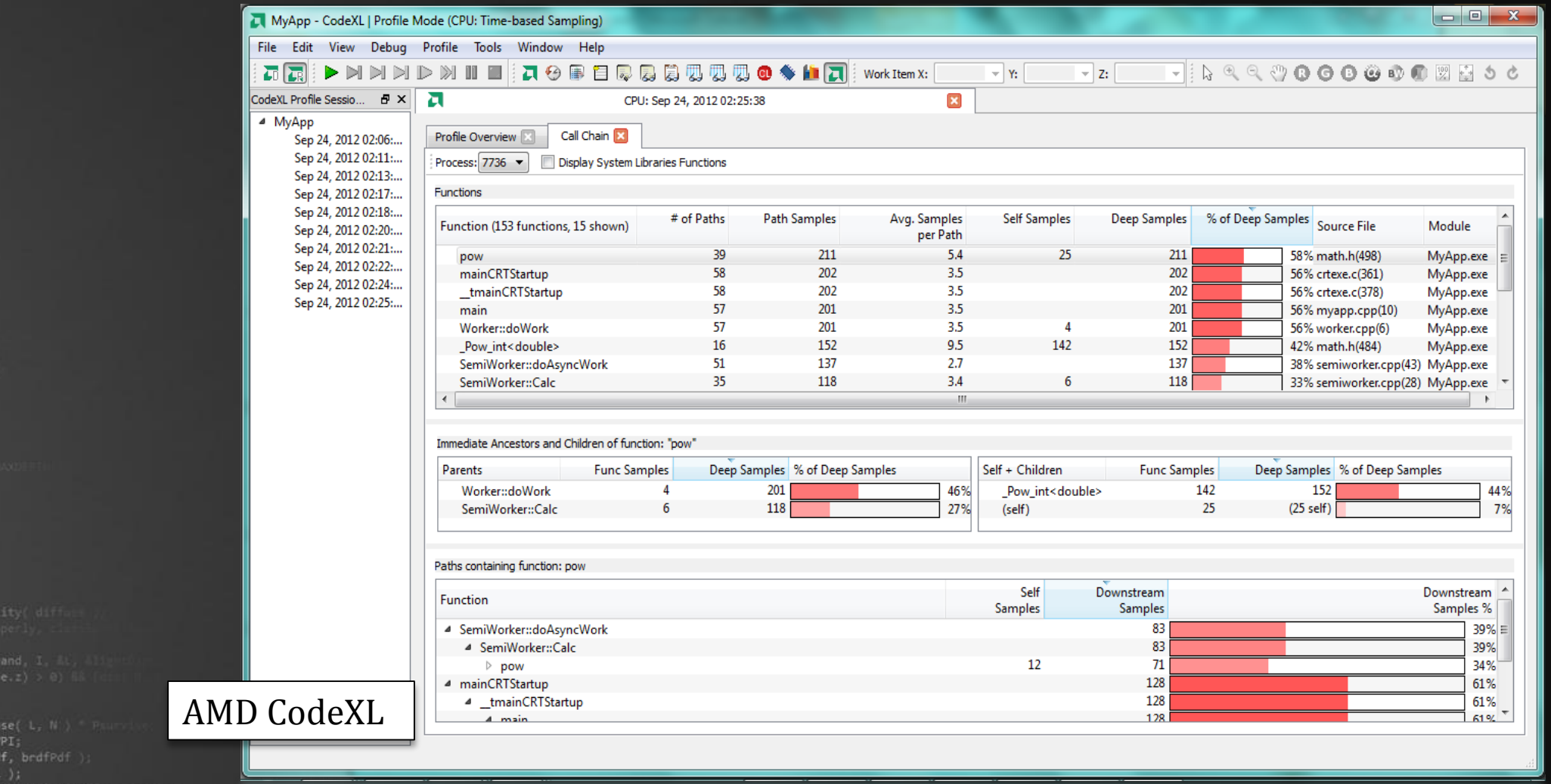
VerySleepy

Tools



Intel VTune

Tools


MyApp - CodeXL | Profile Mode (CPU: Time-based Sampling)
File Edit View Debug Profile Tools Window Help
CodeXL Profile Sessio...
MyApp
Sep 24, 2012 02:06:...
Sep 24, 2012 02:11:...
Sep 24, 2012 02:13:...
Sep 24, 2012 02:17:...
Sep 24, 2012 02:18:...
Sep 24, 2012 02:20:...
Sep 24, 2012 02:21:...
Sep 24, 2012 02:22:...
Sep 24, 2012 02:24:...
Sep 24, 2012 02:25:...
CPU: Sep 24, 2012 02:25:38
Profile Overview
Call Chain
Process: 7736
Display System Libraries Functions
Functions
Function (153 functions, 15 shown) # of Paths Path Samples Avg. Samples per Path Self Samples Deep Samples % of Deep Samples Source File Module
pow 39 211 5.4 25 211 58% math.h(498) MyApp.exe
mainCRTStartup 58 202 3.5 202 56% crtexe.c(361) MyApp.exe
__tmainCRTStartup 58 202 3.5 202 56% crtexe.c(378) MyApp.exe
main 57 201 3.5 201 56% myapp.cpp(10) MyApp.exe
Worker::doWork 57 201 3.5 4 201 56% worker.cpp(6) MyApp.exe
_Pow_int<double> 16 152 9.5 142 152 42% math.h(484) MyApp.exe
SemiWorker::doAsyncWork 51 137 2.7 137 38% semiworker.cpp(43) MyApp.exe
SemiWorker::Calc 35 118 3.4 6 118 33% semiworker.cpp(28) MyApp.exe
Immediate Ancestors and Children of function: "pow"
Parents Func Samples Deep Samples % of Deep Samples
Worker::doWork 4 201 46%
SemiWorker::Calc 6 118 27%
Self + Children Func Samples Deep Samples % of Deep Samples
_Pow_int<double> 142 152 44%
(self) 25 (25 self) 7%
Paths containing function: pow
Function Self Samples Downstream Samples Downstream Samples %
SemiWorker::doAsyncWork 83 39%
SemiWorker::Calc 83 39%
pow 12 71 34%
mainCRTStartup 128 61%
__tmainCRTStartup 128 61%
AMD CodeXL

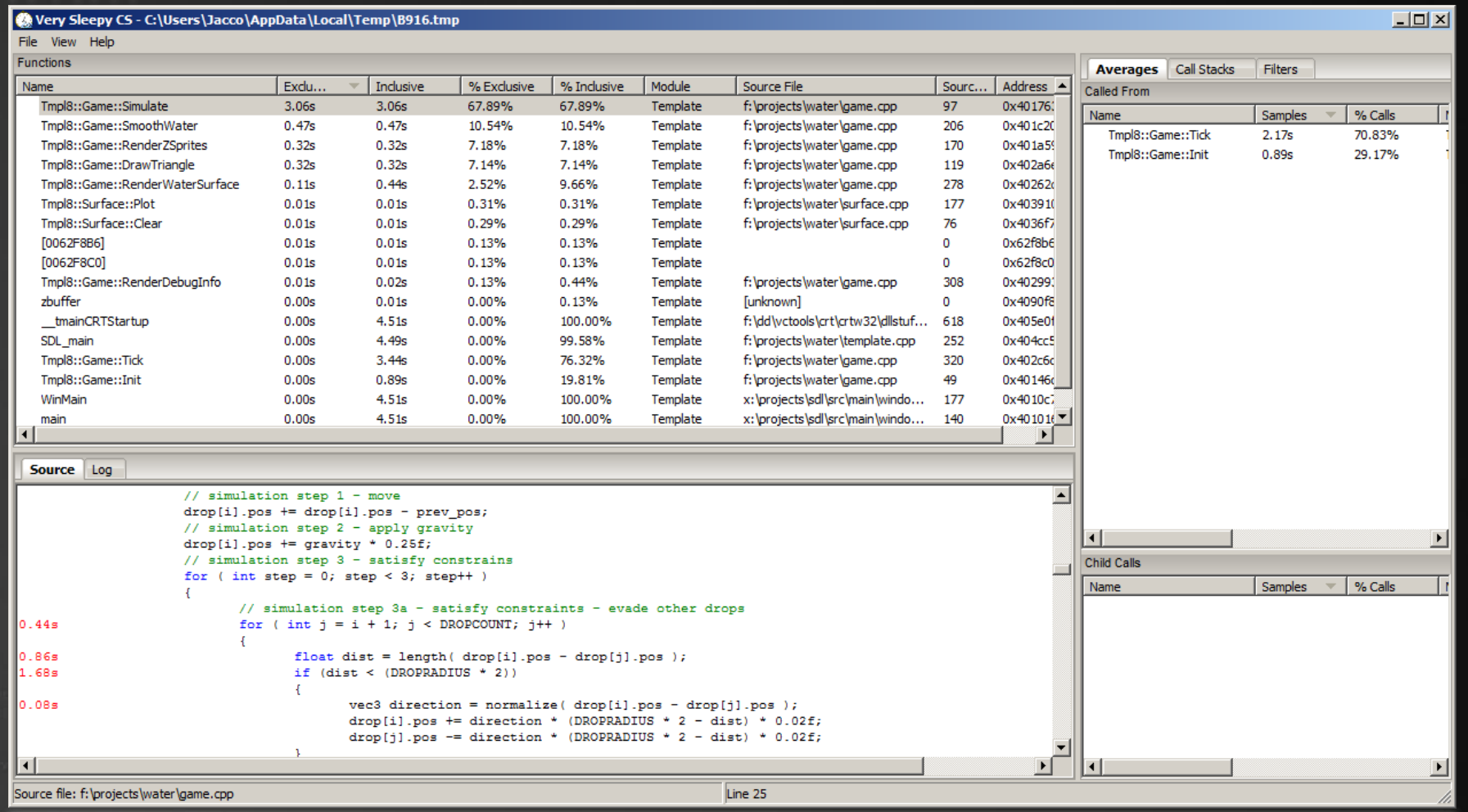
Never Assume

Take-away:

Never assume. Profiling *always* steers optimization.

Optimize in release mode. Enable debug info during this process. Don’t forget to turn it off before distribution.

Profiler Output



Profiler Output

Profiling – Results

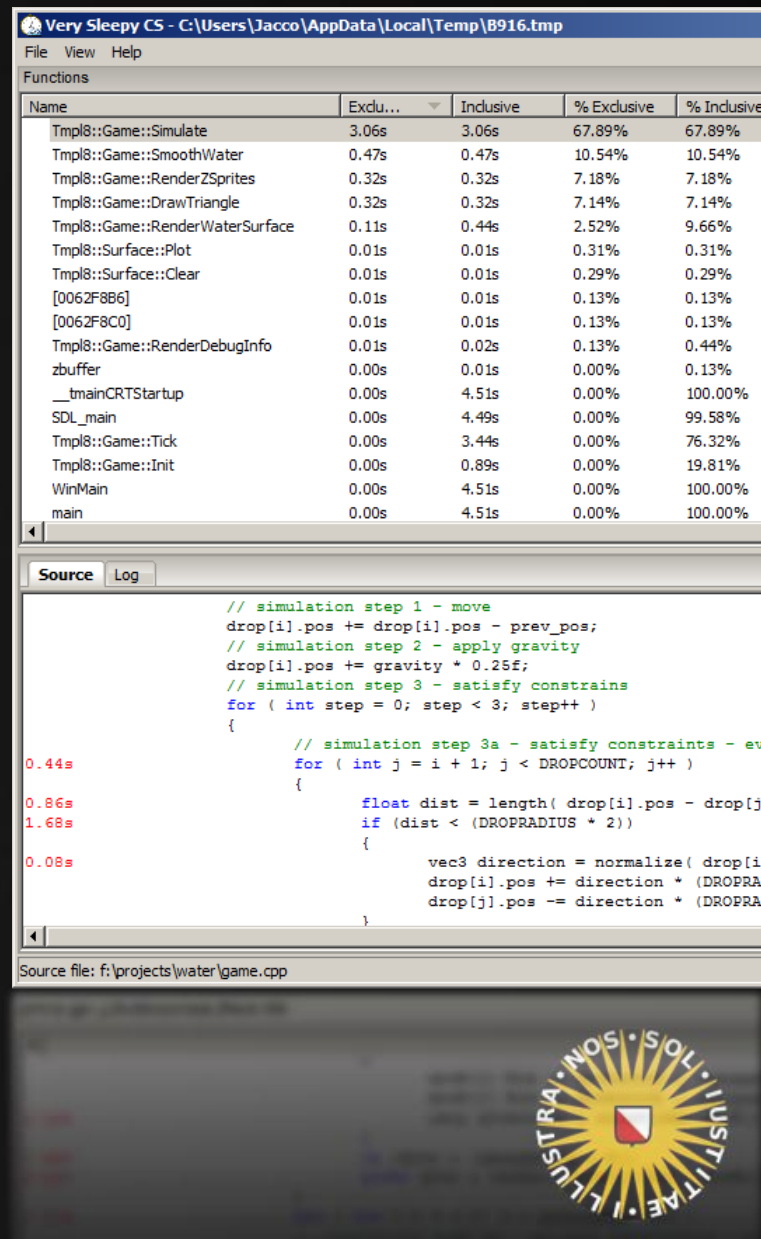
Game::Simulate	67.89%	67.89%
Game::SmoothWater	10.54%	10.54%
Game::RenderZSprites	7.18%	7.18%
Game::Tick	0.00%	76.32%

