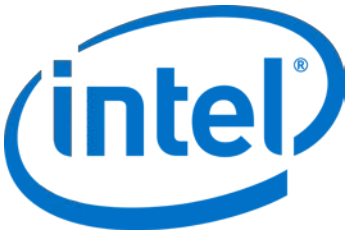


Intel® Architecture Code Analyzer

User's Guide

Document Number: 321356-001US

Revision: 2.0.1

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

Legal Information

Contents

1 Introduction

Intel® Architecture Code Analyzer helps you statically analyze the data dependency, latency, and 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 throughput and latency and reports their cycle counts.
- Identifies the critical path(s).

1.1 Intel® Architecture Code Analyzer Accuracy

The 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 and latency 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

The Intel Architecture Code Analyzer supports analysis for 1st, 2nd and 3rd generation Intel® Core™ processors, which correspond to Intel® microarchitectures codenamed Nehalem (1st gen), Westmere (1st gen), Sandy Bridge (2nd gen) and Ivy Bridge (3rd gen).

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 is capable of analyzing both IA-32 and Intel® 64 code, including Intel® Advanced Vector Extensions (Intel® AVX) instructions.

Intel Architecture Code Analyzer is available on Windows*, Linux*, and Mac OS X* operating systems. Both IA-32 and Intel® 64 operating systems are supported. Intel® 64 code can be analyzed on IA-32 operating systems and vice versa.

NOTE:	Intel® Architecture Code Analyzer has been validated on 64-bit Windows* 7, 64-bit Ubuntu* 10.04, and Mac OS X* 10.6 and 10.7. It should work on other versions of Windows*, Linux*, and Mac OS X* operating systems.

2 Analysis

Intel® Architecture Code Analyzer performs two different types of analysis: Throughput and Latency.

2.1 Throughput Analysis

The 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 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 inter-iteration.
- 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.
- An indication whether the instruction is on the critical path of the analyzed code. The critical path for Throughput Analysis is all instructions that use the throughput bottleneck.
- 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. For 256-bit load operations that keep the port busy for two cycles, the AGU gets freed after the first cycle and can process a store address generation if such micro-op is available for execution.

Following is an example Throughput Analysis report:

```
Throughput Analysis Report
--------------------------
Block Throughput: 28.00 Cycles       Throughput Bottleneck: Divider

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
---------------------------------------------------------------------------
| Cycles | 4.0    28.0 | 1.0  | 1.5    2.0  | 1.5    2.0  | 2.0  | 1.0  |
---------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

| Num Of |                   Ports pressure in cycles                    |    |
|  Uops  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |    |
---------------------------------------------------------------------------------
|   1    |             |      | 1.0    2.0  |             |      |      |    | vmovups ymm0, [rbp-0x70]
|   1    |             |      |             | 1.0    2.0  |      |      |    | vmovups ymm1, [rbp-0x50]
|   1    | 1.0         |      |             |             |      |      |    | vmulps ymm2, ymm0, ymm0
|   1    | 1.0         |      |             |             |      |      |    | vmulps ymm3, ymm1, ymm1
|   1    |             | 1.0  |             |             |      |      |    | vaddps ymm4, ymm2, ymm3
|   3    | 2.0    28.0 |      |             |             |      | 1.0  | CP | vsqrtps ymm5, ymm4
|   2^   |             |      | 0.5         | 0.5         | 2.0  |      |    | vmovups [rbp-0x30], ymm5
```

2.2 Latency Analysis

The Latency Analysis is used to analyze the latency and resource conflicts in a section of code; unlike the throughput analysis, it does not treat the code section as a loop. The Latency Analysis reports the following information:

- Latency of the analyzed code section.
- Resource delay of instructions. A resource delay occurs when all the instruction sources are ready but the execution unit (front end / execution port / divider) is occupied.
- The instructions on a path that has the longest latency (including resource delays) is marked with CP. There may be several critical paths with the same execution latency.
- Total resource conflict delay for each execution unit.
- Performance dependency between instructions.

