AP-526

APPLICATION NOTE

# Optimization's For Intel's 32-Bit Processors

October 1995

Order Number: 242816-001

Information in this document is provided in connection with Intel products. Intel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Intel products except as provided in Intel's Terms and Conditions of Sale for such products.

Intel retains the right to make changes to these specifications at any time, without notice. Microcomputer Products may have minor variations to this specification known as errata.

*Other brands and names are the property of their respective owners.

†Since publication of documents referenced in this document, registration of the Pentium, OverDrive and iCOMP trademarks has been issued to Intel Corporation.

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

COPYRIGHT © INTEL CORPORATION 1995

# CONTENTS

## 1.0. INTRODUCTION

The Intel386™ Architecture Family, including the Intel386, Intel486™, Pentium® and Pentium Pro, represents a series of compatible processors. The newer members of the family are capable of executing any binaries created for members of previous generations. For example, any existing 8086/8088, 80286, Intel386 (DX or SX), and Intel486 processor applications will execute on the Pentium and Pentium Pro processors without any modification or recompilation. However, there are certain code optimization techniques which will make applications execute faster on a specific member of the family with little or no impact on the performance of other members. Most of these optimizations deal with instruction sequence selection and instruction reordering to complement the processor micro-architecture .

The intent of this document is to describe the implementation differences of the processor members and to describe the optimization strategy that gives the best performance for all members of the family.

## 2.0. OVERVIEW OF INTEL386™, INTEL486™, PENTIUM® AND PENTIUM PRO PROCESSORS

### 2.1. The Intel386™ Processor

The Intel386 processor is the first implementation of the 32-bit Intel386 architecture. It includes full 32-bit data paths, rich 32-bit addressing modes and on-chip memory management.

#### 2.1.1. INSTRUCTION PREFETCHER

The instruction prefetcher prefetches the instruction stream from external memory. The prefetched instructions are kept in the four-deep, four-byte-wide prefetch buffers. The instruction decoder operates on the code stream fed to it through the prefetch buffers.

#### 2.1.2. INSTRUCTION DECODER

The instruction decoder places the decoded information in a three-deep FIFO. It is decoupled from both the prefetcher and the execution core and has separate protocols with each.

#### 2.1.3. EXECUTION CORE

The core engine executes the incoming instructions one at a time, but for certain cases, it allows overlapping the last execution cycle of the current instruction with the effective address calculation of the next instruction's memory reference.

From the compiler writer's point of view, the decoupled prefetch/decode/execution stages and the sequential nature-of the core engine has placed few requirements for instruction scheduling. Avoiding the use of an index in the effective address will save one extra clock. A careful choice of instructions to minimize the execution clock counts is the best optimization approach.

### 2.2. The Intel486™ Processor

The Intel486 processor has a full blown integer pipeline delivering a peak throughput of one instruction per clock. It has integrated the first level cache and the floating-point unit on chip, and it has the same on-chip memory management capabilities as the Intel386 processor.

### 2.2.1. INTEGER PIPELINE

The Intel486 processor has a five-stage integer pipeline capable of processing one instruction per clock.

The five pipeline stages are:

1. Prefetch, (PF) - where instructions are fetched from the cache and placed in one of two 16-byte buffers
2. Decode (D1) - where incoming code stream is being decoded. Prefixed instructions stay in D1 for two clocks.
3. Decode (D1) - where incoming code stream is being decoded. Prefixed instructions stay in D1 for two clocks.
4. Address Generation (D2) - where effective address and linear address are calculated in parallel. The address generation can usually be completed in one cycle except in cases where the indexed addressing mode is used. If the index is used, the instruction stays in D2 for two clocks.
5. Execution (E) - where the machine operations are performed. Simple instructions (those with one machine operation) take one cycle to execute giving a maximum throughput of one instruction per clock. The more complex instructions take multiple execution cycles.
6. Writeback (WB) - where the needed register update occurs.

Note that special hardware exists to bypass the writeback stage in special cases, for values calculated in the execution (E) stage, the result of the calculation is available to the next instruction at the end of the execution cycle. The next instruction need not wait for writeback (WB) to complete before using the data.

Also note that a taken branch causes a two clock penalty because the instructions in the decode stages (D1 and D2) are flushed when the taken branch is executed. Whereas the pipeline stream is unaffected by a not-taken branch as the next instructions to execute are already in the pipeline stream.

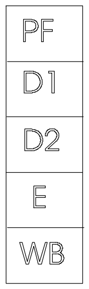


FIG1

**Figure 1. Intel486™ Processor Integer Pipeline**

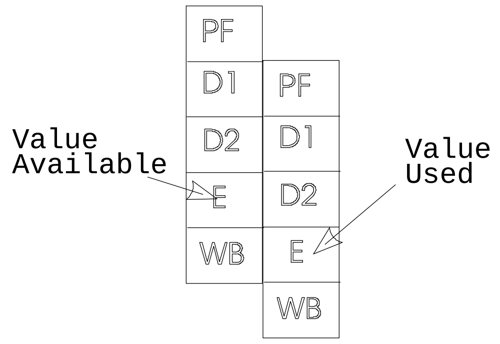


FIG2

**Figure 2. Pipeline Example of Writeback Bypass**

### 2.2.2. ON-CHIP CACHE

The on-chip cache is a combined instruction and data cache. It is 8K bytes in size, four-way set associative with a 16 byte line size and pseudo-LRU replacement algorithm. All data references have priority access to the cache over instruction prefetch cycles.

### 2.2.3. ON-CHIP FLOATING-POINT UNIT

The on-chip floating-point unit utilizes the integer pipeline for early data access. The bus structure allows 64-bit data to be transferred between the cache and the floating-point hardware in one clock. The floating-point design also allows overlapping the floating-point operations with integer operations.

The execution core of the Intel486 processor has been architected to maximize the throughput of a class of "frequently used" instructions. Hence, careful selection of an instruction sequence to perform a given task results in faster execution time. Also, code scheduling to avoid pipeline stalls helps to boost application performance. Most of the optimizations targeted to the Intel486 processor are neutral with respect to the Intel386 processor performance.

## 2.3. The Pentium® Processor

The Pentium processor is an advanced *superscalar* processor. It is built around two general purpose integer pipelines and a pipelined floating-point unit. The Pentium processor can execute two integer instructions simultaneously. A software-transparent dynamic branch-prediction mechanism minimizes pipeline stalls due to branches.

### 2.3.1. INTEGER PIPELINES

The Pentium processor has two parallel integer pipelines, the main pipe (U) which is an enhanced Intel486 processor pipe and the secondary pipe (V) which is similar to the main one but has some limitations on the instructions it can execute. The limitations will be described in more detail in later sections.

The Pentium processor can issue two instructions every cycle. During execution, the next two instructions are checked, and if possible, they are issued such that the first one executes in the U pipe, and the second in the V pipe. (If it is not possible to issue two instructions, then the next instruction is issued to the U pipe and no instruction is issued to the V pipe.)

When instructions execute in the two pipes, their behavior is exactly the same as if they were executed sequentially. When a stall occurs successive instructions are not allowed to pass the stalled instruction in either pipe. In the Pentium processor's pipelines, the D2 stage can perform a multiway add, so there is not a one clock index penalty as with the Intel486 processor pipeline.

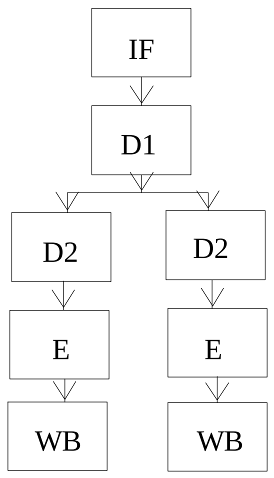


FIG3

**Figure 3. Pentium® Processor Integer Pipelines**

### 2.3.2. CACHES

The on-chip cache subsystem consists of two (instruction and data) 8 Kbyte two-way set associative caches with a cache line length of 32 bytes. There is a 64-bit wide external data bus interface. The caches employ a write back mechanism and an LRU replacement algorithm. The data cache consists of eight banks interleaved on four byte boundaries. The data cache can be accessed simultaneously from both pipes, as long as the references are to different banks. The minimum delay for a cache miss is 3 clocks.

### 2.3.3. INSTRUCTION PREFETCHER

The instruction prefetcher has four buffers, each of which is 32 bytes long. It can fetch an instruction which is split among two cache lines with no penalty. Because the instruction and data caches are separate, instruction prefetches no longer conflict with data references for access to the cache (as in the case of Intel486 processor).

### 2.3.4. BRANCH TARGET BUFFER

The Pentium processor employs a dynamic branch prediction scheme with a 256 entry branch target buffer (BTB). If the prediction is correct, there is no penalty when executing a branch instruction. There is a 3 cycle penalty if the conditional branch was executed in the U pipe or a 4 cycle penalty if it was executed in the V pipe when the branch is mispredicted. Mispredicted calls and unconditional jump instructions have a 3 clock penalty in either pipe. On the Intel486 processor, taken branches always have a two clock penalty.

### 2.3.5. PIPELINED FLOATING-POINT UNIT

The majority of the frequently used instructions are pipelined so that the pipelines can accept a new pair of operands every cycle. Therefore a good code generator can achieve a throughput of almost 1 instruction per cycle (of course this assumes a program with a modest amount of natural parallelism!). The fxch instruction can be executed in parallel with the commonly used floating-point instructions, which lets the code generator or programmer treat the floating-point stack as a regular register set without any performance degradation.

With the superscalar implementation, it is important to schedule the instruction stream to maximize the usage of the two integer pipelines. Since each of the Pentium processor's integer pipelines is enhanced from the pipeline of the Intel486 processor, the instruction scheduling criteria for the Pentium processor is a superset of the Intel486 processor requirements.

## 2.4. The Pentium® Pro Processor

The Pentium Pro processor provides a Dynamic Execution architecture that blends *out-of-order, and speculative execution with hardware register renaming* and *branch prediction*. The processor provides an in-order issue pipeline, which breaks Intel386 processor macroinstructions up into simple, micro-operations called uops, and an out-of-order, superscalar processor core, which executes the uops. The out-of-order core of the processor contains several pipelines to which integer, jump, floating-point, and memory execution units are attached. Several different execution units may be clustered on the same pipeline: e.g. an integer address logic unit, and the floating-point execution units (adder, multiplier, and divider) share a pipeline. The data cache is pseudo-dual ported via interleaving, with one port dedicated to loads and the other to stores. Most simple operations (integer ALU, floating-point add, even floating-point multiply) can be pipelined with a throughput of 1 or 2 operations per cycle. Floating-point divide is not pipelined. Long latency operations can proceed in parallel with short latency operations.

### 2.4.1. IN-ORDER PIPELINE

The In-order Pipeline performs branch prediction, instruction address translation, instruction fetch, instruction decode, and register renaming. Several of these stages have performance implications that can be taken into account by code generators. The pipestages of the In-order Pipeline do not affect performance so long as branches are being correctly predicted by the BTB. When there is a branch misprediction, the pipestages of the In-order Pipeline may affect the latency of fetching the correct instructions to execute.

### 2.4.2. OUT-OF-ORDER CORE

The Out-of-order Cluster resolves data dependencies, dispatches uops to execution units, and buffers the results of uops executed out of order until all previous operations have been completed, so that the permanent processor state can be updated in the correct order.

### 2.4.3. CACHES

The on-chip level one (L1) caches consist of one 8 Kbyte four-way set associative instruction cache unit with a cache line length of 32 bytes and one 8 Kbyte two-way set associative data cache unit. Not all misses in the L1 cache will expose the full memory latency. The level two (L2) cache masks the full latency caused by a L1 cache miss. The minimum delay for a L1 and L2 cache miss is 14 cycles.

### 2.4.4. BRANCH TARGET BUFFER

The Pentium Pro processor uses a branch prediction algorithm that is similar to the Pentium processor. Through the branch target buffer (BTB) branches that have been previously seen are dynamically predicted. Branches that have not yet been seen by the BTB are predicted using a static prediction algorithm. The branch target buffer (BTB) stores the history of the previously seen branches and their targets. When a branch is encountered the BTB feeds the target address directly into the Instruction Fetch Unit (IFU). Once the branch is executed, the BTB is updated with the target address.

Determining the cycle cost for branches is difficult. The Pentium Pro processor has several different levels of branch support which can be quantified in the number of cycles lost:

Branches that are not-taken suffer no penalty. This applies to those branches that are correctly predicted as not taken by the BTB, and to forward branches that are not in the BTB, which are predicted as not taken by default.

Branches which are correctly predicted as taken by the BTB suffer a minor penalty (~ 1 cycle). Instruction fetch is suspended for one cycle. The processor decodes no further instructions in that block, possibly resulting in the issue of less than 4 uops. This minor penalty applies to unconditional branches which have been seen before (i.e. are in the BTB). The minor penalty for correctly predicted taken branches is one lost cycle of instruction fetch, plus issue of no instructions after the branch. This minor penalty usually overlaps with other activities in the processor.

Mispredicted branches suffer a more significant penalty.

The penalty for mispredicted branches is at least 9 cycles (the length of the In-order Issue Pipeline) of lost instruction fetch, plus additional time spent waiting for the mispredicted branch instruction to become the oldest instruction in the machine and retire. This penalty is non-deterministic, dependant upon execution circumstances, but experimentation shows that this is nominally a total of 10-15 cycles.

#### Decoder Shortstop

Branches that are not in the BTB, which are correctly predicted by the decoder shortstop branch prediction mechanism, suffer a small penalty ( ~ 5-6 cycles).This small penalty applies to unconditional direct branches which have never been seen before (i.e. which are not in the BTB). The decoder is always able to correctly predict these!

Conditional branches with negative displacement, such as loop-closing branches, are predicted taken by the shortstop mechanism. They suffer only a small penalty the first time the branch is encountered and a minor penalty on subsequent iterations when the BTB predicts them.

The small penalty for branches that are not in the BTB but which are correctly predicted by the decoder is approximately 5 cycles of lost instruction fetch as opposed to 10-15 cycles for a branch that is incorrectly predicted or that has no prediction.

### 2.4.5. INSTRUCTION PREFETCHER

The Instruction Prefetcher performs aggressive prefetch of straight line code. Obviously, arranging code so that non-loop branches that tend to fall through takes advantage of this prefetch. Additionally, arranging code such that infrequently executed code is segregated to the bottom of the procedure or end of the program where it is not prefetched unnecessarily improves prefetching.

Note that instruction fetch is always for an aligned 16 byte block. Like the Intel486 processor, the Pentium Pro processor reads in instructions from 16-byte aligned boundaries. Therefore, if a branch target address (the address of a label) is, e.g. equal to 14 modulo 16, only two useful instruction bytes will be fetched in the first cycle.

To obtain best performance align JUMP/CALL/RET destinations at 0-mod-16 addresses. However, this may cause the code to grow bigger, and may require extra cycles to decode NOP instructions, process cache misses, etc. The tradeoff between branch alignment and code size is sensitive, and only "important" branch destinations should be aligned. Compilers with profiling feedback and other information about the dynamic frequency of branches may extend this rule.

# 3.0. BLENDED CODE GENERATION CONSIDERATION

Even though each member of the Intel386 processor family has a different micro architecture because of technology versus implementation tradeoffs, the differences create few conflicts in the overall code optimization strategy. In fact, there is a set of "blended" optimizations that will create an optimal binary across the entire family. The "blended" optimizations include:

1. Optimizations that benefit all members.
2. Optimizations that benefit one or more members but do not impact the remaining members.
3. Optimizations that benefit one or more members a lot but only impair the remaining members a little.

For those optimizations that benefit only certain members but cause noticeable degradation to others, it is recommended that they be implemented under switches and left to the user to decide whether maximizing the performance of a specific processor is desirable. Chapter 6 details the recommended switches for Intel Architecture compilers.

## 3.1. Choice of Index Versus Base Register

The Intel386 and the Intel486 processors need an additional clock cycle to generate an effective address when an index register is used. Therefore, if only one indexing component is used (i e.. not both a base register and an index register) and scaling is not necessary, then it is faster to use the register as a base rather than an index. For example:

```
mov  eax, [esi]  ; use esi as base

mov  eax, [esi*] ; use esi as index,
 1 clock penalty
```

It takes the Pentium processor one clock to calculate the effective address even when an index register is used. The Intel486 processor takes 2 clocks in D2 when an index is used. Both the Pentium and Pentium Pro processors are neutral to the choice of index versus base register.

## 3.2. Addressing Modes and Register Usage

1. For the Intel486 processor, when a register is used as the base component, an additional clock cycle is used if that register is the destination of the immediately preceding instruction (assuming all instructions are already in the prefetch queue). For example:

   ```
   add  esi, eax    ; esi is destination register
   mov  eax, [esi]  ; esi is base, 1 clock penalty
   ```

   Since the Pentium processor has two integer pipelines and each pipeline has similar organization as the on the Intel486 processor, a register used as the base or index component of an effective address calculation (in either pipe) causes an additional clock cycle if that register is the destination of either instruction from the immediately preceding cycle (Address Generation Interlock, (AGI)). To avoid the AGI, the instructions should be separated by at least one cycle by placing other instructions between them.

The Pentium Pro processor incurs no penalty for the AGI condition.

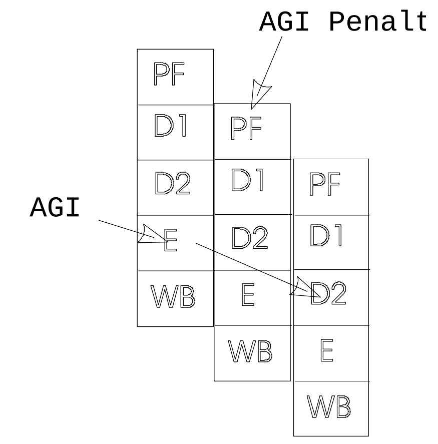


**Figure 4. Pipeline Example of AGI Stall**

2. Note that some instructions have implicit reads/writes to registers. Instructions that generate addresses implicitly through `esp (push,pop/ret/call)` also suffer from the AGI penalty.

   Examples:

   ```
   sub  esp, 24
           ;  1 cycle stall
   push ebx

   mov  esp, ebp
           ;  1 cycle stall
   pop  ebp
   ```

   `Push` and `pop` also implicitly write to `esp`. This, however, does not cause an AGI when the next instruction addresses through esp. Both the Intel486 and the Pentium processors "rename" esp from `Push` and `Pop` instructions to avoid the AGI penalty.

   Example:

   ```
   push edi          ; no stall
   mov  ebx, [esp]
   ```

3. On the Intel486 processor there is a 1 clock penalty for decoding an instruction with either an *index* or an *immediate-displacement* combination. On the Pentium processor, the *immediate-displacement* combination is not pairable. When it is necessary to use constants, it would still be more efficient to use immediate data instead of loading the constant into a register first, but if the same immediate data is used more than once, it would be faster to load the constant in a register and then use the register multiple times.

   ```
   mov  result, 555            ; 555 is immediate, result is displacement
   mov  dword ptr [esp+4], 1   ;   1 is immediate,      4 is displacement
   ```

4. The Intel486 processor has a 1 clock penalty when using a full register immediately after a partial register was written. The Pentium processor is neutral in this respect. This is called a partial stall condition.

   Example (Pentium Processor):
   ```
   mov  al, 0           ; 1
   mov  [ebp], eax      ; 2 - No delay on the Pentium processor
   ```
   Example (Intel486 processor):
   ```
   mov  al, 0       ; 1
                    ; 2  1 clock penalty
   mov  [ebp], eax  ; 3
   ```

The Pentium Pro processor exhibits the same type stall as the Intel386 and Intel486 processors, except that the cost is much higher. The read is stalled until the partial write retires, which can be considerably longer than one cycle.

Avoid using a large register (EAX) after writing a partial register (AL, AH, AX), which is contained in the large register. This causes a partial stall condition on the Pentium Pro processor. This applies to all of the small/large register pairs: AL/AH/AX/EAX, BL/BH/BX/EBX, CL/CH/CX/ECX, DL/DH/DX/ EDX. Additional information on partial register stalls is in section 3.9

## 3.3. Prefetch bandwidth

The Intel486 processor prefetch unit will accesses the on-chip cache to fill the prefetch queue whenever the cache is idle, and there is enough room in the queue for another cache line (16 bytes). If the prefetch queue becomes empty, it can take up to three additional clocks to start the next instruction. The prefetch queue is 32 bytes in size (2 cache lines).

Because data accesses always have priority over prefetch requests, keeping the cache busy with data accesses can lock out the prefetch unit. As a result, optimized code should avoid four consecutive memory instructions that have no stalls. If any of the instructions have an index then there will be an opening for prefetch.

It is important to arrange instructions so that the memory bus is not used continuously by a series of memory-reference instructions. The instructions should be rearranged so that there is a non-memory referencing instruction (such as a register instruction) at least two clocks before the prefetch queue becomes exhausted. This will allow the prefetch unit to transfer a cache line into the queue.

Such arrangement of the instructions will not affect the performance of the Intel386, Pentium and Pentium Pro processors because the Intel386 processor has no cache, and the Pentium and Pentium Pro processors have separate instruction and data caches.

In general, it is difficult for a compiler to model the Intel486 processor prefetch buffer behavior. A sequence of four consecutive memory instructions without stalls (i.e. index penalty) will probably stall because of the prefetch buffers being exhausted.

## 3.4. Alignment

### 3.4.1. CODE

The Intel486 processor has a cache line size of 16 bytes, the Pentium and Pentium Pro processors have a cache line size of 32 bytes. Since the Intel486 processor has only two prefetch buffers (16 bytes each), code alignment has a direct impact on Intel486 processor performance as a result of the prefetch buffer efficiency. The Pentium Pro processor follows the alignment rules of the Intel486 processor. Code alignment has little effect on the Pentium processor performance because it has two prefetch buffers working in tandem. The Intel386 processor with no on-chip cache and a decoupled prefetch unit is not sensitive to code alignment.

For optimal performance across the family, it is recommended that:

- Loop entry labels should be aligned to the next 0 MOD 16 when it is less than 8 bytes away from that boundary.
- Labels that follow a conditional branch should not be aligned.
- Labels that follow an unconditional branch or function call should be aligned to the next 0 MOD 16 when it is less than 8 bytes away from that boundary.

### 3.4.2. DATA

A misaligned access in the data cache or on the bus costs at least an extra 3 cycles on both the Intel486 and Pentium processors. A misaligned access in the data cache costs more cycles on the Pentium Pro processor. It is recommended that data be aligned on the following boundaries to enable more execution performance from all processors

### 3.4.3. 2-BYTE DATA

A 2-byte object should be fully contained within an aligned 4-byte word (i.e., its binary address should be xxxx00, xxxx01, xxxx10, but not xxxx11).

### 3.4.4. 4-BYTE DATA

The alignment of 4-byte object should be on a 4-byte boundary.

### 3.4.5. 8-BYTE DATA

An 8-byte datum (64 bit, e.g., double precision reals) should be aligned on an 8-byte boundary.

## 3.5. Prefixed Opcodes

On the Intel386 processor and the Intel486 processor, all prefix opcodes require an additional clock to decode. On the Pentium processor, an instruction with a prefix is pairable in the U pipe (PU) if the instruction (without the prefix) is pairable in both pipes (UV) or in the U pipe (PU). This is a special case of pairing. The prefixes are issued to the U pipe and get decoded in one cycle for each prefix and then the instruction is issued to the U pipe and may be paired. On the Pentium Pro processor, prefix opcodes may incur severe penalties during decode. It is unlikely that a significant number of prefix opcodes will be generated by modern 32-bit compilers.

For the Pentium and the Pentium Pro processors, the following prefixes: lock, segment override, address size, and operand size belong to this group. On the Pentium processor the two-byte opcode map (0f) prefix also belongs in this group. On the Pentium Pro processor the presence of the two-byte opcode map (0f) in the instruction field does not automatically cause extra cycles for decode. A stall occurs if the size of the instruction is changed by this prefix. Note this includes all the 16 bit instructions when executing in 32 bit mode because an operand size prefix is required (e.g., `mov word ptr [..], add word ptr [..], ...`).

Also, for the Pentium processor, the near jcc prefix behaves differently; it does not take an extra cycle to decode and belongs to PV group. Other `0f` opcodes behave as normal prefixed instructions. For optimized code, prefixed opcodes should be avoided.

When prefixed opcodes must be used, there are two cases in which overlap can be achieved between the extra clock it takes to decode a prefix and a cycle used by the previous instruction executing in the same pipe.

- The one cycle penalty from using the result register of a previous instruction as a base or index (AGI). See Figure 1.
- The last cycle of a preceding multi-cycle instruction.

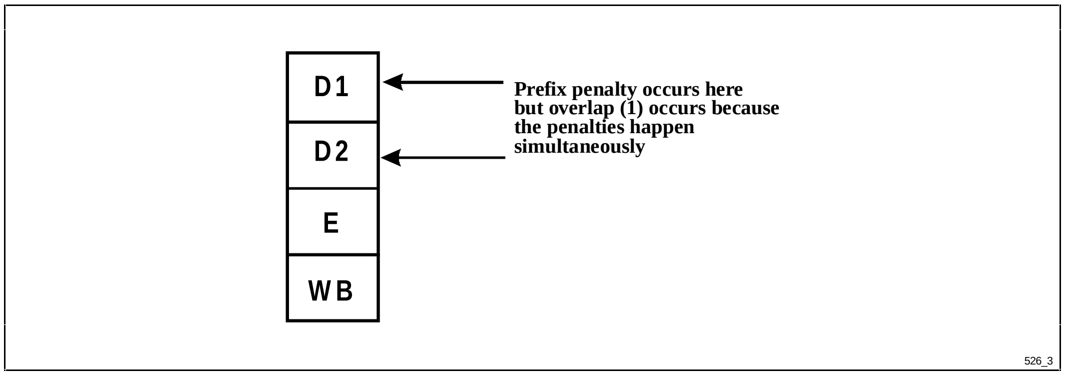

D1
D2
E
WB
Prefix penalty occurs here but overlap (1) occurs because the penalties happen simultaneously
526_3

**Figure 5. Prefix Penalty Example**

## 3.6. Integer Instruction Scheduling

Instruction scheduling is the process of reordering the instructions in a program to avoid stalls and delays and to expose parallelism while maintaining the semantics of the generated code.

Scheduling of integer instructions has two purposes:

1. Eliminate stalls in the Intel486 processor pipeline and each pipe of the Pentium processor.

   There are some conditions where pipe stalls are encountered. The general guideline is to find instructions that can be inserted between the instructions that cause a stall. Since most of the commonly used integer instructions take only one clock, there is not much need to hide latencies. The most common delays which can be avoided through scheduling are AGI 's.
2. Create pairs for maximum throughput from the Pentium processor's dual pipe architecture:

The Pentium processor can issue two instructions for execution simultaneously. This is called *pairing*. There are limitations on which two instructions can be paired and some pairs, even when issued, will not execute in parallel. Pairing details are described in Section 4.1 and in Appendix A.

Scheduling for the Pentium processor's dual pipe is not necessary for the Intel386, Intel486 and Pentium Pro processors but do not adversely impact these processors.

## 3.7. Integer Instruction Selection

The following highlights some instruction sequences to avoid and some sequences to use when generating optimal assembly code. These apply to the Intel486, Pentium and Pentium Pro processors.

1. Lea may be used sometimes as a three/four operand addition instruction (e.g., `lea   ecx, [eax+ebx+4+a]` ).
2. In many cases an `lea` instruction or a sequence of `lea`, `add`, `sub` and `shift` instructions may be used to replace constant multiply instructions. For the Pentium Pro processor the constant multiply is faster relative to other instructions than on the Pentium processor, therefore the trade off between the two options occurs sooner. It is recommended that the integer multiply instruction be used in code designed for Pentium Pro processor execution.

3. This can also be used to avoid copying a register when both operands to an add are still needed after the add, since `lea` need not overwrite its operands.

The disadvantage of the `lea` instruction is that it increases the possibility of an AGI stall with previous instructions. `Lea` is useful for shifts of 2,4,8 because shift takes 2 clocks on Intel486 processor whereas `lea` only takes one. On the Pentium processor, `lea` can execute in either U or V pipes, but shift can only execute in the U pipe. On the Pentium Pro processor, both lea and shift instructions are single uop instructions that execute in one cycle.

**Complex instructions**

Avoid using complex instructions (for example, `enter, leave, loop`). Use sequences of simple instructions instead.

**Zero-Extension of Short**

On the Pentium processor, the `movzx` instruction has a prefix and takes 3 cycles to execute totaling 4 cycles. As with the Intel486 processor, it is recommended the following sequence be used instead:

```
xor  eax, eax
mov  al, mem
```

If this occurs within a loop, it may be possible to pull the `xor` out of the loop if the only assignment to `eax` is the `mov al, mem`. This has greater importance for the Pentium processor since the `movzx` is not pairable and the new sequence may be paired with adjacent instructions.

In order to avoid a partial register stall on the Pentium Pro processor, special hardware has been implemented that allows this code sequence to execute without a stall. Even so, the `movzx` instructions is better for the Pentium Pro processor than the alternative sequences. See Section 3.9 for additional partial stall information.

**Push mem**

The `push` mem instruction takes four cycles for the Intel486 processor. It is recommended to use the following sequence because it takes only two cycles for the Intel486 processor and increases pairing opportunity for the Pentium processor.

```
mov  reg, mem
push reg
```

**Short Opcodes**

Use one byte long instructions as much as possible. This will reduce code size and help increase instruction density in the instruction cache. The most common example is using `inc` and `dec` rather than adding or subtracting the constant 1 with `add` or `sub`. Another common example is using the `push` and `pop` instructions instead of the equivalent sequence.

**8/16 bit operands**

With 8-bit operands, try to use the byte opcodes, rather than using 32-bit operations on sign and zero extended bytes. Prefixes for operand size override apply to 16-bit operands, not to 8-bit operands.

Sign Extension is usually quite expensive. Often, the semantics can be maintained by zero extending 16-bit operands. Specifically, the C code in the following example does not need sign extension nor does it need prefixes for operand size overrides.

```
static short int a, b;
if (a==b) {
   . . .
}
```

Code for comparing these 16-bit operands might be:

```
U pipe            V  pipe

                  xor  eax, eax     xor  ebx, ebx          ; 1
                  movw  ax, [a]                            ; 2 (prefix) + 1
                  movw  bx, [b]                            ; 4 (prefix) +1
                                    cmp  eax, ebx          ; 6
```

Of course, this can only be done under certain circumstances, but the circumstances tend to be quite common. This would not work if the compare was for greater than, less than, greater than or equal, and so on, or if the values in `eax` or `ebx` were to be used in another operation where sign extension was required.