Running ~3 seconds, we spent 0.86s on this line:

```
float dist = length( drop[i].pos - drop[j].pos );
```

and 1.68s on this line:

```
if (dist < (DROPRADIUS * 2))
```



Profiler Output

Profiling – finding hotspots

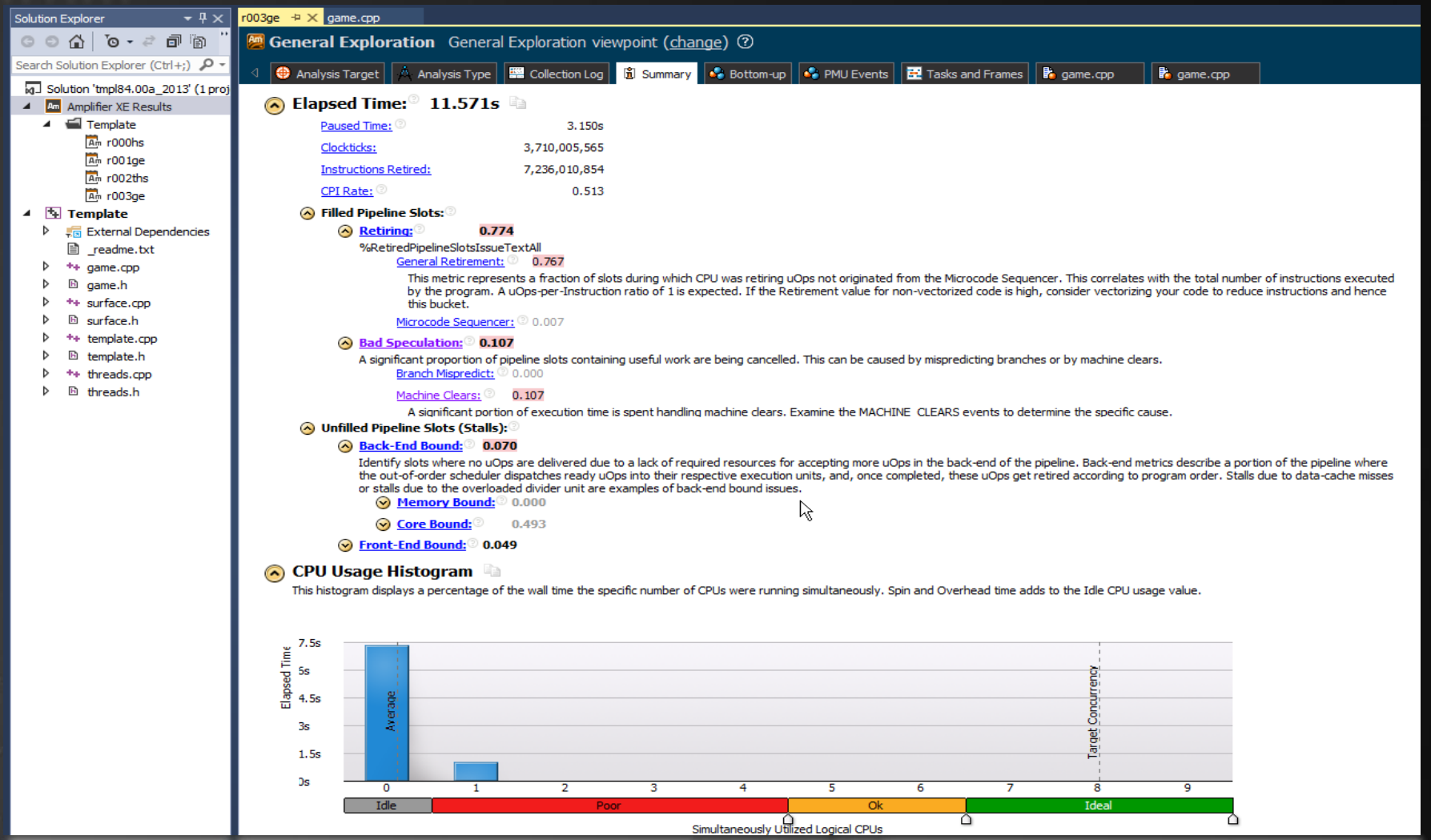
The profiler allows you to quickly find the parts of your program that take most time.

But:

- Mind debug versus release;
- The profiler doesn’t tell you *why* a function is costly
- The profiler doesn’t report scalability
- There is no ‘cost over time’ information

➔ Scalability analysis requires running the program with different work sets (i.e., change N in O(N)).
➔ Determining why a section takes a lot of time requires more in-depth knowledge.
➔ *Solving* the performance issue requires even more in-depth knowledge.

Profiler Output



Profiler Output

So. Lin.	Source	Clockticks	Instructions Retired	CPI Rate	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Spec...	Unfilled Pipeline Slo...: Back-End Bound	Unfilled Pipeline Slo...: Fr. Bo.
129								
130	`// Game::Simulate: do Verlet physics simulation on the particles`							
131	`void Game::Simulate()`							
132	`{`							
133	`ENTER(SIMULATE);`							
134	`for ( int i = 0; i < DROPCOUNT; i++ )`							
135	`{`							
136	`gravity = vec3( sin( angle * PI / 180 ), cos( angle * PI / 180 ), 0 );`							
137	`vec3 prev_pos = drop[i].prev_pos;`							
138	`drop[i].prev_pos = drop[i].pos;`							
139	`// simulation step 1 - move`							
140	`drop[i].pos += drop[i].pos - prev_pos;`							
141	`// simulation step 2 - apply gravity`							
142	`drop[i].pos += gravity * 0.25f;`							
143	`// simulation step 3 - satisfy constrains`							
144	`for ( int step = 0; step < 3; step++ )`	0	2,000,003	0.000	0.000	0.000	1.000	0.000
145	`{`							
146	`// simulation step 3a - satisfy constraints - evade other drops`							
147	`for ( int j = i + 1; j < DROPCOUNT; j++ )`	128,000,192	148,000,222	0.865	0.703	0.000	0.824	0.059
148	`{`							
149	`float dist = length( drop[i].pos - drop[j].pos );`	780,001,170	208,000,312	3.750	1.000	0.000	0.231	0.010
150	`if (dist < (DROPRADIUS * 2))`	1,042,001,563	2,818,004,227	0.370	0.115	0.446	0.381	0.058
151	`{`							
152	`vec3 direction = normalize( drop[i].pos - drop[j].pos );`	64,000,096	86,000,129	0.744	0.820	0.000	0.180	0.000
153	`drop[i].pos += direction * (DROPRADIUS * 2 - dist) * 0.02f;`	22,000,033	0		0.000	1.000	0.000	1.000
154	`drop[j].pos -= direction * (DROPRADIUS * 2 - dist) * 0.02f;`	6,000,009	2,000,003	3.000	0.000	0.000	1.000	0.000
155	`}`							
156	`}`							
157	`// simulation step 3b - satisfy constraints - evade walls`							
158	`if (drop[i].pos.y > 20) drop[i].pos.y = 19.99f - drop[i].pos.z * 0.000`							
159	`if (drop[i].pos.x < -20) drop[i].pos.x = -19.99f + drop[i].pos.z * 0.0`							
160	`if (drop[i].pos.x > 20) drop[i].pos.x = 19.99f - drop[i].pos.z * 0.000`							
161	`if (drop[i].pos.z < -20) drop[i].pos.z = -19.99f + drop[i].pos.z * 0.0`							
162	`if (drop[i].pos.z > 20) drop[i].pos.z = 19.99f - drop[i].pos.z * 0.000`							
163	`}`							
164	`}`							
165	`LEAVE(SIMULATE);`							
166	`}`							

Profiler Output

Take-away:

Free, vendor-agnostic profilers tell you where time is spent in your program (but not *why*).

Vendor-specific tools provide a wealth of information, but generally require knowledge about the hardware processes.

Stalls are generally not vendor-specific and will be similar on similar hardware.

Just timing information is often sufficient to make an educated guess towards improvements.

Custom Profiling

Generic Profiler Downsides

- No ‘performance over time’ measurements
- Requires inclusion of debug information (including source code)
- Not real-time
- Not very intuitive

Using a custom in-app profiler we can drastically improve our profiling information.

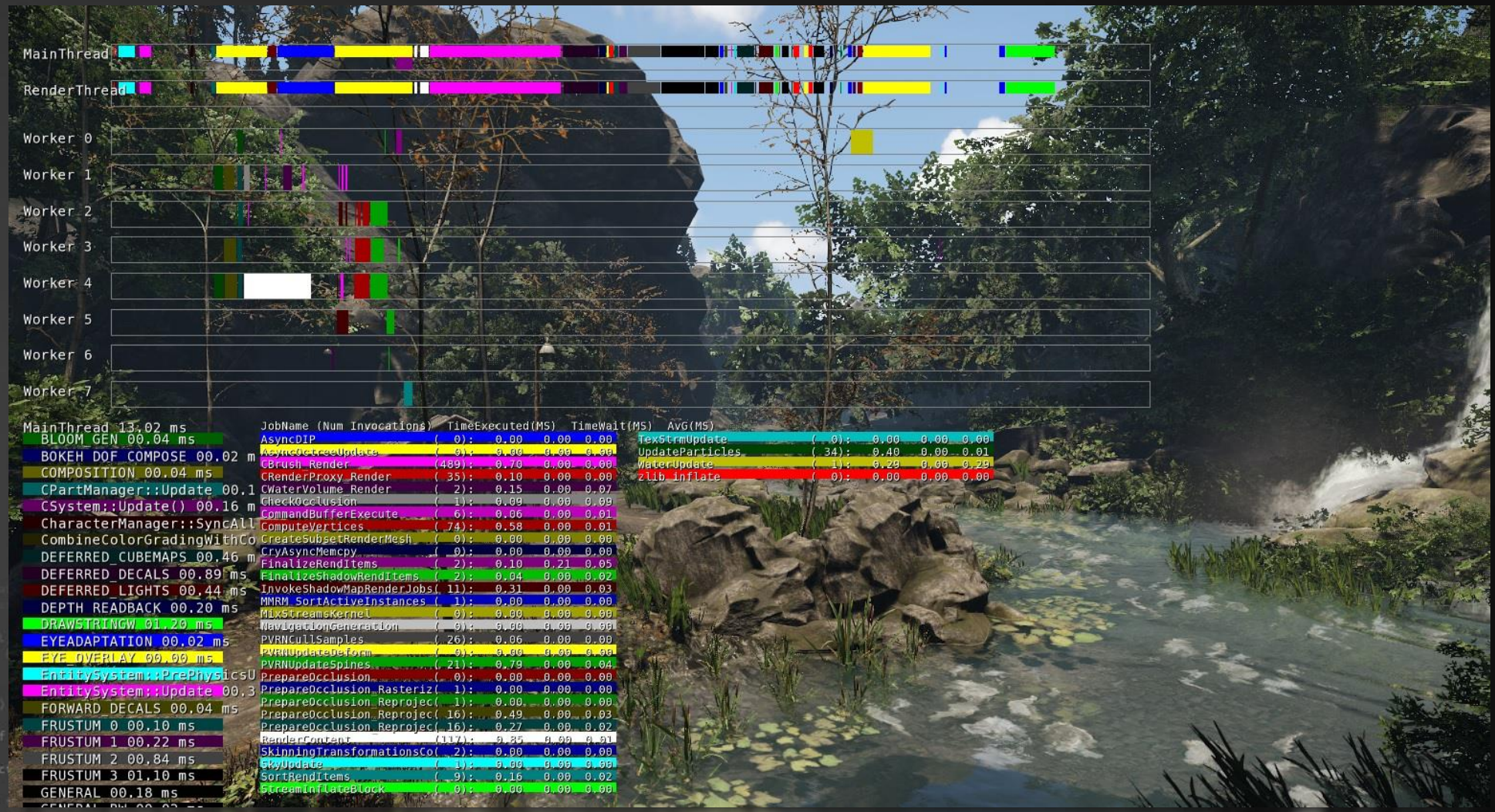
Custom Profiling

Minecraft

Custom Profiling

UnrealEngine 3

Custom Profiling



CryEngine

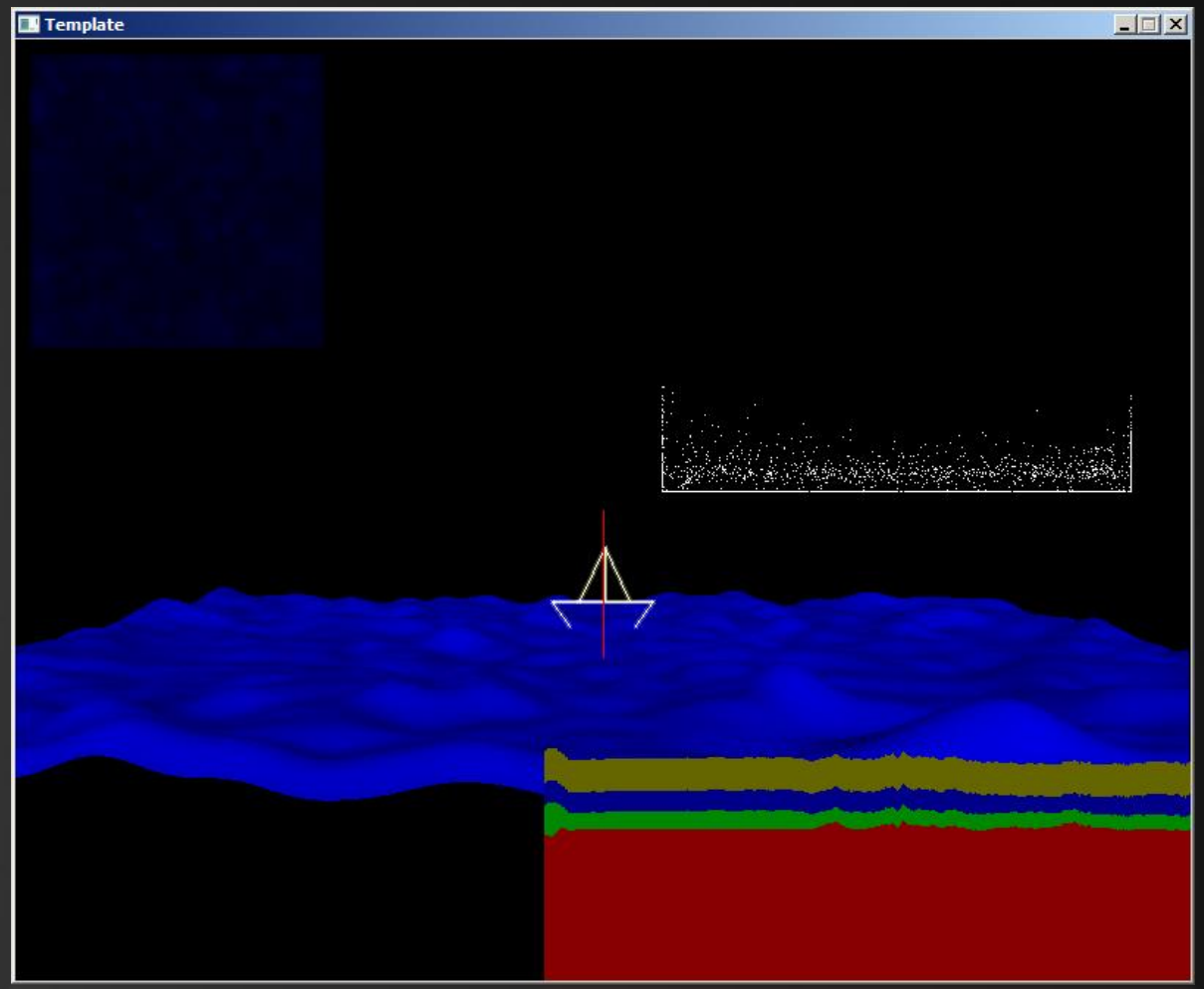
Custom Profiling

StarCraft II

Custom Profiling

StarCraft II

Custom Profiling



Custom Profiling

Take-away:

In-app profiling provides advantages over external profilers:

- You get real-time information, which is easily associated with what is going on in the app;
- You can measure statistics that are not available to the profiler;
- You can present the data in a form that is also useful to people not familiar with the intricacies of the profiler.