Following is an example Latency Analysis report:

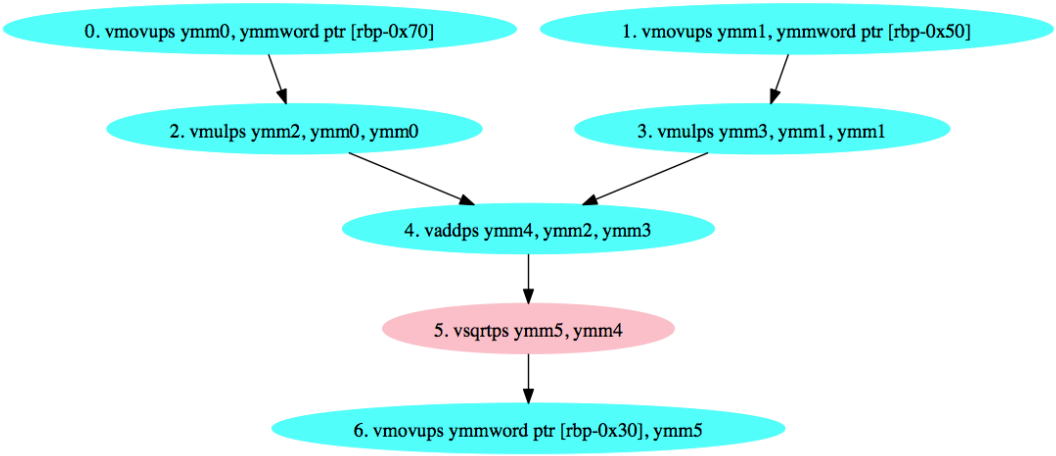
```
Latency Analysis Report
-----------------------
Latency: 59 Cycles

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - Intel(R) AVX to Intel(R) SSE code switch, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

The Resource delay is counted since all the sources of the instructions are ready
and until the needed resource becomes available

| Inst |           Resource Delay In Cycles            |    |                                                  |
| Num  | 0  - DV | 1  | 2  - D  | 3  - D  | 4  | 5  | FE |    |                                                  |
-------------------------------------------------------------------------------------
|  0   |         |    |         |         |    |    |    |    | vmovups ymm0, ymmword ptr [rbp-0x70]             |
|  1   |         |    |         |         |    |    |    | CP | vmovups ymm1, ymmword ptr [rbp-0x50]             |
|  2   |         |    |         |         |    |    |    |    | vmulps ymm2, ymm0, ymm0                          |
|  3   | 1       |    |         |         |    |    |    | CP | vmulps ymm3, ymm1, ymm1                          |
|  4   |         |    |         |         |    |    |    | CP | vaddps ymm4, ymm2, ymm3                          |
|  5   |         |    |         |         |    |    |    | CP | vsqrtps ymm5, ymm4                               |
|  6   |         |    |         |         |    |    | 2  | CP | vmovups ymmword ptr [rbp-0x30], ymm5             |

Resource Conflict on Critical Paths:
-----------------------------------------------------
|  Port  | 0  - DV | 1  | 2  - D  | 3  - D  | 4  | 5  |
-----------------------------------------------------
| Cycles | 1    0  | 0  | 0    0  | 0    0  | 0  | 0  |
-----------------------------------------------------

List Of Delays On Critical Paths
--------------------------------
2 --> 3 1 Cycles Delay On Port0
```

2.3 Graphs

Use the –graph option to set Intel® Architecture Code Analyzer to output the data dependency graph.

TIP: Graph files produced by Intel® Architecture Code Analyzer can be opened with graphviz.

The data dependency graph may be different for throughput analysis as the throughput analysis treats the analyzed code block as an infinite loop block, so there may be inter-iteration dependencies. Red nodes in the graph indicate instructions that are on the critical path for that particular analysis.



2.4 Analysis Report Notes

2.4.1 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.

2.4.2 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.

2.4.3 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 and latency calculations.

In such cases, the summary report includes two additional lines at the top indicating that such instruction(s) exist in your code, and marks the instruction with a '!' character in all columns.