The Pentium Pro processor provides special support to `XOR` a register with itself, recognizing that clearing a register does not depend on the old value of the register. Additionally, special support is provided for the above specific code sequence to avoid the partial stall. See section 3.9 for more information.

The straight-forward method may be slower on the Pentium or Intel486 processors.

```
movsw       eax, a     ; 1  prefix + 3
movsw       ebx, b     ; 5
cmp         ebx, eax   ; 9
```

However, the Pentium Pro processor has improved the performance of the `movzx` instructions in order to reduce the prevalence of partial stalls. Pentium Pro processor specific code should use the `movzx` instructions.

**Compares**

Use `test` when comparing a value in a register with 0. Test essentially '`ands`' the operands together without writing to a destination register. If you '`and`' a value with itself and the result sets the zero condition flag, the value was zero. `Test` is preferred over '`and`' because the '`and`' writes the result register which may subsequently cause an AGI or an artificial output dependence on the Pentium Pro processor. `Test` is better than `cmp` .., `0` because the instruction size is smaller.

Use `test` when comparing the result of a Boolean '`and`' with an immediate constant for equality or inequality if the register is `eax`. `(if (avar & 8) { })`.

On the Pentium processor`, Test` is a one cycle pairable instruction when the form is `eax, imm` or `reg, reg`. Other forms of `test` take two cycles and do not pair.

**Address Calculations**

Pull address calculations into load and store instructions. Internally, memory reference instructions can have 4 operands: a relocatable load-time constant, an immediate constant, a base register, and a scaled index register. (In the segmented model, a segment register may constitute an additional operand in the linear address calculation.) In many cases, several integer instructions can be eliminated by fully using the operands of memory references.

When there is a choice to use either a base or index register, always choose the base because there is a 1 clock penalty on the Intel486 processor for using an index.

### Clearing a Register

The preferred sequence to move zero to a register is `xor reg, reg`. This saves code space but sets the condition codes. In contexts where the condition codes must be preserved, use `mov reg, 0`.

### Integer Divide

Typically, an integer divide is preceded by a `cdq` instruction (Divide instructions use `edx: eax` as the dividend and `cdq` sets up `edx`). It is better to copy `eax` into `edx`, then right shift `edx` 31 places to sign extend. The copy/shift takes the same number of clocks as `cdq` on both the Pentium and Intel486 processors, but the copy/shift scheme allows two other instructions to execute at the same time on the Pentium processor. If you know the value is positive, use `xor edx, edx`.

On the Pentium Pro processor the `cdq` instruction is faster since `cdq` is a single uop instruction as opposed to two instructions for the copy/shift sequence.

### Prolog Sequences

Be careful to avoid AGl's in the prolog due to register `esp`. Since `push` can pair with other `push` instructions, saving callee-saved registers on entry to functions should use these instructions. If possible, load parameters before decrementing `esp`.

In routines that do not call other routines (leaf routines), use `esp` as the base register to free up `ebp`. If you are not using the 32-bit flat model, remember that `ebp` cannot be used as a general purpose base register because it references the stack segment.

### Avoid Compares with Immediate Zero

Often when a value is compared with zero, the operation producing the value sets condition codes which can be tested directly by a `jcc` instruction. The most notable exceptions are `mov` and `lea`. In these cases, use `test.`

### Epilog Sequence

If only 4 bytes were allocated in the stack frame for the current function, instead of incrementing the stack pointer by 4, use `pop` instructions. This avoids AGls and helps both Intel486 and Pentium processors. For Pentium processor use 2 `pops` for eight bytes.

**Integer Multiply by Constant**

The integer multiply by an immediate can usually be replaced by a faster series of shifts, adds, subs, and leas.

1. Binary Method

   In general, if there are 8 or fewer bits set in the binary representation of the constant, it is better not to do the integer multiply. On an Intel486 processor, the break even point is lower: it is profitable if 6 bits or less are in the constant. Basically, shift and add for each bit set. The Pentium Pro processor executes an integer multiply by an immediate in 1 clock, therefore the break even point is 2 bits set.
2. Factorization Method

   This is done by factoring the constant by powers of two plus or minus one, and the constant plus or minus one by powers of two. If the number can be factored by powers of two, then the multiplication can be performed by a series of shifts. If powers of two plus or minus one are included a shift of the previous result and an add or subtract of the previous result can be generated. If the given number plus or minus one can be factored by a power of two, a shift of the previous result and an add or subtract of the original operand can be generated. An iterative for checking powers of two from 31 to 1 can be done. The shift amount needed, and an ordinal to specify an add or subtract is saved for each factor. This information can be used in reverse order to generate the needed instructions.

For example:
```
imul  eax, 217 ; 10 clocks, no pairing
```

In checking powers of two in decreasing order it is found that 217 will divide by 31.

```
217/31 = 7.       31 = (2^^5)-1
save shift = 5 and ordinal = sub_previous_result
```

After a check of 217/31 or 7, it is found that 7+1 is divisible by 8.
```
save shift = 3 and ordinal = sub_operand
```

After factoring the instructions can be generated in reverse.
```
mov    ecx, eax           ; 1
       shl  eax, 3        ; 2
       sub  eax, ecx      ; 3
       mov  ecx, eax      ; 4
       shl  eax, 5        ; 5
       sub  eax, ecx      ; 6
```

This code sequence allows scheduling of other instructions in the Pentium processor's V pipe.

The imul instruction is much faster than most alternative sequences on the Pentium Pro processor. Replace an `imul instruction` by an alternative only if the alternative is one instruction.

## 3.8. Branch Prediction

Branch Optimizations are the most important optimizations for the Pentium Pro processor. These optimizations benefit the Pentium processor and can also help the Intel486 processor, but under different conditions since the Intel486 processor does not have dynamic branch prediction.

### 3.8.1 DYNAMIC PREDICTION

Three elements of dynamic prediction are important:

- If the instruction address is not in the BTB execution is predicted to continue without branching (fall through)
- Predicted taken branches have a 1 clock delay
- The BTB stores a 4-bit history of branch predictions.

The first element suggests that branches should be followed by code that will be executed. Never follow a branch with data.

To avoid the delay of one clock for taken branches simply insert additional work between branches that are expected to be taken. This delay restricts the minimum size of loops to two cycles, if you have a very small loop that takes less than 2 cycles, unroll it.

The branch predictor on the Pentium Pro processor correctly predicts regular patterns of branches. For example it correctly predicts a branch within a loop that is taken on every odd iteration, and not taken on every even iteration.

### 3.8.2 STATIC PREDICTION (PENTIUM® PRO PROCESSOR SPECIFIC)

In the Pentium Pro processor, if a branch does not have a history in the BTB then the branch is predicted using a static algorithm. The static prediction algorithm is as follows:

- Predict unconditional branches taken
- Predict backward conditional branches taken. This rule is suitable for loops.
- Predict forward conditional branches to fall through.

The performance penalty for static prediction is 6 clocks. The penalty for NO prediction or an incorrect prediction is greater than 12 clocks. The following chart illustrates the static branch prediction algorithm:

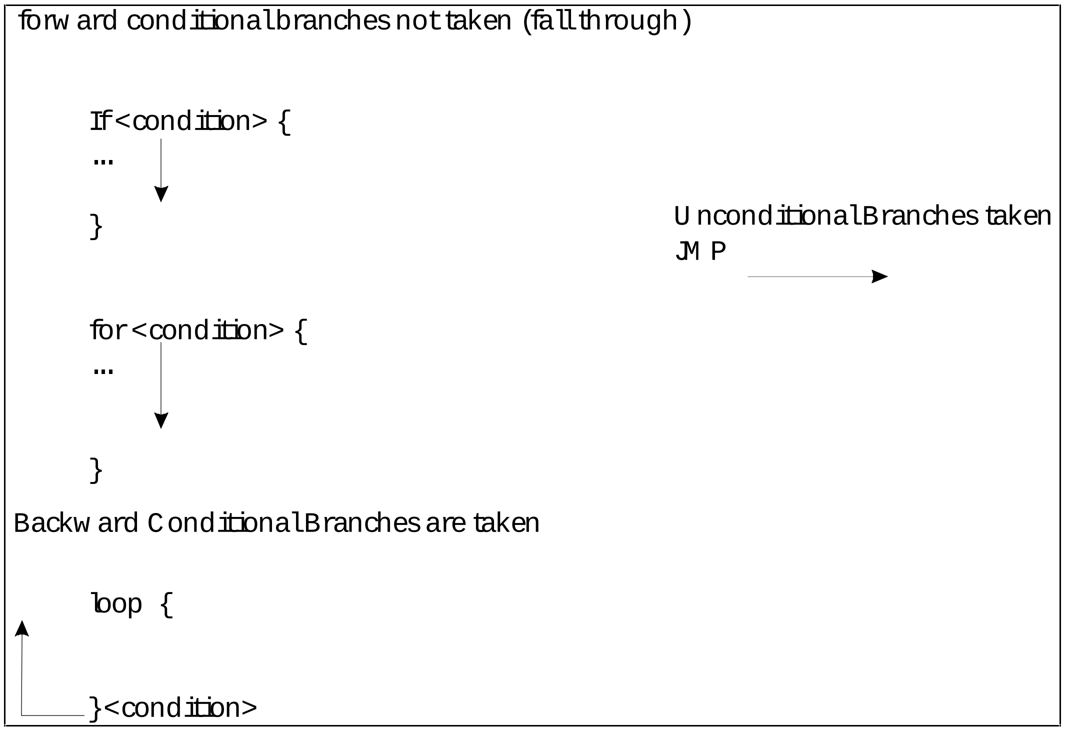

forward conditional branches not taken (fall through)
If<condition> {
...
}
for<condition> {
...
}
Unconditional Branches taken
JMP
Backward Conditional Branches are taken
loop {
}<condition>

**Figure 6. Pentium® Pro Processor Static Branch Prediction Algorithm**

The following examples illustrate the basic rules for the static prediction algorithm.

```
A.Begin:    MOV   EAX, mem32
            AND   EAX, EBX
            IMUL  EAX, EDX
            SHLD  EAX, 7
            JC    Begin
```

In this example, the backwards branch is not in the BTB the first time through therefore it is not predicted. Static prediction will be correct the first time through as taken.

```
B.          MOV   EAX, mem32
            AND   EAX, EBX
            IMUL  EAX, EDX
            SHLD  EAX, 7
            JC    Begin
            MOV   EAX, 0
Begin:      Call  Convert
```

The conditional forward branch is not in the BTB the first time through. Static prediction will predict the branch to fall through. The call will not be predicted in the BTB the first time through. The call will be predicted as taken by the static prediction algorithym.

In these examples the conditional branch has only two alternatives: taken and not taken. Indirect branches, such as switch statements, computed GOTOs or calls through pointers, can go to an arbitrary number of locations. If the branch has a skewed target destination (i.e. 90% of the time it branches to the same address) then the BTB will most of the time predict accurately. However if the target destination is not predictable, performance can degrade quickly. Performance can be improved by changing the indirect branches to conditional branches that can be predicted.

## 3.9. Partial Register Penalties

On the Intel386 and Intel486 processors, if the EAX register is read immediately after the AL register is written, the read is stalled for one cycle. The same stall occurs whenever a small partial register is written just before the large register is read that contains the small partial register.

The Pentium processor eliminates this penalty.

The Pentium Pro processor exhibits the same stall as the Intel386 and Intel486 processors, except that the instructions do not need to be consecutive and the stall is a minimum of 7 cycles and could be much longer. The read of the register waits until the partial write to the register retires. In addition, any uops that follow the stalled uop will also wait until the cycle after the stalled uop continues through the pipe.

In general, avoid reading a large register (EAX) after writing a small register (AL), which is contained in the large register, as it produces a stall.

This applies to all of the small/large register pairs: AL/EAX, BL/EBX, CL/ECX, DL/ EDX. The Pentium Pro processor has a similar stall if a register is not read at the same alignment as it is written: This applies to all of the middle/large register pairs: AH/EAX, BH/EBX, CH/ECX, DH/ EDX.

Special cases of reading and writing small/large register pairs have been implemented in the Pentium Pro processor in order to simplify the blending of code across processors. The special cases include the xor and sub instructions as shown in the following examples:

```
xor eax,eax
movb al,mem8
use eax  <------- no partial stall

xor eax,eax
movw ax,mem16
use eax  <------- no partial stall

xor ax,ax
movb al,mem8
use ax <--------- no partial stall

xor eax,eax
movb al,mem8
use ax <--------- no partial stall

xor ah,ah
movb al,mem8
use ax <----------- no partial stall
```

In general when implementing this sequence, always zero the large register then write to the lower half of the register. The special cases have been implemented for xor and sub when using eax, ebx, ecx, edx, ebp, esp, edi, esi.

## 3.10 Cache Effects

The Pentium Pro Processor has a "write allocate by read-for-ownership" cache, whereas the Pentium processor has a "no-write-allocate; write through on write miss" cache.

On the Pentium Pro processor, when a write occurs and the write misses the cache then the entire 32-byte cache line is fetched. On the Pentium processor, when the same write miss occurs then the write is simply sent out to memory.

Write allocate is generally advantageous, since sequential stores are merged into burst writes, and the data remains in the cache for use by later loads. This is why the Pentium Pro processor adopted this stragegy, and why some of the Pentium processor-based systems designs implement it for the L2 cache, even though the Pentium processor uses writethrough.

Write allocate can be a disadvantage in code which write just one piece of a cache line, does not read the cache line, has strides greater than 32 bytes, and does those writes to a large number of addresses (greater than 8000).

When a large number of writes occur within an application, as in the example program below, and the stride is bigger than the 32-byte cache line, and the array is large, every store, on the Pentium Pro processor, will cause an entire cache line to be fetched. In addition, this fetch will probably be accompanied by the eviction of one (sometimes two) dirty cache lines.

The result is every store causes an additional cache line fetch therefore a slow down of the execution of the program. When many writes occur in a program the performance decrease can be significant. The Sieve of Erastothenes program is a simplistic example that demonstrates these cache effects. In this example, a large array is stepped through in increasing strides while writing a single value of the array with zero.

**Note:**

This is a very simplistic example only used to demonstrate the cache effects, many other optimizations are possible in this code.

**Example: Sieve of Erastothenes**

```
boolean array[2..max]
for(i=2;i<max;i++) {
 array := 1;
}

for(i=2;i<max;i++) {
 if( array[i] ) {
     for(j=2;j<max;j+=i) {
      array[j] := 0;  /* here we assign memory to 0 causing the cache line
                    fetch within the j loop */
     }
 }
}
```

Two optimizations are available for this specific example. One is to pack the array into bits thereby reducing the size of the array, which in turn reduces the number of cache line fetches. The second is to check the value prior to writing thereby reducing the number of writes to memory (dirty cache lines).

## Optimization #1: Boolean

'Boolean', in the program in question, was a char. It may well be better, in some programs, to make the "boolean" array into an array of bits, packed so that read-modify-writes are done (since the cache protocol makes every read into a read-modify-write). But, in this example, the vast majority of strides are greater than 256 bits (one cache line of bits), so the performance increase is not significant.

## Optimization #2: Check before Writing

Another Optimization is to check if the value is already zero before writing.

```
    boolean array[2..max]
    for(i=2;i<max;i++) {
  array := 1;
 }

 for(i=2;i<max;i++) {
  if( array[i] ) {
    for(j=2;j<max;j+=i) {
     if( array[j] != 0 ) {  /* check to see if value is already 0 */
       array[j] := 0;
     }
    }
  }
 }
```

The external bus activity is reduced by half because most of the time in the Sieve program the data is already zero. By checking first you only need one burst bus cycle for the read, and you save the burst bus cycle for every line you do not write.

**Note:**

This operation benefits the Pentium Pro processor but is not a performance optimization for the Pentium processor and, as such, should not be considered generic. Write allocate is generally a performance advantage in system, since sequential stores are merged into burst writes, and the data remains in the cache for use by later loads. This is why the Pentium Pro processor adopted this strategy, and why some of the Pentium processor-based systems designs implement it for the L2 cahce, even though the Pentium processor uses write through.

## 3.11 Profile Guided Optimizations

Our research has shown that by profiling we can determine the best optimizations for each application. The following example illustrates profile guided optimizations.

The block diagram illustrates the program flow as it executes. In order to understand which branches are executed the most times we apply profiling to the application. The program is instrumented as illustrated below.

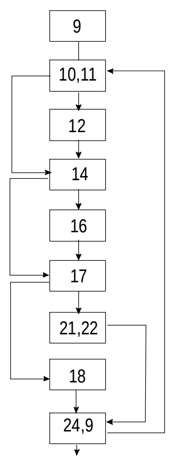


**Figure 7. Profile Guided Optimizations**

**Example:**

```
9     for(i=0;i<3;i++){              Instrument
10          a = i+1;
11    if (a<0){
12                a = f1(a);
13          }
14          if (a >=0){
15          a = f1(a);                 Instrument
16          }
17          if (a > 0){
18                a = a++;
19          }
20          else {
21                a = f1(a+1);
22                a = f1(a);
23          }
24          printf("%d\n", a);
25    }
```

During the instrumentation process the compiler also saves additional information in a file that can be accessed later. Once the instrumentation is completed, the application is executed normally. As execution proceeds, information about the execution is stored in the data file with the previous information.

Following is a representation of how the file might look after executing the application.

**Example:**

```
9    for(i=0;i<3;i++){
10         a = i+1;
11         if (a<0){
12               a = f1(a);
13               }
14         if (a >=0){
15               a = f1(a);
16               }
17         if (a > 0){
18               a = a++;
19               }
20         else {
21               a = f1(a+1);
22               a = f1(a);
23               }
24         printf("%d\n", a);
25         }
```

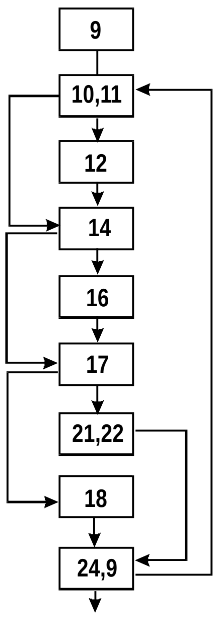


Figure 8.  Profile Guided Optimizations

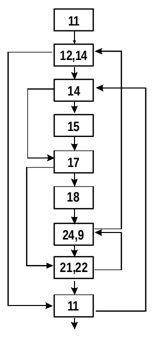
Table 1. Profile Data File Contents

| Source Line Number | # of times Executed | # of times Taken |
|---|---|---|
| 9 | 4 | 3 |
| 11 | 3 | 0 |
| 14 | 3 | 3 |
| 17 | 3 | 3 |

```
9     for(i=0;i<3;i++){
10          a = i+1;
11          if (a<0){
12                a = f1(a);  /* Never Executed*/
13                }
14          if (a >=0){
15                a = f1(a);  /* Always Executed*/
16                }
17          if (a > 0){
18                a = a++;    /* Always Executed*/
19                }
20          else {
21                a = f1(a+1);/* Never Executed*/
22                a = f1(a);
23                }
24          printf("%d\n", a);/* Always Executed*/
25          }
```

The information saved in the data file tells us that the branch at line 11 was executed 3 times but never taken, therefore the code sequence at line 12 was never executed. In addition, the code sequence at lines 20 through 23 are never executed because the branch at line 17 always falls through. For both the Pentium and Pentium Pro processors these code sequences are prefetched needlessly each time this section of code is executed. The best remedy would be to move these sequences out of line of execution.

The compiler will perform this task for us, by recompiling the source and telling the compiler to use the data we have collected, during compilation the compiler accesses the information in the data file and determines the areas of code to be optimized. Following recompilation the code execution flow looks like the following illustration:



526_3

**Figure 9. Optimized Code Flow**

The compiler reorganizes the code sequence, moving the code that is not executed frequently to the end of the program. This allows the processor to prefetch only the code that is frequently executed.

The idea behind profile-guided optimization is to use the compiler to give us the best optimization for our specific application with minimal effort on the part of the programmer. We are using profile guided optimizations to determine optimal register allocation, to define code sequences for inlining and to make decisions about cloning. With register allocation we can give a higher priority to register candidates that are used more based on dynamic, not static, probabilities. When determining whether to inline or clone a code sequence we can identify the frequently called sites to inline and determine the frequency of execution of sites that are candidates for cloning. This gives us the ability to minimize code expansion with increased performance.

# 4.0. PROCESSOR SPECIFIC OPTIMIZATIONS

## 4.1. Pentium® Processor Specific Optimizations

This chapter specifies the Pentium processor specific optimizations.

### 4.1.1. PAIRING

Reordering instructions should be done in order to increase the possibility of issuing two instructions simultaneously. Dependent instructions should be separated by at least one other instruction. Pairing cannot be performed when the following conditions occur:

1. The next two instructions are not pairable instructions (See Appendix A for pairing characteristics of individual instructions.) In general, most simple ALU instructions are pairable.
2. The next two instructions have some type of register contention (implicit or explicit). There are some special exceptions to this rule where register contention can occur with pairing. These are described later.
3. The instructions are not both in the instruction cache. An exception to this which permits pairing is if the first instruction is a one byte instruction.

#### 4.1.1.1. Instruction Set Pairability

##### 4.1.1.1.1 Unpairable Instructions (NP)

1. *shift/rotate* with the shift count in cl
2. Long-Arithmetic instructions for example, `mul`, `div`
3. Extended instructions for example, `ret, enter, pusha, movs, stos, loopnz`
4. Some Floating-Point Instructions for example, `fscale, fldcw, fst`
5. Inter-segment instructions for example, `push sreg, call far`

##### 4.1.1.1.2 Pairable Instructions Issued to U or V Pipes (UV)

1. Most 8/32 bit ALU operations for example, `add, inc, xor`,
2. All 8/32 bit compare instructions for example `cmp, test`
3. All 8/32 bit stack operations using registers for example, `push reg, pop reg`

##### 4.1.1.1.3 Pairable Instructions Issued to U Pipe (PU)

These instructions must be issued to the U pipe and can pair with a suitable instruction in the V-Pipe. These instructions never execute in the V pipe.

1. Carry and borrow instructions for example, `adc, sbb`
2. Prefixed instructions (see next section)

3. Shift with immediate
4. Some Floating-Point Operations for example, `fadd, fmul, fld`

#### 4.1.1.1.4 Pairable Instructions Issued to V Pipe (PV)

These instructions can execute in either the U pipe or the V pipe but they are only paired when they are in the V pipe. Since these instructions change the instruction pointer (eip), they cannot pair in the U pipe since the next instruction may not be adjacent. Even when a branch in the U pipe is predicted "not taken", it will not pair with the following instruction.

1. Simple control transfer instructions for example - `call near, jmp near, jcc`. This includes both the `jcc` short and the `jcc` near (which has a `0f` prefix) versions of the conditional jump instructions.
2. `fxch`

### 4.1.1.2. Unpairability Due To Register Dependencies

The pairability of an instruction is also affected by its operands. The following are the combinations that are not pairable due to *register contention*. Exceptions to these rules are given in the next section.

1. The first instruction writes to a register that the second one reads from *(flow-dependence)*.

   Example:
   ```
   mov  eax, 8
   mov  [ebp], eax
   ```
2. Both instructions write to the same register *(output-dependence)*.

   Example:
   ```
   mov  eax, 8
   mov  eax, [ebp]
   ```

This limitation does not apply to a pair of instructions which write to the eflags register (e.g. two ALU operations that change the condition codes). The condition code after the paired instructions execute will have the condition from the V pipe instruction.

Note that a pair of instructions in which the first reads a register and the second writes to it (anti- dependence) is pairable.

Example:
```
mov  eax, ebx
mov  ebx, [ebp]
```

For purposes of determining register contention, a reference to a byte or word register is treated as a reference to the containing 32-bit register. Hence,

```
mov  al, 1
mov  ah, 0
```

do not pair due to apparent output dependencies on `eax`.

### 4.1.1.3. Special Pairs

There are some instructions that can be paired although the general rule prohibits this. These special pairs overcome register dependencies. Most of these exceptions involve implicit reads/writes to the `esp` register or implicit writes to the condition codes:

Stack Pointer:

1. `push reg/imm; push reg/imm`
2. `push reg/imm; call`
3. `pop reg     ; pop reg`

Condition Codes:

1. `cmp         ; jcc`
2. `add         ; jne`

Note that the special pairs that consist of `push/pop` instructions may have only *immediate* or *register* operands, not memory operands.

### 4.1.1.4. Restrictions On Pair Execution

There are some pairs that may be issued simultaneously but will not execute in parallel:

1. If both instructions access the same data-cache memory bank then the second request (V pipe) must wait for the first request to complete. A bank conflict occurs when bits 2-4 are the same in the two physical addresses. This is because the cache is organized as 8 banks of 32-bit wide data entries. A bank conflict incurs a one clock penalty on the V pipe instruction .
2. Inter-pipe concurrency in execution preserves memory-access ordering. A multi-cycle instruction in the U pipe will execute alone until its last memory access.

```
add  eax, meml
add  ebx, mem2      ; 1
(add)   (add)       ; 2  2-cycle
```

The instructions above add the contents of the register and the value at the memory location, then put the result in the register. An add with a memory operand takes two clocks to execute. The first clock loads the value from cache, and the second clock performs the addition. Since there is only one memory access in the U pipe instruction, the add in the V pipe can start in the same cycle.

```
add  meml, eax          ; 1
(add)                   ; 2
(add) add  mem2, ebx    ; 3
(add)                   ; 4
(add)                   ; 5
```

The above instructions add the contents of the register to the memory location and store the result at the memory location. An add with a memory result takes 3 clocks to execute. The first clock loads the value, the second performs the addition, and the third stores the result. When paired, the last cycle of the U pipe instruction overlaps with the first cycle of the V pipe instruction execution.

No other instructions may begin execution until the instructions already executing have completed.

To expose the opportunities for scheduling and pairing, it is better to issue a sequence of simple instructions rather than a complex instruction that takes the same number of cycles. The simple instruction sequence can take advantage of more issue slots. Compiler writers/programmers can also choose to reconstruct the complex form if the pairing opportunity does not materialize. The load/store style code generation requires more registers and increases code size. This impacts Intel486 processor performance, although only as a second order effect. To compensate for the extra registers needed, extra effort should be put into register allocation and instruction scheduling so that extra registers are only used when parallelism increases.

### 4.1.2. PENTIUM® PROCESSOR FLOATING-POINT OPTIMIZATIONS

The Pentium processor is the first generation of the Intel386 family that implements a pipelined floating-point unit however, in order to achieve maximum throughput from the Pentium processor floating-point unit, specific optimizations must be done.

#### 4.1.2.1. Floating-Point Example

```
Source code:
       A = B + C + D;
       E = F + G + E;

Assembly code:

       fld  B

       fadd C
       fadd D
       fstp A
       fld  F
       fadd G
       fadd H
       fstp E

Total: 20 Cycles
```

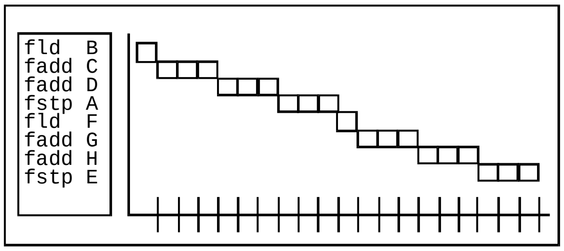


**Figure 10. Floating Point Example**

To exploit the parallel capability of the Pentium processor we need to determine which instructions can be executed in parallel. The two statements are independent, therefore their assembly instructions can be scheduled to improve the execution speed

Source code

```
                  A= B + C + D;
                  E = F + G + E;
        fld   B                 fld   F
        fadd  C                 fadd  G
        fadd  D                 fadd  H
        fstp  A                 fstp  E
```

Most floating-point operations require that one operand and the result use the top of stack. This makes each instruction dependent on the previous instruction and inhibits overlapping the instructions.

One obvious way to get around this is to change the architecture and have floating-point registers, rather than a stack. The code would look like this:

```
fld   B     →     F1
fadd  F1, C →     F1
fld   F     →     F2
fadd  F2,G  →     F2
fadd  F1,D  →     F1
fadd  F2,H  →     F2
fstp  F1    →     A
fstp  F2    →     E
```

Unfortunately, upward and downward compatibility would be lost. Instead, the `fxch` instruction was made "fast". This provides us another way to avoid the top of stack dependencies. The `fxch` instructions can be paired with the common floating-point operations, so there is no penalty on the Pentium processor.

```
                                        STO   ST1
fld   B     Þ     F1          fld B       B
fadd  F1, C Þ     F1          fadd C      B+C
fld   F     Þ     F2          fld F       F     B+C
fadd  F2,G  Þ     F2          fadd G      F+G   B+C
                              fxch ST(1)  B+C   F+G
fadd  F1,D  Þ     F1          fadd D      B+C+D F+G
                              fxch ST(1)  F+G   B+C+D
fadd  F2,H  Þ     F2          fadd H      F+G+H B+C+D
                              fxch ST(1)  B+C+D F+G+H
fstp  F1    Þ     A           fstp A      F+G+H
fstp  F2    Þ     E           fstp E
```

On the Intel486 processor, the index penalty and the added cost of `fxch` are apparent. The index penalty does not overlap with the `fxch` instruction.

On the Pentium processor, the `fxch` instructions pair with preceding `fadd` instructions and execute in parallel with them. The `fxch` instructions move an operand into position for the next floating-point instruction. The result is an improvement in execution speed.

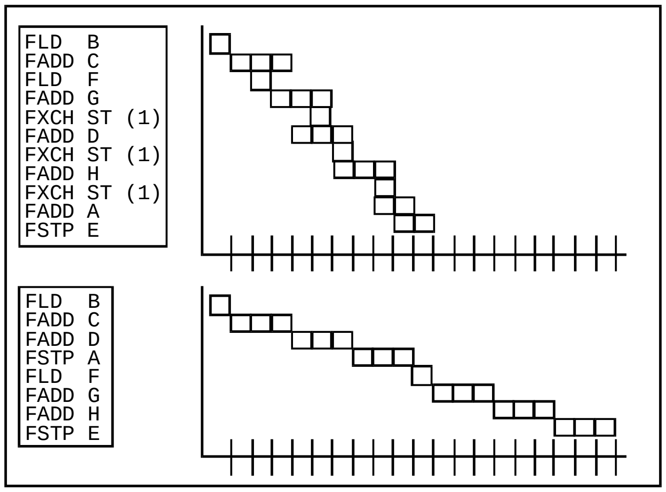


**Figure 11. Floating-Point Example Before and After Optimization**

### 4.1.2.2. FXCH RULES AND REGULATIONS

The `fxch` instruction can be executed for "free" when all of the following conditions occur:

An FP instruction follows the fxch instruction.

