Awareness of Architecture in Programming: Intel Sandy Bridge Front-End and Execution Core optimizations (Case Study)

Oscar Peredo

FCFM, DCC, Universidad de Chile

2015 - Santiago, Chile

Outline

Introduction

Intel Sandy Bridge Overview

- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques

- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

Outline

Introduction

Intel Sandy Bridge Overview

- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques

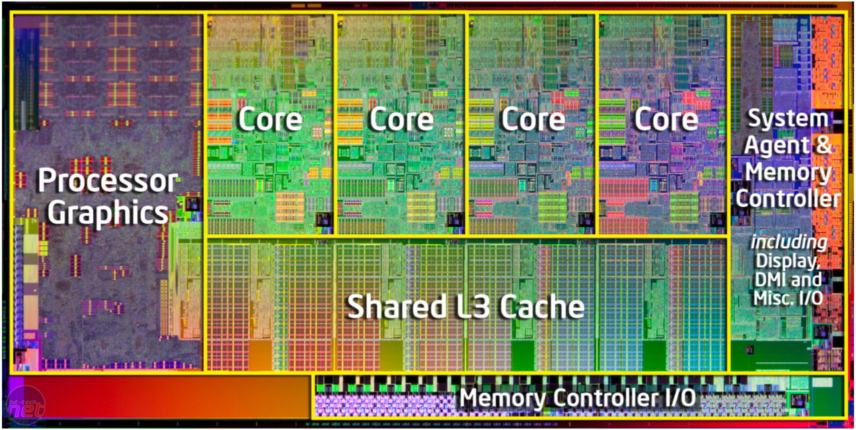
- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

Introduction

- Real scenario: Intel Sandy Bridge Microarchitecture.
- Intel Sandy Bridge Microarchitecture.
- Review of Intel 64 and IA-32 Architectures Optimization Reference Manual:
  - Pipeline overview: Front-End, Out-of-Order Execution Core.
  - General optimization techniques for these components.
  - We focus in high-level optimizations.
- Our first approach to understand technical documents on code optimization (for Intel processors).

Intel Sandy Bridge



http://techreport.com/review/20188/intel-sandy-bridge-core-processors

Outline

Introduction

Intel Sandy Bridge Overview

- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques

- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

Outline

Introduction

Intel Sandy Bridge Overview

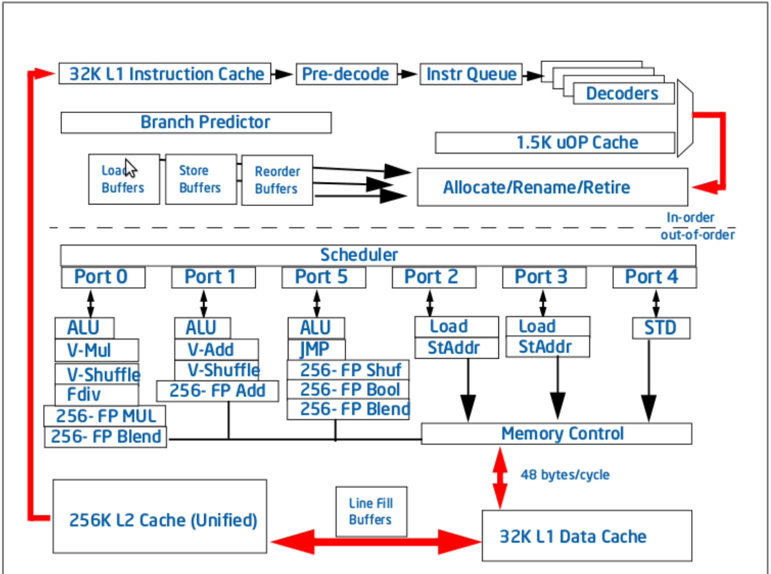
- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques

- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

Pipeline overview



Intel 64 and IA-32 Architectures Optimization Reference Manual (March 2014)

Pipeline overview

The pipeline consists of:

- An in-order issue **front end** that fetches instructions and decodes them into micro-ops (micro-operations). The front end feeds the next pipeline stages with a continuous stream of micro-ops from the most likely path that the program will execute.
- An **out-of-order, superscalar execution engine** that dispatches up to six micro-ops to execution, per cycle. The allocate/rename block reorders micro-ops to *dataflow* order so they can execute as soon as their sources are ready and execution resources are available.
- An in-order retirement unit that ensures that the results of execution of the micro-ops, including any exceptions they may have encountered, are visible according to the original program order.