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 **ebx** register in IA-32 code. As a result, the compiler saves this register just before the macro and restores it immediately after the macro. This adds POP and PUSH instructions at the beginning and end of the analyzed block. By default, Intel® Architecture Code Analyzer ignores those instructions, as they are not part of the original code. See section 3.2 how to force the tool analyze those instructions.

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

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

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>`:

`-32`	32-bit input file (default)
`-64`	64-bit input file (required for 64-bit object files only)
`-arch <type>`	Architecture type. These are the available types: `NHM`, `WSM`, `SNB`, `IVB`
`-analysis <type>`	Analysis type: `LATENCY`, `THROUGHPUT` (default)
`-o <file>`	Specifies an output file. The default is `stdout`. To ensure your output appears correctly, specify an output file. The stdout output line widh is limited to 80 characters, but output files have no line width limit.
`–graph <file>`	Specifies an output file for the analysis graph, which can be viewed with graphviz.
`-ignore <boolean>`	Ignores added `pop ebx / push ebx` due to Intel Architecture Code Analyzer Markers. `true` ignores, `false` does not.
`-report`	Generate error report.

3.3 Analysis Errors

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

Error message	Possible Cause
`COULD NOT OPEN FILE - <file name>`	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 - <offset>`	Code contains an illegal instruction in the specified byte offset.
`INCORRECT XED2 VERSION`	Mixed files between multiple Intel® Architecture Code Analyzer releases.
`COULD NOT FIND START_MARKER` `COULD NOT FIND END_MARKER`	Code did not contain the proper marker(s). See section 3.1 for more details.
`CAN'T DETERMINE MODE, PLEASE USE ONE OF -32/-64 COMMAND LINE OPTIONS`	Intel® Architecture Code Analyzer cannot determine the supplied file format (32-bit or 64-bit). Use the -32 or -64 option to specify.

4 Examples

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

4.1 Throughput Analysis – 4x4 Matrix Multiply

This example performs a multiply of two 4x4 matrices using Intel® AVX. The initial code and throughput analysis report are shown below.

4.1.1 Initial Code Version

```
Throughput Analysis Report
--------------------------
Block Throughput: 12.00 Cycles       Throughput Bottleneck: Port5

Port Binding In Cycles Per Iteration:
-------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
-------------------------------------------------------------------------
| Cycles | 8.0    0.0  | 6.0  | 4.0    4.0  | 4.0    4.0  | 4.0  | 12.0 |
-------------------------------------------------------------------------

| Num Of |             Ports pressure in cycles                |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
---------------------------------------------------------------------
|   2^   |           |     | 1.0   1.0 |           |     | 1.0 | CP | vbroadcastf128 ymm9, xmmword ptr [rcx]
|   2^   |           |     |           | 1.0   1.0 |     | 1.0 | CP | vbroadcastf128 ymm10, xmmword ptr [rcx+0x10]
|   2^   |           |     | 1.0   1.0 |           |     | 1.0 | CP | vbroadcastf128 ymm11, xmmword ptr [rcx+0x20]
|   2^   |           |     |           | 1.0   1.0 |     | 1.0 | CP | vbroadcastf128 ymm12, xmmword ptr [rcx+0x30]
|   1    |           |     | 1.0   2.0 |           |     |     |    | vmovaps ymm0, ymmword ptr [rax]
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm1, ymm0, 0x0
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm2, ymm0, 0x55
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm3, ymm0, 0xcc
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm4, ymm0, 0xff
|   1    |           |     |           | 1.0   2.0 |     |     |    | vmovaps ymm0, ymmword ptr [rax+0x20]
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm5, ymm0, 0x0
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm6, ymm0, 0x55
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm7, ymm0, 0xcc
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm8, ymm0, 0xff
|   1    | 1.0       |     |           |           |     |     |    | vmulps ymm1, ymm1, ymm9
|   1    | 1.0       |     |           |           |     |     |    | vmulps ymm2, ymm2, ymm10
|   1    | 1.0       |     |           |           |     |     |    | vmulps ymm3, ymm3, ymm11
|   1    | 1.0       |     |           |           |     |     |    | vmulps ymm4, ymm4, ymm12
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm1, ymm1, ymm2
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm3, ymm3, ymm4
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm1, ymm1, ymm3
|   1    | 1.0       |     |           |           |     |     |    | vmulps ymm5, ymm5, ymm9
|   1    | 1.0       |     |           |           |     |     |    | vmulps ymm6, ymm6, ymm10
|   1    | 1.0       |     |           |           |     |     |    | vmulps ymm7, ymm7, ymm11
|   1    | 1.0       |     |           |           |     |     |    | vmulps ymm8, ymm8, ymm12
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm5, ymm5, ymm6
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm7, ymm7, ymm8
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm5, ymm5, ymm7
|   2^   |           |     | 1.0       |           | 2.0 |     |    | vmovaps ymmword ptr [rdx], ymm1
|   2^   |           |     |           | 1.0       | 2.0 |     |    | vmovaps ymmword ptr [rdx+0x20], ymm5
```

