/INFOMOV/

Optimization & Vectorization

J. Bikker - April-June 2024 - Lecture 2: “Low Level”

Welcome!

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

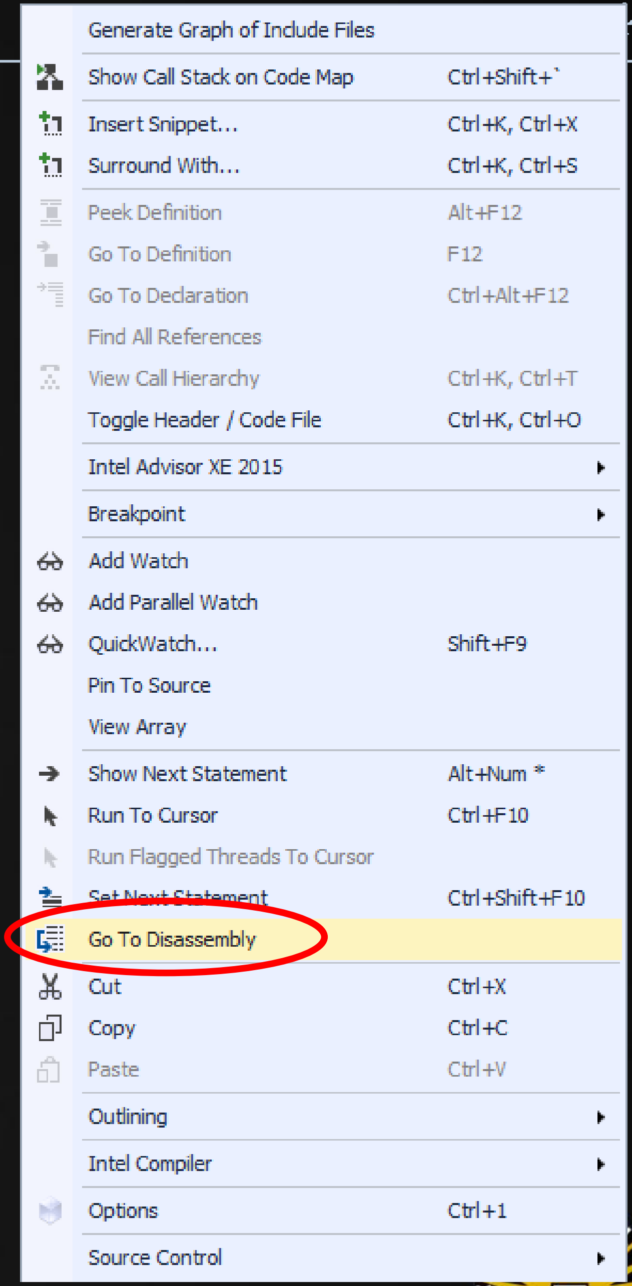
Better solution:

- 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 = 1, y = 1;
unsigned int i = 0, j = 0x28929227;
for( int k = 0; k < ITERATIONS; k++ )
{
    // ensure we feed our line with fresh data
    x += y, y *= 0.9999f;
    // 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, [illegible]						
ADC, [illegible]						
CMP						
CMP						
INC, D[illegible]						
INC, D[illegible]						
AAA						
AAS						
DAA						
DAS						
AAD						
AAM						
MUL, I[illegible]						
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	

Instruction	Operands	Ops	Latency	Reciprocal throughput	Execution pipe	Notes
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 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 IMUL							
MUL IMUL							
MUL IMUL							
MUL IMUL							
MUL IMUL							
MUL IMUL							
MUL IMUL							
MUL IMUL							
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

AMD “Zen 4” 2022

Arithmetic instructions						
ADD, SUB	r,r	1	1	0.25		
ADD, SUB	r,i	1	1	0.25		
ADD, SUB	r,m	1		0.33		
ADD, SUB	m,r8/16	2	7-8	1		
ADD, SUB	m,r32/64	2	1	1		may mirror
ADC, SBB	r,r	1	1	1		
ADC, SBB	r,i	1	1	1		
ADC, SBB	r,m	1	1	1		
ADC, SBB	m,r8/16	2	8	1		
ADC, SBB	m,r32/64	2	1	1		may mirror
ADCX ADOX	r,r	1	1	1		ADX
CMP						
CMP						
CMP						
CMP						
INC, DEC, I						
INC, DEC, I						
AAA, AAS						
DAA						
DAS						
AAD						
AAM		4	13	4		
MUL, IMUL	r8/m8	1	3	1		
MUL, IMUL	r16/m16	3	3	2		
MUL, IMUL	r32/m32	2	3	1		
MUL, IMUL	r64/m64	2	3	1		
IMUL	r,r	1	3	1		
IMUL	r,m	1		1		
IMUL	r16,r16,i	2	4	1		

FSQRT			1	25	10	
FLDPI, etc.			1		1	
FSIN			12-60	50-200		P0 P1
FCOS			18-60	60-150		P0 P1
FSINCOS			12-100	80-150		P0 P1
FPTAN			10-60	60-120		P0 P1
FPATAN			10-100	50-190		P0 P1
FSCALE			8	11	4	P0 P1
FXTRACT			13	12	5	P0 P1
F2XM1			10-18	50-60		P0 P1
FYL2X			10-60	40-60		P0 P1
FYL2XP1			70	~170		P0 P1

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 — Decode the byte code
3. Execute — Execute the instruction
4. Writeback — Write result to RAM/registers

```
xor esi,esi
add ecx, edx
dec ecx
```

i	1	2	3	4	5	6	7	8	9	10	11	12
i_0	IF	ID	EX	WB								
i_1					IF	ID	EX	WB				
i_2									IF	ID	EX	WB

t

CPI = 4

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*.

i	1	2	3	4	5	6	7	8	9
i_0	IF	ID	EX	WB					
i_1		IF	ID	EX	WB				
i_2			IF	ID	EX	WB			
i_3				IF	ID	EX	WB		
i_4					IF	ID	EX	WB	
i_5						IF	ID	EX	WB

t

At the same clock speed, we get four times the throughput (CPI = IPC = 1).

