/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2019 - Lecture 2: “Low Level”

Welcome!


Home
11
43
Jonathan Adamczewski Retweeted
Mike Conley @mconley@mastodon.social @mike_conley · 13h
Are you interested in helping us to make Firefox mind-meltingly fast?
Come work with me!
careers.mozilla.org/position/gh/17...
2
21
27

chat - bikker.j@gmail.com | Google Calendar - Week of S | Home / Twitter | Senior Firefox Performance E | Senior Firefox Performance E
https://careers.mozilla.org/position/gh/1769843/
gmail | news | misc | tech | games | coding | research | tools | ompf2 | huis | Netflix | Vulkan-glTF-PBR/V...

Senior Firefox Performance Engineer

Apply for this job

Team:
Engineering

Locations:
Remote Canada, Toronto, Vancouver

Why join Mozilla engineering?

Engineering: Open Positions

- [Cloud Operations Engineer]
- [Engineering Manager, Desktop Firefox Frontend]
- [Engineering Manager, WebAssembly]
- [Firefox Security Student Worker (Werkstudent)]

Paint Shop Pro - I... | Senior Firefox Perf... | INFOMOV | #optimization-and... | Slack | Ebilkill | Paral... | lecture2 - low level.... | ENG 09:30

/INFOMOV/
Optimization & Vectorization

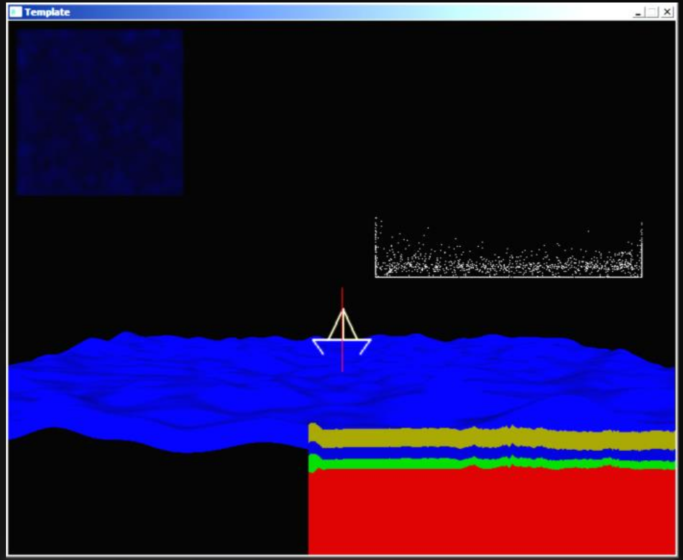
J. Bikker - Sep-Nov 2019 - Lecture 2: “Low Level”

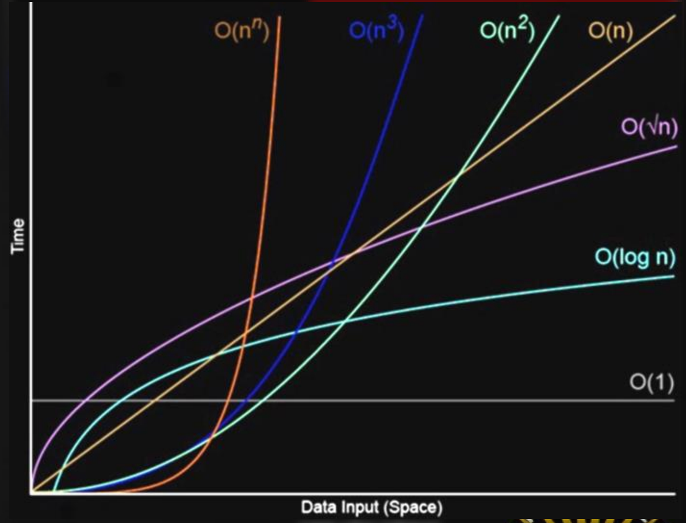
Welcome!

Previously in INFOMOV…

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.





