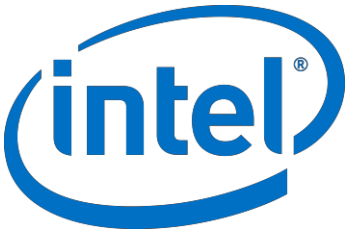

# Intel® Architecture Code Analyzer

## User's Guide



Document Number: 321356-001US

Revision: 3.0

World Wide Web: http://www.intel.com

# Legal Information

# Contents

# 1 Introduction

Intel® Architecture Code Analyzer helps you statically analyze the throughput of instruction sequences (kernels) on Intel® microarchitectures.

For a given binary, Intel Architecture Code Analyzer:

- Identifies the binding of the kernel instructions to the processor ports under ideal front-end, out-of-order engine and memory hierarchy conditions.
- Performs static analysis of the kernel throughput and reports its cycle count.

## 1.1 Intel® Architecture Code Analyzer Accuracy

Intel Architecture Code Analyzer enables you to do a first order **estimate** of the relative performance of sections of code on different microarchitectures. It **does not** provide absolute performance numbers.

The performance data reported by the tool may significantly deviate from actual performance observed on an Intel® processor. You can achieve the most accurate throughput measurements by executing the analyzed code on the processor itself. The Intel® Architecture Code Analyzer complements such measured data with information on port binding, bottlenecks, and critical paths.

## 1.2 Processor Support

Intel Architecture Code Analyzer supports analysis for 4th to 6th generation Intel® Core™ processors, which correspond to Intel® microarchitectures codenamed Haswell (4th gen), Broadwell (5th gen) and Skylake (6th gen), including Skylake Server.

## 1.3 Platform Support

Intel Architecture Code Analyzer is a command-line utility that can analyze a binary file that contains code with special markers that delimit the analyzed code. The tool analyses Intel® 64 bit code including Intel® Advanced Vector Extensions (Intel® AVX), AVX2 and AVX-512 instructions.

Intel Architecture Code Analyzer is available on Windows*, Linux* and Mac OS X* operating systems (64-bit editions).

| | |
|---|---|
| ***NOTE:*** | Intel® Architecture Code Analyzer has been validated on 64-bit SUSE* 11, Mac OS X* 10.12.1 and Microsoft* Windows 8.1 64-bit. It should work on other versions of Linux*, Mac OS X* and Microsoft* Windows operating systems. |

# 2 Analysis

## 2.1 Throughput Analysis

Throughput Analysis is used to analyze the throughput and bottlenecks of a loop body; it treats the contents of the analyzed block as an infinite loop, including considering inter-iteration dependencies between instructions within the analyzed block. The Throughput Analysis report provides the following information:

- Throughput of the whole analyzed block, counted in cycles. The block throughput is calculated as the maximum between:
  - Throughput of the processor's ports
  - Maximum front-end throughput (4 micro-ops per cycle)
  - Divider unit throughput
- Bottleneck source that limited the throughput: front-end, port number, divider unit, or long dependency chains.
- Total number of cycles each processor port was bound by micro-ops.

The detailed section of the throughput analysis report contains one line for each instruction in the analyzed block. Each line contains:

- Number of the instruction micro-ops.
- Average number of cycles per iteration that the instruction was bound to each processor port. For most instructions this simply means the number of cycles the instruction was bound to each port. However, if a particular micro-op may execute on more than one port, the average number of cycles per iteration may be a partial cycle for each port because that micro-op may bind to a different port on each iteration.
- Instruction disassembly in Intel® Software Developer's Manual (MASM) style

Some ports have both a regular pipe and a secondary pipe. These ports are separated by a hyphen, and look like two separate ports in the detailed report. Specifically:

- Port 0 has the Divider pipe split from it. In the first cycle they are both busy, then port 0 is available for the next micro-op and the Divider pipe is kept busy for the duration of the divide operation.
- Load ports 2 and 3 have an Address Generation Unit (AGU) split from them.