4.1.2 Optimization

The Throughput Analysis Report shows that the total throughput (Block Throughput) is 12 cycles, and port 5 was most pressured (Throughput Bottleneck), with 12 micro-ops allocated to it.

Examination of the instructions that bind to port 5 in the instruction analysis report shows that the instructions were **broadcasts** and **vpermilps**. The broadcasts can only execute on port 5, but replacing them with **128-bit loads** followed by **vinsertf128** instructions reduces the pressure on port 5 because **vinsertf128** can execute on port 0. These changes reduced the throughput to 10 cycles.

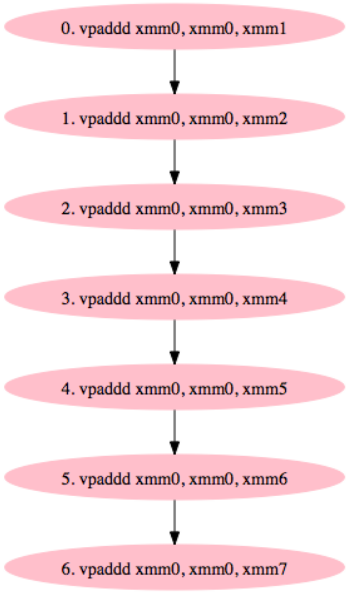
```
Throughput Analysis Report
--------------------------
Block Throughput: 10.00 Cycles       Throughput Bottleneck: Port0, Port5

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
---------------------------------------------------------------------
| Cycles | 10.0   0.0  | 6.0  | 6.0    6.0  | 6.0    6.0  | 4.0  | 10.0 |
---------------------------------------------------------------------

| Num Of |             Ports pressure in cycles              |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
---------------------------------------------------------------------
|   1    |           |     | 1.0   1.0 |           |     |     | vmovaps xmm9, xmmword ptr [rcx]
|   1    |           |     |           | 1.0   1.0 |     |     | vmovaps xmm10, xmmword ptr [rcx+0x10]
|   1    |           |     | 1.0   1.0 |           |     |     | vmovaps xmm11, xmmword ptr [rcx+0x20]
|   1    |           |     |           | 1.0   1.0 |     |     | vmovaps xmm12, xmmword ptr [rcx+0x30]
|   2    | 0.1       |     | 1.0   1.0 |           |     | 0.9 | CP | vinsertf128 ymm9, ymm9, xmmword ptr [rcx], 0x1
|   2    | 0.9       |     |           | 1.0   1.0 |     | 0.1 | CP | vinsertf128 ymm10, ymm10, xmmword ptr [rcx+0x10], 0x1
|   2    |           |     | 1.0   1.0 |           |     | 1.0 | CP | vinsertf128 ymm11, ymm11, xmmword ptr [rcx+0x20], 0x1
|   2    | 1.0       |     |           | 1.0   1.0 |     |     | CP | vinsertf128 ymm12, ymm12, xmmword ptr [rcx+0x30], 0x1
|   1    |           |     | 1.0   2.0 |           |     |     |    | vmovaps ymm0, ymmword ptr [rax]
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm1, ymm0, 0x0
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm2, ymm0, 0x55
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm3, ymm0, 0xcc
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm4, ymm0, 0xff
|   1    |           |     |           | 1.0   2.0 |     |     |    | vmovaps ymm0, ymmword ptr [rax+0x20]
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm5, ymm0, 0x0
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm6, ymm0, 0x55
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm7, ymm0, 0xcc
|   1    |           |     |           |           |     | 1.0 | CP | vpermilps ymm8, ymm0, 0xff
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm1, ymm1, ymm9
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm2, ymm2, ymm10
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm3, ymm3, ymm11
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm4, ymm4, ymm12
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm1, ymm1, ymm2
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm3, ymm3, ymm4
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm1, ymm1, ymm3
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm5, ymm5, ymm9
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm6, ymm6, ymm10
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm7, ymm7, ymm11
|   1    | 1.0       |     |           |           |     |     | CP | vmulps ymm8, ymm8, ymm12
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm5, ymm5, ymm6
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm7, ymm7, ymm8
|   1    |           | 1.0 |           |           |     |     |    | vaddps ymm5, ymm5, ymm7
|   2^   |           |     | 1.0       |           | 2.0 |     |    | vmovaps ymmword ptr [rdx], ymm1
|   2^   |           |     |           | 1.0       | 2.0 |     |    | vmovaps ymmword ptr [rdx+0x20], ymm5
```