Today’s Agenda:

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

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

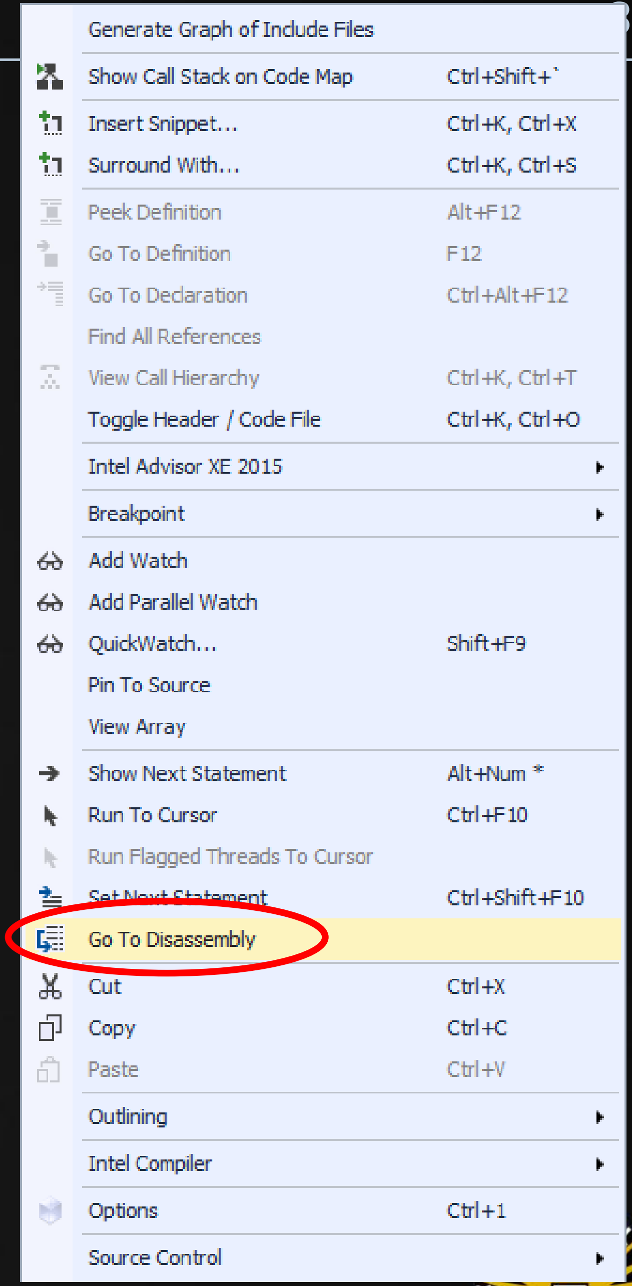
Better solution:

- 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	XMM0..XMM15
			YMM0..YMM15
			ZMM0..ZMM31

* 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						
CMP						
CMP						
INC, DEC, NEG						
INC, DEC, NEG						
AAA, AAS						
DAA						
DAS						
AAD						
AAM						
MUL, IMUL						latency ax=3, dx=4
MUL, IMUL	r16/m16	3	3	2	ALU0_1	
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, S						
ADC, S						
CMP						
CMP						
INC, D						
INC, D						
AAA						
AAS						
DAA						
DAS						
AAD						
AAM						
MUL, I						
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