Outline

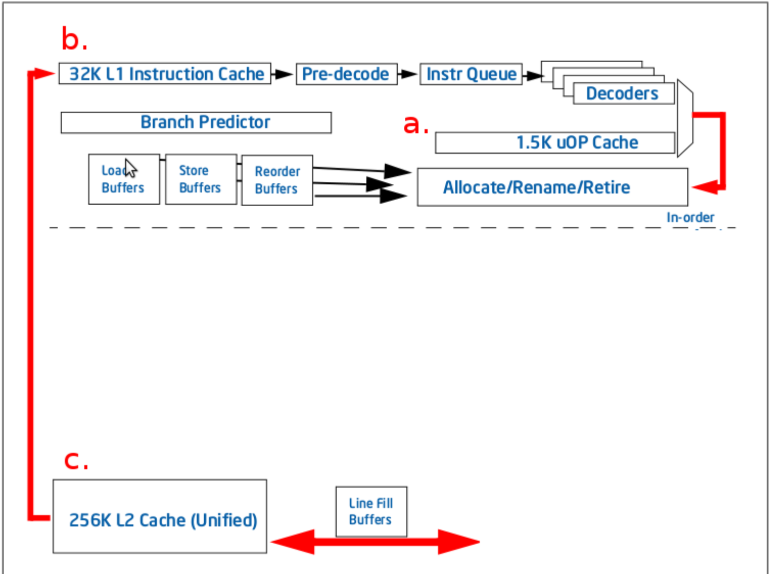
Introduction

Intel Sandy Bridge Overview
- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques
- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

Front-end overview



Intel 64 and IA-32 Architectures Optimization Reference Manual (March 2014)

Front-end overview

1. The Branch Prediction Unit chooses the next block of code to execute from the program. **The processor searches for the code in the following resources, in this order:**
   a. **Decoded ICache** (1.5K uOP Cache)
   b. **Instruction Cache**, via activating the legacy decode pipeline
   c. **L2 cache**, last level cache (LLC) and memory, as necessary
2. New concepts:
   ○ **Micro-fusion**: fuses multiple micro-ops from the same instruction into a single complex micro-op (improves instruction bandwidth delivered from decode to retirement, saving power).
   ○ **Macro-fusion**: merges two instructions into a single micro-op (hardware optimization depending on the first and second merged instructions).
3. The micro-ops corresponding to this code are sent to the Rename/Retire block. They enter into the scheduler in program order, but execute and are de-allocated from the scheduler according to data-flow order.
4. OOO Execution Core starts its work...

Outline

Introduction

Intel Sandy Bridge Overview

- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques

- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

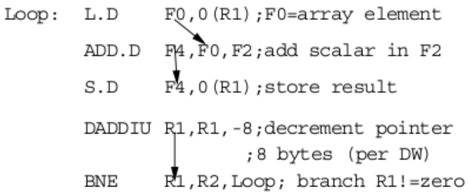
Aside: ILP and OOOE (from Hennessy & Patterson, 4th ed.)

ILP= Instruction-Level Parallelism. OOOE= Out-of-Order Exec.

Q: Which instructions can be executed in parallel in the pipeline?
A: Those who are data independent.

Data dependences:

- **True data dependence**: instruction A produces a result that may be used by $A + i$, or instruction $A + i$ is data dependent on instruction $A + j$ ($j < i$), and $A + j$ is data dependent on A.