And Finally:

Profiling:

Without it, no optimization – we need to *know*

How to profile: tools, custom timers, CPU + GPU

What to profile: realistically (release!), raw performance, scalability
(but also: cache misses, pipelining, branch prediction)

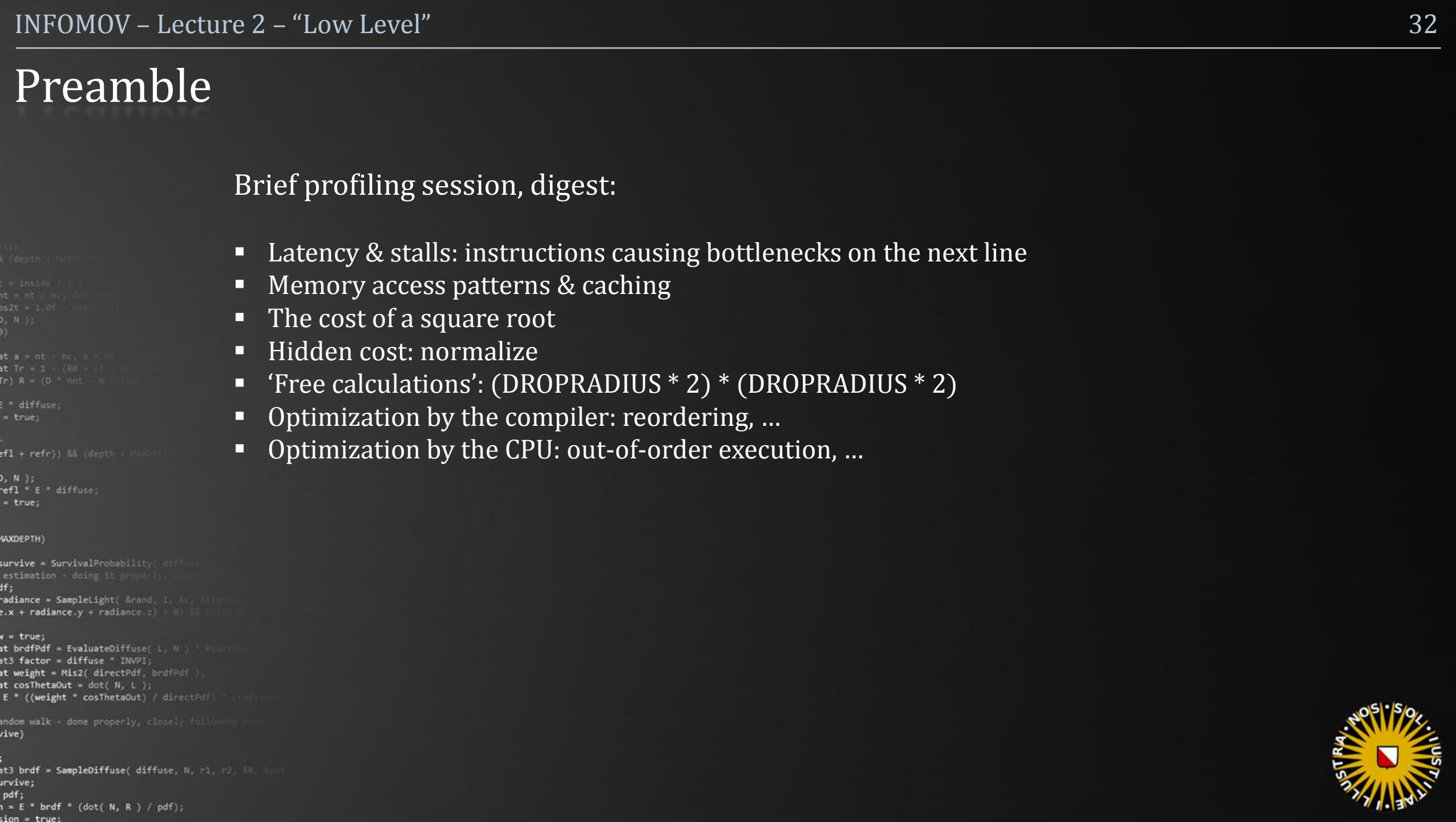
Keep in mind: profiling takes time too

Repeated profiling: things change, if you’re doing it right. Stay informed.

Preamble

Brief profiling session, digest:

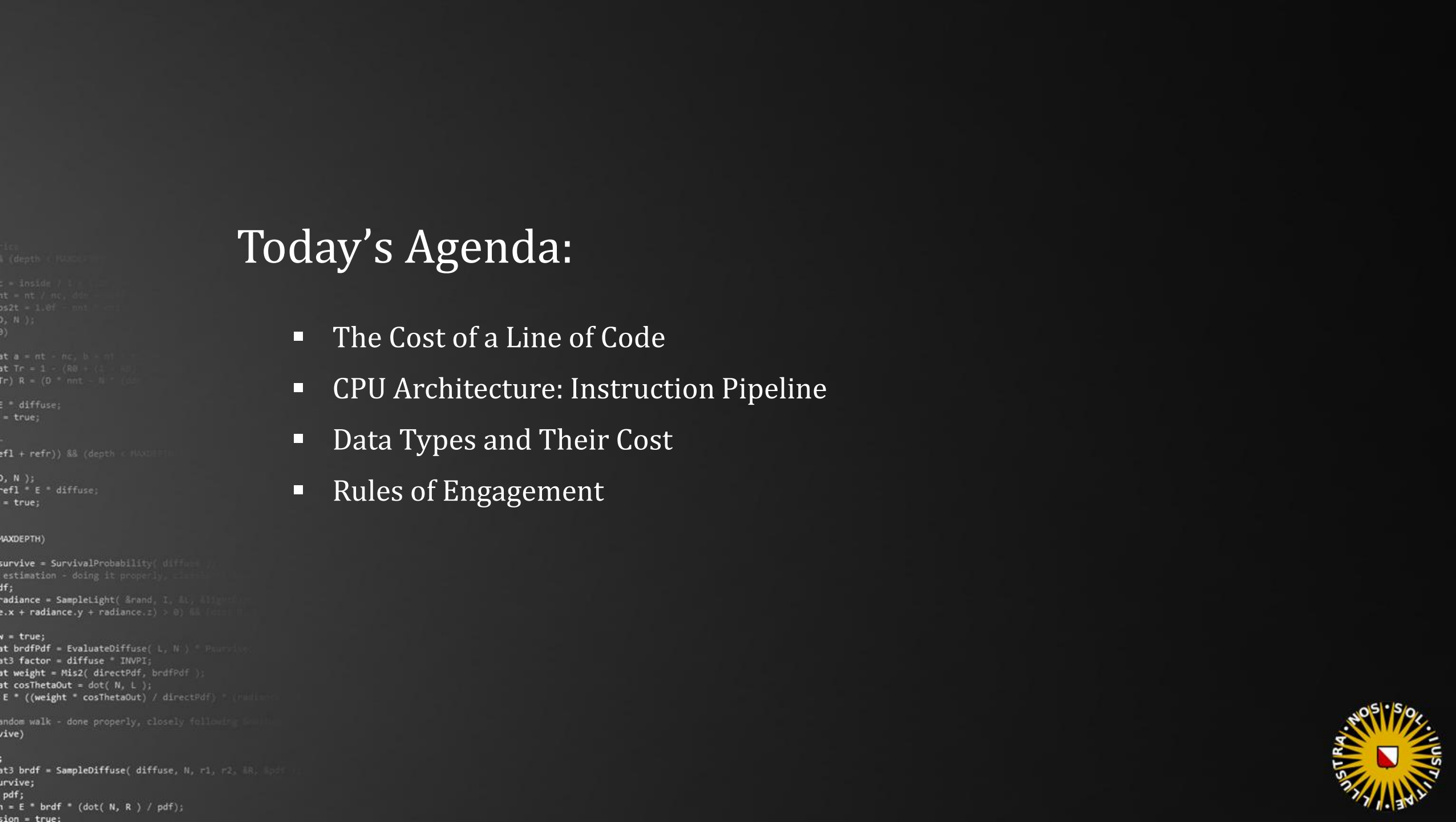
- Latency & stalls: instructions causing bottlenecks on the next line
- Memory access patterns & caching
- The cost of a square root
- Hidden cost: normalize
- ‘Free calculations’: (DROPRADIUS * 2) * (DROPRADIUS * 2)
- Optimization by the compiler: reordering, …
- Optimization by the CPU: out-of-order execution, …



Preamble

Today’s Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement




AND
TAKE
A
BREAK

Instruction Cost

What is the ‘cost’ of a multiply?

```
starttimer();
float x = 0;
for( int i = 0; i < 1000000; i++ ) x *= y;
stoptimer();
```

- Actual measured operations:
  - Timer operations;
  - Initializing ‘x’ and ‘i’;
  - Comparing ‘i’ to 1000000 (x 1000000);
  - Increasing ‘i’ (x 1000000);
  - Jump instruction to start of loop (x 1000000).
- Compiler outsmarts us!
  - No work at all unless we use x
  - x += 1000000 * y

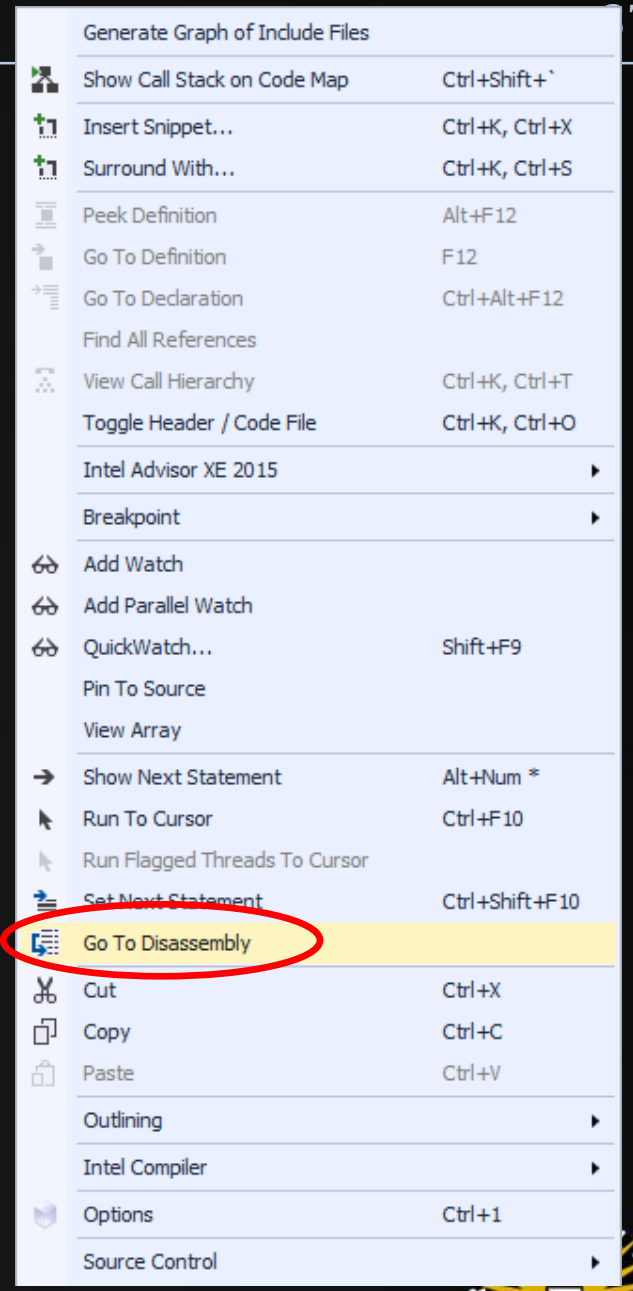
Better solution:

- Create an arbitrary loop
- Measure time with and without the instruction we want to time

Instruction Cost

What is the ‘cost’ of a multiply?

```
float x = 0, y = 0.1f;
unsigned int i = 0, j = 0x28929227;
for( int k = 0; k < ITERATIONS; k++ )
{
   // ensure we feed our line with fresh data
   x += y, y *= 1.01f;
   // integer operations to free up fp execution units
   i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
   // operation to be timed
   if (with) x *= y;
   // integer operations to free up fp execution units
   i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
}
dummy = x + (float)i;
```



Instruction Cost

x86 assembly in 5 minutes:

Modern CPUs still run x86 machine code, based on Intel’s 1978 8086 processor. The original processor was 16-bit, and had 8 ‘general purpose’ 16-bit registers*:

AX (‘accumulator register’)	AH, AL (8-bit)	EAX (32-bit)	RAX (64-bit)
BX (‘base register’)	BH, BL	EBX	RBX
CX (‘counter register’)	CH, CL	ECX	RCX
DX (‘data register’)	DH, DL	EDX	RDX
BP (‘base pointer’)		EBP	RBP
SI (‘source index’)		ESI	RSI
DI (‘destination index’)		EDI	RDI
SP (‘stack pointer’)		ESP	RSP
			R8..R15
		st0..st7	
		XMM0..XMM7	

* More info: http://www.swansontec.com/sregisters.html

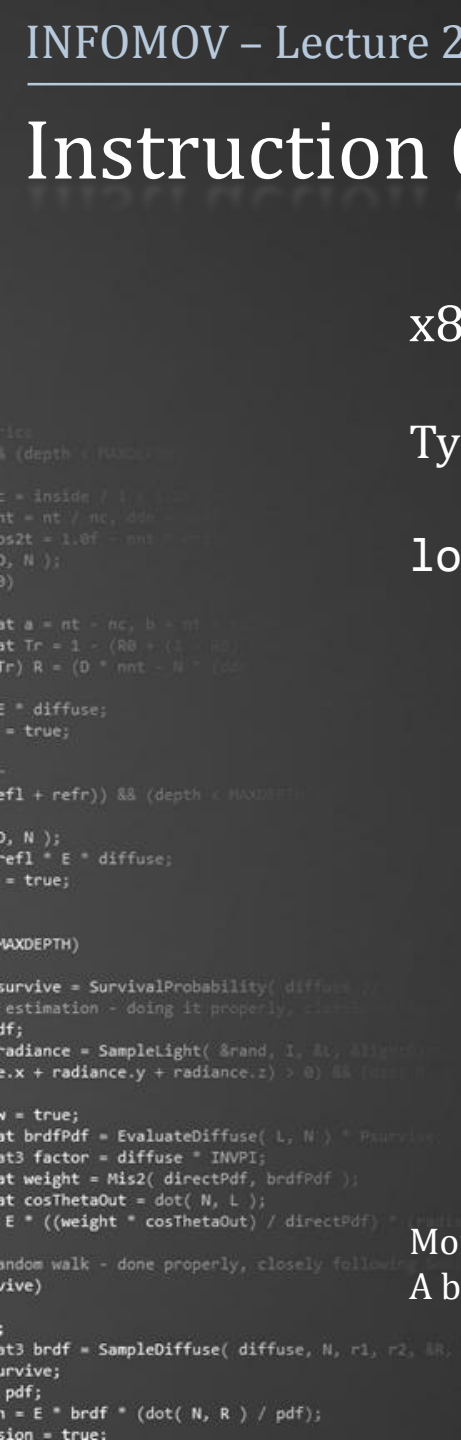
Instruction Cost

x86 assembly in 5 minutes:

Typical assembler:

```
loop:
    mov eax, [0x1008FFA0]     // read from address into register
    shr eax, 5                // shift eax 5 bits to the right
    add eax, edx              // add registers, store in eax
    dec ecx                   // decrement ecx
    jnz loop                  // jump if not zero
    fld [esi]                 // load from address [esi] onto FPU
    fld st0                   // duplicate top float
    faddp                     // add top two values, push result
```

More on x86 assembler: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
A bit more on floating point assembler: https://www.cs.uaf.edu/2007/fall/cs301/lecture/11_12_floating_asm.html



Instruction Cost

What is the ‘cost’ of a multiply?

```
float x = 0, y = 0.1f;
unsigned int i = 0, j = 0x28929227;
for( int k = 0; k < ITERATIONS; k++ )
{
    // ...
    x += y, y *= 1.01f;
    // ...
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
    // ...
    if (with) x *= y;
    // ...
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
}
dummy = x + (float)i;
```

```
fldz
xor ecx, ecx
fld dword ptr ds:[405290h]
mov edx, 28929227h
fld dword ptr ds:[40528Ch]
push esi
mov esi, 0C350h      = 50000
```

```
add ecx, edx
mov eax, 91D2A969h
xor edx, 17737352h
shr ecx, 1
mul eax, edx
fld st(1)
faddp st(3), st

mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1),st
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Eh
dec esi
jne tobetimed<0>+1Fh
```

$= \frac{2^{46}}{28763}$ (!!)