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

	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		

Instruction Cost

Intel Skylake 2015

ADD SUB	r,r/i	1	1	p0156	1	0.25	
ADD SUB	r,m	1	2	p0156 p23		0.5	
ADD SUB	m,r/i	2	4	2p0156 2p237 p4	5	1	
ADC SBB	r,r/i	1	1	p06	1	1	
ADC SBB	r,m	2	2	p06 p23		1	
ADC SBB	m,r/i	4	6	3p0156 2p237 p4	5	2	
CMP	r,r/i	1	1	p0156	1	0.25	
CMP	m,r/i	1	2	p0156 p23	1	0.5	
INC DEC NEG NOT	r	1	1	p0156	1	0.25	
INC DEC NOT	m	3	4	p0156 2p237 p4	5-6	1	
NEG	m	2	4	p0156 2p237 p4	5-6	1	
AAA		2	2	p1 p56	4		not 64 bit
AAS							
DAA DAS							
AAD							
AAM							
MUL IMU							
MUL IMU							
MUL IMU							
MUL IMU							
MUL IMU							
MUL IMU							
MUL IMU							
MUL IMU							
IMUL							
IMUL	r,m	1	2	p1 p23		1	
IMUL	r16,r16,i	2	2	p1 p0156	4	1	
IMUL	r32,r32,i	1	1	p1	3	1	
IMUL	r64,r64,i	1	1	p1	3	1	
IMUL	r16,m16,i	2	3	p1 p0156 p23		1	
IMUL	r32,m32,i	1	2	p1 p23		1	
IMUL	r64,m64,i	1	2	p1 p23		1	
MULX	r32,r32,r32	3	3	p1 2p056	4	1	BMI2
MULX	r32,r32,m32	3	4	p1 2p056 p23		1	BMI2
MULX	r64,r64,r64	2	2	p1 p5	4	1	BMI2

Math							
FSCALE		27	27		130	130	
FXTRACT		17	17		11	11	
FSQRT		1	1	p0	14-21	4-7	
FSIN		53-105			50-120		
FCOS		53-105			50-130		
FSINCOS		55-120			55-150		
F2XM1		16-90			65-80		
FYL2X		40-100			103		
FYL2XP1		56			77		
FPTAN		40-112			140-160		
FPATAN		30-160			100-160		

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:

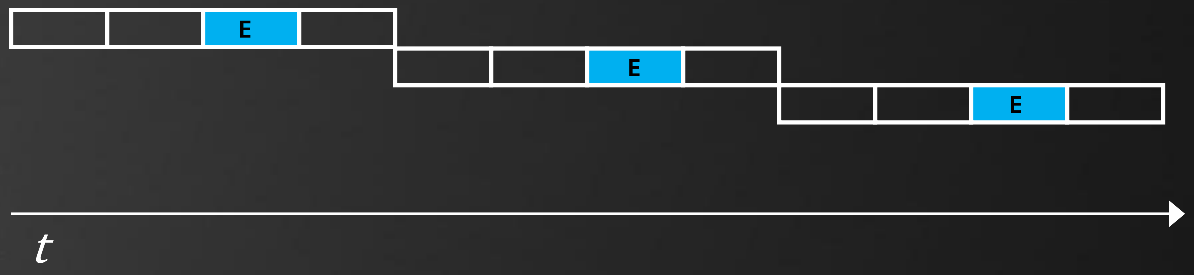
- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement