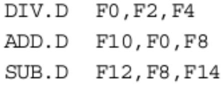


```
Loop:  L.D     F0,0(R1);F0=array element
       ADD.D   F4,F0,F2;add scalar in F2
       S.D     F4,0(R1);store result
       DADDIU  R1,R1,-8;decrement pointer
                     ;8 bytes (per DW)
       BNE     R1,R2,Loop; branch R1!=zero
```

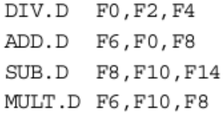
- **Name dependence**: occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name.
- **Control dependence**: depends on branch instructions results.

Aside: ILP and OOOE

- Data dependences may create **Data Hazards** (Read-after-Write, Write-after-Read, Write-after-Write).
- **Data Hazard triggers a pipeline stall** (some cycles are lost waiting for the hazard to resolve).
- Out-of-Order execution idea:



```
DIV.D   F0,F2,F4
ADD.D   F10,F0,F8
SUB.D   F12,F8,F14
```



```
DIV.D   F0,F2,F4
ADD.D   F6,F0,F8
SUB.D   F8,F10,F14
MULT.D  F6,F10,F8
```

  - ○ Left-code shows a RaW hazard (ADD.D reads F0, which is being written by DIV.D), so a stall is triggered. However, SUB.D does not have any dependency with ADD.D or DIV.D, but it has to wait until the hazard resolves. **If SUB.D could be executed Out-of-Order, no stall will incur in its processing.**
  - ○ Right-code shows WaW (SUB.D and ADD.D) and WaR (ADD.D and MULT.D) hazards. **Using register renaming and OOOE, we can avoid the corresponding stalls triggered by those hazards.**

Aside: ILP and OOOE

- **Dynamically scheduled pipeline**: instructions are issued *in-order* and executed *out-of-order*, following some algorithm.
- Well-known algorithms: **Scoreboarding** and **Tomasulo's algorithm**.

Aside: ILP and OOOE

Scoreboarding method: developed by Control Data Corporation in 1964, first implemented in CDC 6600 mainframe computer.

http://en.wikipedia.org/wiki/CDC_6600

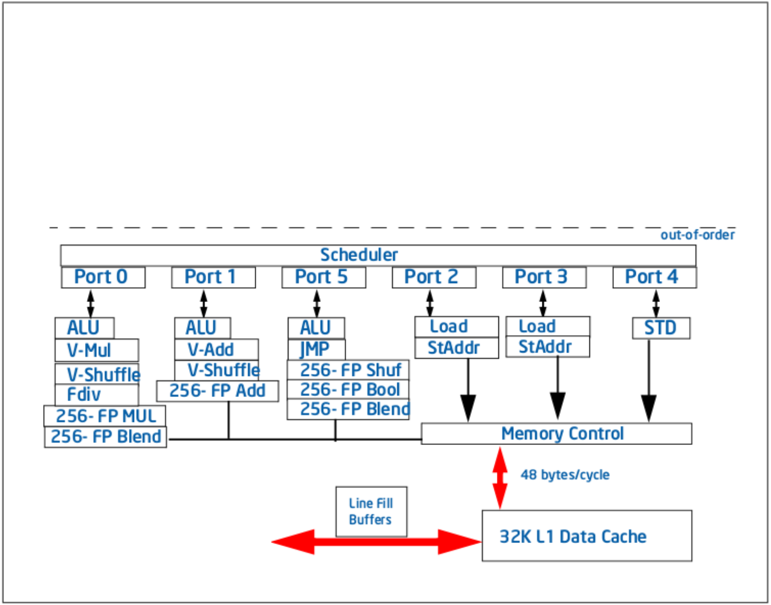
Aside: ILP and OOOE

Tomasulo's algorithm: developed by Robert Tomasulo at IBM in 1967, first implemented in IBM System/360 mainframe computer model 91.

http://en.wikipedia.org/wiki/IBM_System/360

Back to Intel Sandy Bridge...