Following is an example Throughput Analysis report:

*Analysis*

```
Throughput Analysis Report
--------------------------
Block Throughput: 5.00 Cycles       Throughput Bottleneck: Dependency chains
Loop Count:  23
Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------------------------
|  Port  |   0   -  DV   |   1   |   2   -  D    |   3   -  D    |   4   |   5   |   6   |   7   |
---------------------------------------------------------------------------------------------
| Cycles |  2.5     0.0  |  2.5  |  2.0     2.0  |  2.0     1.0  |  1.0  |  4.0  |  2.0  |  0.0  |
---------------------------------------------------------------------------------------------

DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3)
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion occurred
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of   |                    Ports pressure in cycles                         |      |
|  Uops    |  0  - DV    |  1   |  2  -  D    |  3  -  D    |  4   |  5   |  6   |  7   |
---------------------------------------------------------------------------------------
|   1      |             |      | 1.0     1.0 |             |      |      |      |      | mov r10, qword ptr [rbp+0x170]
|   1      |             | 1.0  |             |             |      |      |      |      | lea r9d, ptr [r8*4]
|   1      |             |      |             |             |      |      | 1.0  |      | movsxd r9, r9d
|   1      |             |      |             |             |      |      | 1.0  |      | inc r8d
|   1      |             |      |             | 1.0     1.0 |      |      |      |      | mov r11, qword ptr [rbp+0x178]
|   1      |             |      | 1.0     1.0 |             |      |      |      |      | vmovups xmm3, xmmword ptr [r10+r9*4]
|   1      |             |      |             |             |      | 1.0  |      |      | vpslldq xmm2, xmm3, 0x4
|   1      |             |      |             |             |      | 1.0  |      |      | vpslldq xmm4, xmm3, 0x8
|   1      | 1.0         |      |             |             |      |      |      |      | vaddps xmm6, xmm2, xmm3
|   1      |             |      |             |             |      | 1.0  |      |      | vpslldq xmm5, xmm3, 0xc
|   1      | 0.5         | 0.5  |             |             |      |      |      |      | vaddps xmm7, xmm4, xmm5
|   1      | 0.5         | 0.5  |             |             |      |      |      |      | vaddps xmm8, xmm6, xmm7
|   1      | 0.5         | 0.5  |             |             |      |      |      |      | vaddps xmm9, xmm8, xmm0
|   1      |             |      |             |             |      | 1.0  |      |      | vshufps xmm0, xmm9, xmm9, 0xff
|   2      |             |      |             | 1.0         | 1.0  |      |      |      | vmovups xmmword ptr [r11+r9*4], xmm9
|   1*     |             |      |             |             |      |      |      |      | cmp r8d, esi
|   0*F    |             |      |             |             |      |      |      |      | jl 0xfffffffffffffb0
Total Num Of Uops: 17
```

## 2.2 Trace

To generate a trace use '–trace <path>' option to generate a trace file in <path>.