An FP instruction belonging to the following list immediately precedes the `fxch` instruction: `fadd, fsub, fmul, fld, fcom, fucom, fchs, ftst, fabs, fdiv`.

This `fxch` instruction has already been executed. This is because the instruction boundaries in the cache are marked the first time the instruction is executed, so pairing only happens the second time this instruction is executed from the cache.

This means that this instruction is almost "free" and can be used to access elements in the deeper levels of the FP stack instead of storing them and then loading them again.

### 4.1.2.3. MEMORY OPERANDS

Performing a floating-point operation on a memory operand instead of on a stack register costs no cycles. In the integer part of the Pentium processor, it was better to avoid memory operands. In the floating-point part, you are encouraged to use memory operands. Be aware that memory operands may cause a data cache miss, causing a penalty. Also, floating-point operands are 64-bit operands which need to be 8-byte aligned or an additional penalty will occur.

### 4.1.2.4. Floating-Point Stalls

There are cases where a delay occurs between two operations. Instructions should be inserted between the pair that cause the pipe stall. These instructions could be integer instructions or floating-point instructions that will not cause a new stall themselves. The number of instructions that should be inserted depends on the delay length.

One example of this is when a floating-point instruction depends on the result of the immediately preceding instruction which is also a floating-point instruction. In this case, it would be advantageous to move integer instructions between the two FP instructions, even if the integer instructions perform loop control. The following example restructures a loop in this manner:

```
for (i=0; i<Size; i++)
      array1 [i] += array2 [i];
```

```
             Pentium Processor        Intel486 Processor
                              Clocks                      Clocks
TopOfLoop:
  flds   [eax + array2]         2 - AGI                     3
  fadds  [eax + array1]         1                           3
  fstps  [eax + array1]         5 - Wait for fadds         14 - Wait for fadds
  add    eax, 4                 1                           1
  jnz    TopOfLoop              0 - Pairs with add          3
                              ------                     ------
                                9                          24
```

```
                         Pentium Processor         Intel486 Processor
                              Clocks                      Clocks
TopOfLoop:
  fstps [eax + array1]          4 - Wait for fadds,AGI     10 - Wait for fadds
LoopEntryPoint:
  flds  [eax + array2]          1                           3
  fadds [eax + array1]          1                           3
  add eax, 4                    1                           1
  jnz TopOfLoop                 0 - Pairs with add          3
                              ------                     ------
                                7                          20
```

By moving the integer instructions between the fadds and fstps, both processors can execute the integer instructions while the fadds is completing in the floating-point unit and before the fstps begins execution. Note that this new loop structure requires a separate entry point for the first iteration because the loop needs to begin with the flds. Also, there needs to be an additional fstps after the conditional jump to finish the final loop iteration.

1. Floating-Point Stores

   A floating-point store must wait an extra cycle for its floating-point operand. After an `fld`, an `fst` must wait one clock. After the common arithmetic operations, `fmul` and `fadd`, which normally have a latency of three, `fst` waits an extra cycle for a total of four[1].

[1] This set also includes the `faddp, fsubrp, ...` instructions

```
fld   meml          ; 1 fld takes 1 clock
                          ; 2 fst waits, schedule something here
      fst   mem2          ; 3,4 fst takes 2 clocks

      fadd  meml          ; 1 add takes 3 clocks
                          ; 2 add, schedule something here
                          ; 3 add, schedule something here
                          ; 4 fst waits, schedule something here
      fst   mem2          ; 5,2 fst takes 2 clocks
```

In the next example, the store is not dependent on the previous load:

```
      fld   meml          : 1
      fld   mem2          ; 2
      fxch  st(l)         ; 2
      fst   mem3          ; 3 stores values loaded from meml
```

2. A register may be used immediately after it has been loaded (with `fld`).

```
      fld   mem1          ; l
      fadd  mem2          ; 2,3,4
```

3. Use of a register by a floating-point operation immediately after it has been written by another `fadd, fsub`, or `fmul` causes a 2 cycle delay. If instructions are inserted between these two, then latency and a potential stall can be hidden.
4. There are multi-cycle floating-point instructions (`fdiv` and `fsqrt`) that execute in the floating-point unit pipe. While executing these instructions in the floating-point unit pipe, integer instructions can be executed in parallel. Emitting a number of integer instructions after such an instruction will keep the integer execution units busy (the exact number of instructions depends on the floating-point instruction's cycle count)
5. The integer multiply operations, `mul` and `imul`, are executed in the floating-point unit so these instructions cannot be executed in parallel with a floating-point instruction.
6. A floating-point multiply instruction (`fmul`) delays for one cycle if the immediately preceding cycle executed an `fmul` or an `fmul / fxch` pair. The multiplier can only accept a new pair of operands every other cycle.
7. Transcendental operations execute in the U pipe and nothing can be overlapped with them, so an integer instruction following such an instruction will wait until that instruction completes.
8. Floating-point operations that take integer operands (`fiadd` or `fisub ..`) should be avoided. These instructions should be split into two instructions: `fild` and a floating-point operation. The number of cycles before another instruction can be issued (throughput) for fiadd is 4, while for fild and simple floating-point op it is 1.

   Example:

```
      Complex Instructions          Better for Potential Overlap

      fiadd  [ebp] ; 4              fild   [ebp]  ; 1
                                    faddp  st(l)  ; 2
```

   Using the `fild - faddp` instructions yields 2 free cycles for executing other instructions.

9. The `fstsw` instruction that usually appears after a floating-point comparison instruction (`fcom, fcomp, fcompp`) delays for 3 cycles. Other instructions may be inserted after the comparison instruction in order to hide the latency.
10. Moving a floating-point memory/immediate to memory should be done by integer moves (if precision conversion is not needed) instead of doing `fld - fstp`.

    Examples for floating-point moves:

double precision: 4 vs. 2 cycles

```
fld  [ebp]     ; 1           mov  eax, [ebp]        ; 1
                      ; 2           mov  edx, [ebp+4]      ; 1
        fstp [edi]     ; 3,4         mov  [edi], eax        ; 2
                                    mov  [edi+4], edx      ; 2
```

single precision: 4 vs. 2 cycles

```
        fld  [ebp]     ; 1           mov  eax, [ebp]        ; 1
                      ; 2           mov  [edi], eax        ; 2
        fstp [edi]     ; 3,4
```

This optimization also applies to the Intel486 processor.

11. Transcendental operations execute on the Pentium processor much faster than on the Intel486 processor. It may be worthwhile inlining some of these math library calls because of the fact that the `call` and *prologue/epilogue* overhead involved with the library calls is no longer negligible. Emulating these operations in software will not be faster than the hardware unless accuracy is sacrificed.
12. Integer instructions generally overlap with the floating-point operations except when the last floating-point operation was fxch. In this case there is a one-cycle delay.

```
U pipe              V  pipe

fadd                fxch                          ; 1
                                                  ; 2 fxch delay
mov   eax, 1        inc       edx                 ; 3
```

## 4.2. PENTIUM® PRO PROCESSOR SPECIFIC OPTIMIZATIONS

This section contains information about the optimizations specific to the Pentium Pro processor and a specification of the new instructions on the Pentium Pro processor.

### 4.2.1. OPTIMIZATION SUMMARY

The most important optimizations for the Pentium Pro processor are to improve branch prediction, eliminate partial stalls, and to use Intel486 processor alignment rules. Each of these improvements are addressed in the blended section of this document. An additional list of guidelines are shown in Appendix C.

Specifics on the timing and latency of instructions is in Appendix D.

### 4.2.2. INSTRUCTION SET

The Pentium Pro processor instruction set includes the following extensions to that of the Pentium processor:

- Conditional move instructions, CMOV for integer and FCMOV for floating-point, which permit performance improvement by eliminating branches.

Both CMOV and FCMOV are controlled by the integer condition codes.

- Instructions to move floating-point conditions to the integer condition codes: FCOMI, FUCOMI, FCOMIP, FUCOMIP, floating-point comparisons which set the integer condition codes directly.

These instructions are required because FCMOV is controlled by the integer condition codes.

- Instructions to read the Pentium Pro processor performance counters: RDMPC.

In addition, several minor extensions are made to existing instructions:

- The CPUID instruction is updated for the Pentium Pro processor.
- The definition of the CPUID instruction is extended to include capability bits to indicate the presence or absence of Pentium Pro processor features in future processors.
- The CPUID instruction is also extended to flush the framebuffer cache.
- The PGE bit of CR4 determines whether MOV to CR3 flushes all PTEs from the TLB, or only those for which the global bit is clear. The PGE bit controls the TLB invalidations that occur during control transfers through task gates similarly.
- The RDTSC instruction no longer faults in VM86 mode.
- UD2, an official, two byte long, undefined instruction is defined.
  - This permits Intel validation tests to be portable, instead of breaking every time a new instruction is defined.
  - This is required by software that provides virtual machine capabilities.

### 4.2.2.1. New Instructions

The Pentium Pro processor implements a limited set of new instructions, defined on the following pages. These new instructions function only on Pentium Pro processors; they create an illegal instruction trap on previous processors. Table 2 summarizes the new instruction encodings.

#### 4.2.2.1.1. CMOV - Conditional Move

**Syntax:**

```
      CMOVcc r16,r/m16
      CMOVcc r32,r/m32
```

**Encoding:**

```
      0F 4x /r
0000 1111   0100 cccc   modrm...
```

**Operation:**

```
CMOV reg1,reg2
      IF condition THEN
            reg1 := reg2
      ENDIF
CMOV reg1,mem
      tmp := load( mem )
      IF condition THEN
            reg1 := tmp
      ENDIF
```

**Description:**

The second operand is read, then the flags are tested for the condition specified in the opcode (cc). If the specified condition is true, the value of the second operand is written to the first operand destination register.

Table 2. New Instruction Encodings

| Function | Instruction | Hex | Binary | Where Described |
|---|---|---|---|---|
| Integer Conditional Move | CMOVcc | 0F 4x /r | 0000 1111<br>0100 cccc | Section 4.2.2.1.1 |
| Floating-Point Conditional Move | FCMOVB<br>FCMOVE<br>FCMOVBE<br>FCMOVU<br><br>FCMOVNB<br>FCMOVNE<br>FCMOVNBE<br>FCMOVNU | DA C0 + i<br>DA C8 + i<br>DA D0 + i<br>DA D8 + I<br><br>DB C0 + i<br>DB C8 + i<br>DB D0 + i<br>DB D8 + i | 1101 101n<br>11cccrrr | Section 4.2.2.1.2 |
| Floating-Point Compare setting Integer Flags | FCOMI<br>FCOMIP<br>FUCOMI<br>FUCOMIP | DB F0+i<br>DF F0+i<br>DB E8+i<br>DF E8+i | 1101 101u<br>1111 p iii | Section 4.2.2.1.3 |
| Read Performance Monitoring Counters | RDMPC | OF 33 | 0000 1111<br>0011 0011 | Section 4.2.2.1.4 |

Note that the memory operand is always read, and may therefore create an exception even if the condition is false.

The operands are always 16 or 32 bit integers, as specified by the current operating mode and operand size prefix.The condition specified by the cccc bits of the opcode is exactly the same as the conditions for Jcc and SETcc.

**Example:**

The branchful code

```
      TST ECX
      JNE 1F
      MOV EAX,EBX
1H:
```

can be replaced by CMOV as follows

```
      TST ECX
      CMOVEQ EAX,EBX
```

Eliminating a branch that may be unpredictable.

**Flags Affected:**

None.

**Exceptions:**

Only those that can be predicted by the memory read access of `CMOV reg,mem`

#### 4.2.2.1.2 FCMOV - Floating-Point Conditional Move

```
Syntax:
      FCMOVcc ST, STi
```

**Encoding:**

```
      n=0
            DA C0 + i         FCMOVB            ST, STi
            DA C8 + i         FCMOVE            ST, STi
            DA D0 + i         FCMOVBE           ST, STi
            DA D8 + i         FCMOVU            ST, STi
      n=1
            DB C0 + i         FCMOVNB           ST, STi
            DB C8 + i         FCMOVNE           ST, STi
            DB D0 + i         FCMOVNBE          ST, STi
            DB D8 + i         FCMOVNU           ST, Sti
```

```
|1101 101n      |     |11cccrrr          |
```

**Operation:**

```
      tmpi := Sti
      tmp0 := ST0
IF condition THEN
            ST0 := STi
ENDIF
```

**Description:**

The integer flags are tested for the condition specified in the opcode. If the specified condition is true, the value of the second operand is read, and written to the destination (stack top).

The conditions tested by FCMOV are as listed in Table 2. These conditions are convenient for testing any of the 6 arithmetic relations as well as ordered or unordered after an FCOM or FTST operation. Conditional assignment for other conditions, such as unordered or denormalized numbers, must be performed by FSTSW and branches.

Note that the FCMOV algorithm described above unconditionally reads both sources, ST0 and Sti. An FCMOV will cause a floating-point stack underflow exception if either of its sources is a stack underflow. FCMOV will not create a floating-point stack overflow.

**Table 3. FCMOV$_{CC}$ Conditions**

| Mnemonic | Meaning | Instruction Subcode | | Conditional Tested Integer Flags | Corres FP Flags | Description |
|---|---|---|---|---|---|---|
| B | Below | 0 | 00 | CF=1 | $C_0$=1 | < |
| NB | Not Below | 1 | 00 | CF=0 | $C_0$=0 | >= |
| E | Equal | 0 | 01 | ZF=1 | $C_3$=1 | = |
| NE | Not equal | 1 | 01 | ZF=0 | $C_3$=0 | != |
| BE | Below or equal | 0 | 10 | CF=1 or ZF=1 | $C_0$=1 or $C_3$=1 | <= |
| NBE | Not below or equal | 1 | 10 | CF=0 and ZF=0 | $C_0$=0 and $C_3$=0 | > |

Table 3. FCMOV$_{CC}$ Conditions

| Mnemonic | Meaning | Instruction Subcode | | Conditional Tested Integer Flags | Corres FP Flags | Description |
|---|---|---|---|---|---|---|
| U | Unordered | 0 | 11 | PF = 1 | $C_2$=1 | Not comparable |
| NU | Not Unordered | 1 | 11 | PF != 1 | $C_2$!=1 | Not comparable |

FPU Flags Affected:

In normal operation, none.

If there is a stack exception, the FSW.C1 flag bit will be set as usual to indicate overflow or underflow (O/ U#).

The FTW field corresponding to the top of stack ST0 will be marked FULL unless there is an unmasked stack underflow exception.

Table 4. FCMOV Stack Exception Behavior

| Initial | | Final | | |
|---|---|---|---|---|
| ST0 | STi | Stack Underflow | FSW.C1 | ST0 |
| Masked Stack Underflow | | | | |
| FULL | FULL | None | Unchanged | Unchanged |
| FULL | EMPTY | Sti | Cleared | Unchanged |
| EMPTY | FULL | ST0 | Cleared | FULL |
| EMPTY | EMPTY | ST0 & STI | Cleared | FULL |
| Unmasked Stack Underflow | | | | |
| FULL | FULL | None | Unchanged | Unchanged |
| FULL | EMPTY | STi | Cleared | Unchanged |
| EMPTY | FULL | ST0 | Cleared | Unchanged |
| EMPTY | EMPTY | ST0 & STi | Cleared | Unchanged |

**NOTES:**

The encodings chosen for FCMOV$_{CC}$ produce an illegal instruction exception on the Intel486 CPU and Pentium processor CPU, but are treated like FNOP by the Intel387 math coprocessor.

**Numeric Exceptions:**
`IS`

**Protected Mode Exceptions:**
#NM if either EM or TS in CR0 is set.

**Real Address Mode Exceptions:**
Interrupt 7 if either EM or TS in CR0 is set.

**Virtual 8086 Mode Exceptions:**
Interrupt 7 if either EM or TS in CR0 is set.

**Example: Floating-Point MIN**

Straightforward code for floating-point minimum on existing processors would be as follows:

```
Intel386 code
      FCOM ST0, ST1
      FNSTSW AX
      SAHF[2]
      JB 1F
      XCHG ST0,ST1
1H:
```

Using FCMOV, this becomes (note: only requiring ST0 to be the minimum effort, not keeping all other FP stack elements identical)

```
FCMOV
      FCOM ST0, ST1
      FNSTSW AX
      SAHF
      FCMOVNB ST0,ST1
```

Better code is obtained by using both FCMOV and the new FCOMI instructions:

```
FCMOV+FCOMI
      FCOMI ST0, ST1
      FCMOVNB ST0,ST1
```

Slightly more code is required if minimum is supposed to take unordered numbers into account.

#### 4.2.2.1.3. FCOMI - Floating-Point Comparison Setting Integer Flags

```
Syntax:
      FCOMI             ST0,STi
      FCOMIP            ST0,STi
      FUCOMI            ST0,STi
      FUCOMIP           ST0,STi
```

**Encoding:**

```
      FCOMI             DB F0+i
      FCOMIP            DF F0+i
      FUCOMI            DB E8+i
      FUCOMIP           DF E8+i
1101 101u              1111 p iii
```

[2]Most compilers emit AND tests instead of SAHF because SAHF is slow on previous processors. We will use SAHF in the example because it is slightly more obvious for didactic purposes.

**Operation:**

```
CASE (relation of operands) OF
      Not comparable:              ZF,PF,CF := 111
      ST0 > STi:                   ZF,PF,CF := 000
      ST0 < STi:                   ZF,PF,CF := 001
      ST0 = STi:                   ZF,PF,CF := 100
CASE floating-point stack status OF
      overflow:              SF := 1
      underflow:             SF := 0
      otherwise:             SF := 0
CASE instruction OF
      FCOMI, FUCOMI:               no FP stack adjustment
      FCOMIP, FUCOMIP:       pop ST
```

**Description:**

These instructions all compare the stack top to the source, which is a floating-point stack register, and sets the integer flags according to the results.

**Example:**

FCOMI is equivalent to, but faster than, the floating-point compare sequence on previous processors:

i387:

```
            FCOM ST0,ST1
            FNSTSW AX
            SAHF
            Jcc label
            ...
      label:
```

FCOMI:

```
            FCOMI ST0, ST1
            Jcc label
            ...
      label:
```

**FPU Flags Affected:**

In normal operation, none.

Floating-point exceptions will set FSW.C1 to indicate whether a stack under/overflow exception occurred.

**NOTE**

The encodings chosen for FCMOV$_{CC}$ produce an illegal instruction exception on the Intel486 and Pentium processors, but are treated like FNOP by the Intel387™ math coprocessor.

**Numeric Exceptions:**

D, I, IS

**Protected Mode Exceptions:**
#NM if either EM or TS in CR0 is set.

**Real Address Mode Exceptions:**
Interrupt 7 if either EM or TS in CR0 is set.

**Virtual 8086 Mode Exceptions:**
Interrupt 7 if either EM or TS in CR0 is set.

**NOTES:**

- For FCOMI or FUCOMIP, if either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid operand exception is raised and the flags are set to "unordered".
- For FUCOMI or FUMCOMPI, if either operand is an SNaN or is in an undefined format, or if a stack fault occurs, the invalid operand exception is raised, and the flags are set to "unordered".
- For FUCOMI or FUMCOMPI, if either operand is a QNaN , the flags are set to "unordered". Unlike the ordinary FCOMI and FCOMIP instructions, the unordered compare instructions do not raise the invalid-operand exception on account of a QNaN operand.
- The sign of zero is ignored, so that -0.0 = +0.0.

#### 4.2.2.1.4 RDPMC - Read Performance Monitoring Counters

**Syntax:**

```
RDPMC
```

**Encoding: 0F 33**

| | |
|---|---|
| 0000 1111 | 0011 0011 |

**Operation:**

```
IF CPL <> 0 AND CR4.PCE[3] == 0 THEN #GP(0)
IF ECX = 0 THEN EDX:EAX := PerfCnt0
:: ECX = 1 THEN EDX:EAX := PerfCnt1
ELSE #GP(0)
END IF
```

**Description:**

The Pentium Pro processor contains two performance counters, each of 40 bits, in the Model Specific Register space, which can be accessed using the privileged RDMSR and WRMSR instructions.

The RDPMC instruction provides a standard means of accessing the performance counters, if they are present in an implementation.. Furthermore, the RDPMC instruction permits the performance counters to be accessible from user mode (subject to OS control via the CR4.PCE bit). This permits user applications to be instrumented at a fine granularity,

[3]CR4.PCE = bit 8 of PCE

e.g. procedure call granularity, with far less overhead than can be obtained by making a system call to read the performance counters via RDMSR.

The number of performance counters, the events counted by the performance counters, and the means of programming the performance counters to select particular events is implementation specific.

Note that there is no CPUID feature bit for RDPMC. This is absent because RDPMC is considered to be inherently model specific.

*Serialization*: RDPMC does not imply serialization. It does not imply, for example, that all events caused by preceding instructions have completed, nor that no events caused by succeeding instructions have begun. For example:

```
MOV ECX, 0
RDPMC1
MOV EBX,EAX
MOV EBP,EDX
MOV EAX, mem
RDPMC2
```

If performance counter 0 is set to count the number of instructions fully executed (in Pentium Pro processor terminology, the number of instructions retired) then the difference between the first and second values returned by RDPMC may well be 5, as might be expected above - but it is also possible that the difference is 6 or more, or that the difference is negative, because the first RDPMC may execute after the second. (This does not occur in the Pentium Pro processor implementation, however.)

If a truly reliable RDPMC value is desired, software can employ a serializing instruction such as CPUID before and/or after RDPMC.

*16 bit code*: RDPMC executes in 16 bit modes, including VM86 mode, but will provide a full 32 bit result. Moreover, it will always use the full ECX to determine which performance counter to access.

**Flags Affected:**

None.

**Exceptions:**

GP(0), as indicated above

## 5.0. COMPILER SWITCHES RECOMMENDATION

The following section summarizes the compiler switch recommendations for Intel Architecture Compilers. The default for compilers should be a blended switch that optimizes for the family of processors. Switches specific to each processor should be offered as an alternative for application programmers.

### 5.1. Default (blended code)

Generates blended code. Code compiled with this switch will execute on all Intel Architecture processors (Intel386, Intel486, Pentium and Pentium Pro). This switch is intended for code which will possibly run on more than one processor. There should be no partial stalls not covered by idioms.

## 5.2. Processor Specific Switches

**Target Processor -Intel486™ Processor**

Generates the best Intel486 processor code. Code will run on all 32-bit Intel Architecture processors. This is intended for code which will run only on the Intel486 processor.

**Target Processor - Pentium® Processor**

Generates best the Pentium processor code. Code will run on all 32-bit Intel Architecture processors. This is intended for code which will run only on the Pentium processor.

**Target Processor - Pentium® Pro Processor**

Generates the best Pentium Pro processor code. Code will run on all 32-bit Intel Architecture processors. This is intended for code which will run only on the Pentium Pro processor. There should be no partial stalls.

## 5.3. Other Switches

**Pentium® Pro Processor New Instructions**

This will use the new Pentium Pro processor specific instructions: cmov, fcmov, fcomi. This is independent of the Pentium Pro processor specific switch . If the target processor Pentium Pro processor switch is also specified, the 'if to cmov' optimization will be done depending on Pentium Pro processor style cost analysis.

**Optimize for Small Code Size**

This switch optimizes for small code size. Execution speed will be sacrificed when necessary. An example is to use pushes rather than stores. This is intended for programs with high instruction cache miss rates. This switch also turns off code alignment, regardless of target processor.

## 6.0. SUMMARY

The following tables summarize the micro architecture differences among Intel386, Intel486 and Pentium and Pentium Pro processors and the corresponding code generation consideration. It is possible to derive a set of code generation strategies that provide the optimal performance across the various members of the Intel386 processor family except for the use of FXCH to maximize the Pentium processor floating-point throughput which can be implemented under a user-directed option.

**Table 5. Intel Microprocessor Architecture Differences**

| | Intel386™ Processor | Intel486™ Processor | Pentium® Processor | Pentium Pro Processor |
|---|---|---|---|---|
| **Cache** | None | 8K Combined | 8K Code, 8K Data | 8K Code, 8K Data |
| **Prefetch** | 4x4b filled by external memory access | 2x6b shared bus to cache | 4x32b private bus to cache | 4x32 private bus to cache |
| **Decoder** | 3 deep decoded FIFO | part of core pipeline | part of core pipeline | part of core pipeline |
| **Core** | some instruction overlap | 5 stages pipeline | 5 stages pipeline & superscalar | 12 stages pipeline & Dynamic Execution |
| **Math** | Co-processor | On-Chip | On-Chip & pipelined | On-Chip and pipelined |

Table 6. Code Generation Recommendation By Architecture

| **Processor Characteristics** | **Optimizations** | **Intel386™ Processor** | **Intel486™ Processor** | **Pentium® Processor** | **Pentium Pro Processor** |
|---|---|---|---|---|---|
| Cache | Interleave mem with non-mem | don't care | Interleave if 4 consecutive | don't care | don't care |
| Prefetcher | Alignment | 0-MOD-4 | 0-MOD-16 | don't care | 0-Mod-16 |
| Pipelined execution core | Base Vs index | don't care | use base | don't care | don't care |
| | Avoid AGI | don't care | next instr | next 3 instr | don't care |
| | Instruction selection | 1 clk penalty | short instr | short instr | short instr |
| Superscalar | Pairing | don't care | don't care | pair | don't care |
| pipelined FPU with FXCH | more scheduling | 18 clk penalty | 4 clk penalty | schedule | schedule |

Recommendations for Blended:

1. Interleave mem with non-mem: do nothing
2. Code alignment: 0-mod-16 on loop
3. Base Vs index: use base
4. Avoid AGI: next 3 instructions
5. Instruction selection: short instructions sequence
6. Pairing: pair
7. FP scheduling: avoid FXCH

## APPENDIX A. INTEGER PAIRING TABLE

The following abbreviations are used in the Pairing column of the integer table in this appendix:

NP— Not pairable, executes in U-pipe

PU— Pairable if issued to U-pipe

PV— Pairable if issued to V-pipe

UV— Pairable in either pipe

The I/O instructions are not pairable.