Instruction Cost

What is the ‘cost’ of a multiply?

Observations:

- Compiler reorganizes code
- Compiler cleverly evades division
- Loop counter *decreases*
- Presence of integer instructions affects timing
  (to the point where the mul is free)

But also:

- It is really hard to measure the cost of a line of code.

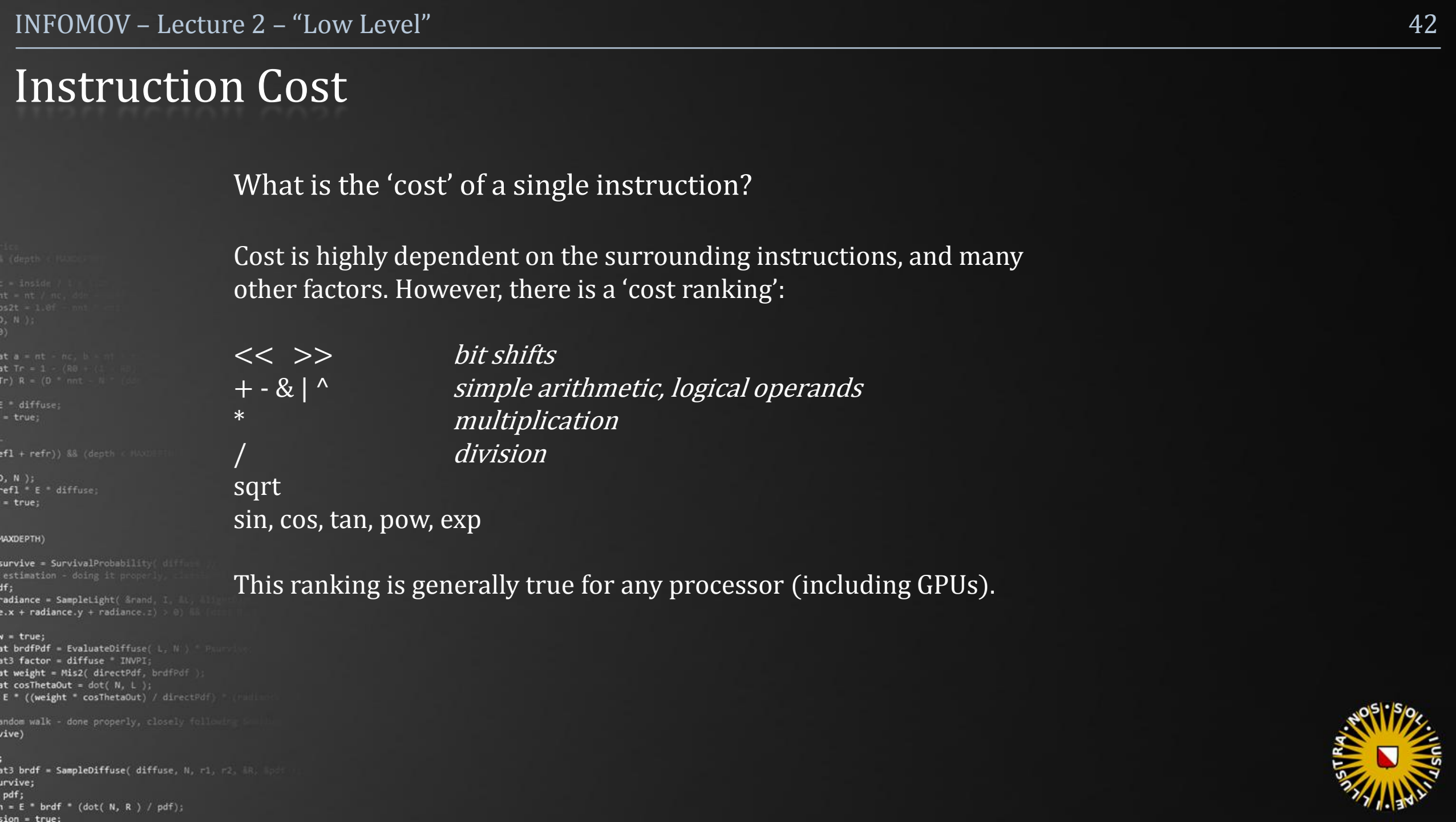
Instruction Cost

What is the ‘cost’ of a single instruction?

Cost is highly dependent on the surrounding instructions, and many other factors. However, there is a ‘cost ranking’:

<< >>	*bit shifts*
+ - & | ^	*simple arithmetic, logical operands*
*	*multiplication*
/	*division*
sqrt	
sin, cos, tan, pow, exp	

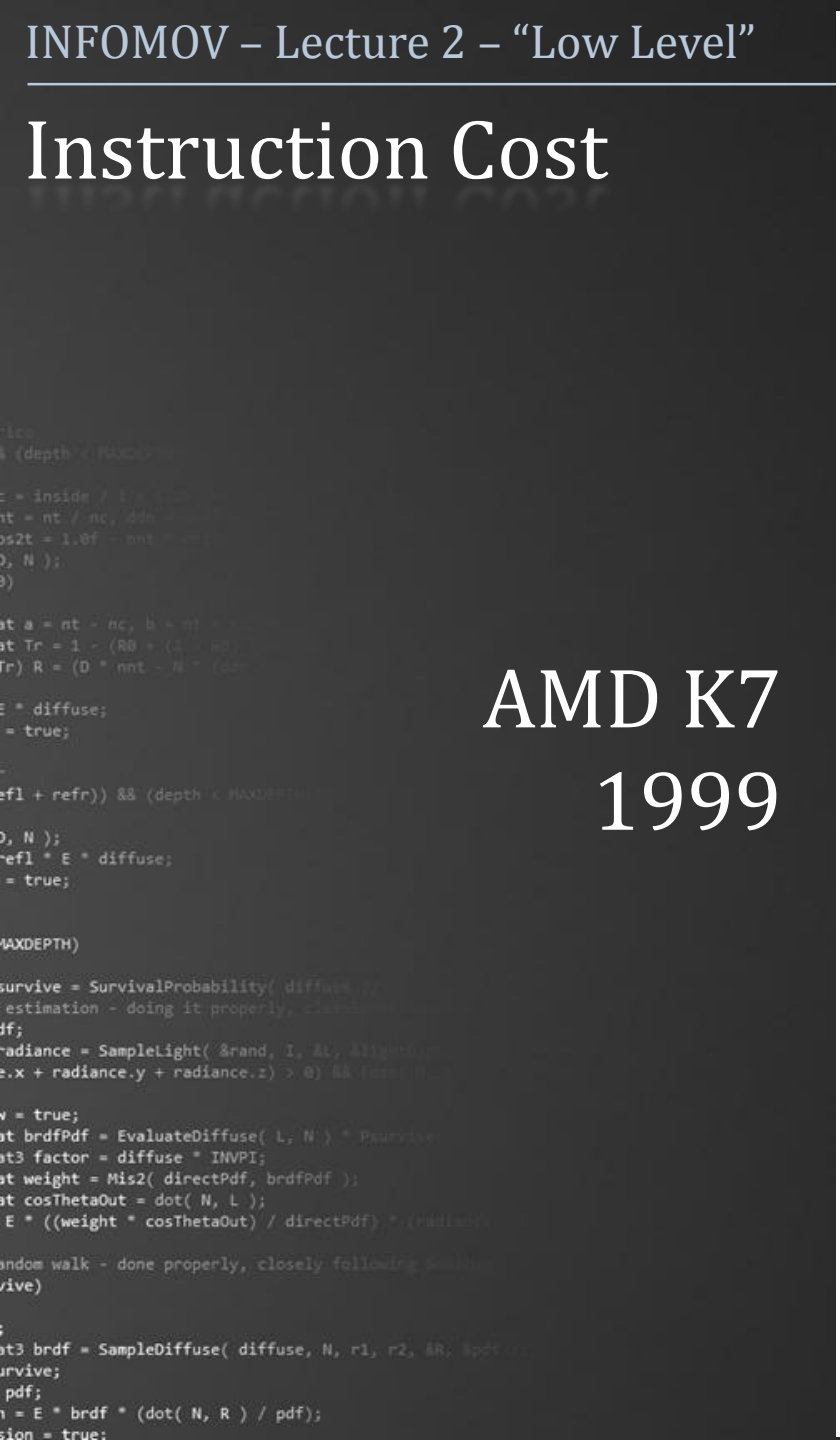
This ranking is generally true for any processor (including GPUs).



Instruction Cost

AMD K7
1999

Instruction	Operands	Ops	Latency	Reciprocal throughput	Execution unit	Notes
Arithmetic instructions						
ADD, SUB	r,r/i	1	1	1/3	ALU	
ADD, SUB	r,m	1	1	1/2	ALU, AGU	
ADD, SUB	m,r	1	7	2,5	ALU, AGU	
ADC, SBB	r,r/i	1	1	1/3	ALU	
ADC, SBB	r,m	1	1	1/2	ALU, AGU	
ADC, SBB	m,r	1	7	2,5	ALU, AGU	
CMP	r,r/i	1	1	1/3	ALU	
CMP	r,m	1	1	1/2	ALU, AGU	
INC, DEC, NEG	r	1	1	1/3	ALU	
INC, DEC, NEG	m	1	7	2,5	ALU, AGU	
AAA, AAS		4	4	4	ALU0	
DAA		2	2	2	ALU0	
DAS		3	3	3	ALU0	
AAD		4	4	4	ALU0	
AAM		4	4	4	ALU0	
MUL, IMUL	r8/m8	3	3	2	ALU0_1	
MUL, IMUL	r16/m16	3	3	2	ALU0_1	latency ax=3, dx=4
MUL, IMUL	r32/m32	3	4	3	ALU0_1	
IMUL	r16,r16/m16	2	3	2	ALU0	
IMUL	r32,r32/m32	2	4	2,5	ALU0	
IMUL	r16,(r16),i	2	4	1	ALU0	
IMUL	r32,(r32),i	2	5	2	ALU0	
IMUL	r16,m16,i	3		2	ALU0	
IMUL	r32,m32,i	3		2	ALU0	
DIV	r8/m8	32	24	23	ALU	
DIV	r16/m16	47	24	23	ALU	
DIV	r32/m32	79	40	40	ALU	
IDIV	r8	41	17	17	ALU	
IDIV	r16	56	25	25	ALU	
IDIV	r32	88	41	41	ALU	
IDIV	m8	42	17	17	ALU	
IDIV	m16	57	25	25	ALU	
IDIV	m32	89	41	41	ALU	
Math						
FSQRT			1	35	12	FMUL
FSIN			44	90-100		
FCOS			51	90-100		
FSINCOS			76	100-150		
FPTAN			46	100-200		
FPATAN			72	160-170		
FSCALE			5	8		
FXTRACT			7	11		
F2XM1			8	27		
FYL2X			49	126		
FYL2XP1			63	147		



Instruction Cost

AMD Jaguar 2013

Instruction	Operands	Ops	Latency	Reciprocal throughput	Execution pipe	Notes
Arithmetic instructions						
ADD, SUB	r,r/i	1	1	0.5	I0/1	
ADD, SUB	r,m	1		1		
ADD, SUB	m,r	1	6	1		
ADC, SBB	r,r/i	1	1	1	I0/1	
ADC,						
ADC,						
CMP						
CMP						
INC, D						
INC, D						
AAA						
AAS						
DAA						
DAS						
AAD						
AAM						
MUL,						
MUL, IMUL	r16/m16	3	3	3	I0	
MUL, IMUL	r32/m32	2	3	2	I0	
MUL, IMUL	r64/m64	2	6	5	I0	
IMUL	r16,r16/m16	1	3	1	I0	
IMUL	r32,r32/m32	1	3	1	I0	
IMUL	r64,r64/m64	1	6	4	I0	
IMUL	r16,(r16),i	2	4	1	I0	
IMUL	r32,(r32),i	1	3	1	I0	
IMUL	r64,(r64),i	1	6	4	I0	
DIV	r8/m8	1	11-14	11-14	I0	
DIV	r16/m16	2	12-19	12-19	I0	
DIV	r32/m32	2	12-27	12-27	I0	
DIV	r64/m64	2	12-43	12-43	I0	
IDIV	r8/m8	1	11-14	11-14	I0	
IDIV	r16/m16	2	12-19	12-19	I0	
IDIV	r32/m32	2	12-27	12-27	I0	
IDIV	r64/m64	2	12-43	12-43	I0	

Math					
FSQRT	1	35	35		FP1
FLDPI, etc.	1		1		FP0
FSIN	4-44	30-139	30-151		FP0, FP1
FCOS	11-51	38-93			FP0, FP1
FSINCOS	11-76	55-122	55-180		FP0, FP1
FPTAN	11-45	55-177	55-177		FP0, FP1
FPATAN	9-75	44-167	44-167		FP0, FP1
FSCALE	5	27			FP0, FP1
FXTRACT	7	9	6		FP0, FP1
F2XM1	8	32-37			FP0, FP1
FYL2X	8-51	30-120	30-120		FP0, FP1
FYL2XP1	61	~160	~160		FP0, FP1