4.2 Latency and Graph Analysis – Add Reduction

This example performs an add reduction on 8 XMM registers. The initial code, latency analysis report, and dependency graph (produced with the –graph option) are shown below.

4.2.1 Initial Code Version

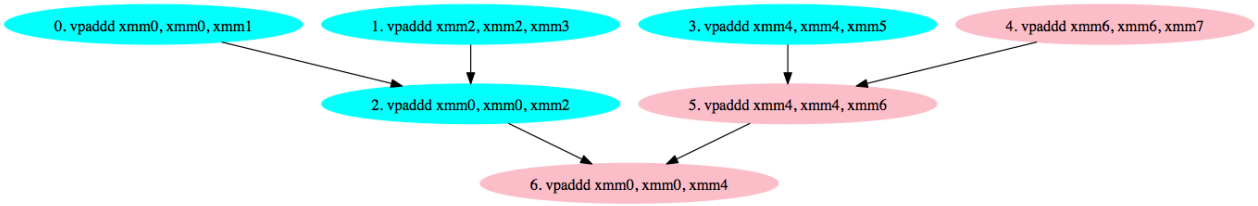
```
Latency Analysis Report
-----------------------
Latency: 7 Cycles

| Inst |            Resource Delay In Cycles             |    |
| Num  | 0  - DV | 1  | 2  - D  | 3  - D  | 4  | 5  | FE |    |
-------------------------------------------------------------------
|  0   |         |    |         |         |    |    |    | CP | vpaddd xmm0, xmm0, xmm1
|  1   |         |    |         |         |    |    |    | CP | vpaddd xmm0, xmm0, xmm2
|  2   |         |    |         |         |    |    |    | CP | vpaddd xmm0, xmm0, xmm3
|  3   |         |    |         |         |    |    |    | CP | vpaddd xmm0, xmm0, xmm4
|  4   |         |    |         |         |    |    |    | CP | vpaddd xmm0, xmm0, xmm5
|  5   |         |    |         |         |    |    |    | CP | vpaddd xmm0, xmm0, xmm6
|  6   |         |    |         |         |    |    |    | CP | vpaddd xmm0, xmm0, xmm7

Resource Conflict on Critical Paths:
-----------------------------------------------------
|  Port  | 0  - DV | 1  | 2  - D  | 3  - D  | 4  | 5  |
-----------------------------------------------------
| Cycles | 0    0  | 0  | 0    0  | 0    0  | 0  | 0  |
-----------------------------------------------------

List Of Delays On Critical Paths
--------------------------------
```



4.2.2 Optimization 1

The analysis report and graph show that all instructions are on the same data dependency path because they all depend on xmm0. We can optimize this code by constructing an add tree, which reduces the dependency between instructions. This change reduced the latency from 7 to 5 cycles.