**Table 7. Integer Instruction Pairing**

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| AAA — ASCII Adjust after Addition | | NP |
| AAD — ASCII Adjust AX before Division | | NP |
| AAM — ASCII Adjust AX after Multiply | | NP |
| AAS — ASCII Adjust AL after Subtraction | | NP |
| ADC — ADD with Carry | | PU |
| ADD — Add | | UV |
| AND — Logical AND | | UV |
| ARPL — Adjust RPL Field of Selector | | NP |
| BOUND — Check Array Against Bounds | | NP |
| BSF — Bit Scan Forward | | NP |
| BSR — Bit Scan Reverse | | NP |
| BSWAP — Byte Swap | | NP |
| BT — Bit Test | | NP |
| BTC — Bit Test and Complement | | NP |
| BTR — Bit Test and Reset | | NP |
| BTS — Bit Test and Set | | NP |
| CALL — Call Procedure (in same segment) | | |
| direct | 1110 1000 : full displacement | PV |
| register indirect | 1111 1111 : 11 010 reg | NP |
| memory indirect | 1111 1111 : mod 010 r/m | NP |
| CALL — Call Procedure (in other segment) | | NP |
| CBW — Convert Byte to Word<br>CWDE — Convert Word to Doubleword | | NP |
| CLC — Clear Carry Flag | | NP |
| CLD — Clear Direction Flag | | NP |

Table 7. Integer Instruction Pairing

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| CLI — Clear Interrupt Flag | | NP |
| CLTS — Clear Task-Switched Flag in CR0 | | NP |
| CMC — Complement Carry Flag | | NP |
| CMP — Compare Two Operands | | UV |
| CMPS/CMPSB/CMPSW/CMPSD — Compare String Operands | | NP |
| CMPXCHG — Compare and Exchange | | NP |
| CMPXCHG8B — Compare and Exchange 8 Bytes | | NP |
| CWD — Convert Word to Dword<br>CDQ — Convert Dword to Qword | | NP |
| DAA — Decimal Adjust AL after Addition | | NP |
| DAS — Decimal Adjust AL after Subtraction | | NP |
| DEC — Decrement by 1 | | UV |
| DIV — Unsigned Divide | | NP |
| ENTER — Make Stack Frame for Procedure Parameters | | NP |
| HLT — Halt | | |
| IDIV — Signed Divide | | NP |
| IMUL — Signed Multiply | | NP |
| INC — Increment by 1 | | UV |
| INT n — Interrupt Type n | | NP |
| INT — Single-Step Interrupt 3 | | NP |
| INTO — Interrupt 4 on Overflow | | NP |
| INVD — Invalidate Cache | | NP |
| INVLPG — Invalidate TLB Entry | | NP |
| IRET/IRETD — Interrupt Return | | NP |
| Jcc — Jump if Condition is Met | | PV |
| JCXZ/JECXZ — Jump on CX/ECX Zero | | NP |
| JMP — Unconditional Jump (to same segment) | | |
| short | 1110 1011 : 8-bit displacement | PV |
| direct | 1110 1001 : full displacement | PV |
| register indirect | 1111 1111 : 11 100 reg | NP |
| memory indirect | 1111 1111 : mod 100 r/m | NP |
| JMP — Unconditional Jump (to other segment) | | NP |

**Table 7. Integer Instruction Pairing**

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| LAHF — Load Flags into AH Register | | NP |
| LAR — Load Access Rights Byte | | NP |
| LDS — Load Pointer to DS | | NP |
| LEA — Load Effective Address | | UV |
| LEAVE — High Level Procedure Exit | | NP |
| LES — Load Pointer to ES | | NP |
| LFS — Load Pointer to FS | | NP |
| LGDT — Load Global Descriptor Table Register | | NP |
| LGS — Load Pointer to GS | | NP |
| LIDT — Load Interrupt Descriptor Table Register | | NP |
| LLDT — Load Local Descriptor Table Register | | NP |
| LMSW — Load Machine Status Word | | NP |
| LOCK — Assert LOCK# Signal Prefix | | |
| LODS/LODSB/LODSW/LODSD — Load String Operand | | NP |
| LOOP — Loop Count | | NP |
| LOOPZ/LOOPE — Loop Count while Zero/Equal | | NP |
| LOOPNZ/LOOPNE — Loop Count while not Zero/Equal | | NP |
| LSL — Load Segment Limit | | NP |
| LSS — Load Pointer to SS | 0000 1111 : 1011 0010 : mod reg r/m | NP |
| LTR — Load Task Register | | NP |
| MOV — Move Data | | UV |
| MOV — Move to/from Control Registers | | NP |
| MOV — Move to/from Debug Registers | | NP |
| MOV — Move to/from Segment Registers | | NP |
| MOVS/MOVSB/MOVSW/MOVSD — Move Data from String to String | | NP |
| MOVSX — Move with Sign-Extend | | NP |
| MOVZX — Move with Zero-Extend | | NP |
| MUL — Unsigned Multiplication of AL, AX or EAX | | NP |
| NEG — Two's Complement Negation | | NP |
| NOP — No Operation | 1001 0000 | UV |
| NOT — One's Complement Negation | | NP |

Table 7. Integer Instruction Pairing

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| OR — Logical Inclusive OR | | UV |
| POP — Pop a Word from the Stack | | |
| reg | 1000 1111 : 11 000 reg | UV |
| or | 0101 1 reg | UV |
| memory | 1000 1111 : mod 000 r/m | NP |
| POP — Pop a Segment Register from the Stack | | NP |
| POPA/POPAD — Pop All General Registers | | NP |
| POPF/POPFD — Pop Stack into FLAGS or EFLAGS Register | | NP |
| PUSH — Push Operand onto the Stack | | |
| reg | 1111 1111 : 11 110 reg | UV |
| or | 0101 0 reg | UV |
| memory | 1111 1111 : mod 110 r/m | NP |
| immediate | 0110 10s0 : immediate data | UV |
| PUSH — Push Segment Register onto the Stack | | NP |
| PUSHA/PUSHAD — Push All General Registers | | NP |
| PUSHF/PUSHFD — Push Flags Register onto the Stack | | NP |
| RCL — Rotate thru Carry Left | | |
| reg by 1 | 1101 000w : 11 010 reg | PU |
| memory by 1 | 1101 000w : mod 010 r/m | PU |
| reg by CL | 1101 001w : 11 010 reg | NP |
| memory by CL | 1101 001w : mod 010 r/m | NP |
| reg by immediate count | 1100 000w : 11 010 reg : imm8 data | PU |
| memory by immediate count | 1100 000w : mod 010 r/m : imm8 data | PU |
| RCR — Rotate thru Carry Right | | |
| reg by 1 | 1101 000w : 11 011 reg | PU |
| memory by 1 | 1101 000w : mod 011 r/m | PU |
| reg by CL | 1101 001w : 11 011 reg | NP |
| memory by CL | 1101 001w : mod 011 r/m | NP |
| reg by immediate count | 1100 000w : 11 011 reg : imm8 data | PU |
| memory by immediate count | 1100 000w : mod 011 r/m : imm8 data | PU |

**Table 7. Integer Instruction Pairing**

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| RDMSR — Read from Model-Specific Register | | NP |
| REP LODS — Load String | | NP |
| REP MOVS — Move String | | NP |
| REP STOS — Store String | | NP |
| REPE CMPS — Compare String (Find Non-Match) | | NP |
| REPE SCAS — Scan String (Find Non-AL/AX/EAX) | | NP |
| REPNE CMPS — Compare String (Find Match) | | NP |
| REPNE SCAS — Scan String (Find AL/AX/EAX) | | NP |
| RET — Return from Procedure (to same segment) | | NP |
| RET — Return from Procedure (to other segment) | | NP |
| ROL — Rotate (not thru Carry) Left | | |
| reg by 1 | 1101 000w : 11 000 reg | PU |
| memory by 1 | 1101 000w : mod 000 r/m | PU |
| reg by CL | 1101 001w : 11 000 reg | NP |
| memory by CL | 1101 001w : mod 000 r/m | NP |
| reg by immediate count | 1100 000w : 11 000 reg : imm8 data | PU |
| memory by immediate count | 1100 000w : mod 000 r/m : imm8 data | PU |
| ROR — Rotate (not thru Carry) Right | | |
| reg by 1 | 1101 000w : 11 001 reg | PU |
| memory by 1 | 1101 000w : mod 001 r/m | PU |
| reg by CL | 1101 001w : 11 001 reg | NP |
| memory by CL | 1101 001w : mod 001 r/m | NP |
| reg by immediate count | 1100 000w : 11 001 reg : imm8 data | PU |
| memory by immediate count | 1100 000w : mod 001 r/m : imm8 data | PU |
| RSM — Resume from System Management Mode | | NP |
| SAHF — Store AH into Flags | | NP |
| SAL — Shift Arithmetic Left same instruction as SHL | | |
| SAR — Shift Arithmetic Right | | |
| reg by 1 | 1101 000w : 11 111 reg | PU |
| memory by 1 | 1101 000w : mod 111 r/m | PU |
| reg by CL | 1101 001w : 11 111 reg | NP |

Table 7.  Integer Instruction Pairing

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| memory by CL | 1101 001w : mod 111 r/m | NP |
| reg by immediate count | 1100 000w : 11 111 reg : imm8 data | PU |
| memory by immediate count | 1100 000w : mod 111 r/m : imm8 data | PU |
| SBB — Integer Subtraction with Borrow | | PU |
| SCAS/SCASB/SCASW/SCASD — Scan String | | NP |
| SETcc — Byte Set on Condition | | NP |
| SGDT — Store Global Descriptor Table Register | | NP |
| SHL — Shift Left | | |
| reg by 1 | 1101 000w : 11 100 reg | PU |
| memory by 1 | 1101 000w : mod 100 r/m | PU |
| reg by CL | 1101 001w : 11 100 reg | NP |
| memory by CL | 1101 001w : mod 100 r/m | NP |
| reg by immediate count | 1100 000w : 11 100 reg : imm8 data | PU |
| memory by immediate count | 1100 000w : mod 100 r/m : imm8 data | PU |
| SHLD — Double Precision Shift Left | | |
| register by immediate count | 0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8 | NP |
| memory by immediate count | 0000 1111 : 1010 0100 : mod reg r/m : imm8 | NP |
| register by CL | 0000 1111 : 1010 0101 : 11 reg2 reg1 | NP |
| memory by CL | 0000 1111 : 1010 0101 : mod reg r/m | NP |
| SHR — Shift Right | | |
| reg by 1 | 1101 000w : 11 101 reg | PU |
| memory by 1 | 1101 000w : mod 101 r/m | PU |
| reg by CL | 1101 001w : 11 101 reg | NP |
| memory by CL | 1101 001w : mod 101 r/m | NP |
| reg by immediate count | 1100 000w : 11 101 reg : imm8 data | PU |
| memory by immediate count | 1100 000w : mod 101 r/m : imm8 data | PU |
| SHRD — Double Precision Shift Right | | |
| register by immediate count | 0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8 | NP |

Table 7.  Integer Instruction Pairing

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| memory by immediate count | 0000 1111 : 1010 1100 : mod reg r/m : imm8 | NP |
| register by CL | 0000 1111 : 1010 1101 : 11 reg2 reg1 | NP |
| memory by CL | 0000 1111 : 1010 1101 : mod reg r/m | NP |
| SIDT — Store Interrupt Descriptor Table Register | | NP |
| SLDT — Store Local Descriptor Table Register | | NP |
| SMSW — Store Machine Status Word | | NP |
| STC — Set Carry Flag | | NP |
| STD — Set Direction Flag | | NP |
| STI — Set Interrupt Flag | | |
| STOS/STOSB/STOSW/STOSD — Store String Data | | NP |
| STR — Store Task Register | | NP |
| SUB — Integer Subtraction | | UV |
| TEST — Logical Compare | | |
| reg1 and reg2 | 1000 010w : 11 reg1 reg2 | UV |
| memory and register | 1000 010w : mod reg r/m | UV |
| immediate and register | 1111 011w : 11 000 reg : immediate data | NP |
| immediate and accumulator | 1010 100w : immediate data | UV |
| immediate and memory | 1111 011w : mod 000 r/m : immediate data | NP |
| VERR — Verify a Segment for Reading | | NP |
| VERW — Verify a Segment for Writing | | NP |
| WAIT — Wait | 1001 1011 | NP |
| WBINVD — Write-Back and Invalidate Data Cache | | NP |
| WRMSR — Write to Model-Specific Register | | NP |
| XADD — Exchange and Add | | NP |
| XCHG — Exchange Register/Memory with Register | | NP |
| XLAT/XLATB — Table Look-up Translation | | NP |
| XOR — Logical Exclusive OR | | UV |

## APPENDIX B. FLOATING-POINT PAIRING TABLE

In the floating-point table in this appendix:

FX— Pairs with FXCH

NP— No pairing.

**Table 8. Floating-Point Instruction Pairings**

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| F2XM1 — Compute $2^{ST(0)}$ — 1 | | NP |
| FABS — Absolute Value | | FX |
| FADD — Add | | FX |
| FADDP — Add and Pop | | FX |
| FBLD — Load Binary Coded Decimal | | NP |
| FBSTP — Store Binary Coded Decimal and Pop | | NP |
| FCHS — Change Sign | | FX |
| FCLEX — Clear Exceptions | | NP |
| FCOM — Compare Real | | FX |
| FCOMP — Compare Real and Pop | | FX |
| FCOMPP — Compare Real and Pop Twice | | |
| FCOS — Cosine of ST(0) | | NP |
| FDECSTP — Decrement Stack-Top Pointer | | NP |
| FDIV — Divide | | FX |
| FDIVP — Divide and Pop | | FX |
| FDIVR — Reverse Divide | | FX |
| FDIVRP — Reverse Divide and Pop | | FX |
| FFREE — Free ST(i) Register | | NP |
| FIADD — Add Integer | | NP |
| FICOM — Compare Integer | | NP |
| FICOMP — Compare Integer and Pop | | NP |
| FIDIV | | NP |
| FIDIVR | | NP |
| FILD — Load Integer | | NP |
| FIMUL | | NP |
| FINCSTP — Increment Stack Pointer | | NP |
| FINIT — Initialize Floating-Point Unit | | NP |
| FIST — Store Integer | | NP |

**Table 8. Floating-Point Instruction Pairings**

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| FISTP — Store Integer and Pop | | NP |
| FISUB | | NP |
| FISUBR | | NP |
| FLD — Load Real | | |
| 32-bit memory | 11011 001 : mod 000 r/m | FX |
| 64-bit memory | 11011 101 : mod 000 r/m | FX |
| 80-bit memory | 11011 011 : mod 101 r/m | NP |
| ST(i) | 11011 001 : 11 000 ST(i) | FX |
| FLD1 — Load +1.0 into ST(0) | | NP |
| FLDCW — Load Control Word | | NP |
| FLDENV — Load FPU Environment | | NP |
| FLDL2E — Load $\log_2$(e) into ST(0) | | NP |
| FLDL2T — Load $\log_2$(10) into ST(0) | | NP |
| FLDLG2 — Load $\log_{10}$(2) into ST(0) | | NP |
| FLDLN2 — Load $\log_e$(2) into ST(0) | | NP |
| FLDPI — Load p into ST(0) | | NP |
| FLDZ — Load +0.0 into ST(0) | | NP |
| FMUL — Multiply | | FX |
| FMULP — Multiply | | FX |
| FNOP — No Operation | | NP |
| FPATAN — Partial Arctangent | | NP |
| FPREM — Partial Remainder | | NP |
| FPREM1 — Partial Remainder (IEEE) | | NP |
| FPTAN — Partial Tangent | | NP |
| FRNDINT — Round to Integer | | |
| FRSTOR — Restore FPU State | | NP |
| FSAVE — Store FPU State | | NP |
| FSCALE — Scale | | NP |
| FSIN — Sine | | NP |
| FSINCOS — Sine and Cosine | | NP |
| FSQRT — Square Root | | NP |
| FST — Store Real | | NP |

Table 8.  Floating-Point Instruction Pairings

| INSTRUCTION | FORMAT | Pairing |
|---|---|---|
| FSTCW — Store Control Word | | NP |
| FSTENV — Store FPU Environment | | NP |
| FSTP — Store Real and Pop | | NP |
| FSTSW — Store Status Word into AX | | NP |
| FSTSW — Store Status Word into Memory | | NP |
| FSUB — Subtract | | FX |
| FSUBP — Subtract and Pop | | FX |
| FSUBR — Reverse Subtract | | FX |
| FSUBRP — Reverse Subtract and Pop | | FX |
| FTST — Test | | FX |
| FUCOM — Unordered Compare Real) | | FX |
| FUCOMP — Unordered Compare and Pop | | FX |
| FUCOMPP — Unordered Compare and Pop Twice | | FX |
| FXAM — Examine | | NP |
| FXCH — Exchange ST(0) and ST(i) | | |
| FXTRACT — Extract Exponent and Significant | | NP |
| FYL2X — ST(1) ´ $\log_2$(ST(0)) | | NP |
| FYL2XP1 — ST(1) ´ $\log_2$(ST(0) + 1.0) | | NP |
| FWAIT — Wait until FPU Ready | | |

# APPENDIX C. THE PENTIUM® PRO PROCESSOR SPECIFIC GUIDELINES

## C.1. COMPILER WRITER'S RULES

This section describes the very small list of optimizations that are necessary to get most of the possible performance optimizations for the Pentium Pro processor.

Generate good basic code to begin with. See [3].

Consider optimizations that improve branch predictability:

- Avoid unnecessary branches
- Loops. Arrange code so that the most frequently executed path of backward branches is taken.
- Fall-through. Arrange code so that the most frequently executed path of forward branches is fall-through.

Arrange code to minimize instruction cache misses and optimize prefetch.

Avoid generating code that exercises the Pentium Pro processor's major stall conditions:

1. Partial stalls
    - For example, do not write AL and then read EAX .
    - In FP compares, use

      ```
      FSTSW AX;
      SAHF
      ```

      instead of:

      ```
      FSTSW AX;
      AND EAX, mask
      ```

    - Avoid testing flags after multibit shifts
2. Avoid misaligned memory accesses

Optimize for instruction parallelism:

1. Flatten calculation graphs as much as possible without introducing new memory traffic
2. Use FXCH to interleave independent FP computations.

Optimize memory traffic

1. Remove unnecessary memory accesses. For example, accesses to FP locations in cache or memory are not free
2. Assume that stores are roughly 2x more expensive than loads in calculating spills
3. Calculate store addresses as early as possible to avoid stores blocking loads

Generate simple REG, REG and REG, MEM instructions that match the decoder template. Avoid MEM, REG instructions that read-modify-write memory

Take advantage of hardware register renaming: do not worry about false Write-after-Write and Write-after-Read dependencies, which are handled by the hardware

## C.2. PROGRAMMER'S RULES

The assembly language programmer can use all of the compiler optimizations. Even the high level language programmer can write faster code when cognizant of machine limitations.

Some optimizations are easier for the programmer to perform than the compiler writer. These include:

· Indirect branches . Indirect branches often have poor branch prediction, if they frequently go to more than one location. Simple techniques can reduce this non-predictability.

· Data structure rearrangement to improve cache locality . Compilers are often limited as to the possible optimizations in this area because of language standards.

## C.3. CODE GENERATION RULES

There are 16 code generation rules:

1. Perform optimizations to improve branch prediction, as these are the most important class of optimizations for the Pentium Pro processor.
2. Arrange code so that forward conditional branches are usually not taken, and backward conditional branches are usually taken.
3. Always pair CALLs and RETurns.
4. Avoid self-modifying code.
5. Avoid placing data in the code segment.
6. Avoid instructions that contain three or more uops. If possible, use instructions that require one uop.
7. Avoid instructions that contain both an immediate and a displacement.
8. Use the REG,MEM forms whenever possible to reduce register pressure.
9. Avoid reading a large register (EAX) after writing a small register (AL), which is contained in the large register, as it produces a stall.
10. Avoid reading the middle of a register (AH) after writing the register as a large register (EAX), as it produces a stall.
11. Avoid using two 8-bit loads to produce a 16-bit load.
12. Cleanse partial registers before calling callee-save procedures.
13. Resolve blocking conditions, such as store addresses, as far as possible away from loads they may block.
14. Use integer multiplication instead of shift-and-add operations..
15. Calculate store addresses as soon as possible.
16. In general, an N-byte quantity which is directly supported by the processor (8-bit bytes, 16-bit words, 32-bit double words, and 32-bit, 64-bit, and 80-bit floating-point numbers) should be aligned on the next highest power-of-two boundary. Avoid misaligned data. Align 8-bit data on any boundary, 16-bit data on any even boundary, 32-bit data on any boundary which is a multiple of four, and 64-bit data on any boundary which is a multiple of 8. Align 80-bit data on a 128-bit boundary, i.e. any boundary which is a multiple of 16 bytes.

# APPENDIX D. PENTIUM® PRO PROCESSOR INSTRUCTION TIMING INFORMATION

## TIMING INSTRUCTION INDEX

The following timing tables are provided to allow you to estimate the performance of a specific piece of code on a Pentium Pro processor. To understand the performance of a specific set of instructions you need to understand how the instructions flow through the Pentium Pro processor pipelines. Note that the Pentium Pro processor has more than one pipeline.

To understand the timing of a piece of code on the Pentium Pro processor, you need the following:

- An understanding of the Pentium Pro processor pipeline including any special constraints or bottlenecks in each of the stages of the pipeline
- The uop breakdown for each macro instruction, a specification of which pipelines each uop flows through and the number of cycles each uop takes in the pipelines. All uops flow through the in-order issue and retirement pipelines, however uops flow through different execution pipes in the out-of-order core.
- The data flow dependencies of the uops that make up each macro-instruction

## D.1. Overall Pentium® Pro Processor Pipeline

The following figure illustrates the Pentium Pro processor architecture.

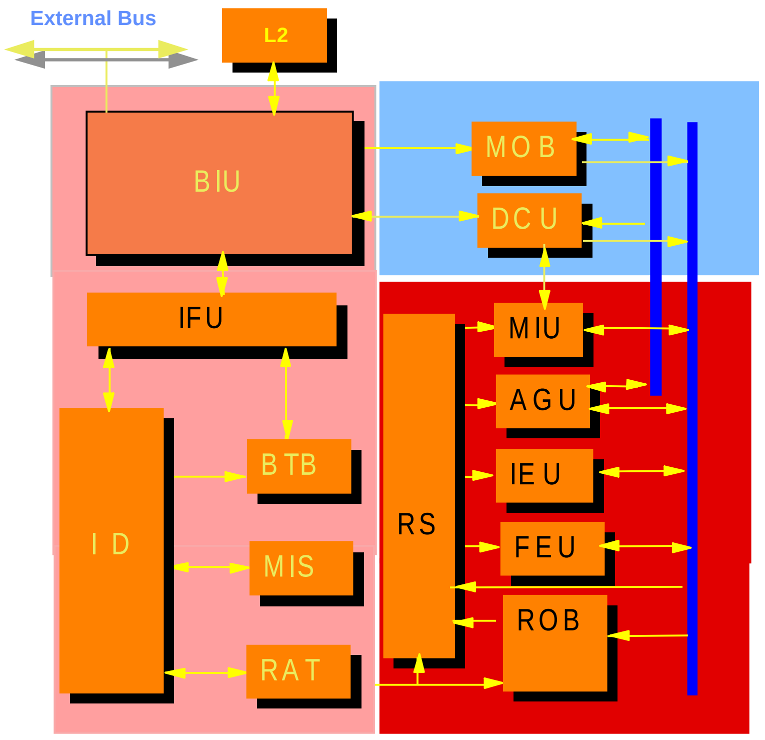
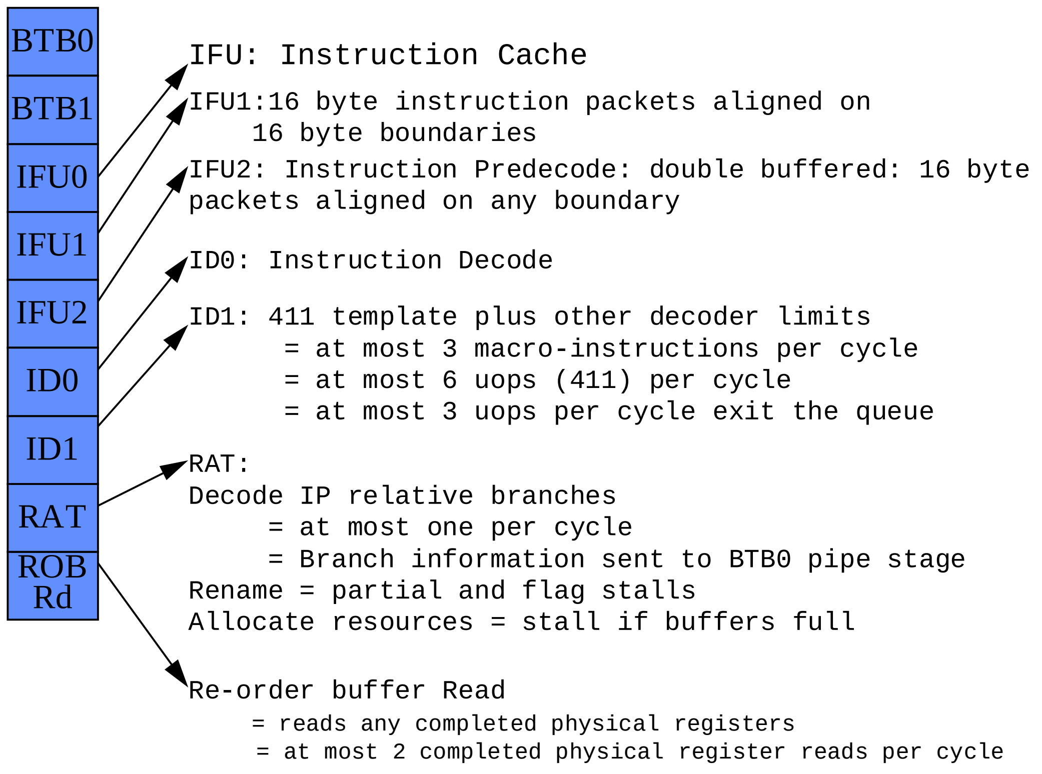


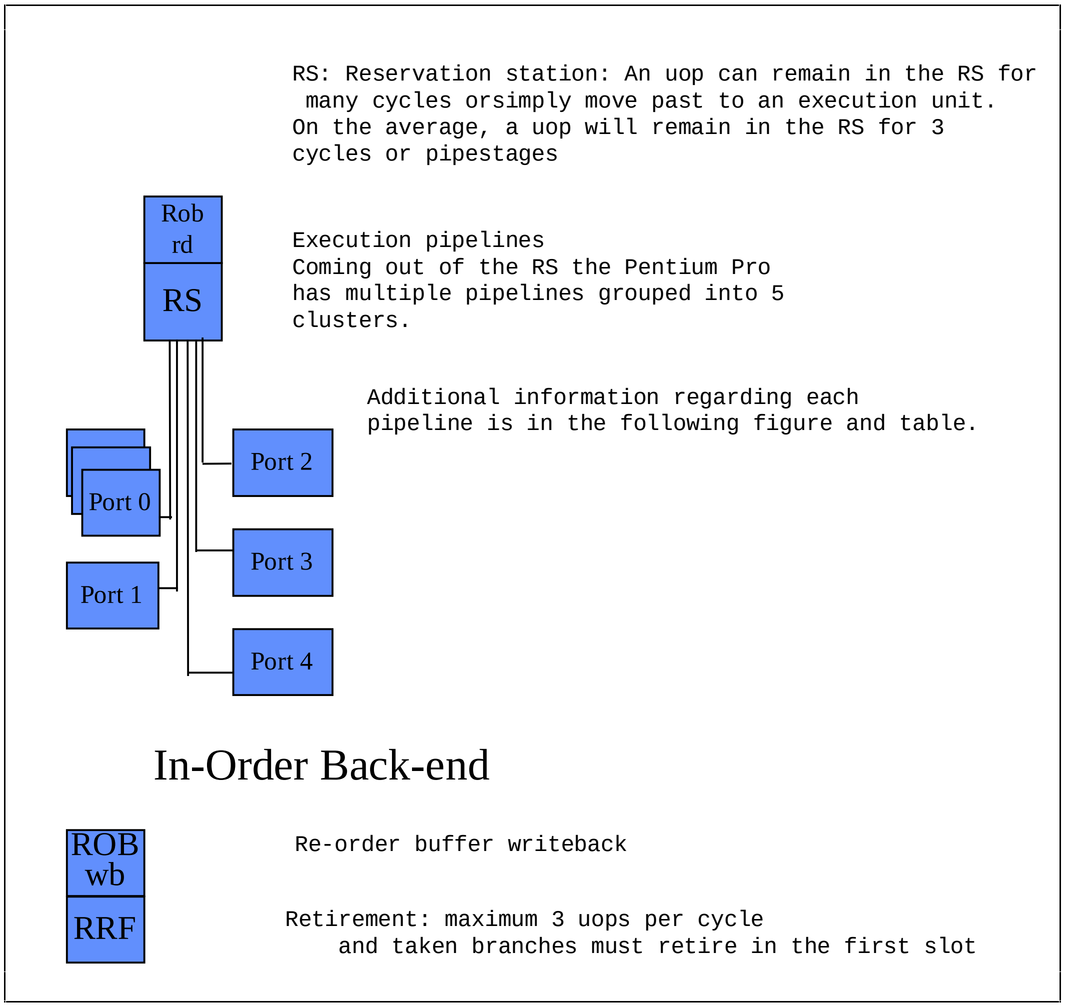
**Figure 12. Pentium® Pro Architecture**

The Pentium Pro processor pipeline is comprised of three parts, the In-Order Issue Front-end, the Out-of-Order Core and the In-Order Retirement unit.



```
IFU: Instruction Cache
IFU1:16 byte instruction packets aligned on
    16 byte boundaries
IFU2: Instruction Predecode: double buffered: 16 byte
packets aligned on any boundary

ID0: Instruction Decode

ID1: 411 template plus other decoder limits
      = at most 3 macro-instructions per cycle
      = at most 6 uops (411) per cycle
      = at most 3 uops per cycle exit the queue

RAT:
Decode IP relative branches
     = at most one per cycle
     = Branch information sent to BTB0 pipe stage
Rename = partial and flag stalls
Allocate resources = stall if buffers full

Re-order buffer Read
    = reads any completed physical registers
    = at most 2 completed physical register reads per cycle
```

**Figure 13. In-Order Issues Front-End**



RS: Reservation station: An uop can remain in the RS for many cycles orsimply move past to an execution unit. On the average, a uop will remain in the RS for 3 cycles or pipestages

Execution pipelines
Coming out of the RS the Pentium Pro has multiple pipelines grouped into 5 clusters.

Additional information regarding each pipeline is in the following figure and table.

In-Order Back-end

Re-order buffer writeback

Retirement: maximum 3 uops per cycle
and taken branches must retire in the first slot

**Figure 14. Out-of-Order Core**

**Table 9. Pentium® Pro Processor Execution Unit Pipelines**

| Port | Execution Units | Latency/Thruput |
|---|---|---|
| 0 | Integer ALU<br>LEAs<br>Shifts<br>FADD<br>FMUL<br>IntegerMuls<br>FDIV | Latency 1, Thruput 1/cycle<br>Latency 1, Thruput 1/cycle<br>Latency 1, Thruput 1/cycle<br>Latency 3, Thruput 1/cycle<br>Latency 5, Thruput 1/2cycle[1]<br>Latency 4, Thruput 1/cycle[2]<br>Latency long and data dep., Thruput non-pipelined |
| 1 | Integer ALU | Latency 1, Thruput 1/cycle |
| 2 | Loads | Latency 3 on a cache hit, Thruput 1/cycle[4] |
| 3 | Store Address | Latency 3 (not applicable) Thruput 1/cycle[3] |
| 4 | Store Data | Latency 1 (not applicable) Thruput 1/cycle |

**Notes:**

1. The FMUL unit cannot accept a second FMUL the cycle after it has accepted the first. This is NOT the same as only being able to do FMULs on even cycles.
2. FMUL is pipelined one every 2 cycles, one way of thinking about it is to imagine that the Pentium Pro processor only has a 32x32->32 multiply pipelined.
3. Stores don't really have a latency that is important from a dataflow perspective- although they do have a latency that matters with respect to determining when they can retire, and be actually done. They also have a different latency with respect to load forwarding. For example, if the store address and store data of a particular address, 100 dispatch in cycle10, a load (of the same size and shape) to the same address 100 can dispatch in the same cycle 10, and not be stalled.
4. A load and store to the same address can dispatch in the same cycle.

## D.2. Uop Breakdown

In addition to the above information, you need to know the breakdown from macro-instruction to uops, on which execution unit pipelines each uop can execute, and what the latencies and dependencies are between the uops for each instruction.

For each macro-instruction in your piece of code you need to create a dataflow graph with “terminals” that you can "plug in" to the dataflow graphs of other macroinstructions, so that you can figure out the dataflow graph of the entire piece of code.

## D.3. Uop Pseudocode for Macroinstructions

Following is a description of the timing tables.

Example timing:

```
HDR: "IMUL rm32"                    :  1111011.1    11.101.sss ------
```

-
  ```
  FLOW  tmp0 := int_mul.port0.latency4(EAX, REG_sss)
        EAX := move.port01.latency1(tmp0);
        EDX := port0.latency1(Tmp0, const);
  ```

Where:

"IMUL rm32"

Is the Intel Macro Instruction:

On the Pentium Pro processor internally, we decode the instruction by generating:

- "rm32" - register is extracted from the memory part, bits 0-2, of modrm byte (for which modrm[67] = 11).
- 
- "r32" - register is extracted from the non-memory part, bits 3-5, of the modrm byte.
- 
-

1111011.1 11.101.sss Is the binary specification of the instruction.

- bits must be set.
- bits must be clear.
- ".", periods, are shown just to clarify the representation.
- "sss" or "ddd" are three bit field where any bit pattern is possible
-

These fields are inserted into the pseudocode where you see a construct like REG_sss.

```
tmp0 := int_mul.port0.latency4 (EAX, REG_sss);
```

This is an integer multiply uop. Its input operands, which you need to be known for the purposes of characterizing dataflow dependencies (as well as things like partial stalls), are EAX, as well as the register which is specified by the "sss" field of the binary opcode - in this case bits 0-2 of the modrm byte.