Note: Two micro-operations can execute simultaneously if they go to different execution pipes

Instruction Cost

Intel Silvermont 2014

	Operands	µops	Unit	Latency	Reciprocal throughput	Remarks
Arithmetic instructions						
ADD SUB	r,r/i	1	IP0/1	1	1/2	
ADD SUB	r,m	1	IP0/1, Mem		1	
ADD SUB	m,r/i	1	IP0/1, Mem	6	1	
ADC SBB	r,r/i	1	IP0/1	2	2	
ADC SBB	r,m	1			2	
ADC SBB	m,r/i	1		6	2	
CMP	r,r/i	1	IP0/1	1	1/2	
CMP	m,r/i	1			1	
INC DEC	r	1	IP0/1	1	1/2	latency to flag=2
NEG NOT						
INC DEC						
AAA						
AAS						
DAA						
DAS						
AAD						
AAM						
MUL IMUL						
MUL IMUL						
MUL IMUL						
MUL IMUL						
IMUL	r16,r16	2	IP0	4	4	
IMUL	r32,r32	1	IP0	3	1	
IMUL	r64,r64	1	IP0	5	2	
IMUL	r16,r16,i	2	IP0	4	4	
IMUL	r32,r32,i	1	IP0	3	1	
IMUL	r64,r64,i	1	IP0	5	2	
MUL IMUL	m8	3	IP0			
MUL IMUL	m16	5	IP0			
MUL IMUL	m32	4	IP0			
MUL IMUL	m64	4	IP0	14		
DIV	r/m8	9	IP0, FP0	24	19	
DIV	r/m16	12	IP0, FP0	25-29	19-23	
DIV	r/m32	12	IP0, FP0	25-39	19-31	
DIV	r/m 64	23	IP0, FP0	34-94	25-94	
IDIV	r/m8	26	IP0, FP0	24-35	25	
IDIV	r/m16	29	IP0, FP0	37-41	30-32	
IDIV	r/m32	29	IP0, FP0	29-46	29-38	
IDIV	r/m64	44	IP0, FP0	47-107	47-107	

Math				
FSCALE	27			66
FXTRACT	15		20	20
FSQRT	1		13-40	13-40
FSIN FCOS	18		40-170	40-170
FSINCOS	110		40-170	
F2XM1	9		39-90	
FYL2X	34		80-140	
FYL2XP1	61		154	
FPTAN	101		45-200	
FPATAN	63		85-190	

Note: This is a low-power processor (ATOM class).

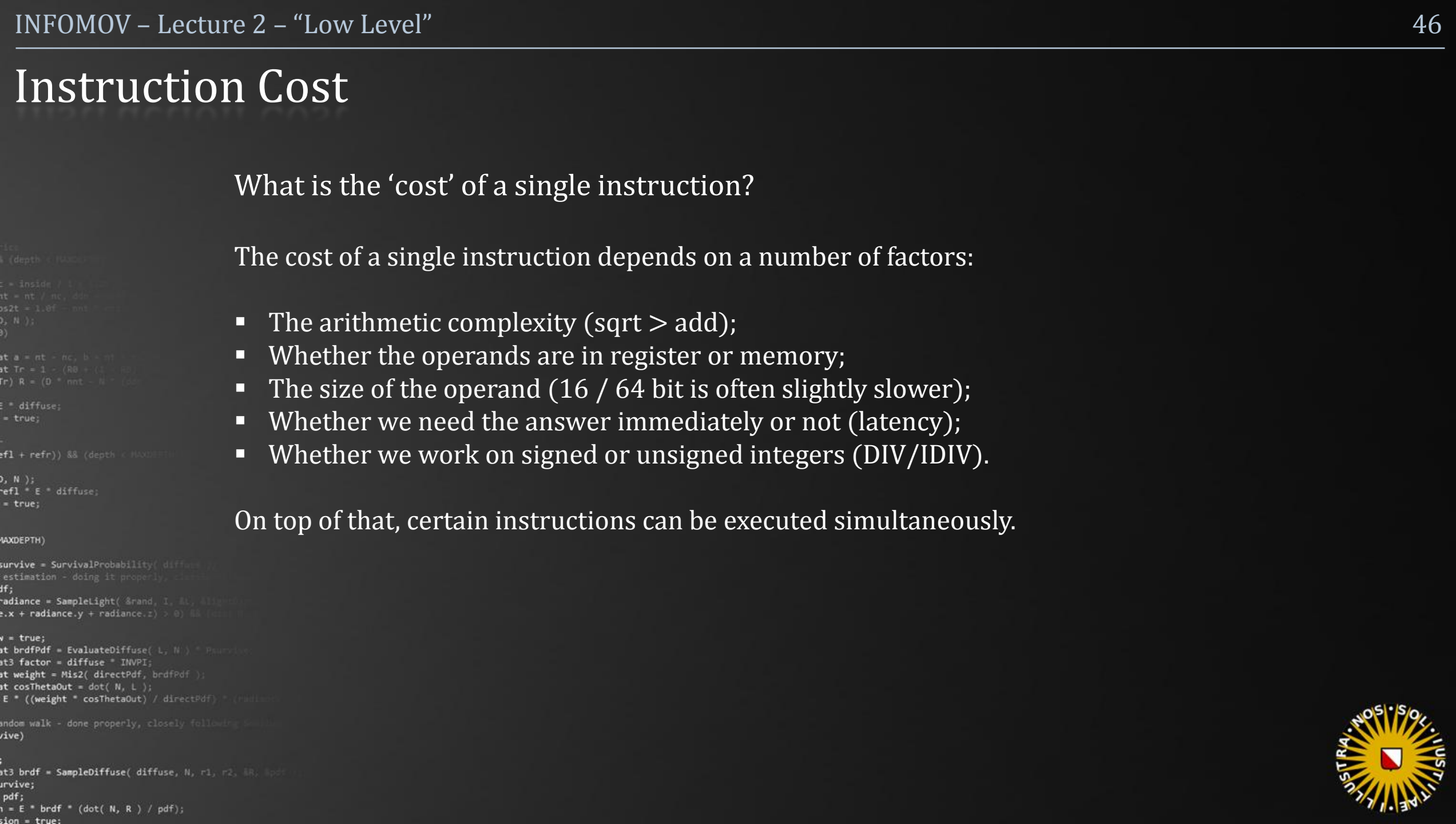
Instruction Cost

What is the ‘cost’ of a single instruction?

The cost of a single instruction depends on a number of factors:

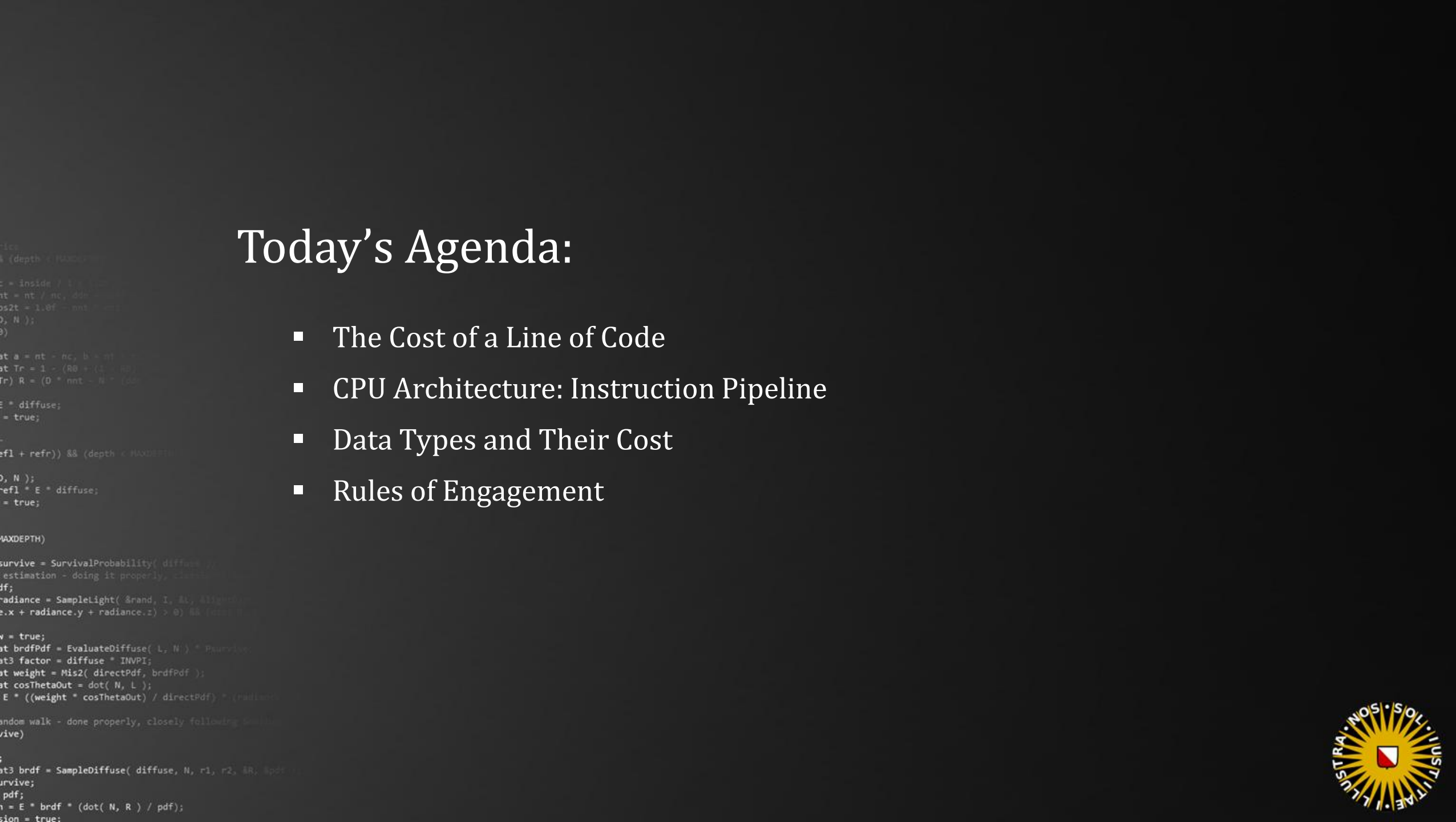
- The arithmetic complexity (sqrt > add);
- Whether the operands are in register or memory;
- The size of the operand (16 / 64 bit is often slightly slower);
- Whether we need the answer immediately or not (latency);
- Whether we work on signed or unsigned integers (DIV/IDIV).

On top of that, certain instructions can be executed simultaneously.



Today’s Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement

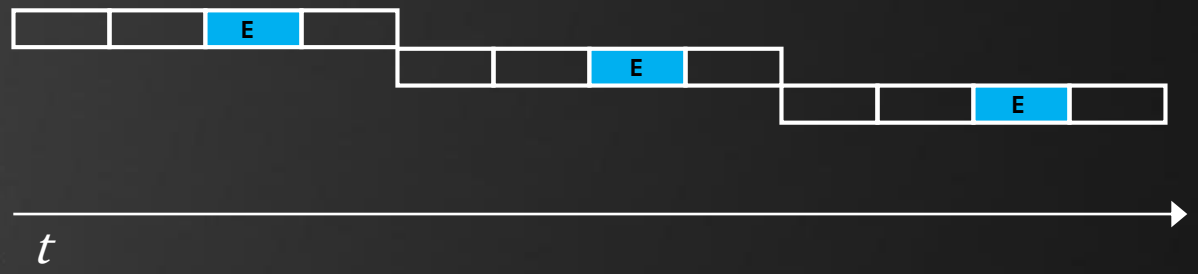


Pipeline

CPU Instruction Pipeline

Instruction execution is typically divided in four phases:

1. Fetch — Get the instruction from RAM
2. Decode — The byte code is decoded
3. Execute — The instruction is executed
4. Writeback — The results are written to RAM/registers

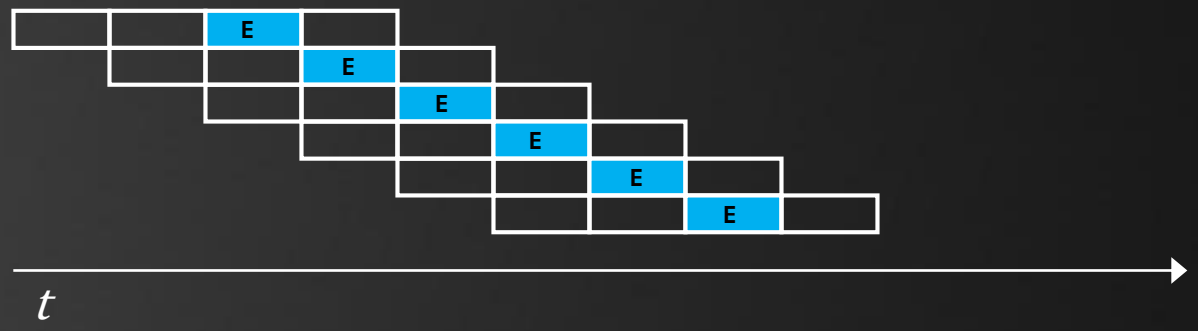



```
fldz
xor ecx, ecx
fld dword ptr ds:[405290h]
mov edx, 28929227h
fld dword ptr ds:[40528Ch]
push esi
mov esi, 0C350h
add ecx, edx
mov eax, 91D2A969h
xor edx, 17737352h
shr ecx, 1
mul eax, edx
fld st(1)
faddp st(3), st
mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1),st
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Eh
dec esi
jne tobetimed<0>+1Fh
```

Pipeline

CPU Instruction Pipeline

For each of the stages, different parts of the CPU are active.
To use its transistors more efficiently, a modern processor overlaps these phases in a *pipeline*.

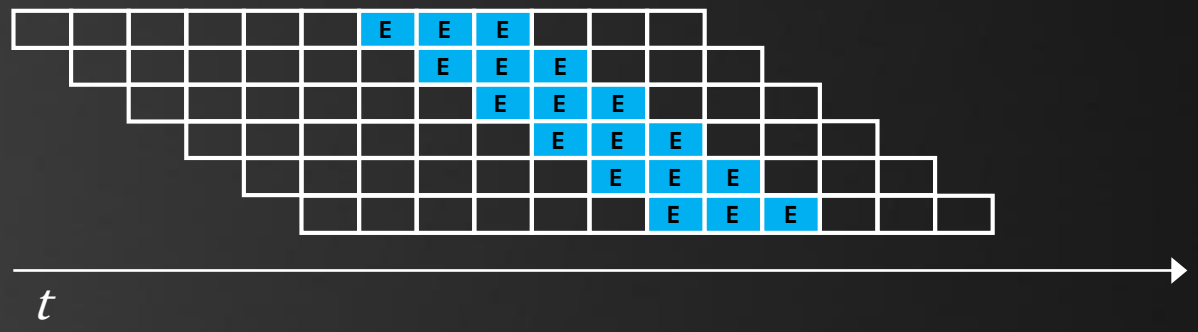


At the same clock speed, we get four times the throughput.