Traces include in-depth information about different operation stages inside the processor. A trace can be used to identify bottlenecks and pressure points.

```
it|in|Dissasembly                                  :0123456789012345678901234567890123456
 0| 0|mov r10, qword ptr [rbp+0x170]               :          |         |         |
 0| 0|    TYPE_LOAD (1 uops)                       :s---deeeew----R-------p       |
 0| 1|lea r9d, ptr [r8*4]                         :          |         |         |
 0| 1|    TYPE_OP (1 uops)                        :sdw----------R-------p       |
 0| 2|movsxd r9, r9d                              :          |         |         |
 0| 2|    TYPE_OP (1 uops)                        :A-dw----------R-------p       |
 0| 3|inc r8d                                     :          |         |         |
 0| 3|    TYPE_OP (1 uops)                        :sdw----------R-------p       |
 0| 4|mov r11, qword ptr [rbp+0x178]              :          |         |         |
 0| 4|    TYPE_LOAD (1 uops)                      : s---deeeew----R-------p      |
 0| 5|vmovups xmm3, xmmword ptr [r10+r9*4]        :          |         |         |
 0| 5|    TYPE_LOAD (1 uops)                      : A-------deeeeew----R-------p |
 0| 6|vpslldq xmm2, xmm3, 0x4                     :          |         |         |
 0| 6|    TYPE_OP (1 uops)                        : A-------------dw----R-------p|
 0| 7|vpslldq xmm4, xmm3, 0x8                     :          |         |         |
 0| 7|    TYPE_OP (1 uops)                        : A-------------cdw----R-------p
 0| 8|vaddps xmm6, xmm2, xmm3                     :          |         |         |
 0| 8|    TYPE_OP (1 uops)                        :  A-------------deeew----R-------p
 0| 9|vpslldq xmm5, xmm3, 0xc                     :          |         |         |
 0| 9|    TYPE OP (1 uops)                        :  A------------ccdw------R-------p
```

Above is an example of a trace output.

The kernel instructions are modeled, in order, from top to bottom while the processor's cycles run from left to right. The 'it' column shows the iteration count of the entire kernel, the 'in' column shows the instruction count within the kernel and the 'Disassembly' column shows the instruction's disassembly, along with the micro-architectural instruction fragment information. By default the first 150 cycles of the modeled execution are displayed.

Each instruction is represented by at most 4 instruction-fragments (OP, STORE DATA, STORE ADDRESS,LOAD). The trace displays the micro-architectural stage of each fragment inside the processor at any given cycle from allocation to retire and even post retire. If two stages happen at the same cycle the most important one of is shown. Specifically when Alloc & sready stages happen at the same time the sready stage is shown.

The stages and possible states are:
[A] – Allocated
[s] – Sources ready
[c] – Port conflict
[d] – Dispatched for execution
[e] – Execute
[w] – Writeback
[R] – Retired
[p] – Post Retire
[-] – pending
[_] – Stalled due to unavailable resources

# 2.3 Analysis Report Notes

## 2.3.1 Unbound Instructions

Some instructions do not require a processor functional unit to complete their execution. For example, a `xor eax, eax` instruction does not require an execution port because the register is directly set to 0. As a result, their micro-ops are not bound to any port. Instructions that are not bound to a port are marked with a '*' character next to their number of micro-ops.

## 2.3.2 Combining 256-bit Intel® AVX and Legacy Intel® SSE

Transitioning between 256-bit Intel® AVX instructions and legacy Intel Streaming SIMD Extensions (Intel® SSE) instructions will cause performance penalties. Intel® Architecture Code Analyzer detects these transitions between 256-bit Intel® AVX and legacy Intel® SSE within the analyzed block, and **ignores** the associated performance penalty in the total throughput and total latency summary report. Instead, the summary report includes two additional lines at the top indicating that such sequence(s) exist in the analyzed block, and marks the first transition instruction with a '@' character in the Num of Uops columns.

For more information on transitions between Intel® AVX and Intel® SSE, see Avoiding AVX-SSE Transition Penalties.

## 2.3.3 Unsupported Instructions

Intel® Architecture Code Analyzer does not support a small subset of the Intel® Architecture Instruction Set. When it reaches an unsupported instruction in the analyzed block it ignores the instruction. It does not take the instruction into account in the port binding analysis or in the throughput calculations.

In such cases, the summary report includes a line indicating that such instruction(s) exist in the code, and marks the instruction with an 'X' character in all columns.

## 2.3.4 Bubbles in the execution of the front end

The Intel® Architecture Code Analyzer models some of the internal resources of the microarchitecture front end. It may report "front end bubbles" if some or any of these resources become a bottleneck.