This uop writes to a microcode temporary register, refered to here as "tmp0". The temporary registers are specified as "tmp1", "tmp2", etc.

As indicated in the pseudocode and detailed in Table 9 above, integer multiplies execute only on RS port 0 and have a latency of 4 cycles. The second uop executes as a part of its internal function a mov from the temporary register into the EAX register as required.

```
EAX := move.port01.latency1 (tmp0);
```

The third uop does internal housekeeping..

What you need to know about the second uop is:

- it executes on either port 0 or 1
- it has a latency of 1
- it takes as input the output of the previous, integer multiplication, uop
- it writes EAX.

By looking at the first and second uops, you can deduce that a valid EAX can be obtained with a latency of 5 cycles through this particular IMUL macro-instruction - as compared to, for example, IMUL opcode 69, or IMUL opcode 0F AF, which only have a latency of 4 cycles (because they don't have these extra housecleaning uops).

```
EDX := port1.latency1.thruput1(tmp0);
```

Similar to the previous uop, except that this uop can only execute on port 0.

Knowing which port uops execute on can make a big difference. One of my best tuning examples so far was on a toy program (that a customer thought was important), whose speed I increased by 50% by changing register-to-register MOV's into Loads - because the load port, port 2, was underutilized, while ports 0 and 1 were saturated.

The superscalar, out-of-order nature of the Pentium Pro processor makes it difficult to accurately predict the execution times of instructions. Therefore, this document includes a description of each instruction which includes the mneumonic, a binary representation of the opcode, and specific information about the dispatch port and the latency of each instruction.

The information is specified using the following format:

```
"ADC r8,rm8":           ( 00.010.01.0   11.ddd.sss )
"ADC r16/32,rm16/32":   ( 00.010.01.1   11.ddd.sss )
"SBB r8,rm8":           ( 00.011.01.0   11.ddd.sss )
"SBB r16/32,rm16/32":   ( 00.011.01.1   11.ddd.sss )
------------------------------------------------------------
TMP0 := port1.latency_1(ArithFLAGS, REG_ddd);
reg_ddd := port01.latency_1(CF_data, REG_sss);
```

The example above shows the instructions (using bit patterns to designate macroinstructions) and the internal and external dependency structure. The format allows show the instruction dispatch port, the latency of the uop and the throughput as illustrated below.

The first uop

```
TMP0 := port1.latency_1(ArithFLAGS, REG_ddd);
```

Dispatch Port: 1 Uop Latency: 1 cycle

The second uop:

```
1 FLOW : reg_ddd := port01.latency_1(CF_data, REG_sss);
```

Dispatch Port: 0 or 1 Uop Latency: 1 cycle

Several special instructions have multiple values in the latency fields. The following example illustrates the uops:

```
HDR:  "FADD ST,ST( i)":                   ( 11011.00.0 )
HDR:  "FMUL ST,ST( i)":                   ( 11011.00.0 )
```

```
HDR:  "FSUB ST,ST( i)":                    ( 11011.00.0 )
HDR:  "FDIV ST,ST( i)":                    ( 11011.00.0 )

1     FLOW:       ST0   =      port_0.latency_4_or_5_or_99(ST0, ST(i))
```

The above latency means that an FADD will have a latency of 4 a FMUL will have a latency of 5 and the FDIV has a latency depending upon the precision . Single precision takes 17 clocks, double precision takes 32 clocks and extended precision takes 37 clocks.

```
HDR:  "FSQRT":                             ( 11011.001 11111010 -------- )

1     FLOW:       ST0   =      fp_sqrt.Port_0.Latency_66(ST0, CONST)
```

The FSQRT instruction has a latency of 28 clocks for single precision, 57 clocks for double precision and 68 clocks for extended precision.