OOO Execution Core overview



OOO Execution Core overview

1. Micro-op execution is executed using execution resources arranged in three stacks.
2. **Branch mispredictions are signaled at branch execution. It re-steers the front end which delivers micro-ops from the correct path.**
3. **Memory operations are managed and reordered to achieve parallelism and maximum performance.** Misses to the L1 data cache go to the L2 cache. The data cache is non-blocking and can handle multiple simultaneous misses.
4. Exceptions (Faults, Traps) are signaled at retirement (or attempted retirement) of the faulting instruction.

Outline

Introduction

Intel Sandy Bridge Overview

- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques

- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

Outline

Introduction

Intel Sandy Bridge Overview

Pipeline

The Front-End

The Out-of-Order Execution Core

Optimization techniques

Optimizing the Front-End

Optimizing the OOO Execution Core

Bibliography

Optimizing the Front-End

Optimizing the front end covers two aspects:

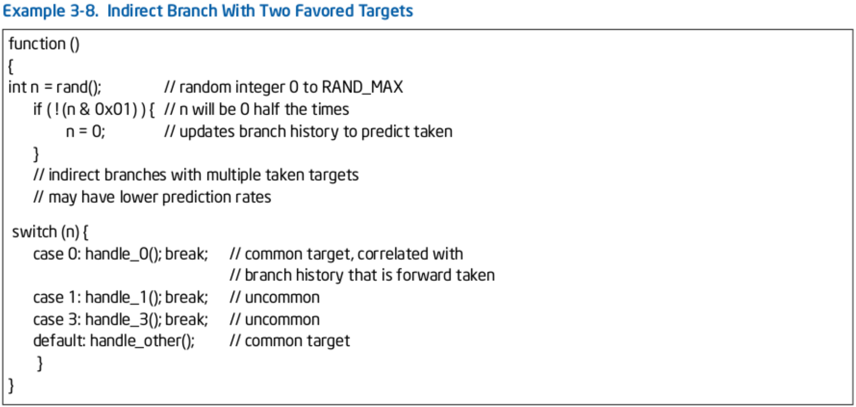
- **Maintaining steady supply of micro-ops to the execution engine.**
  Mispredicted branches can disrupt streams of micro-ops, or cause the execution engine to waste execution resources on executing streams of micro-ops in the non-architected code path. Much of the tuning in this respect focuses on working with the Branch Prediction Unit.
- **Supplying streams of micro-ops to utilize the execution bandwidth and retirement bandwidth as much as possible.**
  For Intel Core microarchitecture and Intel Core Duo processor family, this aspect focuses maintaining high decode throughput. In Intel microarchitecture code name Sandy Bridge, this aspect focuses on keeping the hod code running from Decoded ICache.

Branch Prediction Optimizations

Optimizations that help branch prediction are:

- Keep code and data on separate pages. (hardware opts)
- **Eliminate branches whenever possible.** (user/compiler + hardware opts)
- **Arrange code to be consistent with the static branch prediction algorithm.** (user/compiler + hardware opts)
- Use the PAUSE instruction in spin-wait loops. (hardware opts)
- **Inline functions and pair up calls and returns.** (user/compiler + hardware opts)
- **Unroll as necessary** so that repeatedly-executed loops have sixteen or fewer iterations (unless this causes an excessive code size increase). (user/compiler + hardware opts)
- **Separate branches so that they occur no more frequently than every three micro-ops where possible.** (user/compiler + hardware opts)