Pipeline

CPU Instruction Pipeline

Maximum clockspeed is determined by the most complex of the four stages. For higher clockspeeds, it is advantageous to increase the number of stages (thereby reducing the complexity of each individual stage).



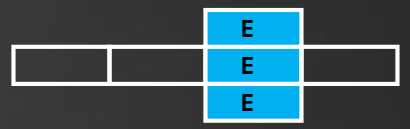
Stages	
7	PowerPC G4e
8	Cortex-A9
10	Athlon
12	Pentium Pro/II/III, Athlon 64
14	Core 2, Apple A7/A8
14/19	Core i2/i3 Sandy Bridge
16	PowerPC G5, Core i*1 Nehalem
18	Bulldozer, Steamroller
20	Pentium 4
31	Pentium 4E Prescott

Obviously, ‘execution’ of different instructions requires different functionality.

Pipeline

CPU Instruction Pipeline

Different execution units for different (classes of) instructions:



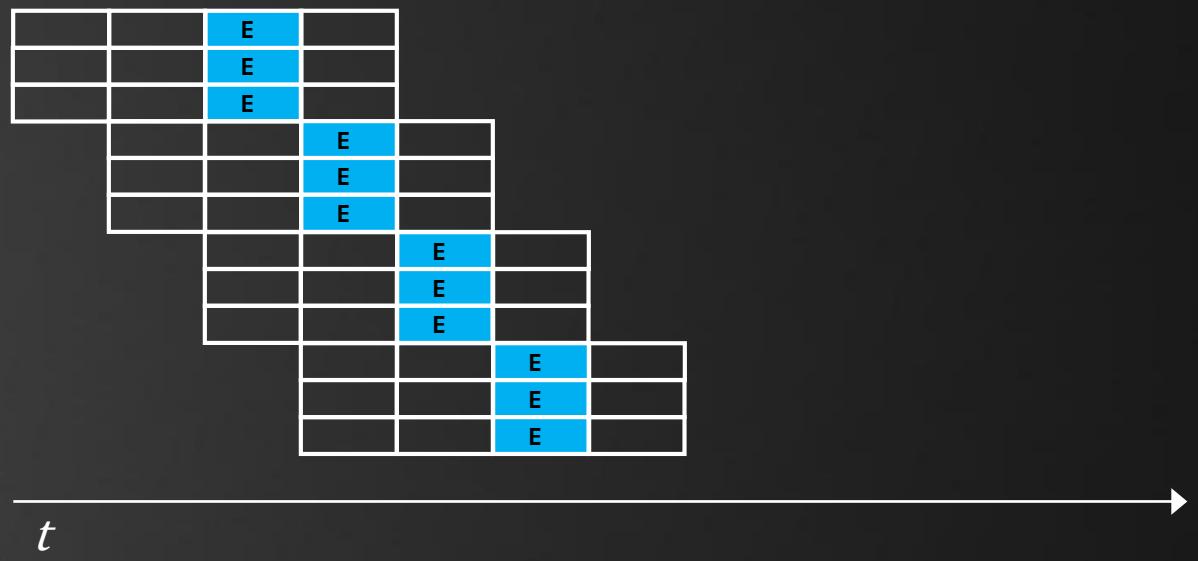
Here, one execution unit handles floats;
one handles integer;
one handles memory operations.

Since the execution logic is typically the most complex part, we might just as well duplicate the other parts:

Pipeline

CPU Instruction Pipeline

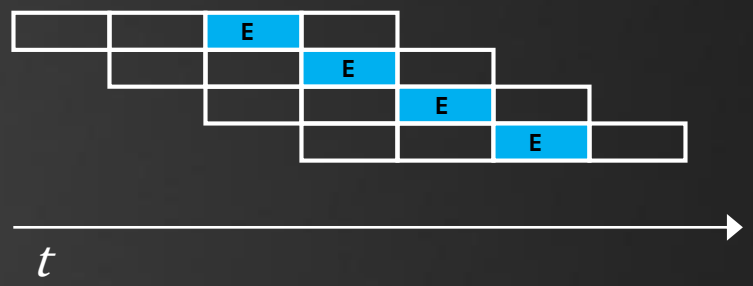
This leads to the superscalar processor, which can execute multiple instructions in the same clock cycle, assuming not all instruction require the same execution logic.



Pipeline

CPU Instruction Pipeline

Using a pipeline has consequences.
Consider the following situation:



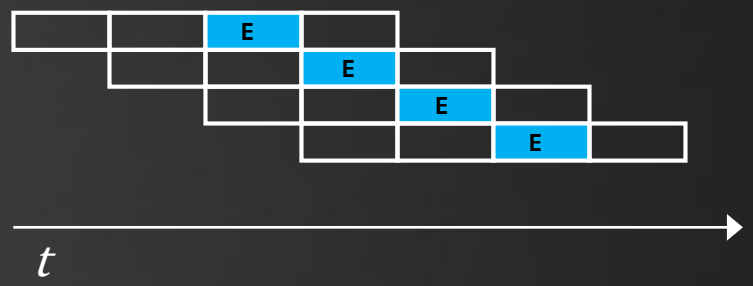

```
a = b * c;
d = a + 1;
```

Here, the second instruction needs the result of the first, which is available one clock tick too late. As a consequence, the pipeline stalls briefly.

Pipeline

CPU Instruction Pipeline

Using a pipeline has consequences.
Consider the following situation:




```
a = b * c;
jump if a is not zero
```

In this scenario, a conditional jump makes it hard for the CPU to determine what to feed into the pipeline after the jump.

Pipeline

CPU Instruction Pipeline - Digest

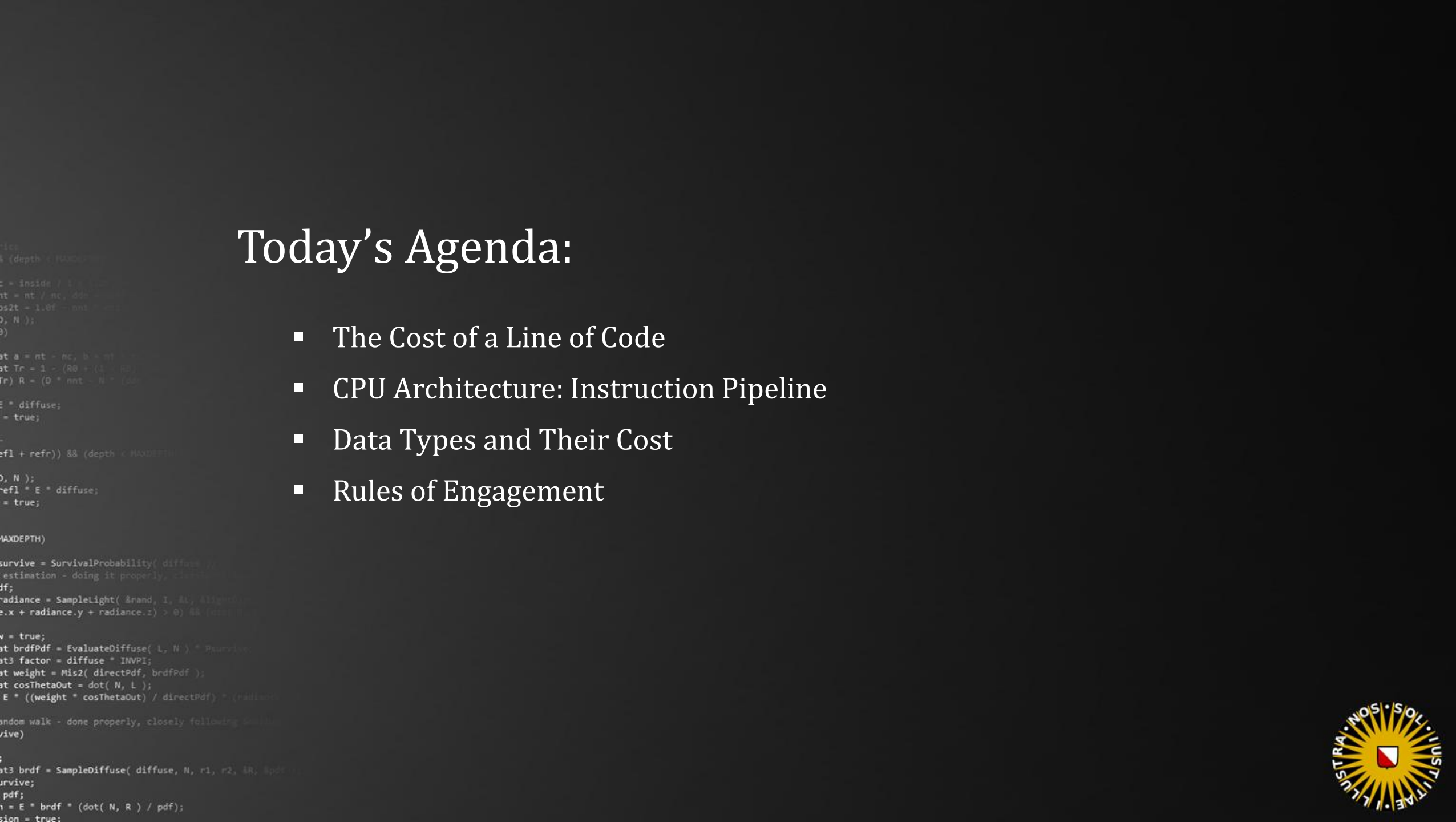
For a more elaborate explanation of the pipeline, see this document:
http://www.lighterra.com/papers/modernmicroprocessors

For now:

- A compiler reorganizes code to prevent latencies
- Feeding mixed code provides the compiler with sufficient opportunities for shuffling
- Branching issues need to be prevented manually

Today’s Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



Data Types

Data types in C++

int
unsigned int

```
Red = u4 & (255 << 16);
Green = u4 & (255 << 8);
Blue = u4 & 255;
```

Size: 32 bit (4 bytes)
Access:

```
union { unsigned int u4; int s4; char s[4]; };
unsigned char v = 100;
s[1] = v;
u4 = (a4 ^ (255 << 8)) | (v << 8);
```

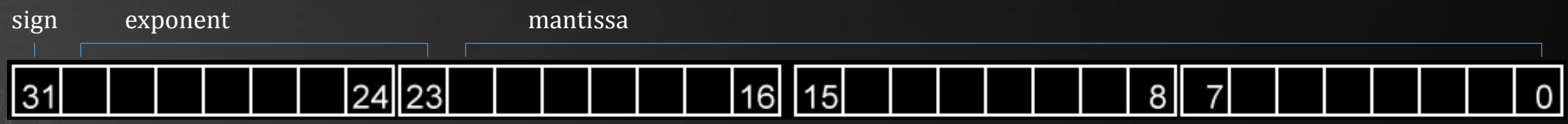
Altering sign bit of s4:
(note: -1 = 0xffffffff)

```
u4 ^= 1 << 31;
```

Data Types

Data types in C++

float



Size: 32 bit (4 bytes)

Exponent: 8 bit; -127 … 128
Mantissa: 23 bit; 0 … 2^{23} -1

Value: sign * mantissa * 2^exponent

Data Types

Data types in C++

double	64 bit (8 bytes)
char, unsigned char	8 bit
short, unsigned short	16 bit
LONG	32 bit (same as int)
LONG LONG, __int64	64 bit
bool	8 bit (!)

Padding:

```
struct Test
{
   unsigned int u;
   bool flag;
};
// sizeof( Test ) is 8
```

```
struct Test2
{
   double d;
   bool flag;
};
// sizeof( Test2 ) is 16
```

Data Types

Data types in C++ - Conversions

Explicit:

```
float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);
```

Implicit:

```
struct Color { unsigned char a, r, g, b; };
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ )
{
   bitmap[i].r *= 0.5f;
   bitmap[i].g *= 0.5f;
   bitmap[i].b *= 0.5f;
}
```

```
// bitmap[i].r *= 0.5f;
movzx       eax,byte ptr [ecx-1]
mov         dword ptr [ebp-4],eax
fild        dword ptr [ebp-4]
fnstcw      word ptr [ebp-2]
movzx       eax,word ptr [ebp-2]
or          eax,0C00h
mov         dword ptr [ebp-8],eax
fmul        st,st(1)
fldcw       word ptr [ebp-8]
fistp       dword ptr [ebp-8]
movzx       eax,byte ptr [ebp-8]
mov         byte ptr [ecx-1],al
```

Data Types

Data types in C++ - Conversions

Explicit:

```
float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);
```

Avoiding conversion:

```
struct Color { unsigned char a, r, g, b; };
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ )
{
    bitmap[i].r >>= 1;
    bitmap[i].g >>= 1;
    bitmap[i].b >>= 1;
}
```

```
// bitmap[i].r >>= 1;
shr         byte ptr [eax-1],1
// bitmap[i].g >>= 1;
shr         byte ptr [eax],1
// bitmap[i].b >>= 1;
shr         byte ptr [eax+1],1
```

Data Types

Data types in C++ - Conversions

Explicit:

float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);

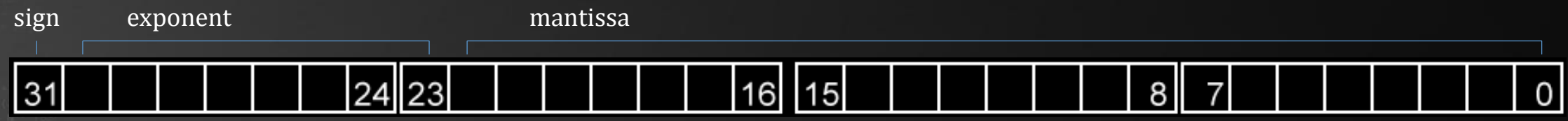
Avoiding conversion (2):

```
struct Color { union { struct { unsigned char a, r, g, b; }; int argb; }; };
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ )
{
    bitmap[i].argb = (bitmap[i].argb >> 1) & 0x7f7f7f;
}
```