The table lists all of the integer and floating-point instructions, however, uop detail is not presented for instructions that produce greater than 4 uops (complex instructions). In this case the following line appears:

```
Complex Instruction Found, Data not Presented
HDR:   "ADD r8,rm8":                       ( 00.000.01.0 11.ddd.sss -------- )
HDR:   "ADD r16/32,rm16/32":               ( 00.000.01.1 11.ddd.sss -------- )
HDR:   "SUB r8,rm8":                       ( 00.101.01.0 11.ddd.sss -------- )
HDR:   "SUB r16/32,rm16/32":               ( 00.101.01.1 11.ddd.sss -------- )
HDR:   "AND r8,rm8":                       ( 00.100.01.0 11.ddd.sss -------- )
HDR:   "AND r16/32,rm16/32":               ( 00.100.01.1 11.ddd.sss -------- )
HDR:   "OR  r8,rm8":                       ( 00.001.01.0 11.ddd.sss -------- )
HDR:   "OR  r16/32,rm16/32":               ( 00.001.01.1 11.ddd.sss -------- )
HDR:   "XOR r8,rm8":                       ( 00.110.01.0 11.ddd.sss -------- )
HDR:   "XOR r16/32,rm16/32":               ( 00.110.01.1 11.ddd.sss -------- )

1     FLOW:       REG_ddd     =      port_01.latency_1(REG_ddd, REG_sss)
----------------------------------------------

HDR:  "ADD r8,m8":                         ( 00.000.01.0 00.ddd.MMM -------- )
HDR:                                       ( 00.000.01.0 01.ddd.MMM -------- )
HDR:                                       ( 00.000.01.0 10.ddd.MMM -------- )
HDR:  "ADD r16/32,m16/32":                 ( 00.000.01.1 00.ddd.MMM -------- )
HDR:                                       ( 00.000.01.1 01.ddd.MMM -------- )
HDR:                                       ( 00.000.01.1 10.ddd.MMM -------- )
HDR:  "SUB r8,m8":                         ( 00.101.01.0 00.ddd.MMM -------- )
HDR:                                       ( 00.101.01.0 01.ddd.MMM -------- )
HDR:                                       ( 00.101.01.0 10.ddd.MMM -------- )
HDR:  "SUB r16/32,m16/32":                 ( 00.101.01.1 00.ddd.MMM -------- )
HDR:                                       ( 00.101.01.1 01.ddd.MMM -------- )
HDR:                                       ( 00.101.01.1 10.ddd.MMM -------- )
HDR:  "AND r8,m8":                         ( 00.100.01.0 00.ddd.MMM -------- )
HDR:                                       ( 00.100.01.0 01.ddd.MMM -------- )
HDR:                                       ( 00.100.01.0 10.ddd.MMM -------- )
HDR:  "AND r16/32,m16/32":                 ( 00.100.01.1 00.ddd.MMM -------- )
HDR:                                       ( 00.100.01.1 01.ddd.MMM -------- )
HDR:                                       ( 00.100.01.1 10.ddd.MMM -------- )
HDR:  "OR  r8,m8":                         ( 00.001.01.0 00.ddd.MMM -------- )
HDR:                                       ( 00.001.01.0 01.ddd.MMM -------- )
HDR:                                       ( 00.001.01.0 10.ddd.MMM -------- )
HDR:  "OR  r16/32,m16/32":                 ( 00.001.01.1 00.ddd.MMM -------- )
```

```
HDR:                                     ( 00.001.01.1 01.ddd.MMM -------- )
HDR:                                     ( 00.001.01.1 10.ddd.MMM -------- )
HDR:  "XOR r8,m8":                       ( 00.110.01.0 00.ddd.MMM -------- )
HDR:                                     ( 00.110.01.0 01.ddd.MMM -------- )
HDR:                                     ( 00.110.01.0 10.ddd.MMM -------- )
HDR:  "XOR r16/32,m16/32":               ( 00.110.01.1 00.ddd.MMM -------- )
HDR:                                     ( 00.110.01.1 01.ddd.MMM -------- )
HDR:                                     ( 00.110.01.1 10.ddd.MMM -------- )


1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       REG_ddd     =      port_01.latency_1(REG_ddd, TMP0)
-----------------------

HDR:  "ADC r8,rm8":                      ( 00.010.01.0 11.ddd.sss -------- )
HDR:  "ADC r16/32,rm16/32":              ( 00.010.01.1 11.ddd.sss -------- )
HDR:  "SBB r8,rm8":                      ( 00.011.01.0 11.ddd.sss -------- )
HDR:  "SBB r16/32,rm16/32":              ( 00.011.01.1 11.ddd.sss -------- )


1     FLOW:       TMP0        =      Port_01.latency_1(ArithFLAGS, REG_ddd)
2     FLOW:       REG_ddd     =      port_01.latency_1(TMP0, REG_sss)
-----------------------

HDR:  "ADC r8,m8":                       ( 00.010.01.0 00.ddd.MMM -------- )
HDR:                                     ( 00.010.01.0 01.ddd.MMM -------- )
HDR:                                     ( 00.010.01.0 10.ddd.MMM -------- )
HDR:  "ADC r16/32,m16/32":               ( 00.010.01.1 00.ddd.MMM -------- )
HDR:                                     ( 00.010.01.1 01.ddd.MMM -------- )
HDR:                                     ( 00.010.01.1 10.ddd.MMM -------- )
HDR:  "SBB r8,m8":                       ( 00.011.01.0 00.ddd.MMM -------- )
HDR:                                     ( 00.011.01.0 01.ddd.MMM -------- )
HDR:                                     ( 00.011.01.0 10.ddd.MMM -------- )
HDR:  "SBB r16/32,m16/32":               ( 00.011.01.1 00.ddd.MMM -------- )
HDR:                                     ( 00.011.01.1 01.ddd.MMM -------- )
HDR:                                     ( 00.011.01.1 10.ddd.MMM -------- )


1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP1        =      Port_01.latency_1(ArithFLAGS, REG_ddd)
3     FLOW:       REG_ddd     =      Port_01.latency_1(TMP1, TMP0)
-----------------------

HDR:  "ADD rm8,r8":                      ( 00.000.00.0 11.ddd.sss -------- )
HDR:  "ADD rm16/32,r16/32":              ( 00.000.00.1 11.ddd.sss -------- )
HDR:  "SUB rm8,r8":                      ( 00.101.00.0 11.ddd.sss -------- )
HDR:  "SUB rm16/32,r16/32":              ( 00.101.00.1 11.ddd.sss -------- )
HDR:  "AND rm8,r8":                      ( 00.100.00.0 11.ddd.sss -------- )
HDR:  "AND rm16/32,r16/32":              ( 00.100.00.1 11.ddd.sss -------- )
HDR:  "OR  rm8,r8":                      ( 00.001.00.0 11.ddd.sss -------- )
HDR:  "OR  rm16/32,r16/32":              ( 00.001.00.1 11.ddd.sss -------- )
HDR:  "XOR rm8,r8":                      ( 00.110.00.0 11.ddd.sss -------- )
HDR:  "XOR rm16/32,r16/32":              ( 00.110.00.1 11.ddd.sss -------- )

1     FLOW:       REG_sss     =      port_01.latency_1(REG_sss, REG_ddd)
-----------------------
```

```
HDR:  "ADD m8,r8":                      ( 00.000.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.000.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.000.00.0 10.ddd.MMM -------- )
HDR:  "ADD m16/32,r16/32":              ( 00.000.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.000.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.000.00.1 10.ddd.MMM -------- )
HDR:  "SUB m8,r8":                      ( 00.101.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.101.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.101.00.0 10.ddd.MMM -------- )
HDR:  "SUB m16/32,r16/32":              ( 00.101.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.101.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.101.00.1 10.ddd.MMM -------- )
HDR:  "AND m8,r8":                      ( 00.100.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.100.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.100.00.0 10.ddd.MMM -------- )
HDR:  "AND m16/32,r16/32":              ( 00.100.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.100.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.100.00.1 10.ddd.MMM -------- )
HDR:  "OR  m8,r8":                      ( 00.001.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.001.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.001.00.0 10.ddd.MMM -------- )
HDR:  "OR  m16/32,r16/32":              ( 00.001.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.001.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.001.00.1 10.ddd.MMM -------- )
HDR:  "XOR m8,r8":                      ( 00.110.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.110.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.110.00.0 10.ddd.MMM -------- )
HDR:  "XOR m16/32,r16/32":              ( 00.110.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.110.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.110.00.1 10.ddd.MMM -------- )


1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      port_01.latency_1(TMP0, REG_ddd)
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "LOCK ADD m8,r8":                 ( 00.000.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.000.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.000.00.0 10.ddd.MMM -------- )
HDR:  "LOCK ADD m16/32,r16/32":         ( 00.000.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.000.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.000.00.1 10.ddd.MMM -------- )
HDR:  "LOCK SUB m8,r8":                 ( 00.101.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.101.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.101.00.0 10.ddd.MMM -------- )
HDR:  "LOCK SUB m16/32,r16/32":         ( 00.101.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.101.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.101.00.1 10.ddd.MMM -------- )
HDR:  "LOCK AND m8,r8":                 ( 00.100.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.100.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.100.00.0 10.ddd.MMM -------- )
HDR:  "LOCK AND m16/32,r16/32":         ( 00.100.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.100.00.1 01.ddd.MMM -------- )
```

```
HDR:                                    ( 00.100.00.1 10.ddd.MMM -------- )
HDR:  "LOCK OR  m8,r8":                 ( 00.001.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.001.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.001.00.0 10.ddd.MMM -------- )
HDR:  "LOCK OR  m16/32,r16/32":         ( 00.001.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.001.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.001.00.1 10.ddd.MMM -------- )
HDR:  "LOCK XOR m8,r8":                 ( 00.110.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.110.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.110.00.0 10.ddd.MMM -------- )
HDR:  "LOCK XOR m16/32,r16/32":         ( 00.110.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.110.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.110.00.1 10.ddd.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "ADC rm8,r8":                     ( 00.010.00.0 11.ddd.sss -------- )
HDR:  "ADC rm16/32,r16/32":             ( 00.010.00.1 11.ddd.sss -------- )
HDR:  "SBB rm8,r8":                     ( 00.011.00.0 11.ddd.sss -------- )
HDR:  "SBB rm16/32,r16/32":             ( 00.011.00.1 11.ddd.sss -------- )


1     FLOW:       TMP0        =      Port_01.latency_1(ArithFLAGS, REG_sss)
2     FLOW:       REG_sss     =      port_01.latency_1(TMP0, REG_ddd)
-----------------------

HDR:  "ADC m8,r8":                      ( 00.010.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.010.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.010.00.0 10.ddd.MMM -------- )
HDR:  "ADC m16/32,r16/32":              ( 00.010.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.010.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.010.00.1 10.ddd.MMM -------- )
HDR:  "SBB m8,r8":                      ( 00.011.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.011.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.011.00.0 10.ddd.MMM -------- )
HDR:  "SBB m16/32,r16/32":              ( 00.011.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.011.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.011.00.1 10.ddd.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "LOCK ADC m8,r8":                 ( 00.010.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.010.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.010.00.0 10.ddd.MMM -------- )
HDR:  "LOCK ADC m16/32,r16/32":         ( 00.010.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.010.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.010.00.1 10.ddd.MMM -------- )
HDR:  "LOCK SBB m8,r8":                 ( 00.011.00.0 00.ddd.MMM -------- )
HDR:                                    ( 00.011.00.0 01.ddd.MMM -------- )
HDR:                                    ( 00.011.00.0 10.ddd.MMM -------- )
HDR:  "LOCK SBB m16/32,r16/32":         ( 00.011.00.1 00.ddd.MMM -------- )
HDR:                                    ( 00.011.00.1 01.ddd.MMM -------- )
HDR:                                    ( 00.011.00.1 10.ddd.MMM -------- )
```

```
Complex Instruction Found, Data not Presented

HDR:  "ADD r8,imm8":                     ( 100000.s.0 11.000.sss -------- )
HDR:  "ADD r16/32,imm16/32":             ( 100000.0.1 11.000.sss -------- )
HDR:  "ADD r16/32,imm8":                 ( 100000.1.1 11.000.sss -------- )
HDR:  "SUB r8,imm8":                     ( 100000.s.0 11.101.sss -------- )
HDR:  "SUB r16/32,imm16/32":             ( 100000.0.1 11.101.sss -------- )
HDR:  "SUB r16/32,imm8":                 ( 100000.1.1 11.101.sss -------- )
HDR:  "AND r8,imm8":                     ( 100000.s.0 11.100.sss -------- )
HDR:  "AND r16/32,imm16/32":             ( 100000.0.1 11.100.sss -------- )
HDR:  "AND r16/32,imm8":                 ( 100000.1.1 11.100.sss -------- )
HDR:  "OR  r8,imm8":                     ( 100000.s.0 11.001.sss -------- )
HDR:  "OR  r16/32,imm16/32":             ( 100000.0.1 11.001.sss -------- )
HDR:  "OR  r16/32,imm8":                 ( 100000.1.1 11.001.sss -------- )
HDR:  "XOR r8,imm8":                     ( 100000.s.0 11.110.sss -------- )
HDR:  "XOR r16/32,imm16/32":             ( 100000.0.1 11.110.sss -------- )
HDR:  "XOR r16/32,imm8":                 ( 100000.1.1 11.110.sss -------- )

1     FLOW:       REG_sss     =      port_01.latency_1(REG_sss, IMM)
-----------------------

HDR:  "ADD m8,imm8":                     ( 100000.s.0 00.000.sss -------- )
HDR:                                     ( 100000.s.0 01.000.sss -------- )
HDR:                                     ( 100000.s.0 10.000.sss -------- )
HDR:  "ADD m16/32,imm16/32":             ( 100000.s.1 00.000.sss -------- )
HDR:                                     ( 100000.s.1 01.000.sss -------- )
HDR:                                     ( 100000.s.1 10.000.sss -------- )
HDR:  "SUB m8,imm8":                     ( 100000.s.0 00.101.sss -------- )
HDR:                                     ( 100000.s.0 01.101.sss -------- )
HDR:                                     ( 100000.s.0 10.101.sss -------- )
HDR:  "SUB m16/32,imm16/32":             ( 100000.s.1 00.101.sss -------- )
HDR:                                     ( 100000.s.1 01.101.sss -------- )
HDR:                                     ( 100000.s.1 10.101.sss -------- )
HDR:  "AND m8,imm8":                     ( 100000.s.0 00.100.sss -------- )
HDR:                                     ( 100000.s.0 01.100.sss -------- )
HDR:                                     ( 100000.s.0 10.100.sss -------- )
HDR:  "AND m16/32,imm16/32":             ( 100000.s.1 00.100.sss -------- )
HDR:                                     ( 100000.s.1 01.100.sss -------- )
HDR:                                     ( 100000.s.1 10.100.sss -------- )
HDR:  "OR  m8,imm8":                     ( 100000.s.0 00.001.sss -------- )
HDR:                                     ( 100000.s.0 01.001.sss -------- )
HDR:                                     ( 100000.s.0 10.001.sss -------- )
HDR:  "OR  m16/32,imm16/32":             ( 100000.s.1 00.001.sss -------- )
HDR:                                     ( 100000.s.1 01.001.sss -------- )
HDR:                                     ( 100000.s.1 10.001.sss -------- )
HDR:  "XOR m8,imm8":                     ( 100000.s.0 00.110.sss -------- )
HDR:                                     ( 100000.s.0 01.110.sss -------- )
HDR:                                     ( 100000.s.0 10.110.sss -------- )
HDR:  "XOR m16/32,imm16/32":             ( 100000.s.1 00.110.sss -------- )
HDR:                                     ( 100000.s.1 01.110.sss -------- )
HDR:                                     ( 100000.s.1 10.110.sss -------- )


1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      port_01.latency_1(TMP0, IMM)
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
```

```
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "LOCK ADD m8,imm8":                 ( 100000.s.0 00.000.sss -------- )
HDR:                                      ( 100000.s.0 01.000.sss -------- )
HDR:                                      ( 100000.s.0 10.000.sss -------- )
HDR:  "LOCK ADD m16/32,imm16/32":         ( 100000.s.1 00.000.sss -------- )
HDR:                                      ( 100000.s.1 01.000.sss -------- )
HDR:                                      ( 100000.s.1 10.000.sss -------- )
HDR:  "LOCK SUB m8,imm8":                 ( 100000.s.0 00.101.sss -------- )
HDR:                                      ( 100000.s.0 01.101.sss -------- )
HDR:                                      ( 100000.s.0 10.101.sss -------- )
HDR:  "LOCK SUB m16/32,imm16/32":         ( 100000.s.1 00.101.sss -------- )
HDR:                                      ( 100000.s.1 01.101.sss -------- )
HDR:                                      ( 100000.s.1 10.101.sss -------- )
HDR:  "LOCK AND m8,imm8":                 ( 100000.s.0 00.100.sss -------- )
HDR:                                      ( 100000.s.0 01.100.sss -------- )
HDR:                                      ( 100000.s.0 10.100.sss -------- )
HDR:  "LOCK AND m16/32,imm16/32":         ( 100000.s.1 00.100.sss -------- )
HDR:                                      ( 100000.s.1 01.100.sss -------- )
HDR:                                      ( 100000.s.1 10.100.sss -------- )
HDR:  "LOCK OR  m8,imm8":                 ( 100000.s.0 00.001.sss -------- )
HDR:                                      ( 100000.s.0 01.001.sss -------- )
HDR:                                      ( 100000.s.0 10.001.sss -------- )
HDR:  "LOCK OR  m16/32,imm16/32":         ( 100000.s.1 00.001.sss -------- )
HDR:                                      ( 100000.s.1 01.001.sss -------- )
HDR:                                      ( 100000.s.1 10.001.sss -------- )
HDR:  "LOCK XOR m8,imm8":                 ( 100000.s.0 00.110.sss -------- )
HDR:                                      ( 100000.s.0 01.110.sss -------- )
HDR:                                      ( 100000.s.0 10.110.sss -------- )
HDR:  "LOCK XOR m16/32,imm16/32":         ( 100000.s.1 00.110.sss -------- )
HDR:                                      ( 100000.s.1 01.110.sss -------- )
HDR:                                      ( 100000.s.1 10.110.sss -------- )

Complex Instruction Found, Data not Presented

HDR:  "ADC r8,imm8":                      ( 100000.s.0 11.010.sss -------- )
HDR:  "ADC r16/32,imm16/32":              ( 100000.s.1 11.010.sss -------- )
HDR:  "SBB r8,imm8":                      ( 100000.s.0 11.011.sss -------- )
HDR:  "SBB r16/32,imm16/32":              ( 100000.s.1 11.011.sss -------- )


1     FLOW:       TMP0        =      Port_01.latency_1(ArithFLAGS, REG_sss)
2     FLOW:       REG_sss     =      port_01.latency_1(TMP0, IMM)
-----------------------

HDR:  "ADC m8,imm8":                      ( 100000.s.0 00.010.MMM -------- )
HDR:                                      ( 100000.s.0 01.010.MMM -------- )
HDR:                                      ( 100000.s.0 10.010.MMM -------- )
HDR:  "ADC m16/32,imm16/32":              ( 100000.s.1 00.010.MMM -------- )
HDR:                                      ( 100000.s.1 01.010.MMM -------- )
HDR:                                      ( 100000.s.1 10.010.MMM -------- )
HDR:  "SBB m8,imm8":                      ( 100000.s.0 00.011.MMM -------- )
HDR:                                      ( 100000.s.0 01.011.MMM -------- )
HDR:                                      ( 100000.s.0 10.011.MMM -------- )
HDR:  "SBB m16/32,imm16/32":              ( 100000.s.1 00.011.MMM -------- )
```

```
HDR:                                    ( 100000.s.1 01.011.MMM -------- )
HDR:                                    ( 100000.s.1 10.011.MMM -------- )
Complex Instruction found, Data not presented

-----------------------

HDR:  "LOCK ADC m8,imm8":               ( 100000.s.0 00.010.MMM -------- )
HDR:                                    ( 100000.s.0 01.010.MMM -------- )
HDR:                                    ( 100000.s.0 10.010.MMM -------- )
HDR:  "LOCK ADC m16/32,imm16/32":       ( 100000.s.1 00.010.MMM -------- )
HDR:                                    ( 100000.s.1 01.010.MMM -------- )
HDR:                                    ( 100000.s.1 10.010.MMM -------- )
HDR:  "LOCK SBB m8,imm8":               ( 100000.s.0 00.011.MMM -------- )
HDR:                                    ( 100000.s.0 01.011.MMM -------- )
HDR:                                    ( 100000.s.0 10.011.MMM -------- )
HDR:  "LOCK SBB m16/32,imm16/32":       ( 100000.s.1 00.011.MMM -------- )
HDR:                                    ( 100000.s.1 01.011.MMM -------- )
HDR:                                    ( 100000.s.1 10.011.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "ADD AL,imm8":                    ( 00.000.10.0 -------- -------- )
HDR:  "SUB AL,imm8":                    ( 00.101.10.0 -------- -------- )
HDR:  "AND AL,imm8":                    ( 00.100.10.0 -------- -------- )
HDR:  "OR  AL,imm8":                    ( 00.001.10.0 -------- -------- )
HDR:  "XOR AL,imm8":                    ( 00.110.10.0 -------- -------- )
HDR:  "ADD eAX,imm16/32":               ( 00.000.10.1 -------- -------- )
HDR:  "SUB eAX,imm16/32":               ( 00.101.10.1 -------- -------- )
HDR:  "AND eAX,imm16/32":               ( 00.100.10.1 -------- -------- )
HDR:  "OR  eAX,imm16/32":               ( 00.001.10.1 -------- -------- )
HDR:  "XOR eAX,imm16/32":               ( 00.110.10.1 -------- -------- )

1     FLOW:       EAX         =      port_01.latency_1(EAX, IMM)
-----------------------

HDR:  "ADC AL,imm8":                    ( 00.010.10.0 -------- -------- )
HDR:  "SBB AL,imm8":                    ( 00.011.10.0 -------- -------- )
HDR:  "ADC eAX,imm16/32":               ( 00.010.10.1 -------- -------- )
HDR:  "SBB eAX,imm16/32":               ( 00.011.10.1 -------- -------- )


1     FLOW:       TMP0        =      Port_01.latency_1(ArithFLAGS, EAX)
2     FLOW:       EAX         =      port_01.latency_1(TMP0, IMM)
-----------------------

HDR:  "INC  rm8":                       ( 1111111.0 11.000.sss -------- )
HDR:  "INC  rm16/32":                   ( 1111111.1 11.000.sss -------- )
HDR:  "DEC  rm8":                       ( 1111111.0 11.001.sss -------- )
HDR:  "DEC  rm16/32":                   ( 1111111.1 11.001.sss -------- )

1     FLOW:       REG_sss     =      port_01.latency_1(REG_sss, CONST)
-----------------------

HDR:  "INC  m8":                        ( 1111111.0 00.000.MMM -------- )
HDR:                                    ( 1111111.0 01.000.MMM -------- )
HDR:                                    ( 1111111.0 10.000.MMM -------- )
```

```
HDR:  "INC  m16/32":                     ( 1111111.1 00.000.MMM -------- )
HDR:                                     ( 1111111.1 01.000.MMM -------- )
HDR:                                     ( 1111111.1 10.000.MMM -------- )
HDR:  "DEC  m8":                         ( 1111111.0 00.001.MMM -------- )
HDR:                                     ( 1111111.0 01.001.MMM -------- )
HDR:                                     ( 1111111.0 10.001.MMM -------- )
HDR:  "DEC  m16/32":                     ( 1111111.1 00.001.MMM -------- )
HDR:                                     ( 1111111.1 01.001.MMM -------- )
HDR:                                     ( 1111111.1 10.001.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      port_01.latency_1(TMP0, CONST)
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "LOCK INC  m8":                    ( 1111111.0 00.000.MMM -------- )
HDR:                                     ( 1111111.0 01.000.MMM -------- )
HDR:                                     ( 1111111.0 10.000.MMM -------- )
HDR:  "LOCK INC  m16/32":                ( 1111111.1 00.000.MMM -------- )
HDR:                                     ( 1111111.1 01.000.MMM -------- )
HDR:                                     ( 1111111.1 10.000.MMM -------- )
HDR:  "LOCK DEC  m8":                    ( 1111111.0 00.001.MMM -------- )
HDR:                                     ( 1111111.0 01.001.MMM -------- )
HDR:                                     ( 1111111.0 10.001.MMM -------- )
HDR:  "LOCK DEC  m16/32":                ( 1111111.1 00.001.MMM -------- )
HDR:                                     ( 1111111.1 01.001.MMM -------- )
HDR:                                     ( 1111111.1 10.001.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "INC  r16/32":                     ( 01.000.ddd -------- -------- )
HDR:  "DEC  r16/32":                     ( 01.001.ddd -------- -------- )

1     FLOW:       REG_ddd     =      port_01.latency_1(REG_ddd, CONST)
-----------------------

HDR:  "NOT  rm8":                        ( 1111011.0 11.010.sss -------- )
HDR:  "NOT  rm16/32":                    ( 1111011.1 11.010.sss -------- )

1     FLOW:       REG_sss     =      xor.Port_01.latency_1(REG_sss, CONST)
-----------------------

HDR:  "NOT  m8":                         ( 1111011.0 00.010.MMM -------- )
HDR:                                     ( 1111011.0 01.010.MMM -------- )
HDR:                                     ( 1111011.0 10.010.MMM -------- )
HDR:  "NOT  m16/32":                     ( 1111011.1 00.010.MMM -------- )
HDR:                                     ( 1111011.1 01.010.MMM -------- )
HDR:                                     ( 1111011.1 10.010.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      xor.Port_01.latency_1(TMP0, CONST)
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------
```

```
HDR:  "LOCK NOT  m8":                    ( 1111011.0 00.010.MMM -------- )
HDR:                                     ( 1111011.0 01.010.MMM -------- )
HDR:                                     ( 1111011.0 10.010.MMM -------- )
HDR:  "LOCK NOT  m16/32":                ( 1111011.1 00.010.MMM -------- )
HDR:                                     ( 1111011.1 01.010.MMM -------- )
HDR:                                     ( 1111011.1 10.010.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "NEG  rm8":                        ( 1111011.0 11.011.sss -------- )
HDR:  "NEG  rm16/32":                    ( 1111011.1 11.011.sss -------- )

1     FLOW:       REG_sss     =      sub.Port_01.latency_1(CONST, REG_sss)
-----------------------

HDR:  "NEG  m8":                         ( 1111011.0 00.011.MMM -------- )
HDR:                                     ( 1111011.0 01.011.MMM -------- )
HDR:                                     ( 1111011.0 10.011.MMM -------- )
HDR:  "NEG  m16/32":                     ( 1111011.1 00.011.MMM -------- )
HDR:                                     ( 1111011.1 01.011.MMM -------- )
HDR:                                     ( 1111011.1 10.011.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      sub.Port_01.latency_1(CONST, TMP0)
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "LOCK NEG  m8":                    ( 1111011.0 00.011.MMM -------- )
HDR:                                     ( 1111011.0 01.011.MMM -------- )
HDR:                                     ( 1111011.0 10.011.MMM -------- )
HDR:  "LOCK NEG  m16/32":                ( 1111011.1 00.011.MMM -------- )
HDR:                                     ( 1111011.1 01.011.MMM -------- )
HDR:                                     ( 1111011.1 10.011.MMM -------- )

Complex Instruction Found, Data not Presented-----------------------

HDR:  "AAS":                             ( 00111111 -------- -------- )

1     FLOW:       AX    =      Port_1.latency_1(ArithFLAGS, AX)
-----------------------

HDR:  "DAA":                             ( 00100111 -------- -------- )

1     FLOW:       AL    =      Port_1.latency_1(ArithFLAGS, AL)
-----------------------

HDR:  "DAS":                             ( 00101111 -------- -------- )

1     FLOW:       AL    =      Port_1.latency_1(ArithFLAGS, AL)
-----------------------

HDR:  "AAD":                             ( 11010101 -------- -------- )

1     FLOW:       TMP0        =      mul.Port_0.Latency_4(AH, IMM)
2     FLOW:       AH          =      move.Port_01.latency_1(CONST)
```

```
3     FLOW:       AL          =      add.Port_01.latency_1(AL, TMP0)
-----------------------

HDR:  "AAM":                              ( 11010100 -------- -------- )

1     FLOW:       TMP0        =      div.Port_0.Latency_99(AL, IMM)
2     FLOW:       AH          =      move.Port_01.latency_1(TMP0)
3     FLOW:       TMP0        =      Port_0.latency_1(TMP0, CONST)
4     FLOW:       AL          =      add.Port_01.latency_1(TMP0, CONST)
-----------------------

HDR:  "INVD":                             ( 00001111 0000.1000 -------- )

Complex Instruction Found, Data not Presented

HDR:  "WBINVD":                           ( 00001111 0000.1001 -------- )

Complex Instruction Found, Data not Presented

HDR:  "INVLPG m":                         ( 00001111 0000.0001 00.111.sss )
HDR:                                      ( 00001111 0000.0001 01.111.sss )
HDR:                                      ( 00001111 0000.0001 10.111.sss )

Complex Instruction Found, Data not Presented -----------------------

HDR:  "CMP AL, imm8":                     ( 0011110.0 -------- -------- )
HDR:  "CMP eAX,imm16/32":                 ( 0011110.1 -------- -------- )

1     FLOW:       sink        =      sub.Port_01.latency_1(EAX, IMM)
-----------------------
HDR:  "CMP rm8,imm8":                     ( 100000.0.0 11.111.sss -------- )
HDR:  "CMP rm8,imm8":                     ( 100000.1.0 11.111.sss -------- )
HDR:  "CMP rm16/32,imm16/32":             ( 100000.0.1 11.111.sss -------- )
HDR:  "CMP rm16/32,imm8":                 ( 100000.1.1 11.111.sss -------- )

1     FLOW:       sink        =      sub.Port_01.latency_1(REG_sss, IMM)
-----------------------

HDR:  "CMP m8, imm8":                     ( 100000.0.0 00.111.MMM -------- )
HDR:                                      ( 100000.0.0 01.111.MMM -------- )
HDR:                                      ( 100000.0.0 10.111.MMM -------- )
HDR:  "CMP m8, imm8":                     ( 100000.1.0 00.111.MMM -------- )
HDR:                                      ( 100000.1.0 01.111.MMM -------- )
HDR:                                      ( 100000.1.0 10.111.MMM -------- )
HDR:  "CMP m16/32, imm16/32":             ( 100000.0.1 00.111.MMM -------- )
HDR:                                      ( 100000.0.1 01.111.MMM -------- )
HDR:                                      ( 100000.0.1 10.111.MMM -------- )
HDR:  "CMP m16/32, imm8":                 ( 100000.1.1 00.111.MMM -------- )
HDR:                                      ( 100000.1.1 01.111.MMM -------- )
HDR:                                      ( 100000.1.1 10.111.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       sink        =      sub.Port_01.latency_1(TMP0, IMM)    1
-----------------------

HDR:  "CMP rm8,r8":                       ( 001110.0.0 11.ddd.sss -------- )
```

```
HDR:  "CMP rm16/32,r16/32":              ( 001110.0.1 11.ddd.sss -------- )

1     FLOW:       sink        =      sub.Port_01.latency_1(REG_sss, REG_ddd)
-----------------------

HDR:  "CMP m8,r8":                       ( 001110.0.0 00.ddd.MMM -------- )
HDR:                                     ( 001110.0.0 01.ddd.MMM -------- )
HDR:                                     ( 001110.0.0 10.ddd.MMM -------- )
HDR:  "CMP m16/32,r16/32":               ( 001110.0.1 00.ddd.MMM -------- )
HDR:                                     ( 001110.0.1 01.ddd.MMM -------- )
HDR:                                     ( 001110.0.1 10.ddd.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       sink        =      sub.Port_01.latency_1(TMP0, REG_ddd)
-----------------------

HDR:  "CMP r8,rm8":                      ( 001110.1.0 11.ddd.sss -------- )
HDR:  "CMP r16/32,rm16/32":              ( 001110.1.1 11.ddd.sss -------- )

1     FLOW:       sink        =      sub.Port_01.latency_1(REG_ddd, REG_sss)
-----------------------

HDR:  "CMP r8,m8":                       ( 001110.1.0 00.ddd.MMM -------- )
HDR:                                     ( 001110.1.0 01.ddd.MMM -------- )
HDR:                                     ( 001110.1.0 10.ddd.MMM -------- )
HDR:  "CMP r16/32,m16/32":               ( 001110.1.1 00.ddd.MMM -------- )
HDR:                                     ( 001110.1.1 01.ddd.MMM -------- )
HDR:                                     ( 001110.1.1 10.ddd.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       sink        =      sub.Port_01.latency_1(REG_ddd, TMP0)
-----------------------

HDR:  "TEST rm8,r8":                     ( 10.000.10.0 11.ddd.sss -------- )
HDR:  "TEST rm16/32,r16/32":             ( 10.000.10.1 11.ddd.sss -------- )

1     FLOW:       sink        =      and.Port_01.latency_1(REG_ddd, REG_sss)
-----------------------

HDR:  "TEST m8,r8":                      ( 10.000.10.0 00.ddd.MMM -------- )
HDR:                                     ( 10.000.10.0 01.ddd.MMM -------- )
HDR:                                     ( 10.000.10.0 10.ddd.MMM -------- )
HDR:  "TEST m16/32,r16/32":              ( 10.000.10.1 00.ddd.MMM -------- )
HDR:                                     ( 10.000.10.1 01.ddd.MMM -------- )
HDR:                                     ( 10.000.10.1 10.ddd.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       sink        =      and.Port_01.latency_1(REG_ddd, TMP0)
-----------------------

HDR:  "TEST AL,imm8":                    ( 10101000 -------- -------- )
HDR:  "TEST eAX,imm16/32":               ( 10101001 -------- -------- )

1     FLOW:       sink        =      and.Port_01.latency_1(EAX, IMM)
-----------------------
```

```
HDR:  "TEST rm8,imm8":                    ( 11110110 11.00-.sss -------- )
HDR:  "TEST rm16/32,imm16/32":            ( 11110111 11.00-.sss -------- )

1     FLOW:       sink        =      and.Port_01.latency_1(REG_sss, IMM)
-----------------------

HDR:  "TEST m8,imm8":                     ( 11110110 00.000.MMM -------- )
HDR:                                      ( 11110110 01.000.MMM -------- )
HDR:                                      ( 11110110 10.000.MMM -------- )
HDR:  "TEST m8,imm8":                     ( 11110110 00.001.MMM -------- )
HDR:                                      ( 11110110 01.001.MMM -------- )
HDR:                                      ( 11110110 10.001.MMM -------- )
HDR:  "TEST m16/32,imm16/32":             ( 11110111 00.000.MMM -------- )
HDR:                                      ( 11110111 01.000.MMM -------- )
HDR:                                      ( 11110111 10.000.MMM -------- )
HDR:  "TEST m16/32,imm16/32":             ( 11110111 00.001.MMM -------- )
HDR:                                      ( 11110111 01.001.MMM -------- )
HDR:                                      ( 11110111 10.001.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       sink        =      and.Port_01.latency_1(TMP0, IMM)
-----------------------

HDR:  "SETO  rm8":                        ( 00001111 1001.0000 11.---.sss )
HDR:  "SETNO rm8":                        ( 00001111 1001.0001 11.---.sss )
HDR:  "SETB/NAE/C rm8":                   ( 00001111 1001.0010 11.---.sss )
HDR:  "SETNB/AE/NC rm8":                  ( 00001111 1001.0011 11.---.sss )
HDR:  "SETE/Z rm8":                       ( 00001111 1001.0100 11.---.sss )
HDR:  "SETNE/NZ rm8":                     ( 00001111 1001.0101 11.---.sss )
HDR:  "SETBE/NA rm8":                     ( 00001111 1001.0110 11.---.sss )
HDR:  "SETNBE/A rm8":                     ( 00001111 1001.0111 11.---.sss )
HDR:  "SETS rm8":                         ( 00001111 1001.1000 11.---.sss )
HDR:  "SETNS rm8":                        ( 00001111 1001.1001 11.---.sss )
HDR:  "SETP/PE rm8":                      ( 00001111 1001.1010 11.---.sss )
HDR:  "SETNP/PO rm8":                     ( 00001111 1001.1011 11.---.sss )
HDR:  "SETL/NGE rm8":                     ( 00001111 1001.1100 11.---.sss )
HDR:  "SETNL/GE rm8":                     ( 00001111 1001.1101 11.---.sss )
HDR:  "SETLE/NG rm8":                     ( 00001111 1001.1110 11.---.sss )
HDR:  "SETNLE/G rm8":                     ( 00001111 1001.1111 11.---.sss )

1     FLOW:       REG_sss     =      Port_01.latency_1(ArithFLAGS, CONST)
-----------------------

HDR:  "SETO  m8":                         ( 00001111 1001.0000 00.---.MMM )
HDR:                                      ( 00001111 1001.0000 01.---.MMM )
HDR:                                      ( 00001111 1001.0000 10.---.MMM )
HDR:  "SETNO m8":                         ( 00001111 1001.0001 00.---.MMM )
HDR:                                      ( 00001111 1001.0001 01.---.MMM )
HDR:                                      ( 00001111 1001.0001 10.---.MMM )
HDR:  "SETB/NAE/C m8":                    ( 00001111 1001.0010 00.---.MMM )
HDR:                                      ( 00001111 1001.0010 01.---.MMM )
HDR:                                      ( 00001111 1001.0010 10.---.MMM )
HDR:  "SETNB/AE/NC m8":                   ( 00001111 1001.0011 00.---.MMM )
HDR:                                      ( 00001111 1001.0011 01.---.MMM )
HDR:                                      ( 00001111 1001.0011 10.---.MMM )
HDR:  "SETE/Z m8":                        ( 00001111 1001.0100 00.---.MMM )
```

```
HDR:                                     ( 00001111 1001.0100 01.---.MMM )
HDR:                                     ( 00001111 1001.0100 10.---.MMM )
HDR:  "SETNE/NZ m8":                     ( 00001111 1001.0101 00.---.MMM )
HDR:                                     ( 00001111 1001.0101 01.---.MMM )