Branch Type Selection

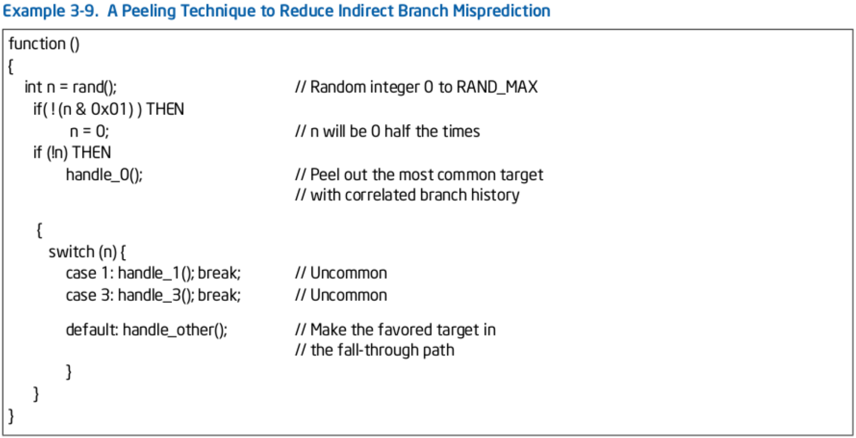


Example 3-8. Indirect Branch With Two Favored Targets

```
function ()
{
int n = rand();             // random integer 0 to RAND_MAX
    if ( ! (n & 0x01) ) {   // n will be 0 half the times
         n = 0;             // updates branch history to predict taken
    }
    // indirect branches with multiple taken targets
    // may have lower prediction rates

 switch (n) {
    case 0: handle_0(); break;     // common target, correlated with
                                   // branch history that is forward taken
    case 1: handle_1(); break;     // uncommon
    case 3: handle_3(); break;     // uncommon
    default: handle_other();       // common target
     }
}
```

Branch Type Selection



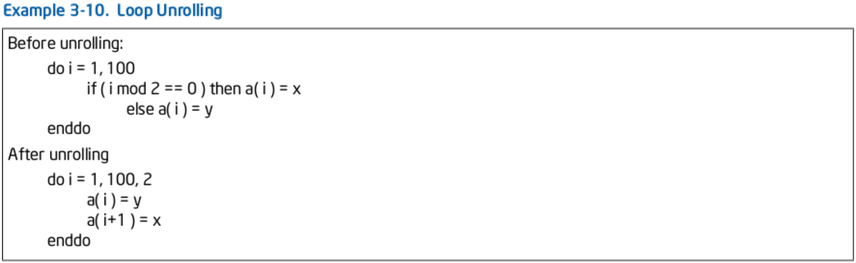
Example 3-9. A Peeling Technique to Reduce Indirect Branch Misprediction

```
function ()
{
   int n = rand();                          // Random integer 0 to RAND_MAX
     if( ! (n & 0x01) ) THEN
            n = 0;                          // n will be 0 half the times
     if (!n) THEN
           handle_0();                      // Peel out the most common target
                                            // with correlated branch history

      {
        switch (n) {
           case 1: handle_1(); break;       // Uncommon
           case 3: handle_3(); break;       // Uncommon

           default: handle_other();         // Make the favored target in
                                            // the fall-through path
           }
      }
}
```