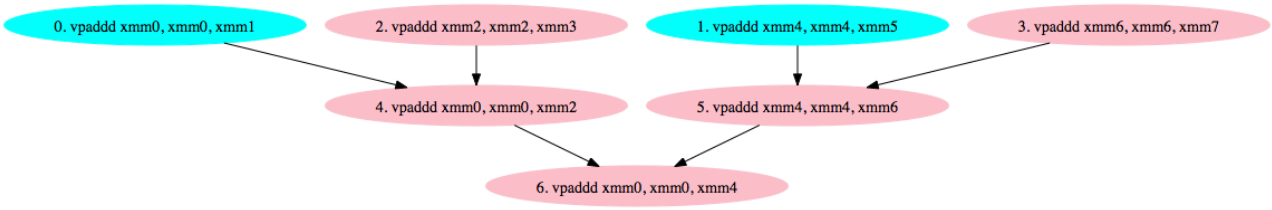
```
Latency Analysis Report
-----------------------
Latency: 5 Cycles

| Inst |            Resource Delay In Cycles             |    |
| Num  | 0  - DV | 1  | 2  - D  | 3  - D  | 4  | 5  | FE |    |
-------------------------------------------------------------------
|  0   |         |    |         |         |    |    |    |    | vpaddd xmm0, xmm0, xmm1
|  1   |         |    |         |         |    |    |    |    | vpaddd xmm2, xmm2, xmm3
|  2   |         |    |         |         |    |    |    |    | vpaddd xmm0, xmm0, xmm2
|  3   |         |    |         |         |    | 1  |    |    | vpaddd xmm4, xmm4, xmm5
|  4   |         | 1  |         |         |    |    | 1  | CP | vpaddd xmm6, xmm6, xmm7
|  5   |         |    |         |         |    |    |    | CP | vpaddd xmm4, xmm4, xmm6
|  6   |         |    |         |         |    |    |    | CP | vpaddd xmm0, xmm0, xmm4

Resource Conflict on Critical Paths:
-----------------------------------------------------
|  Port  | 0  - DV | 1  | 2  - D  | 3  - D  | 4  | 5  |
-----------------------------------------------------
| Cycles | 0    0  | 1  | 0    0  | 0    0  | 0  | 0  |
-----------------------------------------------------

List Of Delays On Critical Paths
--------------------------------
2 --> 4 1 Cycles Delay On Port1
```



4.2.3 Optimization 2

The analysis report tells us that instruction 4 (vpaddd xmm6, xmm6, xmm7) was delayed by instruction 2 due to a resource conflict, and that instruction 4 is on a critical path. Because instruction 5 depends on instruction 4 and instruction 6 depends on instruction 5, both of these instructions are also delayed, and these last three add insturctions can only be executed at a rate of one per cycle. The result from instruction 2 (vpaddd xmm0, xmm0, xmm2) are not needed until instruction 6 (vpaddd xmm0, xmm0 xmm4), so we can resolve this issue by moving vpaddd xmm0, xmm0, xmm2 lower in the add tree, which enables the code to fully utilize the resources, reducing the latency to 4 cycles.

```
latency Analysis Report
-----------------------
Latency: 4 Cycles

| Inst |            Resource Delay In Cycles             |    |
| Num  | 0  - DV | 1  | 2  - D  | 3  - D  | 4  | 5  | FE |    |
-------------------------------------------------------------------
|  0   |         |    |         |         |    |    |    | vpaddd xmm0, xmm0, xmm1
|  1   |         |    |         |         |    |    |    | vpaddd xmm4, xmm4, xmm5
|  2   |         | 1  |         |         |    |    | CP | vpaddd xmm2, xmm2, xmm3
|  3   |         |    |         |         |    | 1  | CP | vpaddd xmm6, xmm6, xmm7
|  4   |         |    |         |         |    |    | CP | vpaddd xmm0, xmm0, xmm2
|  5   |         |    |         |         |    |    | CP | vpaddd xmm4, xmm4, xmm6
|  6   |         |    |         |         |    |    | CP | vpaddd xmm0, xmm0, xmm4

Resource Conflict on Critical Paths:
-----------------------------------------------------
|  Port  | 0  - DV | 1  | 2  - D  | 3  - D  | 4  | 5  |
-----------------------------------------------------
| Cycles | 0    0  | 1  | 0    0  | 0    0  | 0  | 1  |
-----------------------------------------------------

List Of Delays On Critical Paths
--------------------------------
0 --> 2 1 Cycles Delay On Port1
1 --> 3 1 Cycles Delay On Port5
```



5 Release Contents

This section lists the files required for running on Windows*, Linux*, and Mac OS X* operating systems to analyze IA-32 and Intel® 64 code. Each section also explains which environmental variables to modify.

