Report on Assignment 1

Ayushi Agarwal
[redacted]

1 Introduction

The goal of this study is to analyze the latency and throughput of various x86 instructions to study the performance of an instruction.
Latency and throughput both are important parameters when it comes to analyzing the instructions performance on a processor. Pipelining in computer architecture allows a trade-off between latency and throughput of an instruction. While it is important that the instruction takes less time to execute, it is also important that the number of instructions executed per cycle be high. So, we have seen that some of the architecture choices do take a latency hit if the throughput is improving significantly because that leads to faster program execution.

2 Method

To study the performance of instructions on a microprocessor, multiple variants of the same instruction or multiple instructions are joined in the same line. Different types of addressing modes and operands are used for creating different flavours of the same instruction.

This report studies various instruction sets including general purpose registers, floating point instructions, logical instructions, MMX instructions and SSE instructions. Proper initialization sequences for various registers (GPR, Floating-Point, MMX, XMM) is to be done before executing instructions.
Several macros are created to place 8 instructions (described more in Section 2.1) back-to-back 25 times and the macro is executed 8000 times[1]. We will discuss the macros in Section 2.2.

$$\begin{aligned} Total\ Instructions = (8\ instructions) * \\ (repeat\ 25\ times) * \\ (8000\ times) \\ = 1,600,000\ instructions \end{aligned} \tag{1}$$

The time unit is CPU clock cycles for all the results. The RDTSC (RD Timestamp Counter) instruction is used to get the CPU clock information.

$$CPI = \frac{(Ending\ Clock - Starting\ Clock)}{1,600,000} \tag{2}$$

As in equation (2), CPI stands for clocks per instruction. The Starting and Ending clocks are measured using RDTSC instruction. 1, 600, 000 is the total number of instructions executed as shown in equation (1).
There are some extra clock cycles due to the setup of loops for the instruction block. This is accounted as the overhead clock cycles and is subtracted from the actual calculation. So the modified form of equation (1) where we account for the overhead clocks in the measurement of CPI is shown in equation (3).

$$CPI = \frac{(End\,Clock - Start\,Clock - Overhead)}{1,600,000} \tag{3}$$

2.1 Why Latency and Throughput

Latency: It is the number of clock cycles that the instructions takes to make the data available for the next instruction. It is the delay that any instruction generates in a data dependent chain of instructions.
Throughput: It is measured as the total number of instructions of similar kind that can be executed per clock cycle. Generally, some companies tend to report the reciprocal throughput for their processors. Reciprocal throughput is the number of clock cycles it takes for an instruction to complete its execution and the operands of the instructions should not be dependent on the preceding instructions.
Example: If an instruction has a reciprocal throughput of 3, it means that the next instruction of similar kind will start executing after 3 clock cycles.

- To measure the latency, a dependency chain of identical/similar instructions(in this report we use 8 instructions) is created so that the output of each instruction is needed as input for the next instruction. This helps to measure the delay caused by an instruction in the pipeline.

- To measure the throughput, a sequence of instructions (in this report we use 8 instructions) of the

same kind with no inter-dependence are created. Each instruction uses different pair of registers to avoid data dependency. This is done to avoid any bottlenecks apart from those created by the instruction itself.

This can be explained using some MACRO definitions from the source code for Throughput and latency as below:

1. THPT_X2X(x): This Macro is for calculating the throughput for addressing mode $X2X$. The instructions are combined in a set of 8 instructions such that no instruction is dependent on the preceding instruction. So each instruction actually works on different pair of registers. This can be seen in the source code in file $latthpt\_gnu.h$. So if the first instruction works on xmm0 and xmm1, the next instruction uses xmm2 and xmm3 and so on. There is no data dependency between two adjacent instructions.

2. LAT_X2X(x): This Macro is for calculating the latency for addressing mode $X2X$. The instructions are combined in a set of 8 instructions such that there is a dependency chain where the output of each instruction is needed as input to the next instruction. So if first instruction uses xmm0 and xmm7, the second instruction would use xmm1 and xmm0 (hence creating a dependency) and so on.
For some addressing modes it is difficult to bring out this dependency chain because all instructions work on independent set of operands and registers.
Example: M2X - The instructions here would try to move data from MMX register to XMM register. So each instruction works on reading MMX register and writing to XMM register. There is no dependency between two adjacent instructions. Similar is the case with memory loads and stores until there is a dependency such that there is a store to the memory followed by a load from the same address. We discuss this more in Section 3.

2.2 Adding New Macros

1. **Adding new Register type :** The source code was modified to add a new register type for YMM instructions. Apart from creating just the addressing mode macros for YMM registers, we first have to define the register type for the CPU to understand. So we modified the $cpu.h$ file to add YMMS register group with 8 256-bit registers. Then new macros have to be added for the addressing modes like $Y2Y, Y2m, m2Y$ in the $macro\_asm.h$ file.

2. **Adding new addressing modes:** This is done by modifying the $latthpt\_gnu.h$ file with new macros corresponding to latency or throughput for a particular addressing mode. If the addressing mode is not defined in $macro\_asm.h$ file, then it needs to be defined first before defining LAT/THPT macro's.

2.3 Instructions reported

Table 1 shows all the instructions of MMX and SSE instruction set that were analyzed for their latency and throughput. This is not an exhaustive table for all the results that were obtained by the source code. We report only some interesting observations and newly added instructions. The columns that are left empty means the values were not reported for them.