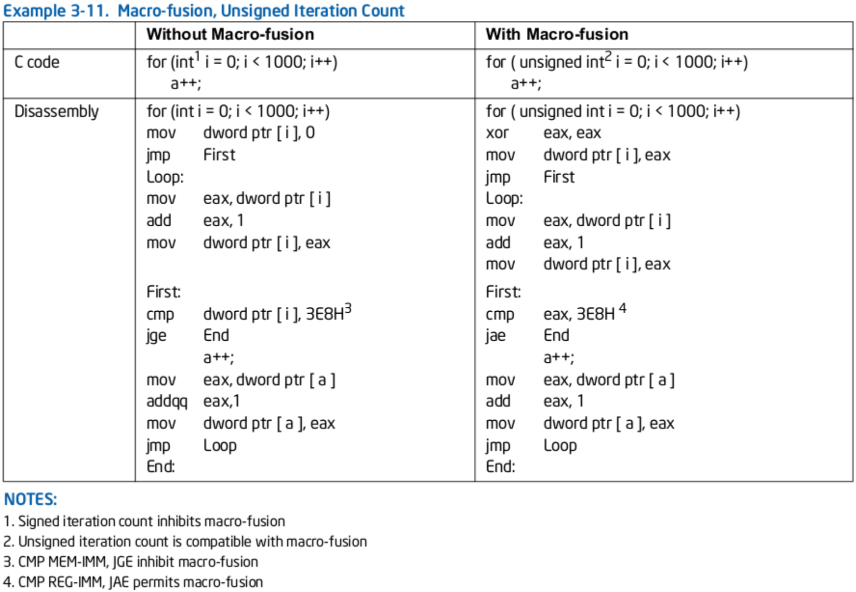
Loop unrolling



Example 3-10. Loop Unrolling

```
Before unrolling:
        do i = 1, 100
                if ( i mod 2 == 0 ) then a( i ) = x
                        else a( i ) = y
        enddo
After unrolling
        do i = 1, 100, 2
                a( i ) = y
                a( i+1 ) = x
        enddo
```

Optimizing for macro-fusion



Example 3-11. Macro-fusion, Unsigned Iteration Count

	Without Macro-fusion	With Macro-fusion

C code	for (int[1] i = 0; i < 1000; i++)
	a++;
		for (unsigned int[2] i = 0; i < 1000; i++)
		a++;

Disassembly	for (int i = 0; i < 1000; i++) mov dword ptr [i], 0 jmp First Loop: mov eax, dword ptr [i] add eax, 1 mov dword ptr [i], eax First: cmp dword ptr [i], 3E8H[3] jge End a++; mov eax, dword ptr [a] addqq eax,1 mov dword ptr [a], eax jmp Loop End:	for (unsigned int i = 0; i < 1000; i++) xor eax, eax mov dword ptr [i], eax jmp First Loop: mov eax, dword ptr [i] add eax, 1 mov dword ptr [i], eax First: cmp eax, 3E8H [4] jae End a++; mov eax, dword ptr [a] add eax, 1 mov dword ptr [a], eax jmp Loop End:

NOTES:

1. Signed iteration count inhibits macro-fusion
2. Unsigned iteration count is compatible with macro-fusion
3. CMP MEM-IMM, JGE inhibit macro-fusion
4. CMP REG-IMM, JAE permits macro-fusion

Outline

Introduction

Intel Sandy Bridge Overview

- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques

- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

Optimizing the OOO Execution Core

General guidelines to make use of the available parallelism are:

- **Follow the rules to maximize useful decode bandwidth and front end throughput (Front-End optimizations)**.
- Maximize rename bandwidth. (hardware opts)
- Scheduling recommendations on sequences of instructions so that multiple dependency chains are alive simultaneously, thus ensuring that your code utilizes maximum parallelism. (hardware opts)
- Avoid hazards, minimize delays that may occur in the execution core, allowing the dispatched micro-ops to make progress and be ready for retirement quickly. (**vectorization**: user/compiler + hardware opts)

Vectorization

- Vectorization is a program transformation that allows special hardware to perform the same operation on multiple data elements at the same time. **SIMD = Single-Instruction Multiple-Data**.
- To help enable the compiler to generate SIMD code, avoid global pointers and global variables.
- Use the smallest possible floating-point or SIMD data type, to enable more parallelism with the use of a (longer) SIMD vector. **For example, use single precision instead of double precision where possible.**
- Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. **Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration**, something which is called a lexically backward dependence.
- **Avoid the use of conditional branches inside loops** and consider using SSE instructions to eliminate branches.
- **Keep induction (loop) variable expressions simple**.

Outline

Introduction

Intel Sandy Bridge Overview

- Pipeline
- The Front-End
- The Out-of-Order Execution Core

Optimization techniques

- Optimizing the Front-End
- Optimizing the OOO Execution Core

Bibliography

Bibliography

- Intel 64 and IA-32 Architectures Optimization Reference Manual (March 2014).
- Hennessy, J. and Patterson, D., *Computer Architecture: A Quantitative Approach*, Morgan-Kauffman 4th Edition, 2007.

Thanks for your attention!

Contact: `operedo [at] gmail.com`