5.1 Windows* OS

Add the `iaca-mac32` directory to the PATH environment variable.

Include `include/iacaMarks.h` in your code.

Filename	Description
`iaca.exe`	Intel® Architecture Code Analyzer command-line tool.
`iacaLoader.dll`	Intel Architecture Code Analyzer shared library.

`iacaLogicNHM.dll`
`iacaLogicWSM.dll`
`iacaLogicSNB.dll`
`iacaLogicIVB.dll`
	Intel Architecture Code Analyzer shared library for each of the supported architectures.

`iacaArchDataNHM.dll`
`iacaArchDataWSM.dll`
`iacaArchDataSNB.dll`
`iacaArchDataIVB.dll`
	Instruction databases for each of the supported architectures.

`XED2NHM.dll`
`XED2WSM.dll`
`XED2SNB.dll`
`XED2IVB.dll`
	XED2 shared libraries for each of the supported architectures.

`iacaMarks.h`	Header file for the start/end markers. Place this file in another directory.
`msvcp100.dll` `msvcr100.dll`	Microsoft Visual Studio* 2010 runtime redistributable packages.

5.2 Linux* OS

Add the `bin/` directory to the `PATH` environment variable.

Add the `lib/` directory to the `LD_LIBRARY_PATH` environment variable.

Include `include/iacaMarks.h` in your code.

Filename	Description
`bin/iaca`	Intel Architecture Code Analyzer command-line tool
`bin/iaca.sh`	Intel Architecture Code Analyzer invocation script
`lib/libiacaLoader.so`	Intel Architecture Code Analyzer shared objects

`lib/libiacaLogicNHM.so`
`lib/libiacaLogicWSM.so`
`lib/libiacaLogicSNB.so`
`lib/libiacaLogicIVB.so`
	Intel Architecture Code Analyzer shared objects for each of the supported architectures

`lib/libiacaArchDataNHM.so`
`lib/libiacaArchDataWSM.so`
`lib/libiacaArchDataSNB.so`
`lib/libiacaArchDataIVB.so`
	Instruction databases for each of the supported architectures

`lib/libXED2NHM.so`
`lib/libXED2WSM.so`
`lib/libXED2SNB.so`
`lib/libXED2IVB.so`
	XED2 shared objects for each of the supported architectures

`include/iacaMarks.h`	Header file for the start/end markers

5.3 Mac OS X*

Add the `bin/` directory to the `PATH` environment variable.

Add the `lib/` directory to the `DYLD_LIBRARY_PATH` environment variable.

Include `include/iacaMarks.h` in your code.

Filename	Description
`bin/iaca`	Intel Architecture Code Analyzer command-line tool
`bin/iaca.sh`	Intel Architecture Code Analyzer invocation script
`lib/libiacaLoader.so`	Intel Architecture Code Analyzer shared objects

`lib/libiacaLogicNHM.so`
`lib/libiacaLogicWSM.so`
`lib/libiacaLogicSNB.so`
`lib/libiacaLogicIVB.so`
	Intel Architecture Code Analyzer shared objects for each of the supported architectures

`lib/libiacaArchDataNHM.so`
`lib/libiacaArchDataWSM.so`
`lib/libiacaArchDataSNB.so`
`lib/libiacaArchDataIVB.so`
	Instruction databases for each of the supported architectures

`lib/libXED2NHM.so`
`lib/libXED2WSM.so`
`lib/libXED2SNB.so`
`lib/libXED2IVB.so`
	XED2 shared objects for each of the supported architectures

`include/iacaMarks.h`	The header file for the start/end markers