Table 2 shows the instructions for general purpose registers and floating point registers.

Inst	Type	Latency	Thpt
MMX MOVQ	$M \leftarrow M$	3.3	0.5
	$mem \leftarrow M \leftarrow mem$	6.0	
MMX MOVQ2DQ	$X \leftarrow M$		1.0
YMM VMOVDQA	$mem \leftarrow Y$	1.1	
	$Y \leftarrow mem$	0.5	
	$mem \leftarrow Y \leftarrow mem$	2.8	
XMM VMOVDQA	$X \leftarrow X$	0.5	0.5
	$X \leftarrow mem$	0.4	0.4
	$mem \leftarrow X$	0.9	0.9
	$mem \leftarrow X \leftarrow mem$	2.2	
XMM MOVDQ2Q	$M \leftarrow X$		0.9
XMM FP MOVAPD	$X \leftarrow X$	0.6	0.2
	$X \leftarrow mem$	0.5	0.9
	$mem \leftarrow X$	1.0	0.9
	$mem \leftarrow X \leftarrow mem$	2.2	

Table 1: Latency and throughput numbers for MMX, XMM and YMM registers instructions.

3 Results and Analysis

Table 1 and Table 2 in Section 2.3, report the latency and throughput numbers of various instructions that were measured using the method described in Section 2. This Section highlights some of the peculiar observations made during the experiment and a detailed analysis of the reported numbers.

Some Observations :

The first set of instructions in the program encounter more latency and throughput. This is an interesting observation which we think might be caused by the setting of the pipeline in the beginning of the program. We observe that no matter what set of instructions are executed first in the program, the first few instructions (of the order of 800 instructions - this was observed by doing an object dump/*objdump* from the executable code) always incur more clock cycles than later instructions. This is the reason why a simple MMX register to register move shows a latency of 3.3 cycles as reported

Inst	Type	Latency	Thpt
bswap	$E \leftarrow E$	1.6	0.9
bsf	$R \leftarrow mem$	2.8	1.5
mov	$mem \leftarrow E$	0.9	0.9
	$E \leftarrow mem$	0.0	
	$E \leftarrow E$		0.9
	$R \leftarrow mem$	0.5	
	$mem \leftarrow R$	0.9	
	$mem \leftarrow R \leftarrow mem$	2.0	
logical	$mem \leftarrow R$	4.9	5
and	$R \leftarrow mem$	1.0	0.6
mul	$EDA \leftarrow E$	3.3	3.9
XCHG	$R \leftarrow mem$	15.8	15.9
	$E \leftarrow E$		1.8
FADD	$F \leftarrow mem$	2.9	2.6
	after fstp in queue	425.5	318.6
FLD	$F \leftarrow mem$	316.0	315.7
FSTP	$mem \leftarrow F$	392.9	396.1
FST	$mem \leftarrow F$	0.9	0.9

Table 2: Latency and throughput numbers for GPR and FP registers instructions.

in Table 1. If this instruction is executed later in the code, it reports a latency of 1.2 cycles.

Some instructions execution may be interfered by a previous instruction in the queue. Like, if a floating point store is done before a floating point add, the latency of add also increases comparable to the store latency. This is also shown in Table 2. This may be caused by some internal dependencies in the pipeline that are not visible in software. Also, in our method we are using RDTSC counter to count the number of clocks. It can sometimes give inconsistent results. Another possible explanation is that it may be because that part of the code doesn't fit in the L1 cache and keeps getting evicted. More detailed analysis requires some cycle accurate simulator to study the caches as well.

As discussed in Section 2.1, the latency of some versions of addressing modes (example, a memory read or write) cannot be calculated accurately using software methods. Only for instructions that operate with same type of registers(possible to create dependency chain), the latency macros would work correctly. When going from one register type to different register type (like MMX to XMM or Register to memory and vice-versa), the latency cannot be accurately measured as they are implemented differently inside the processor[2].

Hence, we measure the combined latency of memory to register and back to memory ($mem \leftarrow Reg \leftarrow mem$). But there is no clear division between the individual latency's because the processor might be internally doing some forwarding of data to avoid data going to the cache and back.
Hence, some individual latency numbers of memory read/write reported in Table 1 and 2, don't hold much value.

The latency of FST and FSTP instructions differ significantly. Couldn't come up with a clear reasoning for this. Can a stack pop take so many cycles?

A Peculiar Instruction XCHG : This instruction takes many cycles to execute when one of the operands is in the memory. This is because when a memory operand is referenced in a XCHG instruction, the processor implements the locking protocol for the duration of the exchange operation irrespective of the presence or absence of a LOCK prefix[3]. Table 2 shows the difference in latency incurred by this instruction if both the operands are general-purpose registers and there is no memory involvement.

Similar observation is expected for other instructions having a LOCK prefix.

4 Conclusions

This report is not an exhaustive analysis of all the x86 instructions but it brings out important observations of how various instructions are executed and how the execution can vary according to the addressing modes used. We also see that both latency and throughput are important parameters to analyze instruction performance.

5 References

1. Intel Software's Article on Latency and Throughput
2. Agner Fog's Instruction Table
3. Intel x-86 Software Developer ISA Manual