HDR:                                     ( 00001111 1001.0101 10.---.MMM )
HDR:  "SETBE/NA m8":                     ( 00001111 1001.0110 00.---.MMM )
HDR:                                     ( 00001111 1001.0110 01.---.MMM )
HDR:                                     ( 00001111 1001.0110 10.---.MMM )
HDR:  "SETNBE/A m8":                     ( 00001111 1001.0111 00.---.MMM )
HDR:                                     ( 00001111 1001.0111 01.---.MMM )
HDR:                                     ( 00001111 1001.0111 10.---.MMM )
HDR:  "SETS m8":                         ( 00001111 1001.1000 00.---.MMM )
HDR:                                     ( 00001111 1001.1000 01.---.MMM )
HDR:                                     ( 00001111 1001.1000 10.---.MMM )
HDR:  "SETNS m8":                        ( 00001111 1001.1001 00.---.MMM )
HDR:                                     ( 00001111 1001.1001 01.---.MMM )
HDR:                                     ( 00001111 1001.1001 10.---.MMM )
HDR:  "SETP/PE m8":                      ( 00001111 1001.1010 00.---.MMM )
HDR:                                     ( 00001111 1001.1010 01.---.MMM )
HDR:                                     ( 00001111 1001.1010 10.---.MMM )
HDR:  "SETNP/PO m8":                     ( 00001111 1001.1011 00.---.MMM )
HDR:                                     ( 00001111 1001.1011 01.---.MMM )
HDR:                                     ( 00001111 1001.1011 10.---.MMM )
HDR:  "SETL/NGE m8":                     ( 00001111 1001.1100 00.---.MMM )
HDR:                                     ( 00001111 1001.1100 01.---.MMM )
HDR:                                     ( 00001111 1001.1100 10.---.MMM )
HDR:  "SETNL/GE m8":                     ( 00001111 1001.1101 00.---.MMM )
HDR:                                     ( 00001111 1001.1101 01.---.MMM )
HDR:                                     ( 00001111 1001.1101 10.---.MMM )
HDR:  "SETLE/NG m8":                     ( 00001111 1001.1110 00.---.MMM )
HDR:                                     ( 00001111 1001.1110 01.---.MMM )
HDR:                                     ( 00001111 1001.1110 10.---.MMM )
HDR:  "SETNLE/G m8":                     ( 00001111 1001.1111 00.---.MMM )
HDR:                                     ( 00001111 1001.1111 01.---.MMM )
HDR:                                     ( 00001111 1001.1111 10.---.MMM )

1     FLOW:       TMP0        =      Port_01.latency_1(ArithFLAGS, CONST)
2     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
3     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "BT rm16/32, r16/32":              ( 00001111 10100011 11.ddd.sss )

1     FLOW:       sink        =      test_bit.Port_01.latency_1(REG_sss, REG_ddd)
-----------------------

HDR:  "BT m16/32, r16/32":               ( 00001111 10100011 00.ddd.MMM )
HDR:                                     ( 00001111 10100011 01.ddd.MMM )
HDR:                                     ( 00001111 10100011 10.ddd.MMM )

(7) Complex Instruction Found, Data not Presented

HDR:  "BT rm16/32, imm8":                ( 00001111 10111010 11.100.sss )

1     FLOW:       sink        =      test_bit.Port_01.latency_1(REG_sss, IMM)
-----------------------
```

```
HDR:  "BT m16/32, imm8":                    ( 00001111 10111010 00.100.MMM )
HDR:                                        ( 00001111 10111010 01.100.MMM )
HDR:                                        ( 00001111 10111010 10.100.MMM )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       sink        =      test_bit.Port_01.latency_1(TMP0, IMM)
-----------------------

HDR:  "BTC rm16/32, r16/32":                ( 00001111 10.111.011 11.ddd.sss )
HDR:  "BTR rm16/32, r16/32":                ( 00001111 10.110.011 11.ddd.sss )
HDR:  "BTS rm16/32, r16/32":                ( 00001111 10.101.011 11.ddd.sss )

1     FLOW:       REG_sss     =      port_0.latency_1(REG_sss, REG_ddd)
-----------------------

HDR:  "BTC m16/32, r16/32":                 ( 00001111 10.111.011 00.ddd.MMM )
HDR:                                        ( 00001111 10.111.011 01.ddd.MMM )
HDR:                                        ( 00001111 10.111.011 10.ddd.MMM )
HDR:  "BTR m16/32, r16/32":                 ( 00001111 10.110.011 00.ddd.MMM )
HDR:                                        ( 00001111 10.110.011 01.ddd.MMM )
HDR:                                        ( 00001111 10.110.011 10.ddd.MMM )
HDR:  "BTS m16/32, r16/32":                 ( 00001111 10.101.011 00.ddd.MMM )
HDR:                                        ( 00001111 10.101.011 01.ddd.MMM )
HDR:                                        ( 00001111 10.101.011 10.ddd.MMM )

Complex Instruction Found, Data not Presented

HDR:  "LOCK BTC m16/32, r16/32":            ( 00001111 10.111.011 00.ddd.MMM )
HDR:                                        ( 00001111 10.111.011 01.ddd.MMM )
HDR:                                        ( 00001111 10.111.011 10.ddd.MMM )
HDR:  "LOCK BTR m16/32, r16/32":            ( 00001111 10.110.011 00.ddd.MMM )
HDR:                                        ( 00001111 10.110.011 01.ddd.MMM )
HDR:                                        ( 00001111 10.110.011 10.ddd.MMM )
HDR:  "LOCK BTS m16/32, r16/32":            ( 00001111 10.101.011 00.ddd.MMM )
HDR:                                        ( 00001111 10.101.011 01.ddd.MMM )
HDR:                                        ( 00001111 10.101.011 10.ddd.MMM )

Complex Instruction Found, Data not Presented

HDR:  "BTC rm16/32, imm8":                  ( 00001111 10111010 11.111.sss )
HDR:  "BTR rm16/32, imm8":                  ( 00001111 10111010 11.110.sss )
HDR:  "BTS rm16/32, imm8":                  ( 00001111 10111010 11.101.sss )

1     FLOW:       REG_sss     =      port_0.latency_1(REG_sss, IMM)
-----------------------

HDR:  "BTC m16/32, imm8":                   ( 00001111 10111010 00.111.MMM )
HDR:                                        ( 00001111 10111010 01.111.MMM )
HDR:                                        ( 00001111 10111010 10.111.MMM )
HDR:  "BTR m16/32, imm8":                   ( 00001111 10111010 00.110.MMM )
HDR:                                        ( 00001111 10111010 01.110.MMM )
HDR:                                        ( 00001111 10111010 10.110.MMM )
HDR:  "BTS m16/32, imm8":                   ( 00001111 10111010 00.101.MMM )
HDR:                                        ( 00001111 10111010 01.101.MMM )
HDR:                                        ( 00001111 10111010 10.101.MMM )
```

```
1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP1        =      port_0.latency_1(TMP0, IMM)       1
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP1)  2
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "LOCK BTC m16/32, imm8":           ( 00001111 10111010 00.111.MMM )
HDR:                                     ( 00001111 10111010 01.111.MMM )
HDR:                                     ( 00001111 10111010 10.111.MMM )
HDR:  "LOCK BTR m16/32, imm8":           ( 00001111 10111010 00.110.MMM )
HDR:                                     ( 00001111 10111010 01.110.MMM )
HDR:                                     ( 00001111 10111010 10.110.MMM )
HDR:  "LOCK BTS m16/32, imm8":           ( 00001111 10111010 00.101.MMM )
HDR:                                     ( 00001111 10111010 01.101.MMM )
HDR:                                     ( 00001111 10111010 10.101.MMM )

Complex Instruction Found, Data not Presented

HDR:  "BSF r16/32,rm16/32":              ( 00001111 10111100 11.ddd.sss )

1     FLOW:       TMP0        =      Port_1.latency_1(REG_sss)
2     FLOW:       REG_ddd     =      Port_01.latency_1(TMP0, REG_ddd)
-----------------------

HDR:  "BSF r16/32,m16/32":               ( 00001111 10111100 00.ddd.MMM )
HDR:                                     ( 00001111 10111100 01.ddd.MMM )
HDR:                                     ( 00001111 10111100 10.ddd.MMM )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP1        =      Port_1.latency_1(TMP0)
3     FLOW:       REG_ddd     =      Port_01.latency_1(TMP1, REG_ddd)
-----------------------

HDR:  "BSR r16/32,rm16/32":              ( 00001111 10111101 11.ddd.sss )

1     FLOW:       TMP1        =      Port_1.latency_1(REG_sss)
2     FLOW:       REG_ddd     =      Port_01.latency_1(TMP1, REG_ddd)
-----------------------

HDR:  "BSR r16/32,m16/32":               ( 00001111 10111101 00.ddd.MMM )
HDR:                                     ( 00001111 10111101 01.ddd.MMM )
HDR:                                     ( 00001111 10111101 10.ddd.MMM )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP1        =      Port_1.latency_1(TMP0)
3     FLOW:       REG_ddd     =      Port_01.latency_1(TMP1, REG_ddd)
-----------------------

HDR:  "PUSHF/PUSHFD":                    ( 10011100 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "POPFD":                           ( 10011101 -------- -------- )

Complex Instruction Found, Data not Presented
```

```
HDR:  "POPF":                            ( 10011101 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "CLC":                             ( 11111000 -------- -------- )

1     FLOW:       sink        =      add.Port_01.latency_1(CONST,CONST_0)
-----------------------

HDR:  "CMC":                             ( 11110101 -------- -------- )

1     FLOW:       sink        =      Port_01.latency_1(ArithFLAGS, CONST)
-----------------------

HDR:  "STC":                             ( 11111001 -------- -------- )

1     FLOW:       sink        =      Port_01.latency_1 (CONST,CONST)
-----------------------

HDR:  "SAHF":                            ( 10011110 -------- -------- )

1     FLOW:       sink        =      Port_01.latency_1(AH)
-----------------------

HDR: "LAHF":                             ( 10011111 -------- -------- )

1     FLOW:       AH          =      Port_01.latency_1(ArithFLAGS, CONST)
-----------------------

HDR:  "CLI":                             ( 11111010 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "STI":                             ( 11111011 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "CLD":                             ( 11111100 -------- -------- )

1     FLOW:       TMP5        =      Port_01.latency_1(ArithFLAGS, SystemFlags)
2     FLOW:       sink        =      move.Port_01.latency_1(CONST)
3     FLOW: SystemFlags       =      and.Port_01.latency_1(TMP5, CONST)
4     FLOW:       sink        =      move.Port_01.latency_1(CONST)
-----------------------

HDR:  "STD":                             ( 11111101 -------- -------- )

1     FLOW:       TMP5        =      Port_01.latency_1(ArithFLAGS, SystemFlags)
2     FLOW:       TMP5        =      Port_01.latency_1(TMP5, 000010000)
3     FLOW: SystemFlags       =      Port_01.latency_1(TMP5, 000001010)
4     FLOW:       sink        =      move.Port_01.latency_1(CONST)
-----------------------

HDR:  "SALC":                            ( 11010110 -------- -------- )
```

```
1     FLOW:       TMP0        =      merge.Port_0.latency_1(ArithFLAGS, CONST)
2     FLOW:       AL          =      Port_01.latency_1(TMP0, CONST)
-----------------------

HDR:  "Call rel16/32 near":               ( 11101000 -------- -------- )

1     FLOW:       sink        =      M_call.Port_1.latency_1(virt_ip)
2     FLOW:       sink        =      store_data.Port_4.latency_1(next_virt_ip)
3     FLOW   sink             =      store_address.Port_3.latency_1( (ESP)
4     FLOW:       ESP         =      sub.Port_01.latency_1(ESP, REG_OP_Size)
-----------------------

HDR:  "Call r16/32 near":                 ( 11111111 11.010.sss -------- )

Complex Instruction Found, Data not Presented

HDR:  "Call m16/32 near":                 ( 11111111 00.010.MMM -------- )
HDR:                                      ( 11111111 01.010.MMM -------- )
HDR:                                      ( 11111111 10.010.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "Ret near":                         ( 11000011 -------- -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1( (ESP)
2     FLOW:       sink        =      Port_01.latency_1(TMP0, CONST)
3     FLOW:       ESP         =      add.Port_01.latency_1(ESP, )
4     FLOW:       sink        =      Port_1.latency_1(virt_ip, TMP0)
-----------------------

HDR:  "RET near iw":                      ( 11000010 -------- -------- )
Complex Instruction Found, Data not Presented

HDR:  "JO  rel8":                         ( 0111.0000 -------- -------- )
HDR:   "JNO rel8":                        ( 0111.0001 -------- -------- )
HDR:   "JB/NAE/C rel8":                   ( 0111.0010 -------- -------- )
HDR:   "JNB/AE/NC rel8":                  ( 0111.0011 -------- -------- )
HDR:   "JE/Z rel8":                       ( 0111.0100 -------- -------- )
HDR:   "JNE/NZ rel8":                     ( 0111.0101 -------- -------- )
HDR:   "JBE/NA rel8":                     ( 0111.0110 -------- -------- )
HDR:   "JNBE/A rel8":                     ( 0111.0111 -------- -------- )
HDR:   "JS rel8":                         (0111.1000 -------- -------- )
HDR:   "JNS rel8":                        ( 0111.1001 -------- -------- )
HDR:   "JP/PE rel8":                      ( 0111.1010 -------- -------- )
HDR:   "JNP/PO rel8":                     ( 0111.1011 -------- -------- )
HDR:   "JL/NGE rel8":                     ( 0111.1100 -------- -------- )
HDR:   "JNL/GE rel8":                     ( 0111.1101 -------- -------- )
HDR:   "JLE/NG rel8":                     ( 0111.1110 -------- -------- )
HDR:   "JNLE/G rel8":                     ( 0111.1111 -------- -------- )
HDR:   "JO  rel16/32":                    ( 00001111 1000.0000 -------- )
HDR:   "JNO rel16/32":                    ( 00001111 1000.0001 -------- )
HDR:   "JB/NAE/C rel16/32":               ( 00001111 1000.0010 -------- )
HDR:   "JNB/AE/NC rel16/32":              ( 00001111 1000.0011 -------- )
HDR:   "JE/Z rel16/32":                   ( 00001111 1000.0100 -------- )
HDR:   "JNE/NZ rel16/32":                 ( 00001111 1000.0101 -------- )
```

```
HDR:   "JBE/NA rel16/32":                ( 00001111 1000.0110 -------- )
HDR:   "JNBE/A rel16/32":                ( 00001111 1000.0111 -------- )
HDR:   "JS rel16/32":                    ( 00001111 1000.1000 -------- )
HDR:   "JNS rel16/32":                   ( 00001111 1000.1001 -------- )
HDR:   "JP/PE rel16/32":                 ( 00001111 1000.1010 -------- )
HDR:   "JNP/PO rel16/32":                ( 00001111 1000.1011 -------- )
HDR:   "JL/NGE rel16/32":                ( 00001111 1000.1100 -------- )
HDR:   "JNL/GE rel16/32":                ( 00001111 1000.1101 -------- )
HDR:   "JLE/NG rel16/32":                ( 00001111 1000.1110 -------- )
HDR:   "JNLE/G rel16/32":                ( 00001111 1000.1111 -------- )

1     FLOW:       sink        =      Port_1.latency_1(ArithFLAGS, virt_ip)
-----------------------

HDR:  "JCXZ/JECXZ rel8":                 ( 11100011 -------- -------- )

1     FLOW:       TMP0        =      sub.Port_01.latency_1(ECX, CONST)
2     FLOW:       sink        =      Port_1.latency_1(TMP0, virt_ip)     1
-----------------------

HDR:  "JMP rel8":                        ( 11101011 -------- -------- )
HDR:  "JMP rel16/32":                    ( 11101001 -------- -------- )

1     FLOW:       sink        =      Port_1.latency_1(virt_ip)
-----------------------

HDR:  "JMP near reg16/32":               ( 11111111 11.100.sss -------- )

1     FLOW: sink              =      Port_1.latency_1(virt_ip, REG_sss)
-----------------------

HDR:  "JMP near m16/32":                 ( 11111111 00.100.MMM -------- )
HDR:                                     ( 11111111 01.100.MMM -------- )
HDR:                                     ( 11111111 10.100.MMM -------- )
Complex Instruction Found, Data not Presented

HDR:  "LOOP rel8":                       ( 11100010 -------- -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "LOOPE rel8":                      ( 11100001 -------- -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "LOOPNE rel8":                     ( 11100000 -------- -------- )

Complex Instruction Found, Data not Presented/n/n

HDR:  "Halt":                            ( 11110100 -------- -------- )

Complex Instruction Found, Data not Presented
```

```
HDR:  "BOUND r16,m16/32&16/32":           ( 0110.0010 00.ddd.MMM -------- )
HDR:                                      ( 0110.0010 01.ddd.MMM -------- )
HDR:                                      ( 0110.0010 10.ddd.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "INTn":                             ( 1100.1101 -------- -------- )

1     FLOW:       TMP0        =      move.Port_01.latency_1(IMM)
2     FLOW:       TMP1        =      move.Port_01.latency_1(000110101)
3     FLOW:       sink        =      Port_0.latency_1(TMP0, TMP1)   1  2
-----------------------

HDR:  "INT0":                             ( 1100.1110 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "INT3":                             ( 1100.1100 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "INT1":                             ( 1111.0001 -------- -------- )

Complex Instruction Found, Data not Presented

-----------------------

HDR:  "IN eAX, imm8":                     ( 1110.010- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "IN eAX, DX":                       ( 1110.110- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "OUT imm8, eAX":                    ( 1110.011- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "OUT DX, eAX":                      ( 1110.111- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "INSB/W/D m8/16/32,DX":             ( 0110.110.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "OUTSB/W/D DX,m8/16/32":            ( 0110.111.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "REP INSB/W/D m8/16/32,DX":         ( 0110.110.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "REP OUTSB/W/D DX,m8/16/32":        ( 0110.111.- -------- -------- )
```

```
Complex Instruction Found, Data not Presented/n/n-----------------------

HDR:  "LEA r16/32,m":                    ( 10001101 00.ddd.sss -------- )
HDR:                                     ( 10001101 01.ddd.sss -------- )
HDR:                                     ( 10001101 10.ddd.sss -------- )

1     FLOW:       REG_ddd     =      load_ea.Port_0.latency_1 (base_BBB)
-----------------------

HDR:  "NOP":                             ( 10010000 -------- -------- )

1     FLOW:       sink        =      move.Port_01.latency_1(CONST)
-----------------------

HDR:  "BSWAP r32":                       ( 00001111 11001.ddd -------- )


1     FLOW:       TMP0        =      move.Port_01.latency_1(REG_ddd)
2     FLOW:       REG_ddd     =      Port_0.latency_1(TMP0)   1
-----------------------

HDR:  "XADD rm8,r8":                     ( 00001111 1100000.0 11.ddd.sss )
HDR:  "XADD rm16/32,r16/32":             ( 00001111 1100000.1 11.ddd.sss )

1     FLOW:       TMP0        =      move.Port_01.latency_1(REG_sss)
2     FLOW:       TMP1        =      move.Port_01.latency_1(REG_ddd)
3     FLOW:       REG_ddd     =      move.Port_01.latency_1(TMP0)   1
4     FLOW:       REG_sss     =      add.Port_01.latency_1(TMP0, TMP1)    1  2
-----------------------

HDR:  "XADD m8,r8":                      ( 00001111 1100000.0 00.ddd.MMM )
HDR:                                     ( 00001111 1100000.0 01.ddd.MMM )
HDR:                                     ( 00001111 1100000.0 10.ddd.MMM )
HDR:  "XADD m16/32,r16/32":              ( 00001111 1100000.1 00.ddd.MMM )
HDR:                                     ( 00001111 1100000.1 01.ddd.MMM )
HDR:                                     ( 00001111 1100000.1 10.ddd.MMM )

 Complex Instruction Found, Data not Presented

HDR:  "LOCK XADD m8,r8":                 ( 00001111 1100000.0 00.ddd.MMM )
HDR:                                     ( 00001111 1100000.0 01.ddd.MMM )
HDR:                                     ( 00001111 1100000.0 10.ddd.MMM )
HDR:  "LOCK XADD m16/32,r16/32":         ( 00001111 1100000.1 00.ddd.MMM )
HDR:                                     ( 00001111 1100000.1 01.ddd.MMM )
HDR:                                     ( 00001111 1100000.1 10.ddd.MMM )

Complex Instruction Found, Data not Presented

HDR:  "CMPXCHG rm8,r8":                   ( 00001111 1011000.0 11.ddd.sss )
HDR:  "CMPXCHG rm16/32,r16/32":           ( 00001111 1011000.1 11.ddd.sss )

Complex Instruction Found, Data not Presented

HDR:  "CMPXCHG m8,r8":                   ( 00001111 1011000.0 00.ddd.MMM )
HDR:                                     ( 00001111 1011000.0 01.ddd.MMM )
```

```
HDR:                                    ( 00001111 1011000.0 10.ddd.MMM )
HDR:  "CMPXCHG m16/32,r16/32":          ( 00001111 1011000.1 00.ddd.MMM )
HDR:                                    ( 00001111 1011000.1 01.ddd.MMM )
HDR:                                    ( 00001111 1011000.1 10.ddd.MMM )

Complex Instruction Found, Data not Presented

HDR:  "LOCK CMPXCHG m8,r8":             ( 00001111 1011000.0 00.ddd.MMM )
HDR:                                    ( 00001111 1011000.0 01.ddd.MMM )
HDR:                                    ( 00001111 1011000.0 10.ddd.MMM )
HDR:  "LOCK CMPXCHG m16/32,r16/32":     ( 00001111 1011000.1 00.ddd.MMM )
HDR:                                    ( 00001111 1011000.1 01.ddd.MMM )
HDR:                                    ( 00001111 1011000.1 10.ddd.MMM )

Complex Instruction Found, Data not Presented

HDR:  "CMPXCHG8B rm64":                 ( 00001111 11000111 00.---.MMM )
HDR:                                    ( 00001111 11000111 01.---.MMM )
HDR:                                    ( 00001111 11000111 10.---.MMM )

Complex Instruction Found, Data not Presented

HDR:  "LOCK CMPXCHG8B rm64":            ( 00001111 11000111 00.---.MMM )
HDR:                                    ( 00001111 11000111 01.---.MMM )
HDR:                                    ( 00001111 11000111 10.---.MMM )

Complex Instruction Found, Data not Presented

HDR:  "XLAT/B":                         ( 11010111 -------- -------- )

1     FLOW:  TMP0            =      move.Port_01.latency_1(AL)
2     FLOW:       AL         =      load.Port_2.latency_1(EBX)
----------------------

HDR:  "CMOVO  r16/32,r16/32":           ( 00001111 0100.0000 11.ddd.sss )
HDR:  "CMOVNO r16/32,r16/32":           ( 00001111 0100.0001 11.ddd.sss )
HDR:  "CMOVB/NAE/C r16/32,r16/32":      ( 00001111 0100.0010 11.ddd.sss )
HDR:  "CMOVNB/AE/NC r16/32,r16/32":     ( 00001111 0100.0011 11.ddd.sss )
HDR:  "CMOVE/Z r16/32,r16/32":          ( 00001111 0100.0100 11.ddd.sss )
HDR:  "CMOVNE/NZ r16/32,r16/32":        ( 00001111 0100.0101 11.ddd.sss )
HDR:  "CMOVBE/NA r16/32,r16/32":        ( 00001111 0100.0110 11.ddd.sss )
HDR:  "CMOVNBE/A r16/32,r16/32":        ( 00001111 0100.1001 11.ddd.sss )
HDR:  "CMOVS r16/32,r16/32":            ( 00001111 0100.1000 11.ddd.sss )
HDR:  "CMOVNS r16/32,r16/32":           ( 00001111 0100.0111 11.ddd.sss )
HDR:  "CMOVP/PE r16/32,r16/32":         ( 00001111 0100.1010 11.ddd.sss )
HDR:  "CMOVNP/PO r16/32,r16/32":        ( 00001111 0100.1011 11.ddd.sss )
HDR:  "CMOVL/NGE r16/32,r16/32":        ( 00001111 0100.1100 11.ddd.sss )
HDR:  "CMOVNL/GE r16/32,r16/32":        ( 00001111 0100.1101 11.ddd.sss )
HDR:  "CMOVLE/NG r16/32,r16/32":        ( 00001111 0100.1110 11.ddd.sss )
HDR:  "CMOVNLE/G r16/32,r16/32":        ( 00001111 0100.1111 11.ddd.sss )


1     FLOW:       TMP0       =      merge.Port_0.latency_1(ArithFLAGS, REG_sss)
2     FLOW:       REG_ddd    =      Port_01.latency_1(TMP0, REG_ddd)    1
----------------------
```

```
HDR:  "CMOVO  r16/32,m16/32":           ( 00001111 0100.0000 00.ddd.MMM )
HDR:                                    ( 00001111 0100.0000 01.ddd.MMM )
HDR:                                    ( 00001111 0100.0000 10.ddd.MMM )
HDR:  "CMOVNO r16/32,m16/32":           ( 00001111 0100.0001 00.ddd.MMM )
HDR:                                    ( 00001111 0100.0001 01.ddd.MMM )
HDR:                                    ( 00001111 0100.0001 10.ddd.MMM )
HDR:  "CMOVB/NAE/C r16/32,m16/32":      ( 00001111 0100.0010 00.ddd.MMM )
HDR:                                    ( 00001111 0100.0010 01.ddd.MMM )
HDR:                                    ( 00001111 0100.0010 10.ddd.MMM )
HDR:  "CMOVNB/AE/NC r16/32,m16/32":     ( 00001111 0100.0011 00.ddd.MMM )
HDR:                                    ( 00001111 0100.0011 01.ddd.MMM )
HDR:                                    ( 00001111 0100.0011 10.ddd.MMM )
HDR:  "CMOVE/Z r16/32,m16/32":          ( 00001111 0100.0100 00.ddd.MMM )
HDR:                                    ( 00001111 0100.0100 01.ddd.MMM )
HDR:                                    ( 00001111 0100.0100 10.ddd.MMM )
HDR:  "CMOVNE/NZ r16/32,m16/32":        ( 00001111 0100.0101 00.ddd.MMM )
HDR:                                    ( 00001111 0100.0101 01.ddd.MMM )
HDR:                                    ( 00001111 0100.0101 10.ddd.MMM )
HDR:  "CMOVBE/NA r16/32,m16/32":        ( 00001111 0100.0110 00.ddd.MMM )
HDR:                                    ( 00001111 0100.0110 01.ddd.MMM )
HDR:                                    ( 00001111 0100.0110 10.ddd.MMM )
HDR:  "CMOVNBE/A r16/32,m16/32":        ( 00001111 0100.1001 00.ddd.MMM )
HDR:                                    ( 00001111 0100.1001 01.ddd.MMM )
HDR:                                    ( 00001111 0100.1001 10.ddd.MMM )
HDR:  "CMOVS r16/32,m16/32":            ( 00001111 0100.1000 00.ddd.MMM )
HDR:                                    ( 00001111 0100.1000 01.ddd.MMM )
HDR:                                    ( 00001111 0100.1000 10.ddd.MMM )
HDR:  "CMOVNS r16/32,m16/32":           ( 00001111 0100.0111 00.ddd.MMM )
HDR:                                    ( 00001111 0100.0111 01.ddd.MMM )
HDR:                                    ( 00001111 0100.0111 10.ddd.MMM )
HDR:  "CMOVP/PE r16/32,m16/32":         ( 00001111 0100.1010 00.ddd.MMM )
HDR:                                    ( 00001111 0100.1010 01.ddd.MMM )
HDR:                                    ( 00001111 0100.1010 10.ddd.MMM )
HDR:  "CMOVNP/PO r16/32,m16/32":        ( 00001111 0100.1011 00.ddd.MMM )
HDR:                                    ( 00001111 0100.1011 01.ddd.MMM )
HDR:                                    ( 00001111 0100.1011 10.ddd.MMM )
HDR:  "CMOVL/NGE r16/32,m16/32":        ( 00001111 0100.1100 00.ddd.MMM )
HDR:                                    ( 00001111 0100.1100 01.ddd.MMM )
HDR:                                    ( 00001111 0100.1100 10.ddd.MMM )
HDR:  "CMOVNL/GE r16/32,m16/32":        ( 00001111 0100.1101 00.ddd.MMM )
HDR:                                    ( 00001111 0100.1101 01.ddd.MMM )
HDR:                                    ( 00001111 0100.1101 10.ddd.MMM )
HDR:  "CMOVLE/NG r16/32,m16/32":        ( 00001111 0100.1110 00.ddd.MMM )
HDR:                                    ( 00001111 0100.1110 01.ddd.MMM )
HDR:                                    ( 00001111 0100.1110 10.ddd.MMM )
HDR:  "CMOVNLE/G r16/32,m16/32":        ( 00001111 0100.1111 00.ddd.MMM )
HDR:                                    ( 00001111 0100.1111 01.ddd.MMM )
HDR:                                    ( 00001111 0100.1111 10.ddd.MMM )

1     FLOW: TMP0              =      load.Port_2.latency_1(MEM)
2     FLOW: TMP0              =      Port_0.latency_1(ArithFLAGS, TMP0)   1
3     FLOW: REG_ddd           =      Port_01.latency_1(TMP0, REG_ddd)     1  2
-----------------------

HDR: "CPUID":                           ( 00001111 1010.0010 -------- )
```

```
Complex Instruction Found, Data not Presented
-----------------------
HDR:  "MOV rm8,r8":                    ( 100010.0.0 11.ddd.sss -------- )
HDR:  "MOV rm16/32,r16/32":            ( 100010.0.1 11.ddd.sss -------- )

1     FLOW:       REG_sss    =      move.Port_01.latency_1(REG_ddd)
-----------------------

HDR:  "MOV m8,r8":                     ( 1000100.0 00.ddd.MMM -------- )
HDR:                                   ( 1000100.0 01.ddd.MMM -------- )
HDR:                                   ( 1000100.0 10.ddd.MMM -------- )
HDR:  "MOV m16/32,r16/32":             ( 1000100.1 00.ddd.MMM -------- )
HDR:                                   ( 1000100.1 01.ddd.MMM -------- )
HDR:                                   ( 1000100.1 10.ddd.MMM -------- )

1     FLOW:      sink        =      store_data.Port_4.latency_1(REG_ddd)
2     FLOW:      sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "MOV r8,rm8":                    ( 100010.1.0 11.ddd.sss -------- )
HDR:  "MOV r16/32,rm16/32":            ( 100010.1.1 11.ddd.sss -------- )

1     FLOW:      REG_ddd     =      move.Port_01.latency_1(REG_sss)
-----------------------

HDR:  "MOV r8,m8":                     ( 1000101.0 00.ddd.MMM -------- )
HDR:                                   ( 1000101.0 01.ddd.MMM -------- )
HDR:                                   ( 1000101.0 10.ddd.MMM -------- )
HDR:  "MOV r16/32,m16/32":             ( 1000101.1 00.ddd.MMM -------- )
HDR:                                   ( 1000101.1 01.ddd.MMM -------- )
HDR:                                   ( 1000101.1 10.ddd.MMM -------- )

1     FLOW:      REG_ddd     =      load.Port_2.latency_1(MEM)
-----------------------

HDR:  "MOV rm8,imm8":                  ( 1100011.0 11.000.sss -------- )
HDR:  "MOV rm16/32,imm16/32":          ( 1100011.1 11.000.sss -------- )

1     FLOW:      REG_sss     =      move.Port_01.latency_1(IMM)
-----------------------

HDR:  "MOV m8,imm8":                   ( 1100011.0 00.000.MMM -------- )
HDR:                                   ( 1100011.0 01.000.MMM -------- )
HDR:                                   ( 1100011.0 10.000.MMM -------- )
HDR:  "MOV m16/32,imm16/32":           ( 1100011.1 00.000.MMM -------- )
HDR:                                   ( 1100011.1 01.000.MMM -------- )
HDR:                                   ( 1100011.1 10.000.MMM -------- )

1     FLOW:      sink        =      store_data.Port_4.latency_1(IMM)
2     FLOW:      sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "MOV r8,imm8":                   ( 1011.0.ddd -------- -------- )
HDR:  "MOV r16/32,imm16/32":           ( 1011.1.ddd -------- -------- )

1     FLOW:      REG_ddd     =      move.Port_01.latency_1(IMM)
```

```
-----------------------

HDR:  "MOV AL,moffs8":                    ( 101000.0.0 -------- -------- )
HDR:  "MOV EAX,moffs16/32":               ( 101000.0.1 -------- -------- )

1     FLOW:       EAX         =      load.Port_2.latency_1(SEG+DISP)
-----------------------

HDR:  "MOV moffs8,AL":                    ( 101000.1.0 -------- -------- )
HDR:  "MOV moffs16/32,EAX":               ( 101000.1.1 -------- -------- )

1     FLOW:       sink        =      store_data.Port_4.latency_1(EAX)
2     FLOW:       sink        =      store_address.Port_3.latency_1(SEG+DISP)
-----------------------

HDR:  "XCHG rm8,r8":                      ( 1000011.0 11.ddd.sss -------- )
HDR:  "XCHG rm16/32,r16/32":              ( 1000011.1 11.ddd.sss -------- )

1     FLOW:       TMP0        =      move.Port_01.latency_1(REG_ddd)
2     FLOW:       REG_ddd     =      move.Port_01.latency_1(REG_sss)
3     FLOW:       REG_sss     =      move.Port_01.latency_1(TMP0)   1
-----------------------

HDR:  "XCHG eAX,r16/32":                  ( 10010.1-- -------- -------- )
HDR:                                      ( 10010.01- -------- -------- )
HDR:                                      ( 10010.001 -------- -------- )

1     FLOW:       TMP0        =      move.Port_01.latency_1(REG_ddd)
2     FLOW:       REG_ddd     =      move.Port_01.latency_1(EAX)
3     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)   1
-----------------------

HDR:  "XCHG m8,r8":                       ( 1000011.0 00.ddd.MMM -------- )
HDR:                                      ( 1000011.0 01.ddd.MMM -------- )
HDR:                                      ( 1000011.0 10.ddd.MMM -------- )
HDR:  "XCHG m16/32,r16/32":               ( 1000011.1 00.ddd.MMM -------- )
HDR:                                      ( 1000011.1 01.ddd.MMM -------- )
HDR:                                      ( 1000011.1 10.ddd.MMM -------- )
HDR:  "LOCK XCHG m8,r8":                  ( 1000011.0 00.ddd.MMM -------- )
HDR:                                      ( 1000011.0 01.ddd.MMM -------- )
HDR:                                      ( 1000011.0 10.ddd.MMM -------- )
HDR:  "LOCK XCHG m16/32,r16/32":          ( 1000011.1 00.ddd.MMM -------- )
HDR:                                      ( 1000011.1 01.ddd.MMM -------- )
HDR:                                      ( 1000011.1 10.ddd.MMM -------- )

Complex Instruction Found, Data not Presented-----------------------

HDR:  "MOV CR0, r32":                     ( 00001111 0010.0010 --.000.sss )

Complex Instruction Found, Data not Presented

HDR:  "MOV CR2, r32":                     ( 00001111 0010.0010 --.010.sss )

Complex Instruction Found, Data not Presented

HDR:  "MOV CR3, r32":                     ( 00001111 0010.0010 --.011.sss )
```

```
Complex Instruction Found, Data not Presented

HDR:  "MOV CR4, r32":                    ( 00001111 0010.0010 --.100.sss )

Complex Instruction Found, Data not Presented

HDR:  "LMSW r16":                        ( 00001111 0000.0001 11.110.sss )

Complex Instruction Found, Data not Presented

HDR:  "LMSW m16":                        ( 00001111 0000.0001 00.110.MMM )
HDR:                                     ( 00001111 0000.0001 01.110.MMM )
HDR:                                     ( 00001111 0000.0001 10.110.MMM )

Complex Instruction Found, Data not Presented

HDR:  "SMSW m16":                        ( 00001111 0000.0001 00.100.MMM )
HDR:                                     ( 00001111 0000.0001 01.100.MMM )
HDR:                                     ( 00001111 0000.0001 10.100.MMM )

Complex Instruction Found, Data not Presented
-----------------------

HDR:  "MOV DRx, r32":                    ( 00001111 0010.0011 --.---.sss )

Complex Instruction Found, Data not Presented

HDR:  "MOV r32, CR0":                    ( 00001111 0010.0000 --.000.sss )
HDR:  "MOV r32, CR2":                    ( 00001111 0010.0000 --.010.sss )
HDR:  "MOV r32, CR3":                    ( 00001111 0010.0000 --.011.sss )
HDR:  "MOV r32, CR4":                    ( 00001111 0010.0000 --.100.sss )

Complex Instruction Found, Data not Presented

HDR:  "MOV r32, DRx":                    ( 00001111 0010.0001 --.---.sss )

Complex Instruction Found, Data not Presented

HDR:  "WRMSR":                           ( 00001111 0011.0000 -------- )

Complex Instruction Found, Data not Presented

HDR:  "RDTSC":                           ( 00001111 0011.0001 -------- )

Complex Instruction Found, Data not Presented

HDR:  "RDPMC":                           ( 00001111 0011.0011 -------- )

Complex Instruction Found, Data not Presented

HDR:  "RDMSR":                           ( 00001111 0011.0010 -------- )

Complex Instruction Found, Data not Presented

HDR:  "CLTS":                            ( 00001111 0000.0110 -------- )
```

```
Complex Instruction Found, Data not Presented
-----------------------

HDR:  "IMUL r16/32,rm16/32":             ( 00001111 10101111 11.ddd.sss )

1     FLOW:       REG_ddd     =      int_mul.Port_0.Latency_4(REG_ddd, REG_sss)
-----------------------

HDR:  "IMUL r16/32,m16/32":              ( 00001111 10101111 00.ddd.MMM )
HDR:                                     ( 00001111 10101111 01.ddd.MMM )
HDR:                                     ( 00001111 10101111 10.ddd.MMM )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       REG_ddd     =      int_mul.Port_0.Latency_4(REG_ddd, TMP1)
-----------------------

HDR:  "IMUL r16/32,rm16/32,imm8/16/32":  ( 011010.s.1 11.ddd.sss -------- )

1     FLOW:       REG_ddd     =      int_mul.Port_0.Latency_4(REG_sss, IMM)
-----------------------

HDR:  "IMUL r16/32,rm16/32,imm8/16/32":  ( 011010.s.1 00.ddd.MMM -------- )
HDR:                                     ( 011010.s.1 01.ddd.MMM -------- )
HDR:                                     ( 011010.s.1 10.ddd.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       REG_ddd     =      int_mul.Port_0.Latency_4(TMP1, IMM)
-----------------------

HDR:  "IMUL rm8":                        ( 1111011.0 11.101.sss -------- )

1     FLOW: AX                =      int_mul.Port_0.Latency_4(AL, REG_sss)
-----------------------

HDR:  "IMUL m8":                         ( 1111011.0 00.101.MMM -------- )
HDR:                                     ( 1111011.0 01.101.MMM -------- )
HDR:                                     ( 1111011.0 10.101.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       AX          =      int_mul.Port_0.Latency_4(AL, TMP1)
-----------------------

HDR:  "IMUL rm16":                       ( 1111011.1 11.101.sss -------- )

1     FLOW:       TMP0        =      int_mul.Port_0.Latency_4(EAX, REG_sss)
2     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
3     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "IMUL m16":                        ( 1111011.1 00.101.MMM -------- )
HDR:                                     ( 1111011.1 01.101.MMM -------- )
HDR:                                     ( 1111011.1 10.101.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      int_mul.Port_0.Latency_4(EAX, TMP1)
```

```
3     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
4     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
-----------------------

HDR:  "IMUL rm32":                        ( 1111011.1 11.101.sss -------- )

1     FLOW:       TMP0        =      int_mul.Port_0.Latency_4(EAX, REG_sss)
2     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
3     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "IMUL m32":                         ( 1111011.1 00.101.MMM -------- )
HDR:                                      ( 1111011.1 01.101.MMM -------- )