## 2.3.5 VDIV / VSQRT Latency

For some values of their operands (e.g. zero or one) VDIV and VSQRT instructions can produce results earlier than their specified latency. The Intel® Architecture Code Analyzer does not model this behavior. As a result it could be more "pessimistic" for kernels that use these instructions.

# 3 Using Intel® Architecture Code Analyzer

This section explains how to build your binary so that the Intel® Architecture Code Analyzer can analyze it, and it lists the tool command-line options.

## 3.1 Building Your Binary

The file **iacaMarks.h** contains macros to denote the start (IACA_START) and end (IACA_END) of the code section for the Intel® Architecture Code Analyzer to evaluate. The Intel Architecture Code Analyzer is a static tool. It treats the analyzed code section as a single consecutive block of instructions. It does not follow branch instructions, not even unconditional branches.

When analyzing a loop construct, place the macros at the following locations:

```
while ( condition )
{
    IACA_START
    <loop body>
}
IACA_END
```

This placement skips the loop initialization and includes the loop-end branch instruction.

These macros modify the **rbx** register in IA-64 code. As a result, the compiler saves this register just before the macro and restores it immediately after the macro.

Once you insert the macros into your code, build your code into an executable file or an object file.

For Microsoft* Visual C++ compiler, 64-bit version, use **IACA_VC64_START** and **IACA_VC64_END**, instead.

***NOTE:*** Input files generated with the Intel compiler option –Qipo are not supported.

# 3.2 Command Line Options

The following command runs the Intel® Architecture Code Analyzer:

```
iaca <options> <input file name>
```

`<input file name>` represents the name of the input file.

Available `<options>`:

| | |
|---|---|
| `-arch <type>` | Architecture type.<br>These are the available types: `HSW, BDW, SKL and SKX`. |
| `-trace <file>` | Generate an IACA trace and output it to a given file. |
| `-reduceout` | Output is reduced. |
| `-v` | Print version and exit. |
| `-trace-cycle-count` | Specify max cycle number to show in IACA trace |

# 3.3 Analysis Errors

Should the analysis fail, the following error messages may appear:

| Error message | Possible Cause |
|---|---|
| `COULD NOT OPEN FILE - <Error>` | The supplied path for the input or output file was incorrect, the input file is not readable or failed to create the output file. |
| `ILLEGAL INSTRUCTION - <Error num>` | Code contains an illegal instruction or other xed error occurred, xed error number is printed. |
| `COULD NOT FIND START_MARKER`<br>`COULD NOT FIND END_MARKER` | Code did not contain the proper marker(s). See section 3.1 for more details. |

# 4 Examples

This section provides examples of how to analyze and optimize code using Intel® Architecture Code Analyzer.

## 4.1 Throughput Analysis – Unrolling

This example performs summation of array elements (only the loop code is analyzed). The initial code and throughput analysis report for Skylake micro-architecture are shown below.

### 4.1.1 Initial Code

```
Throughput Analysis Report
--------------------------
Block Throughput: 4.00 Cycles       Throughput Bottleneck: Dependency chains
Loop Count:  50
Port Binding In Cycles Per Iteration:
-----------------------------------------------------------------------------------------
|  Port  |   0   -  DV   |   1   |   2   -  D    |   3   -  D    |   4   |   5   |   6   |   7   |
-----------------------------------------------------------------------------------------
| Cycles |  0.5     0.0  |  0.5  |  0.5     0.5  |  0.5     0.5  |  0.0  |  0.5  |  0.5  |  0.0  |
-----------------------------------------------------------------------------------------


| Num Of   |                    Ports pressure in cycles                         |      |
|  Uops    |  0  - DV    |  1   |  2  -  D    |  3  -  D    |  4   |  5   |  6   |  7   |
---------------------------------------------------------------------------------------
|   1      |             |      |             |             |      | 0.5  | 0.5  |      | add rbx, 0x20
|   2^     | 0.5         | 0.5  | 0.5     0.5 | 0.5     0.5 |      |      |      |      | vaddps ymm0, ymm0, ymmword ptr [rbx]
|   1*     |             |      |             |             |      |      |      |      | sub eax, 0x8
|   0*F    |             |      |             |             |      |      |      |      | jnle 0xfffffffffffffff5
Total Num Of Uops: 4
```

## 4.1.2 Optimization

The Throughput Analysis Report shows that the total throughput (Block Throughput) is 4 cycles and so this is the throughput per iteration of the loop, and for each one of these iteration one vaddps instruction is performed.