Data Types

Data types in C++ - Free interpretation

Trick: Cheaper float comparison

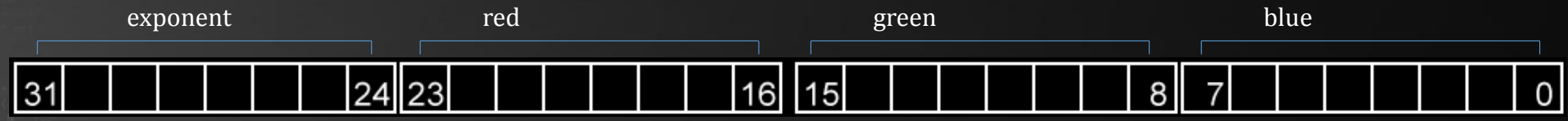



```
union { float v1; unsigned int u1; };
union { float v2; unsigned int u2; };

bool smaller = (v1 < v2);

bool smaller = (u1 < u2);  // same result, if signs of v1 and v2 are equal.
```

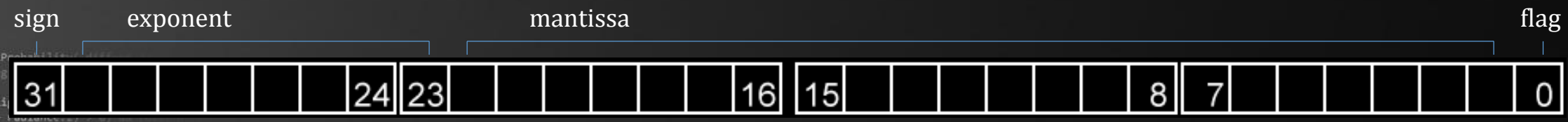
Data Types

Data types in C++ - Rolling your own

HDR color storage

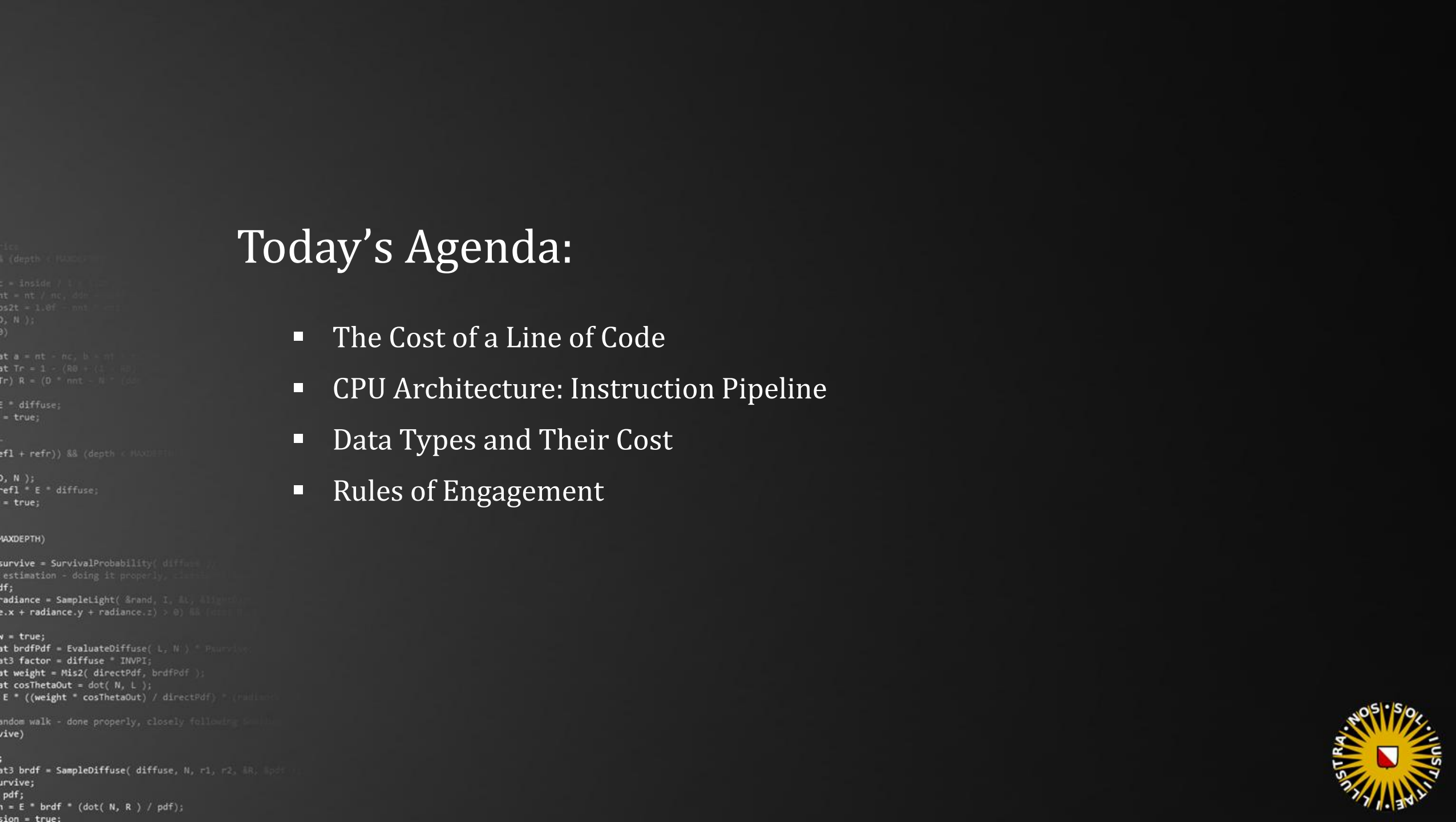


Storing a bit flag in a floating point value



Today’s Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement

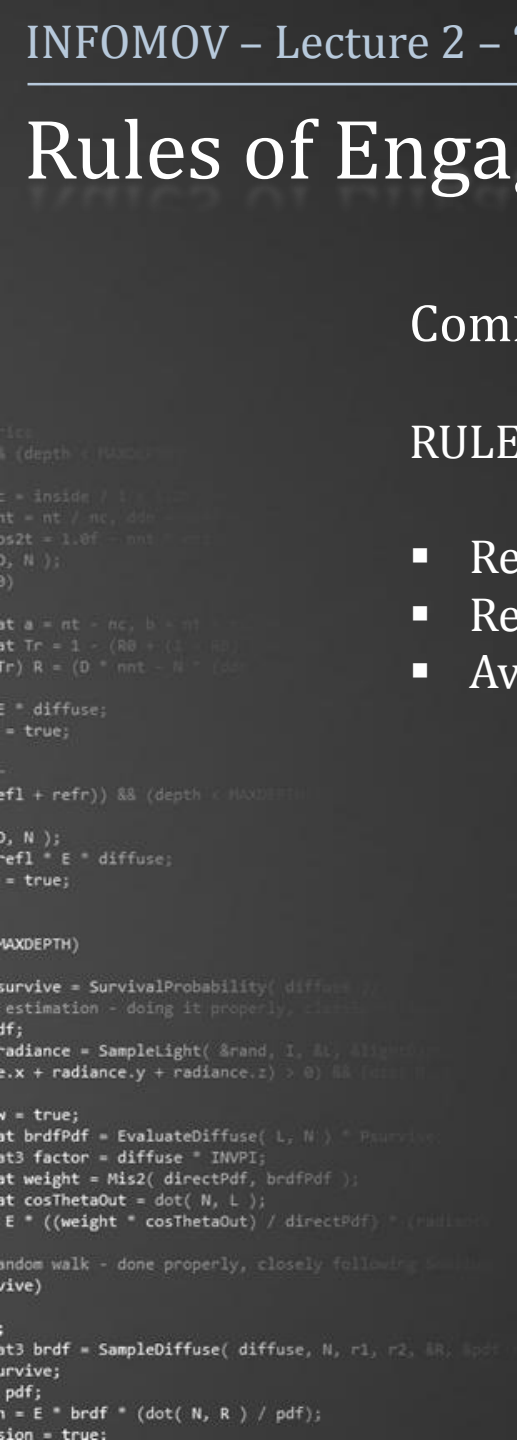


Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 1: Avoid Costly Operations

- Replace multiplications by bitshifts, when possible
- Replace divisions by (reciprocal) multiplications
- Avoid sin, cos, sqrt



Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 2: Precalculate

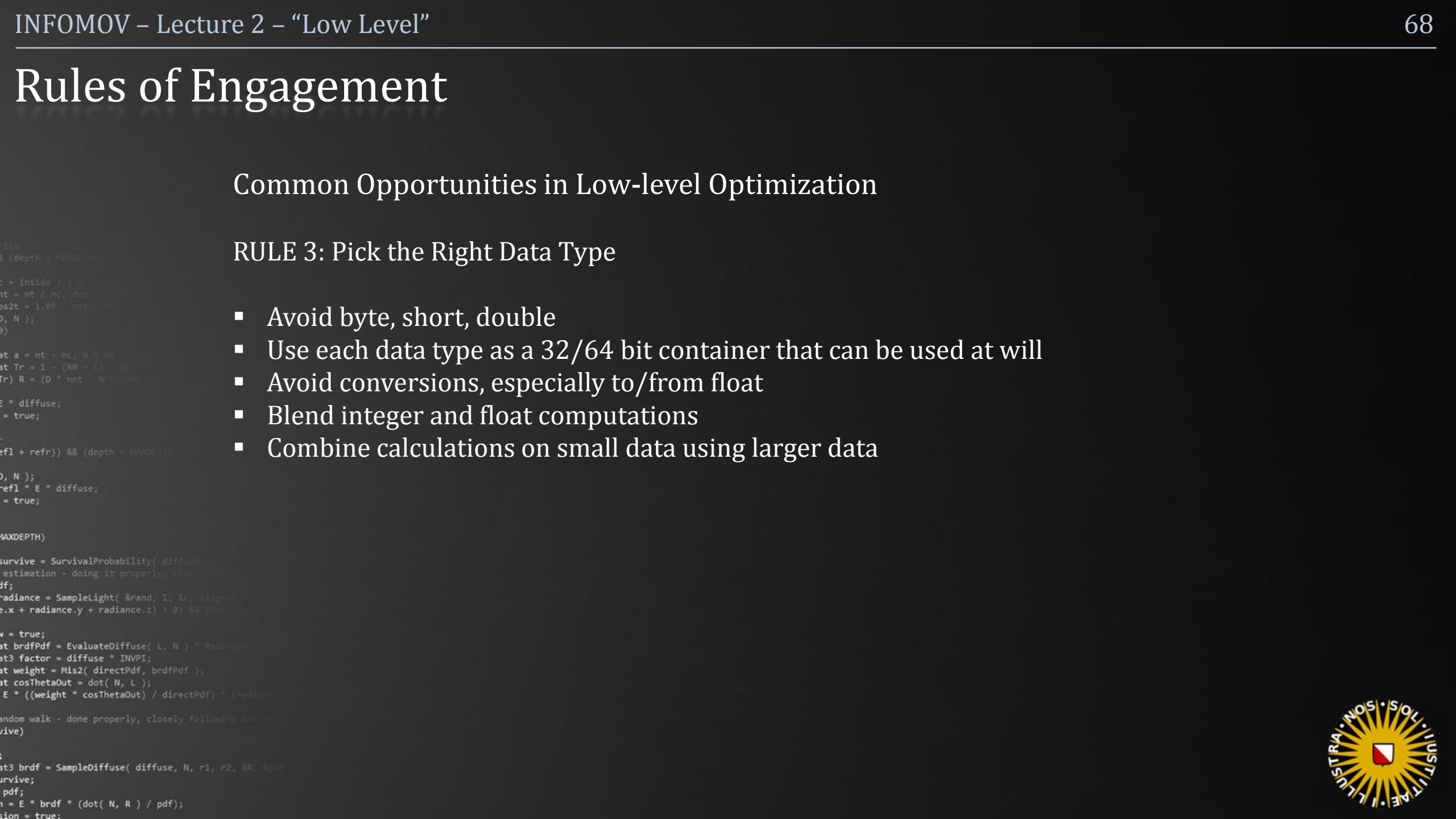
- Reuse (partial) results
- Adapt previous results (interpolation, reprojection, …)
- Loop hoisting
- Lookup tables

Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 3: Pick the Right Data Type

- Avoid byte, short, double
- Use each data type as a 32/64 bit container that can be used at will
- Avoid conversions, especially to/from float
- Blend integer and float computations
- Combine calculations on small data using larger data



Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 4: Avoid Conditional Branches

- if, while, ?, MIN/MAX
- Try to split loops with conditional paths into multiple unconditional loops
- Use lookup tables to prevent conditional code
- Use loop unrolling
- If all else fails: make conditional branches predictable

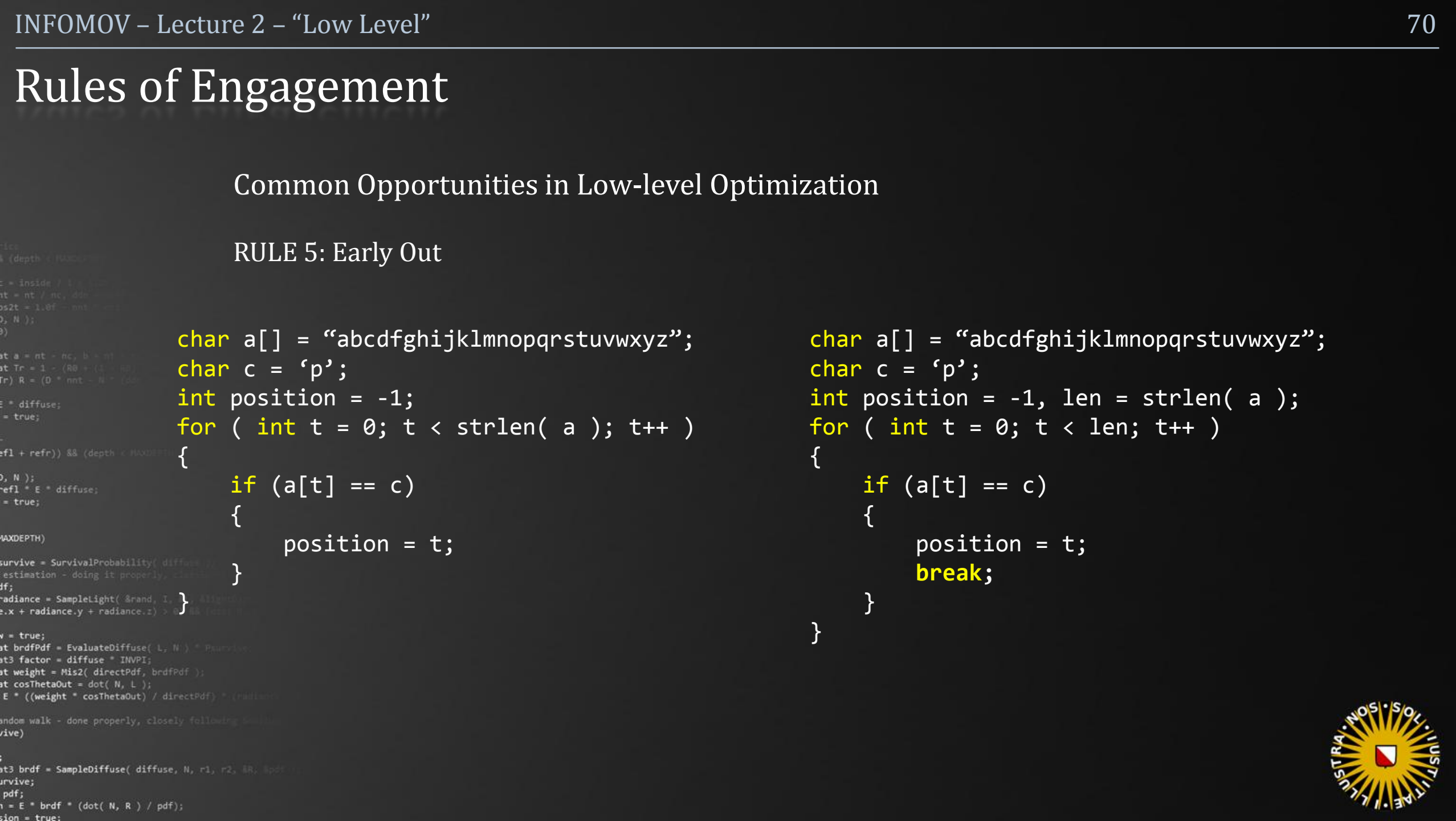
Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 5: Early Out

```
char a[] = “abcdfghijklmnopqrstuvwxyz”;
char c = ‘p’;
int position = -1;
for ( int t = 0; t < strlen( a ); t++ )
{
    if (a[t] == c)
    {
        position = t;
    }
}
```

```
char a[] = “abcdfghijklmnopqrstuvwxyz”;
char c = ‘p’;
int position = -1, len = strlen( a );
for ( int t = 0; t < len; t++ )
{
    if (a[t] == c)
    {
        position = t;
        break;
    }
}
```