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



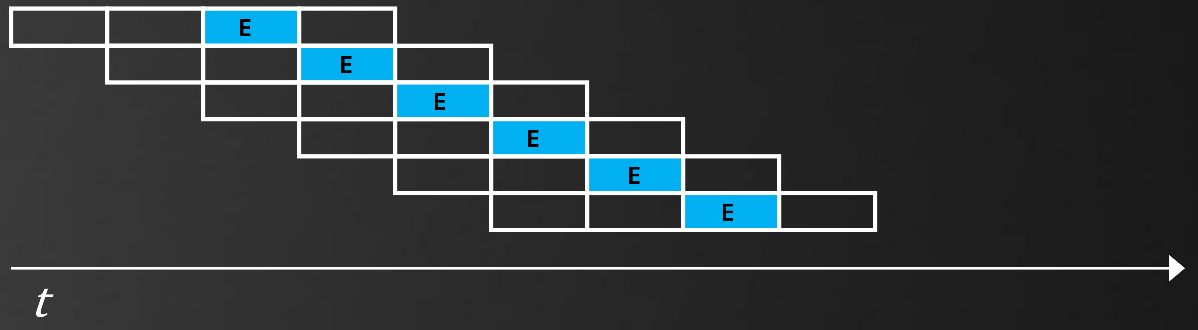
CPI = 4

```
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 (CPI = IPC = 1).

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

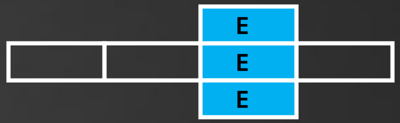
Obviously, ‘execution’ of different instructions requires different functionality.

Superpipelining allows higher clockspeeds and thus higher throughput, but it also increases the latency of individual instructions.

Pipeline

CPU Instruction Pipeline

Different execution units for different (classes of) instructions:



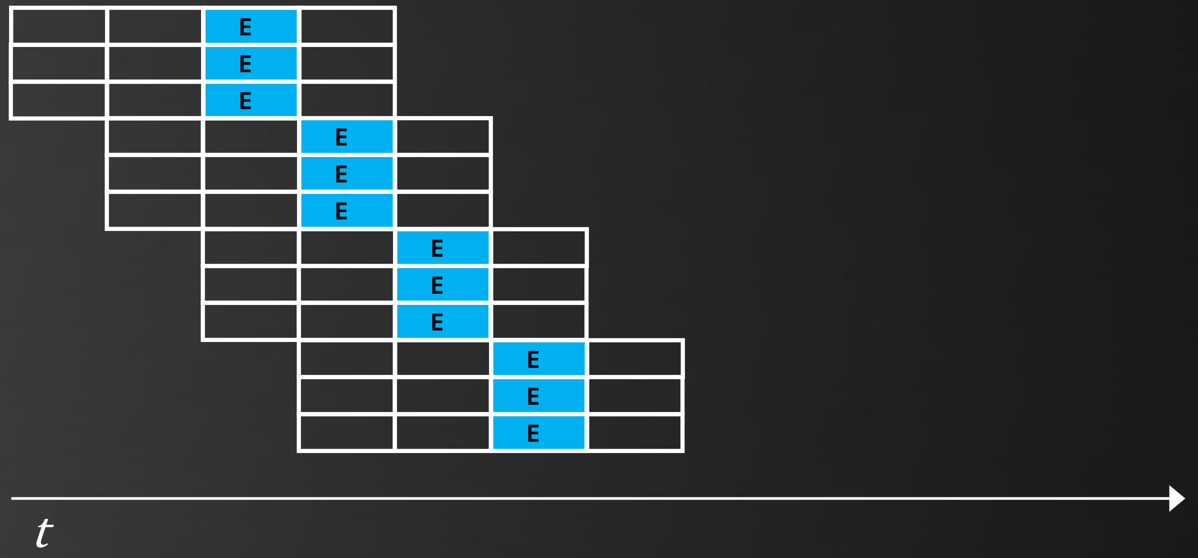
Here, one execution unit handles floats;
one handles integer;
one handles memory operations.

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.

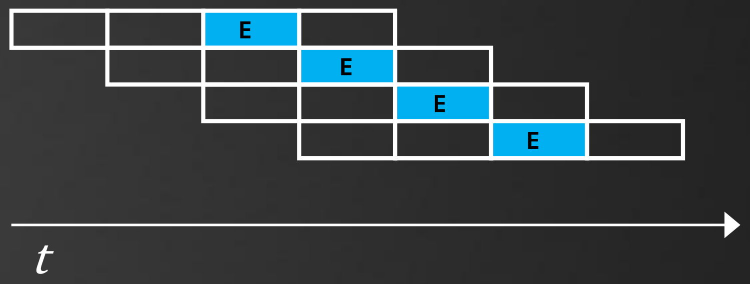


IPC = 3 (or: ILP = 3)

Pipeline

CPU Instruction Pipeline

Using a pipeline has consequences.
Consider the following situation:



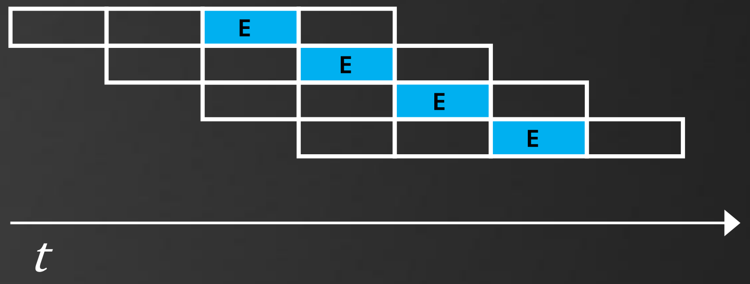