If we perform an unrolling of 8 operations per iteration for this loop we get the same throughput but with a lot more work done, and this is due to the dependency between each vaddps operation that we avoid.

```
Throughput Analysis Report
--------------------------
Block Throughput: 4.00 Cycles       Throughput Bottleneck: Backend
Loop Count:  22
Port Binding In Cycles Per Iteration:
-------------------------------------------------------------------------------------------------
|  Port  |   0   -  DV   |   1   |   2   -  D    |   3   -  D    |   4   |   5   |   6   |   7   |
-------------------------------------------------------------------------------------------------
| Cycles |  4.0     0.0  |  4.0  |  4.0     4.0  |  4.0     4.0  |  0.0  |  0.5  |  0.5  |  0.0  |
-------------------------------------------------------------------------------------------------



| Num Of   |                    Ports pressure in cycles                         |      |
|  Uops    |  0  - DV    |  1   |  2  -  D    |  3  -  D    |  4   |  5   |  6   |  7   |
-----------------------------------------------------------------------------------------
|   1      |             |      |             |             |      | 0.5  | 0.5  |      | add rbx, 0x100
|   2^     |             | 1.0  | 1.0     1.0 |             |      |      |      |      | vaddps ymm0, ymm0, ymmword ptr [rbx]
|   2^     | 1.0         |      |             | 1.0     1.0 |      |      |      |      | vaddps ymm1, ymm1, ymmword ptr [rbx+0x20]
|   2^     |             | 1.0  | 1.0     1.0 |             |      |      |      |      | vaddps ymm2, ymm2, ymmword ptr [rbx+0x40]
|   2^     | 1.0         |      |             | 1.0     1.0 |      |      |      |      | vaddps ymm3, ymm3, ymmword ptr [rbx+0x60]
|   2^     |             | 1.0  | 1.0     1.0 |             |      |      |      |      | vaddps ymm4, ymm4, ymmword ptr [rbx+0x80]
|   2^     | 1.0         |      |             | 1.0     1.0 |      |      |      |      | vaddps ymm5, ymm5, ymmword ptr [rbx+0xa0]
|   2^     |             | 1.0  | 1.0     1.0 |             |      |      |      |      | vaddps ymm6, ymm6, ymmword ptr [rbx+0xc0]
|   2^     | 1.0         |      |             | 1.0     1.0 |      |      |      |      | vaddps ymm7, ymm7, ymmword ptr [rbx+0xe0]
|   1*     |             |      |             |             |      |      |      |      | sub eax, 0x40
|   0*F    |             |      |             |             |      |      |      |      | jnle 0xfffffffffffffc3
Total Num Of Uops: 18
```