Rules of Engagement

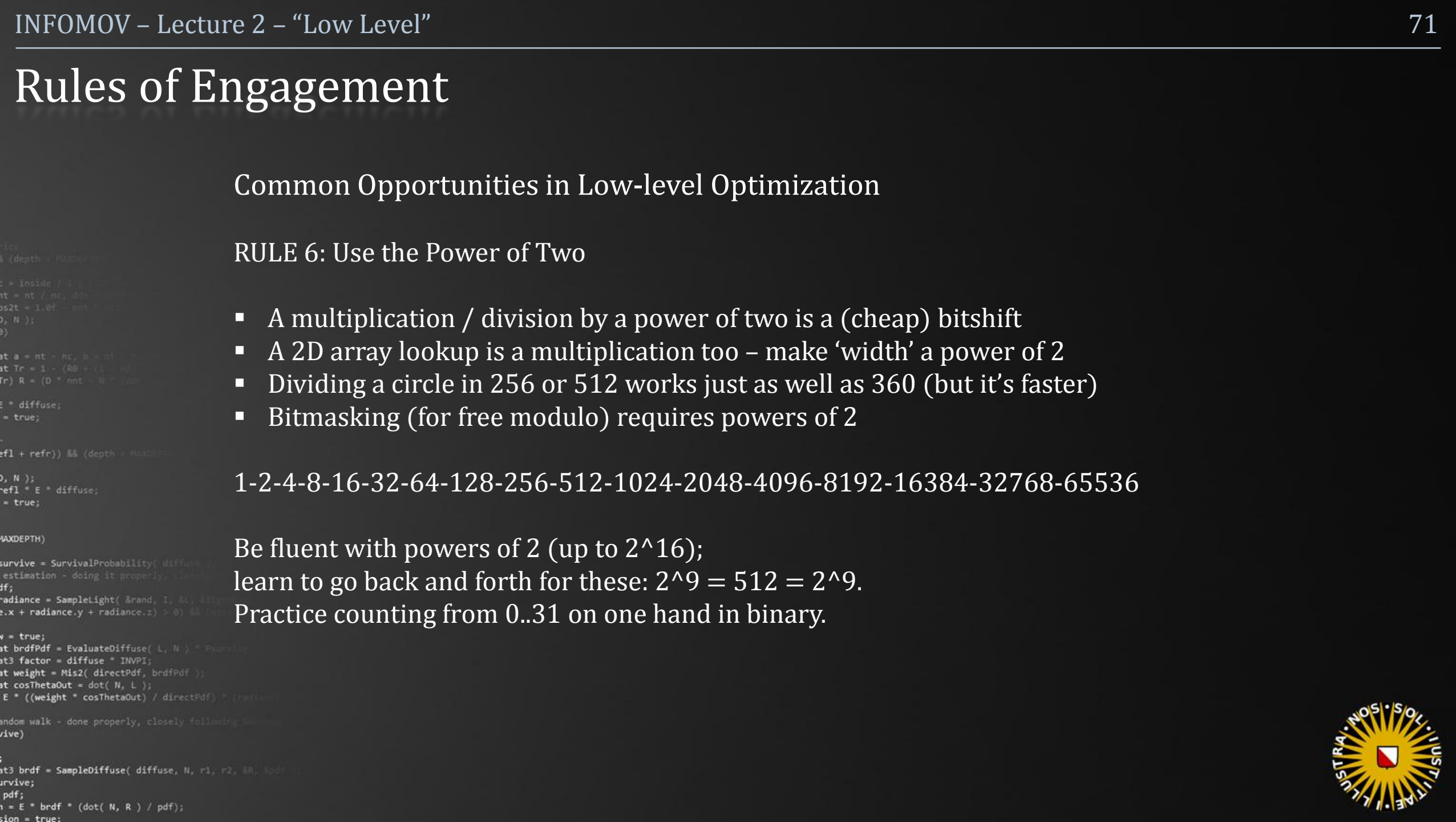
Common Opportunities in Low-level Optimization

RULE 6: Use the Power of Two

- A multiplication / division by a power of two is a (cheap) bitshift
- A 2D array lookup is a multiplication too – make ‘width’ a power of 2
- Dividing a circle in 256 or 512 works just as well as 360 (but it’s faster)
- Bitmasking (for free modulo) requires powers of 2

1-2-4-8-16-32-64-128-256-512-1024-2048-4096-8192-16384-32768-65536

Be fluent with powers of 2 (up to 2^16);
learn to go back and forth for these: 2^9 = 512 = 2^9.
Practice counting from 0..31 on one hand in binary.



Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 7: Do Things Simultaneously

- Use those cores
- An integer holds four bytes; use these for instruction level parallelism
- More on this later.

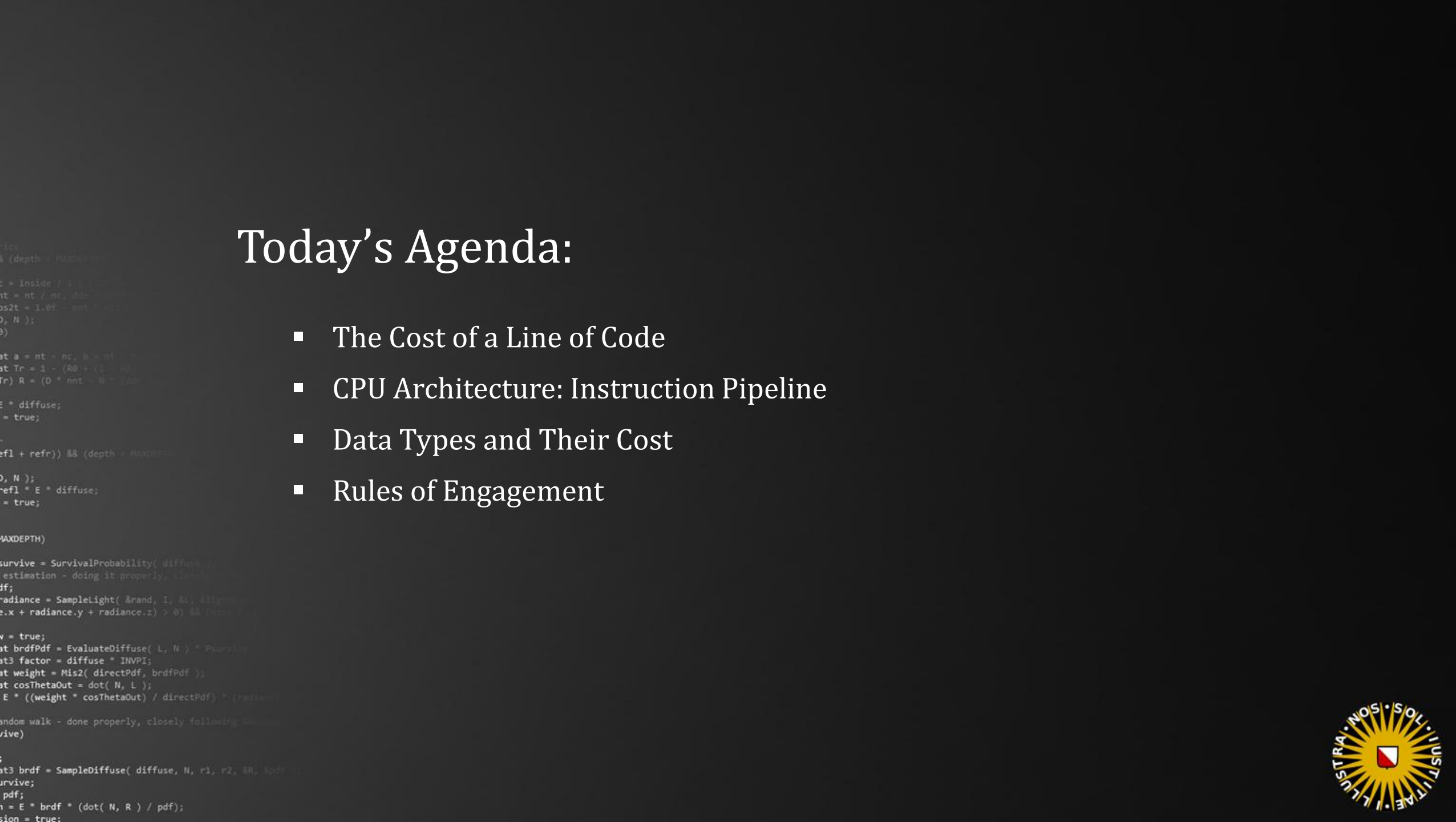
Rules of Engagement

Common Opportunities in Low-level Optimization

1. Avoid Costly Operations
2. Precalculate
3. Pick the Right Data Type
4. Avoid Conditional Branches
5. Early Out
6. Use the Power of Two
7. Do Things Simultaneously

Today’s Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



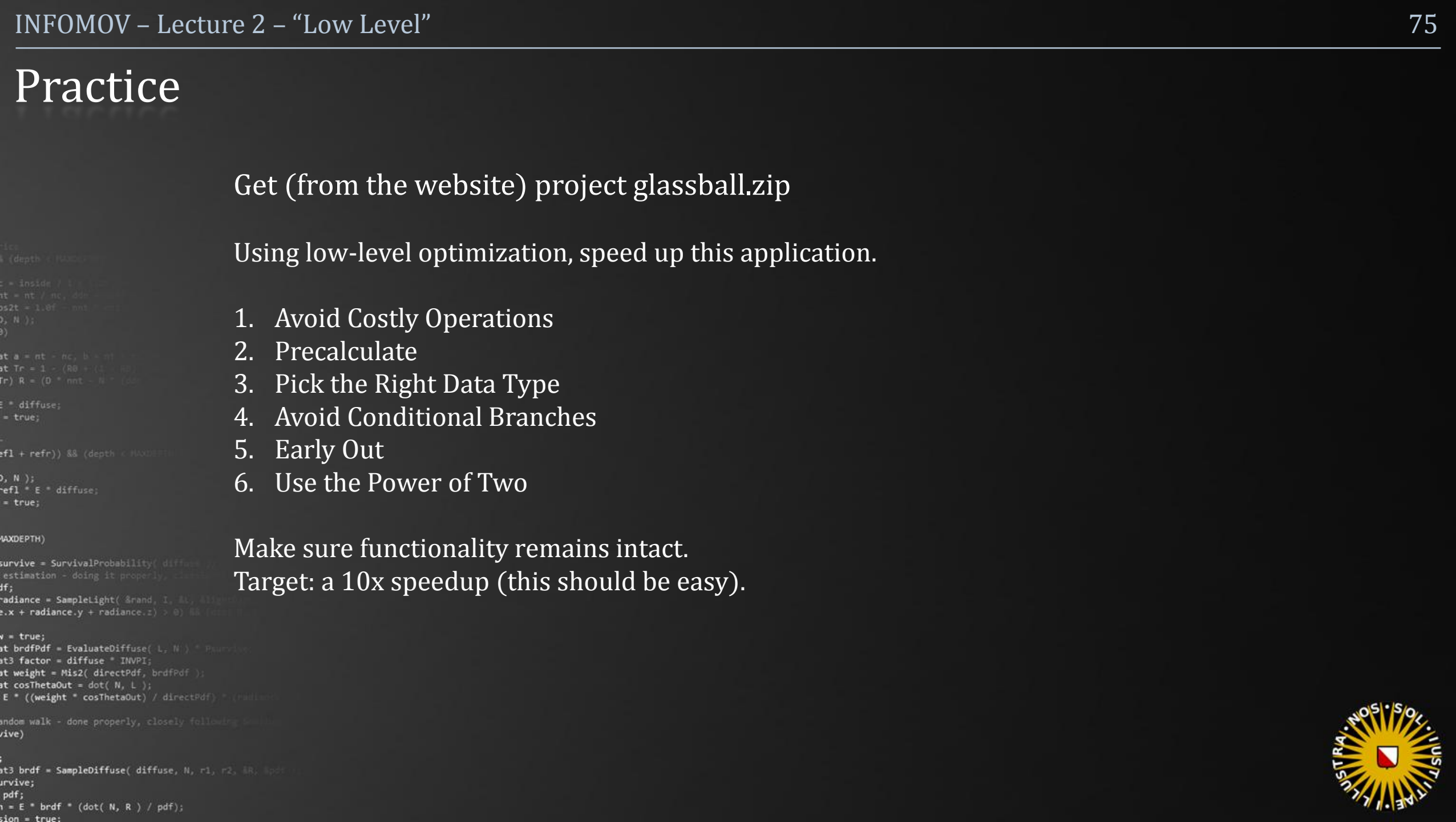
Practice

Get (from the website) project glassball.zip

Using low-level optimization, speed up this application.

1. Avoid Costly Operations
2. Precalculate
3. Pick the Right Data Type
4. Avoid Conditional Branches
5. Early Out
6. Use the Power of Two

Make sure functionality remains intact.
Target: a 10x speedup (this should be easy).



/INFOMOV/

END of “Low Level”

next lecture: “low level (2)”