```
a = b * c;
d = a + 1;
```

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

Or check this very detailed study of the Nehalem architecture:

The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, Thomadakis, 2011.

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

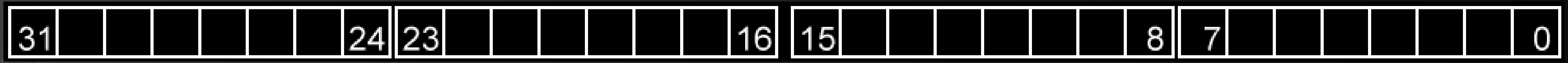
Today's Agenda:

- 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



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)

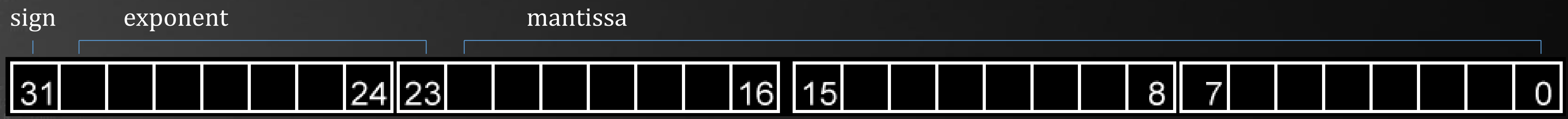
```
u4 ^= 1 << 31;
```

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

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

Exercise: write a function that replaces array a = { 0.5, 0.25, 0.125, 0.0625, ... }.

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
```

*: More on http://www.catb.org/esr/structure-packing

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);

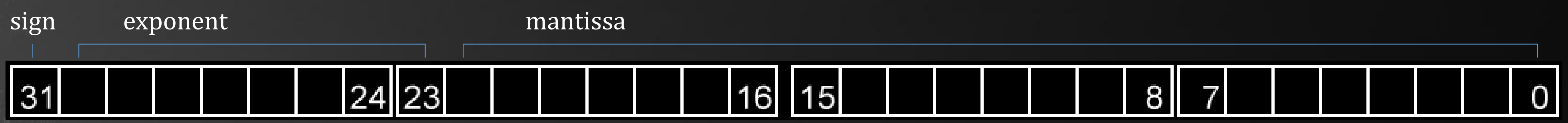
Avoiding conversion (2):

```
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

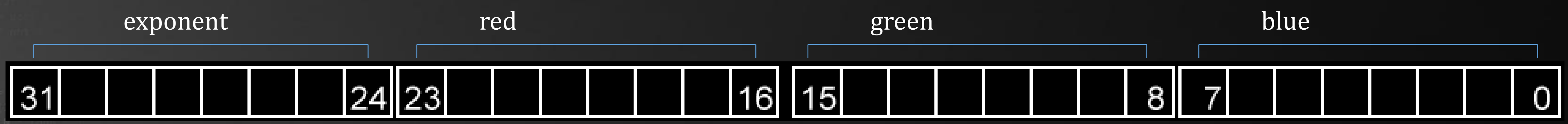



```
union { float v1; unsigned int u1; };
union { float v2; unsigned int u2; };

bool smaller = (v1 < v2);

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

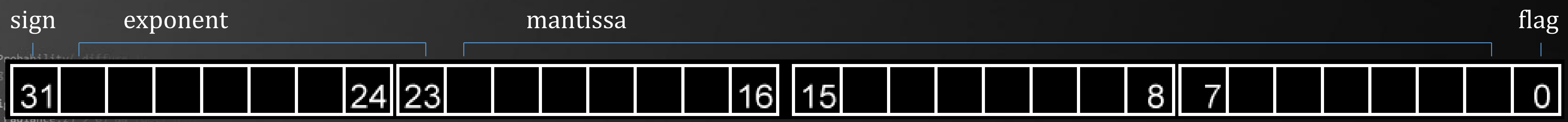
Data Types

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: “caching (1)”