The performance gain as seen in the IACA trace:

```
it|in|Dissasembly                                  :0123456789012345678901234567890123456
 0| 0|add rbx, 0x20                                 :          |         |         |
 0| 0|    TYPE_OP (1 uops)                         :sdw----R-------p    |         |
 0| 1|vaddps ymm0, ymm0, ymmword ptr [rbx]         :          |         |         |
 0| 1|    TYPE_LOAD (1 uops)                       :A-s-deeeeeew----R-------p     |
 0| 1|    TYPE_OP (1 uops)                         :A----------deeew----R-------p |
 0| 2|sub eax, 0x8                                 :          |         |         |
 0| 2|    TYPE_OP (1 uops)                         :sdw----------------R-------p |
 0| 3|jnle 0xfffffffffffffff5                      :          |         |         |
 0| 3|    TYPE_OP (0 uops)                         :w----------------R-------p |
 1| 0|add rbx, 0x20                                :          |         |         |
 1| 0|    TYPE_OP (1 uops)                         :A-dw----------------R-------p |
 1| 1|vaddps ymm0, ymm0, ymmword ptr [rbx]         :          |         |         |
 1| 1|    TYPE_LOAD (1 uops)                       : A-s-deeeeeew----R-------p |
 1| 1|    TYPE_OP (1 uops)                         : A-------------deeew----R-------p
 1| 2|sub eax, 0x8                                 :          |         |         |
 1| 2|    TYPE_OP (1 uops)                         : Aw--------------------R-------p
 1| 3|jnle 0xfffffffffffffff5                      :          |         |         |
 1| 3|    TYPE_OP (0 uops)                         : w---------------------R-------p
```

```
 it|in|Dissasembly                                  :0123456789012345678901234567890123456
 0| 0|add rbx, 0x100                                :          |         |         |
 0| 0|    TYPE_OP (1 uops)                         :sdw----R-------p    |         |
 0| 1|vaddps ymm0, ymm0, ymmword ptr [rbx]         :          |         |         |
 0| 1|    TYPE_LOAD (1 uops)                       :A-s-deeeeeew----R-------p     |
 0| 1|    TYPE_OP (1 uops)                         :A----------deeew----R-------p |
 0| 2|vaddps ymm1, ymm1, ymmword ptr [rbx+0x20]    :          |         |         |
 0| 2|    TYPE_LOAD (1 uops)                       :A-s-deeeeeew---------R-------p |
 0| 2|    TYPE_OP (1 uops)                         :A----------deeew----R-------p |
 0| 3|vaddps ymm2, ymm2, ymmword ptr [rbx+0x40]    :          |         |         |
 0| 3|    TYPE_LOAD (1 uops)                       :A-s-cdeeeeeew---------R-------p |
 0| 3|    TYPE_OP (1 uops)                         :A-----------deeew----R-------p|
 0| 4|vaddps ymm3, ymm3, ymmword ptr [rbx+0x60]    :          |         |         |
 0| 4|    TYPE_LOAD (1 uops)                       : As--deeeeeew---------R-------p|
 0| 4|    TYPE_OP (1 uops)                         : A----------deeew----R-------p|
 0| 5|vaddps ymm4, ymm4, ymmword ptr [rbx+0x80]    :          |         |         |
 0| 5|    TYPE_LOAD (1 uops)                       : As--cdeeeeeew---------R-------p|
 0| 5|    TYPE_OP (1 uops)                         : A-----------deeew----R-------p
```

As seen in the trace output, unrolling the loop gives a significant performance gain.
The vaddps operations seen in the top trace output are dependent due to ymm0, and so each Operation is performed only after the previous operation reached writeback stage.
Unrolling the loop makes use of more registers and so parallel executions are possible.

# 5 Release Contents

This section lists the files required for running on Linux*, and Mac OS X* operating systems to analyze Intel® 64 code. Each section also explains which environmental variables to modify.

## 5.1 Linux* OS

Include `iacaMarks.h` in your code.

| Filename | Description |
|---|---|
| `iaca` | Intel Architecture Code Analyzer command-line tool |
| `iacaMarks.h` | Header file for the start/end markers |

## 5.2 Mac OS X*

Include `iacaMarks.h` in your code.

| Filename | Description |
|---|---|
| `iaca` | Intel Architecture Code Analyzer command-line tool |
| `iacaMarks.h` | Header file for the start/end markers |

## 5.3 Windows* OS

Include `iacaMarks.h` in your code.

| Filename | Description |
|---|---|
| `iaca` | Intel Architecture Code Analyzer command-line tool |
| `iacaMarks.h` | Header file for the start/end markers |