Pipeline

CPU Instruction Pipeline

In practice, each of the pipeline phases takes several cycles to complete.

i	1	2	3	4	5	6	7	8	9
i_0	IF	ID	ID	ID	EX	WB			
i_1		IF			ID	ID	ID	EX	WB

t →

Pipeline

CPU Instruction Pipeline

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

i	1	2	3	4	5	6	7	8	9	10	11	12	13
i_0					EX	EX	EX						
i_1						EX	EX	EX					
i_2							EX	EX	EX				
i_3								EX	EX	EX			
i_4									EX	EX	EX		

t →

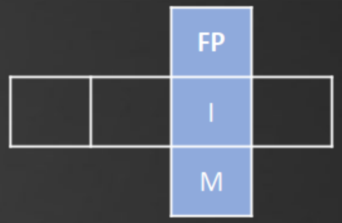
Stages	
7	PowerPC G4e
8	Cortex-A9
10	Athlon
12	Pentium Pro/II/III, Athlon 64
14	Core 2, Apple A7/A8
14/19	Core i2/i3 Sandy Bridge
16	PowerPC G5, Core i*1 Nehalem
18	Bulldozer, Steamroller
20	Pentium 4
31	Pentium 4E Prescott

Super-pipelining allows higher clock speeds and thus higher throughput, but it also increases the latency of individual instructions.

Pipeline

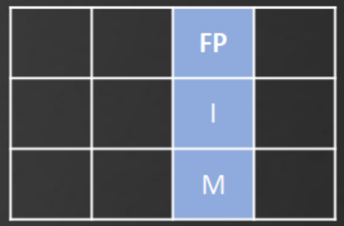
CPU Instruction Pipeline

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.

i	1	2	3	4	5	6
fp_0			EX			
int_0			EX			
m_0			EX			
fp_1				EX		
int_1				EX		
m_1				EX		
fp_2					EX	
int_2					EX	
m_2					EX	

t

IPC = 3 (or: ILP = 3)

Pipeline

CPU Instruction Pipeline

Using a pipeline has consequences. Consider the following situation:

```
a = b * c;
d = a + 1;
y = y >> 1;
z = 0x1a4;
```

i	1	2	3	4	5	6	7	8	9	10
i_0	IF	ID	EX	EX	WB					
i_1		IF	ID		EX	WB				
i_2			IF	ID		EX	WB			
i_3				IF	ID		EX	WB		

t →

Pipeline

CPU Instruction Pipeline

Using a pipeline has consequences. Consider the following situation:

```
a = b * c;
d = a + 1;
y = y >> 1;
z = 0x1a4;
```

i	1	2	3	4	5	6	7	8	9	10
i_0	IF	ID	EX	EX	WB					
i_1		IF	ID			EX	WB			
i_2			IF	ID			EX	WB		
i_3				IF	ID			EX	WB	

t

We now lose one cycle!

Pipeline

CPU Instruction Pipeline

Using a pipeline has consequences. Consider the following situation:

```
a = b * c;
d = a + 1;
y = y >> 1;
z = 0x1a4;
```

i	1	2	3	4	5	6	7	8	9	10
i_0	IF	ID	EX	EX	WB					
i_1		IF	ID			EX	WB			
i_2			IF	ID	EX			WB		
i_3				IF	ID		EX		WB	

t →

Out-of-order execution requires instructions to be *retired* in-order. An instruction is retired when its result is written back to memory.

Pipeline

CPU Instruction Pipeline

A good compiler re-organizes your code to maximize throughput.

```
a = b * c;
y = y >> 1;
d = a + 1;
z = 0x1a4;
```

i	1	2	3	4	5	6	7	8	9	10
i_0	IF	ID	EX	EX	WB					
i_1		IF	ID		EX	WB				
i_2			IF	ID		EX	WB			
i_3				IF	ID		EX	WB		

t →

Pipeline

CPU Instruction Pipeline

So how do we utilize out-of-order execution:

- A compiler reorganizes code to prevent latencies
- The CPU reorganizes instructions to prevent latencies
- Feeding mixed code provides compiler and CPU with opportunities for shuffling

One little problem remains:

What if the CPU doesn't know what the next instruction is?

Pipeline

Speculative Execution

The CPU *guesses* what a branch will do:

```
if (a < b) foo(); else bar();
```

For this it uses *branch prediction*.
Instructions at the predicted location will be executed even if it is not sure that this is correct.

A *branch misprediction* requires that the pipeline is flushed and re-populated…

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;
```

31							24	23							16	15							8	7							0

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);
```

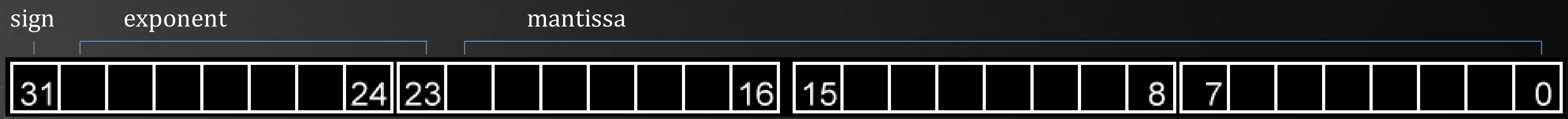
Altering sign bit of s4:
(note: -1 = 0xffffffff)

```
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

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;
}
```

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

/INFOMOV/

END of “Low Level”

next lecture: “Profiling”