HDR:                                      ( 1111011.1 10.101.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      int_mul.Port_0.Latency_4(EAX, TMP1)
3     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
4     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
-----------------------

HDR:  "MUL AL,rm8":                       ( 1111011.0 11.100.sss -------- )

1     FLOW:       AX          =      mul.Port_0.Latency_4(AL, REG_sss)
-----------------------

HDR:  "MUL AL,m8":                        ( 1111011.0 00.100.MMM -------- )
HDR:                                      ( 1111011.0 01.100.MMM -------- )
HDR:                                      ( 1111011.0 10.100.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       AX          =      mul.Port_0.Latency_4(AL, TMP1)
-----------------------

HDR:  "MUL AX,rm16":                      ( 1111011.1 11.100.sss -------- )

1     FLOW:       TMP0        =      mul.Port_0.Latency_4(EAX, REG_sss)
2     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
3     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "MUL AX,m16":                       ( 1111011.1 00.100.MMM -------- )
HDR:                                      ( 1111011.1 01.100.MMM -------- )
HDR:                                      ( 1111011.1 10.100.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      mul.Port_0.Latency_4(EAX, TMP1)
3     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
4     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
-----------------------

HDR:  "MUL EAX,rm32":                     ( 1111011.1 11.100.sss -------- )

1     FLOW:       TMP0        =      mul.Port_0.Latency_4(EAX, REG_sss)
2     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
3     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
```

```
-----------------------

HDR:  "MUL EAX,m32":                        ( 1111011.1 00.100.MMM -------- )
HDR:                                        ( 1111011.1 01.100.MMM -------- )
HDR:                                        ( 1111011.1 10.100.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      mul.Port_0.Latency_4(EAX, TMP1)
3     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
4     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
-----------------------

HDR:  "IDIV AL,rm8":                        ( 1111011.0 11.111.sss -------- )

1     FLOW:       TMP0        =      int_div.Port_0.Latency_99(AX, REG_sss)
2     FLOW:       AL          =      move.Port_01.latency_1(TMP0)
3     FLOW:       AH          =      Port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "IDIV AX,m8":                         ( 1111011.0 00.111.MMM -------- )
HDR:                                        ( 1111011.0 01.111.MMM -------- )
HDR:                                        ( 1111011.0 10.111.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      int_div.Port_0.Latency_99(AX, TMP1)
3     FLOW:       AL          =      move.Port_01.latency_1(TMP0)
4     FLOW:       AH          =      Port_0.latency_1(TMP0, CONST)
-----------------------
HDR:  "IDIV eAX,rm16/32":                   ( 1111011.1 11.111.sss -------- )

1     FLOW:       TMP2        =      Port_0.latency_1(EDX, EAX)
2     FLOW:       TMP0        =      int_div.Port_0.Latency_99(TMP2, REG_sss)
3     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
4     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "IDIV AX,m16/32":                     ( 1111011.1 00.111.MMM -------- )
HDR:                                        ( 1111011.1 01.111.MMM -------- )
HDR:                                        ( 1111011.1 10.111.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP2        =      Port_0.latency_1(EDX, EAX)
3     FLOW:       TMP0        =      int_div.Port_0.Latency_99(TMP2, TMP1)
4     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
-----------------------

HDR:  "DIV AL,rm8":                         ( 1111011.0 11.110.sss -------- )

1     FLOW:       TMP0        =      div.Port_0.Latency_99(AX, REG_sss)
2     FLOW:       AL          =      move.Port_01.latency_1(TMP0)
3     FLOW:       AH          =      Port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "DIV AX,m8":                          ( 1111011.0 00.110.MMM -------- )
HDR:                                        ( 1111011.0 01.110.MMM -------- )
HDR:                                        ( 1111011.0 10.110.MMM -------- )
```

```
1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      div.Port_0.Latency_99(AX, TMP1)
3     FLOW:       AL          =      move.Port_01.latency_1(TMP0)
4     FLOW:       AH          =      Port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "DIV AX,rm16/32":                   ( 1111011.1 11.110.sss -------- )

1     FLOW:       TMP2        =      Port_0.latency_1(EDX, EAX)
2     FLOW:       TMP0        =      div.Port_0.Latency_99(TMP2, REG_sss)
3     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
4     FLOW:       EDX         =      Port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "DIV AX,m16/32":                    ( 1111011.1 00.110.MMM -------- )
HDR:                                      ( 1111011.1 01.110.MMM -------- )
HDR:                                      ( 1111011.1 10.110.MMM -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP2        =      Port_0.latency_1(EDX, EAX)
3     FLOW:       TMP0        =      div.Port_0.Latency_99(TMP2, TMP1)
4     FLOW:       EAX         =      move.Port_01.latency_1(TMP0)
-----------------------

HDR:  "SHL/SAL rm8,1":                    ( 110.1.00.0.0 11.100.sss -------- )
HDR:  "SHL/SAL rm16/32,1":                ( 110.1.00.0.1 11.100.sss -------- )
HDR:  "SAR rm8,1":                        ( 110.1.00.0.0 11.111.sss -------- )
HDR:  "SAR rm16/32,1":                    ( 110.1.00.0.1 11.111.sss -------- )
HDR:  "SHR rm8,1":                        ( 110.1.00.0.0 11.101.sss -------- )
HDR:  "SHR rm16/32,1":                    ( 110.1.00.0.1 11.101.sss -------- )
HDR:  "SHL/SAL rm8,1":                    ( 110.1.00.0.0 11.110.sss -------- )
HDR:  "SHL/SAL rm16/32,1":                ( 110.1.00.0.1 11.110.sss -------- )

1     FLOW:       REG_sss     =      port_0.latency_1(REG_sss, CONST)
-----------------------

HDR:  "SHL/SAL m8,1":                     ( 110.1.00.0.0 00.100.MMM -------- )
HDR:                                      ( 110.1.00.0.0 01.100.MMM -------- )
HDR:                                      ( 110.1.00.0.0 10.100.MMM -------- )
HDR:  "SHL/SAL m16/32,1":                 ( 110.1.00.0.1 00.100.MMM -------- )
HDR:                                      ( 110.1.00.0.1 01.100.MMM -------- )
HDR:                                      ( 110.1.00.0.1 10.100.MMM -------- )
HDR:  "SAR m8,1":                         ( 110.1.00.0.0 00.111.MMM -------- )
HDR:                                      ( 110.1.00.0.0 01.111.MMM -------- )
HDR:                                      ( 110.1.00.0.0 10.111.MMM -------- )
HDR:  "SAR m16/32,1":                     ( 110.1.00.0.1 00.111.MMM -------- )
HDR:                                      ( 110.1.00.0.1 01.111.MMM -------- )
HDR:                                      ( 110.1.00.0.1 10.111.MMM -------- )
HDR:  "SHR m8,1":                         ( 110.1.00.0.0 00.101.MMM -------- )
HDR:                                      ( 110.1.00.0.0 01.101.MMM -------- )
HDR:                                      ( 110.1.00.0.0 10.101.MMM -------- )
HDR:  "SHR m16/32,1":                     ( 110.1.00.0.1 00.101.MMM -------- )
HDR:                                      ( 110.1.00.0.1 01.101.MMM -------- )
HDR:                                      ( 110.1.00.0.1 10.101.MMM -------- )
HDR:  "SHL/SAL m8,1":                     ( 110.1.00.0.0 00.110.MMM -------- )
```

```
HDR:                                     ( 110.1.00.0.0 01.110.MMM -------- )
HDR:                                     ( 110.1.00.0.0 10.110.MMM -------- )
HDR:  "SHL/SAL m16/32,1":                ( 110.1.00.0.1 00.110.MMM -------- )
HDR:                                     ( 110.1.00.0.1 01.110.MMM -------- )
HDR:                                     ( 110.1.00.0.1 10.110.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      port_0.latency_1(TMP0, CONST)
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "SHL/SAL rm8,CL":                  ( 110.1.00.1.0 11.100.sss -------- )
HDR:  "SHL/SAL rm16/32,CL":              ( 110.1.00.1.1 11.100.sss -------- )
HDR:  "SAR rm8,CL":                      ( 110.1.00.1.0 11.111.sss -------- )
HDR:  "SAR rm16/32,CL":                  ( 110.1.00.1.1 11.111.sss -------- )
HDR:  "SHR rm8,CL":                      ( 110.1.00.1.0 11.101.sss -------- )
HDR:  "SHR rm16/32,CL":                  ( 110.1.00.1.1 11.101.sss -------- )
HDR:  "SHL/SAL rm8,CL":                  ( 110.1.00.1.0 11.110.sss -------- )
HDR:  "SHL/SAL rm16/32,CL":              ( 110.1.00.1.1 11.110.sss -------- )

1     FLOW:       REG_sss     =      port_0.latency_1(REG_sss, CL)
-----------------------

HDR:  "SHL/SAL m8,CL":                   ( 110.1.00.1.0 00.100.MMM -------- )
HDR:                                     ( 110.1.00.1.0 01.100.MMM -------- )
HDR:                                     ( 110.1.00.1.0 10.100.MMM -------- )
HDR:  "SHL/SAL m16/32,CL":               ( 110.1.00.1.1 00.100.MMM -------- )
HDR:                                     ( 110.1.00.1.1 01.100.MMM -------- )
HDR:                                     ( 110.1.00.1.1 10.100.MMM -------- )
HDR:  "SAR m8,CL":                       ( 110.1.00.1.0 00.111.MMM -------- )
HDR:                                     ( 110.1.00.1.0 01.111.MMM -------- )
HDR:                                     ( 110.1.00.1.0 10.111.MMM -------- )
HDR:  "SAR m16/32,CL":                   ( 110.1.00.1.1 00.111.MMM -------- )
HDR:                                     ( 110.1.00.1.1 01.111.MMM -------- )
HDR:                                     ( 110.1.00.1.1 10.111.MMM -------- )
HDR:  "SHR m8,CL":                       ( 110.1.00.1.0 00.101.MMM -------- )
HDR:                                     ( 110.1.00.1.0 01.101.MMM -------- )
HDR:                                     ( 110.1.00.1.0 10.101.MMM -------- )
HDR:  "SHR m16/32,CL":                   ( 110.1.00.1.1 00.101.MMM -------- )
HDR:                                     ( 110.1.00.1.1 01.101.MMM -------- )
HDR:                                     ( 110.1.00.1.1 10.101.MMM -------- )
HDR:  "SHL/SAL m8,CL":                   ( 110.1.00.1.0 00.110.MMM -------- )
HDR:                                     ( 110.1.00.1.0 01.110.MMM -------- )
HDR:                                     ( 110.1.00.1.0 10.110.MMM -------- )
HDR:  "SHL/SAL m16/32,CL":               ( 110.1.00.1.1 00.110.MMM -------- )
HDR:                                     ( 110.1.00.1.1 01.110.MMM -------- )
HDR:                                     ( 110.1.00.1.1 10.110.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      port_0.latency_1(TMP0, CL)
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "SHL/SAL rm8,imm8":                ( 110.0.00.0.0 11.100.sss -------- )
```

```
HDR:  "SHL/SAL rm16/32,imm8":            ( 110.0.00.0.1 11.100.sss -------- )
HDR:  "SAR rm8,imm8":                    ( 110.0.00.0.0 11.111.sss -------- )
HDR:  "SAR rm16/32,imm8":                ( 110.0.00.0.1 11.111.sss -------- )
HDR:  "SHR rm8,imm8":                    ( 110.0.00.0.0 11.101.sss -------- )
HDR:  "SHR rm16/32,imm8":                ( 110.0.00.0.1 11.101.sss -------- )
HDR:  "SHL/SAL rm8,imm8":                ( 110.0.00.0.0 11.110.sss -------- )
HDR:  "SHL/SAL rm16/32,imm8":            ( 110.0.00.0.1 11.110.sss -------- )

1     FLOW:       REG_sss     =      port_0.latency_1(REG_sss, IMM)
-----------------------

HDR:  "SHL/SAL m8,imm8":                 ( 110.0.00.0.0 00.100.MMM -------- )
HDR:                                     ( 110.0.00.0.0 01.100.MMM -------- )
HDR:                                     ( 110.0.00.0.0 10.100.MMM -------- )
HDR:  "SHL/SAL m16/32,imm8":             ( 110.0.00.0.1 00.100.MMM -------- )
HDR:                                     ( 110.0.00.0.1 01.100.MMM -------- )
HDR:                                     ( 110.0.00.0.1 10.100.MMM -------- )
HDR:  "SAR m8,imm8":                     ( 110.0.00.0.0 00.111.MMM -------- )
HDR:                                     ( 110.0.00.0.0 01.111.MMM -------- )
HDR:                                     ( 110.0.00.0.0 10.111.MMM -------- )
HDR:  "SAR m16/32,imm8":                 ( 110.0.00.0.1 00.111.MMM -------- )
HDR:                                     ( 110.0.00.0.1 01.111.MMM -------- )
HDR:                                     ( 110.0.00.0.1 10.111.MMM -------- )
HDR:  "SHR m8,imm8":                     ( 110.0.00.0.0 00.101.MMM -------- )
HDR:                                     ( 110.0.00.0.0 01.101.MMM -------- )
HDR:                                     ( 110.0.00.0.0 10.101.MMM -------- )
HDR:  "SHR m16/32,imm8":                 ( 110.0.00.0.1 00.101.MMM -------- )
HDR:                                     ( 110.0.00.0.1 01.101.MMM -------- )
HDR:                                     ( 110.0.00.0.1 10.101.MMM -------- )
HDR:  "SHL/SAL m8,imm8":                 ( 110.0.00.0.0 00.110.MMM -------- )
HDR:                                     ( 110.0.00.0.0 01.110.MMM -------- )
HDR:                                     ( 110.0.00.0.0 10.110.MMM -------- )
HDR:  "SHL/SAL m16/32,imm8":             ( 110.0.00.0.1 00.110.MMM -------- )
HDR:                                     ( 110.0.00.0.1 01.110.MMM -------- )
HDR:                                     ( 110.0.00.0.1 10.110.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      port_0.latency_1(TMP0, IMM)
3     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)
4     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "ROL rm8,1":                       ( 110.1.00.0.0 11.000.sss -------- )
HDR:  "ROL rm16/32,1":                   ( 110.1.00.0.1 11.000.sss -------- )
HDR:  "ROR rm8,1":                       ( 110.1.00.0.0 11.001.sss -------- )
HDR:  "ROR rm16/32,1":                   ( 110.1.00.0.1 11.001.sss -------- )

1     FLOW:       REG_sss     =      port_0.latency_1(REG_sss, CONST)
-----------------------

HDR:  "ROL m8,1":                        ( 110.1.00.0.0 00.000.MMM -------- )
HDR:                                     ( 110.1.00.0.0 01.000.MMM -------- )
HDR:                                     ( 110.1.00.0.0 10.000.MMM -------- )
HDR:  "ROL m16/32,1":                    ( 110.1.00.0.1 00.000.MMM -------- )
HDR:                                     ( 110.1.00.0.1 01.000.MMM -------- )
HDR:                                     ( 110.1.00.0.1 10.000.MMM -------- )
```

```
HDR:  "ROR m8,1":                           ( 110.1.00.0.0 00.001.MMM -------- )
HDR:                                        ( 110.1.00.0.0 01.001.MMM -------- )
HDR:                                        ( 110.1.00.0.0 10.001.MMM -------- )
HDR:  "ROR m16/32,1":                       ( 110.1.00.0.1 00.001.MMM -------- )
HDR:                                        ( 110.1.00.0.1 01.001.MMM -------- )
HDR:                                        ( 110.1.00.0.1 10.001.MMM -------- )

1     FLOW:      TMP0       =      load.Port_2.latency_1(MEM)
2     FLOW:      TMP0       =      port_0.latency_1(TMP0, CONST)
3     FLOW:      sink       =      store_data.Port_4.latency_1(TMP0)
4     FLOW:      sink       =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "ROL rm8,CL":                         ( 110.1.00.1.0 11.000.sss -------- )
HDR:  "ROL rm16/32,CL":                     ( 110.1.00.1.1 11.000.sss -------- )
HDR:  "ROR rm8,CL":                         ( 110.1.00.1.0 11.001.sss -------- )
HDR:  "ROR rm16/32,CL":                     ( 110.1.00.1.1 11.001.sss -------- )

1     FLOW:      REG_sss    =      port_0.latency_1(REG_sss, CL)
-----------------------

HDR:  "ROL m8,CL":                          ( 110.1.00.1.0 00.000.MMM -------- )
HDR:                                        ( 110.1.00.1.0 01.000.MMM -------- )
HDR:                                        ( 110.1.00.1.0 10.000.MMM -------- )
HDR:  "ROL m16/32,CL":                      ( 110.1.00.1.1 00.000.MMM -------- )
HDR:                                        ( 110.1.00.1.1 01.000.MMM -------- )
HDR:                                        ( 110.1.00.1.1 10.000.MMM -------- )
HDR:  "ROR m8,CL":                          ( 110.1.00.1.0 00.001.MMM -------- )
HDR:                                        ( 110.1.00.1.0 01.001.MMM -------- )
HDR:                                        ( 110.1.00.1.0 10.001.MMM -------- )
HDR:  "ROR m16/32,CL":                      ( 110.1.00.1.1 00.001.MMM -------- )
HDR:                                        ( 110.1.00.1.1 01.001.MMM -------- )
HDR:                                        ( 110.1.00.1.1 10.001.MMM -------- )

1     FLOW:  TMP0           =      load.Port_2.latency_1(MEM)
2     FLOW:  TMP0           =      port_0.latency_1(TMP0, CL)
3     FLOW:  sink           =      store_data.Port_4.latency_1(TMP0)
4     FLOW:  sink           =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "ROL rm8,imm8":                       ( 110.0.00.0.0 11.000.sss -------- )
HDR:  "ROL rm16/32,imm8":                   ( 110.0.00.0.1 11.000.sss -------- )
HDR:  "ROR rm8,imm8":                       ( 110.0.00.0.0 11.001.sss -------- )
HDR:  "ROR rm16/32,imm8":                   ( 110.0.00.0.1 11.001.sss -------- )

1     FLOW:      REG_sss    =      port_0.latency_1(REG_sss, IMM)
-----------------------

HDR:  "ROL m8,imm8":                        ( 110.0.00.0.0 00.000.MMM -------- )
HDR:                                        ( 110.0.00.0.0 01.000.MMM -------- )
HDR:                                        ( 110.0.00.0.0 10.000.MMM -------- )
HDR:  "ROL m16/32,imm8":                    ( 110.0.00.0.1 00.000.MMM -------- )
HDR:                                        ( 110.0.00.0.1 01.000.MMM -------- )
HDR:                                        ( 110.0.00.0.1 10.000.MMM -------- )
HDR:  "ROR m8,imm8":                        ( 110.0.00.0.0 00.001.MMM -------- )
HDR:                                        ( 110.0.00.0.0 01.001.MMM -------- )
```

```
HDR:                                     ( 110.0.00.0.0 10.001.MMM -------- )
HDR:  "ROR m16/32,imm8":                 ( 110.0.00.0.1 00.001.MMM -------- )
HDR:                                     ( 110.0.00.0.1 01.001.MMM -------- )
HDR:                                     ( 110.0.00.0.1 10.001.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "RCL rm8,1":                       ( 110.1.00.0.0 11.010.sss -------- )
HDR:  "RCL rm16/32,1":                   ( 110.1.00.0.1 11.010.sss -------- )
HDR:  "RCR rm8,1":                       ( 110.1.00.0.0 11.011.sss -------- )
HDR:  "RCR rm16/32,1":                   ( 110.1.00.0.1 11.011.sss -------- )

1     FLOW:       TMP0        =      Port_01.latency_1(ArithFLAGS, REG_sss)
2     FLOW:       REG_sss     =      port_0.latency_1(TMP0, CONST)
-----------------------

HDR:  "RCL m8,1":                        ( 110.1.00.0.0 00.010.MMM -------- )
HDR:                                     ( 110.1.00.0.0 01.010.MMM -------- )
HDR:                                     ( 110.1.00.0.0 10.010.MMM -------- )
HDR:  "RCL m16/32,1":                    ( 110.1.00.0.1 00.010.MMM -------- )
HDR:                                     ( 110.1.00.0.1 01.010.MMM -------- )
HDR:                                     ( 110.1.00.0.1 10.010.MMM -------- )
HDR:  "RCR m8,1":                        ( 110.1.00.0.0 00.011.MMM -------- )
HDR:                                     ( 110.1.00.0.0 01.011.MMM -------- )
HDR:                                     ( 110.1.00.0.0 10.011.MMM -------- )
HDR:  "RCR m16/32,1":                    ( 110.1.00.0.1 00.011.MMM -------- )
HDR:                                     ( 110.1.00.0.1 01.011.MMM -------- )
HDR:                                     ( 110.1.00.0.1 10.011.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCL rm8,CL":                      ( 110.1.00.1.0 11.010.sss -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCL rm16/32,CL":                  ( 110.1.00.1.1 11.010.sss -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCR rm8,CL":                      ( 110.1.00.1.0 11.011.sss -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCR rm16/32,CL":                  ( 110.1.00.1.1 11.011.sss -------- )

Complex Instruction found, Data not presented

-----------------------
```

```
HDR:  "RCL m8,CL":                        ( 110.1.00.1.0 00.010.MMM -------- )
HDR:                                      ( 110.1.00.1.0 01.010.MMM -------- )
HDR:                                      ( 110.1.00.1.0 10.010.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCL m16/32,CL":                    ( 110.1.00.1.1 00.010.MMM -------- )
HDR:                                      ( 110.1.00.1.1 01.010.MMM -------- )
HDR:                                      ( 110.1.00.1.1 10.010.MMM -------- )

Complex Instruction found, Data not presented

----------------------

HDR:  "RCR m8,CL":                        (110.1.00.1.0 00.011.MMM -------- )
HDR:                                      ( 110.1.00.1.0 01.011.MMM -------- )
HDR:                                      ( 110.1.00.1.0 10.011.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCR m16/32,CL":                    ( 110.1.00.1.1 00.011.MMM -------- )
HDR:                                      ( 110.1.00.1.1 01.011.MMM -------- )
HDR:                                      ( 110.1.00.1.1 10.011.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCL rm8,imm8":                     ( 110.0.00.0.0 11.010.sss -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCL rm16/32,imm8":                 ( 110.0.00.0.1 11.010.sss -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCR rm8,imm8":                     ( 110.0.00.0.0 11.011.sss -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCR rm16/32,imm8":                 ( 110.0.00.0.1 11.011.sss -------- )

Complex Instruction found, Data not presented

-----------------------
```

```
HDR:  "RCL m8,imm8":                     ( 110.0.00.0.0 00.010.MMM -------- )
HDR:                                     ( 110.0.00.0.0 01.010.MMM -------- )
HDR:                                     ( 110.0.00.0.0 10.010.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCL m16/32,imm8":                 ( 110.0.00.0.1 00.010.MMM -------- )
HDR:                                     ( 110.0.00.0.1 01.010.MMM -------- )
HDR:                                     ( 110.0.00.0.1 10.010.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCR m8,imm8":                     ( 110.0.00.0.0 00.011.MMM -------- )
HDR:                                     ( 110.0.00.0.0 01.011.MMM -------- )
HDR:                                     ( 110.0.00.0.0 10.011.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "RCR m16/32,imm8":                 ( 110.0.00.0.1 00.011.MMM -------- )
HDR:                                     ( 110.0.00.0.1 01.011.MMM -------- )
HDR:                                     ( 110.0.00.0.1 10.011.MMM -------- )

Complex Instruction found, Data not presented

-----------------------

HDR:  "SHLD rm16/32,r16/32,imm8":        ( 00001111 10.100.100 11.ddd.sss )

1     FLOW:       TMP0        =      Port_0.latency_1(REG_sss, IMM)
2     FLOW:       REG_sss     =      shl_double.Port_0.latency_1(TMP0, REG_ddd)
        1
-----------------------

HDR:  "SHLD rm16/32,r16/32,CL":          ( 00001111 10.100.101 11.ddd.sss )

1     FLOW:       TMP0        =      Port_0.latency_1(REG_sss, CL)
2     FLOW:       REG_sss     =      shl_double.Port_0.latency_1(TMP0, REG_ddd)
1
-----------------------

HDR:  "SHLD m16/32,r16/32,imm8":         ( 00001111 10.100.100 00.ddd.MMM )
HDR:                                     ( 00001111 10.100.100 01.ddd.MMM )
HDR:                                     ( 00001111 10.100.100 10.ddd.MMM )

Complex Instruction found, Data not presented

-----------------------

HDR:  "SHLD m16/32,r16/32,CL":           ( 00001111 10.100.101 00.ddd.MMM )
```

```
HDR:                                    ( 00001111 10.100.101 01.ddd.MMM )
HDR:                                    ( 00001111 10.100.101 10.ddd.MMM )

Complex Instruction found, Data not presented

-----------------------

HDR:  "SHRD rm16/32,r16/32,imm8":        ( 00001111 10.101.100 11.ddd.sss )

1     FLOW:       TMP0        =      Port_0.latency_1(REG_sss, IMM)
2     FLOW:       REG_sss     =      shr_double.Port_0.latency_1(TMP0, REG_ddd)
1
-----------------------

HDR:  "SHRD rm16/32,r16/32,CL":          ( 00001111 10.101.101 11.ddd.sss )

1     FLOW:       TMP0        =      Port_0.latency_1(REG_sss, CL)
2     FLOW:       REG_sss     =      shr_double.Port_0.latency_1(TMP0, REG_ddd)
1
-----------------------

HDR:  "SHRD m16/32,r16/32,imm8":         ( 00001111 10.101.100 00.ddd.MMM )
HDR:                                    ( 00001111 10.101.100 01.ddd.MMM )
HDR:                                     00001111 10.101.100 10.ddd.MMM )

Complex Instruction found, Data not presented

-----------------------

HDR:  "SHRD m16/32,r16/32,CL":           ( 00001111 10.101.101 00.ddd.MMM )
HDR:                                    ( 00001111 10.101.101 01.ddd.MMM )
HDR:                                    ( 00001111 10.101.101 10.ddd.MMM )

Complex Instruction Found, Data not Presented/n/n

-----------------------

HDR:  "Push r16/32":                     ( 01010.ddd -------- -------- )

1     FLOW:       sink        =      store_data.Port_4.latency_1(REG_ddd)
2     FLOW:       sink        =      store_address.Port_3.latency_1( (ESP)
3     FLOW:       ESP         =      sub.Port_01.latency_1(ESP,REG_OP_Size )
-----------------------

HDR:  "Push r16/32":                     ( 11111111 11.110.sss -------- )

1     FLOW: sink  =      store_data.Port_4.latency_1(REG_sss)
2     FLOW:       sink  =      store_address.Port_3.latency_1( (ESP)
3     FLOW:       ESP   =      sub.Port_01.latency_1(ESP, REG_OP_Size )
-----------------------

HDR:  "Push m16/32":                     ( 11111111 00.110.MMM -------- )
HDR:                                    ( 11111111 01.110.MMM -------- )
HDR:                                    ( 11111111 10.110.MMM -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1(MEM)
```

```
2     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)    1
3     FLOW:       sink        =      store_address.Port_3.latency_1( (ESP)
4     FLOW:       ESP         =      sub.Port_01.latency_1(ESP, REG_OP_Size)
-----------------------

HDR:  "Push imm8":                        ( 011010.1.0 -------- -------- )
HDR:  "Push imm16/32":                    ( 011010.0.0 -------- -------- )

1     FLOW:       sink        =      store_data.Port_4.latency_1(IMM)
2     FLOW:       sink        =      store_address.Port_3.latency_1( (ESP)
3     FLOW:       ESP         =      sub.Port_01.latency_1(ESP, REG_OP_Size )
-----------------------

HDR:  "Pop r16/32":                       ( 01011.0-- -------- -------- )
HDR:                                      ( 01011.101 -------- -------- )
HDR:                                      ( 01011.11- -------- -------- )

1     FLOW:       REG_ddd     =      load.Port_2.latency_1( (ESP)
2     FLOW:       ESP         =      add.Port_01.latency_1(ESP, REG_OP_Size )
-----------------------

HDR:  "Pop eSP":                          ( 10001111 11.000.100 -------- )
HDR:                                      ( 01011.100 -------- -------- )

1     FLOW:       TMP0        =      load.Port_2.latency_1( (ESP)
2     FLOW:       ESP         =      add.Port_01.latency_1(ESP, REG_OP_Size )
3     FLOW:       ESP         =      move.Port_01.latency_1(TMP0)
-----------------------

HDR:  "Pop r16/32":                       ( 10001111 11.000.0-- -------- )
HDR:                                      ( 10001111 11.000.101 -------- )
HDR:                                      ( 10001111 11.000.11- -------- )

1     FLOW:       REG_sss     =      load.Port_2.latency_1( (ESP)
2     FLOW:       ESP         =      add.Port_01.latency_1(ESP, REG_OP_Size )
-----------------------

HDR:  "Pop m16/32":                       ( 10001111 00.000.MMM -------- )
HDR:                                      ( 10001111 01.000.MMM -------- )
HDR:                                      ( 10001111 10.000.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "PUSHA/PUSHAD":                     ( 01100000 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "POPA/POPAD":                       ( 01100001 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "ENTER":                            ( 11001000 -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "leave":                            ( 11001001 -------- -------- )
```

```
1     FLOW:       TMP0        =      move.Port_01.latency_1(EBP)
2     FLOW:       EBP         =      load.Port_2.latency_1( (TMP0)
3     FLOW:       ESP         =      add.Port_01.latency_1(TMP0, REG_OP_Size)

-----------------------

HDR:  "MOVSB/W/D m8/16/32,m8/16/32":      ( 1010010.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "CMPSB/W/D m8/16/32,m8/16/32":      ( 1010011.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "SCASB/W/D m8/16/32,m8/16/32":      ( 1010111.- -------- -------- )

1     FLOW:       TMP1        =      load.Port_2.latency_1 (EDI)
2     FLOW:       TMP2        =      sub.Port_01.latency_1(EAX, TMP1)
3     FLOW:       EDI         =      Port_2.latency_1(LINSEG_SUPOVR: (EDI)
-----------------------

HDR:  "LODSB/W/D m8/16/32,m8/16/32":      ( 1010110.- -------- -------- )

1     FLOW:       EAX         =      load.Port_2.latency_1(ESI)
2     FLOW:       ESI         =      Port_2.latency_1(*LINSEG_SUPOVR: (ESI)
-----------------------

HDR: "STOSB/W/D m8/16/32,m8/16/32":       ( 1010101.- -------- -------- )

1     FLOW:       sink        =      store_data.Port_4.latency_1(EAX)
2     FLOW:       sink        =      store_address.Port_3.latency_1(EDI)
3     FLOW:       EDI         =      Port_2.latency_1(*LINSEG_SUPOVR: (EDI)
-----------------------

HDR:  "REP MOVSB/W/D m8/16/32,m8/16/32":  ( 1010010.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "REP CMPSB/W/D m8/16/32,m8/16/32":  ( 1010011.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "REP SCASB/W/D m8/16/32,m8/16/32":  ( 1010111.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "REP LODSB/W/D m8/16/32,m8/16/32":  ( 1010110.- -------- -------- )

Complex Instruction Found, Data not Presented

HDR:  "REP STOSB/W/D m8/16/32,m8/16/32":  ( 1010101.- -------- -------- )

Complex Instruction Found, Data not Presented-----------------------

HDR: "MOVSX r16,rm8":                     ( 00001111 1011.1.11.0 11.ddd.sss )
```

```
1     FLOW:       REG_ddd     =      move.Port_01.latency_1(REG_sss)
-----------------------

HDR: "MOVSX r32,rm8":                     ( 00001111 1011.1.11.0 11.ddd.sss )

1     FLOW:       REG_ddd     =      move.Port_01.latency_1(REG_sss)
-----------------------

HDR: "MOVSX r32,rm16":                    ( 00001111 1011.1.11.1 11.ddd.sss )

1     FLOW:       REG_ddd     =      move.Port_01.latency_1(REG_sss)
-----------------------

HDR: "MOVSX r16,m8":                      ( 00001111 1011.1.11.0 00.ddd.MMM )
HDR:                                      ( 00001111 1011.1.11.0 01.ddd.MMM )
HDR:                                      ( 00001111 1011.1.11.0 10.ddd.MMM )

1     FLOW:       REG_ddd     =      load.Port_2.latency_1(MEM)
-----------------------

HDR:  "MOVSX r32,m8":                     ( 00001111 1011.1.11.0 00.ddd.MMM )
HDR:                                      ( 00001111 1011.1.11.0 01.ddd.MMM )
HDR:                                      ( 00001111 1011.1.11.0 10.ddd.MMM )

1     FLOW:       REG_ddd     =      load.Port_2.latency_1(MEM)
-----------------------

HDR:  "MOVSX r16/32,m16":                 ( 00001111 1011.1.11.1 00.ddd.MMM )
HDR:                                      ( 00001111 1011.1.11.1 01.ddd.MMM )
HDR:                                      ( 00001111 1011.1.11.1 10.ddd.MMM )

1     FLOW:       REG_ddd     =      load.Port_2.latency_1(MEM)
-----------------------

HDR:  "MOVZX r16,rm8":                    ( 00001111 1011.0.11.0 11.ddd.sss )

1     FLOW:       REG_ddd     =      move.Port_01.latency_1(REG_sss)
-----------------------

HDR:  "MOVZX r32,rm8":                    ( 00001111 1011.0.11.0 11.ddd.sss )

1     FLOW:       REG_ddd     =      move.Port_01.latency_1(REG_sss)
-----------------------

HDR:  "MOVZX r32,rm16":                   ( 00001111 1011.0.11.1 11.ddd.sss )

1     FLOW:       REG_ddd     =      move.Port_01.latency_1(REG_sss)
-----------------------

HDR:  "MOVZX r16,m8":                     ( 00001111 1011.0.11.0 00.ddd.MMM )
HDR:                                      ( 00001111 1011.0.11.0 01.ddd.MMM )
HDR:                                      ( 00001111 1011.0.11.0 10.ddd.MMM )

1     FLOW:       REG_ddd     =      load.Port_2.latency_1(MEM)
-----------------------
```

```
HDR:  "MOVZX r32,m8":                        ( 00001111 1011.0.11.0 00.ddd.MMM )
HDR:                                         ( 00001111 1011.0.11.0 01.ddd.MMM )
HDR:                                         ( 00001111 1011.0.11.0 10.ddd.MMM )

1     FLOW:       REG_ddd     =      load.Port_2.latency_1(MEM)
-----------------------

HDR:  "MOVZX r32,m16":                       ( 00001111 1011.0.11.1 00.ddd.MMM )
HDR:                                         ( 00001111 1011.0.11.1 01.ddd.MMM )
HDR:                                         ( 00001111 1011.0.11.1 10.ddd.MMM )

1     FLOW:       REG_ddd     =      load.Port_2.latency_1(MEM)
-----------------------

HDR:   "CWD/CDQ":                            ( 10011001 -------- -------- )

1     FLOW:       EDX         =      sar.Port_0.latency_1(EAX, CONST)
-----------------------

HDR:  "CBW":                                 ( 10011000 -------- -------- )

1     FLOW:       AX          =      move.Port_01.latency_1(AL)
-----------------------

HDR:  "CWDE":                                ( 10011000 -------- -------- )

1     FLOW: EAX               =      move.Port_01.latency_1(AX)
-----------------------

HDR:  "FADD ST,ST( i)":                      ( 11011.00.0 )
HDR:  "FMUL ST,ST( i)":                      ( 11011.00.0 )
HDR:  "FSUB ST,ST( i)":                      ( 11011.00.0 )
HDR:  "FDIV ST,ST( i)":                      ( 11011.00.0 )

1     FLOW:       ST0         =      port_0.latency_4_or_5_or_99(ST0, ST(i))
-----------------------

HDR:  "FADD ST( i),ST":                      ( 11011.10.0 )
HDR:  "FMUL ST( i),ST":                      ( 11011.10.0 )
HDR:  "FSUBR ST( i),ST":                     ( 11011.10.0 )
HDR:  "FDIVR ST( i),ST":                     ( 11011.10.0 )

1     FLOW:       ST(i)       =      port_0.latency_4_or_5_or_99(ST0, ST(i))
-----------------------

HDR:  "FADDP ST( i),ST":                     ( 11011.11.0 )
HDR:  "FSUBP ST( i),ST":                     ( 11011.11.0 )
HDR:  "FMULP ST( i),ST":                     ( 11011.11.0 )
HDR:  "FDIVP ST( i),ST":                     ( 11011.11.0 )

1     FLOW:       ST(i)       =      port_0.latency_4_or_5_or_99(ST(i), ST0)
-----------------------

HDR:  "FSUB ST( i),ST":                      ( 11011.10.0 )
HDR:  "FDIV ST( i),ST":                      ( 11011.10.0 )
```

```
1     FLOW:      ST(i)      =     port_0.latency_4_or_5_or_99(ST(i), ST0)
-----------------------

HDR:  "FSUBR ST,ST( i)":               ( 11011.00.0 )
HDR:  "FDIVR ST,ST( i)":               ( 11011.00.0 )

1     FLOW:      ST0        =     port_0.latency_4_or_5_or_99(ST(i), ST0)
-----------------------

HDR:  "FSUBRP ST( i),ST":              ( 11011.11.0 )
HDR:  "FDIVRP ST( i),ST":              ( 11011.11.0 )

1     FLOW:      ST(i)      =     port_0.latency_4_or_5_or_99(ST0, ST(i))
-----------------------

HDR:  "FADD m32real":                  ( 11011.00.0 00.000.sss -------- )
HDR:                                   ( 11011.00.0 01.000.sss -------- )
HDR:                                   ( 11011.00.0 10.000.sss -------- )
HDR:  "FADD m64real":                  ( 11011.10.0 00.000.sss -------- )
HDR:                                   ( 11011.10.0 01.000.sss -------- )
HDR:                                   ( 11011.10.0 10.000.sss -------- )
HDR:  "FSUB m32real":                  ( 11011.00.0 00.100.sss -------- )
HDR:                                   ( 11011.00.0 01.100.sss -------- )
HDR:                                   ( 11011.00.0 10.100.sss -------- )
HDR:  "FSUB m64real":                  ( 11011.10.0 00.100.sss -------- )
HDR:                                   ( 11011.10.0 01.100.sss -------- )
HDR:                                   ( 11011.10.0 10.100.sss -------- )
HDR:  "FMUL m32real":                  ( 11011.00.0 00.001.sss -------- )
HDR:                                   ( 11011.00.0 01.001.sss -------- )
HDR:                                   ( 11011.00.0 10.001.sss -------- )
HDR:  "FMUL m64real":                  ( 11011.10.0 00.001.sss -------- )
HDR:                                   ( 11011.10.0 01.001.sss -------- )
HDR:                                   ( 11011.10.0 10.001.sss -------- )
HDR:  "FDIV m32real":                  ( 11011.00.0 00.110.sss -------- )
HDR:                                   ( 11011.00.0 01.110.sss -------- )
HDR:                                   ( 11011.00.0 10.110.sss -------- )
HDR:  "FDIV m64real":                  ( 11011.10.0 00.110.sss -------- )
HDR:                                   ( 11011.10.0 01.110.sss -------- )
HDR:                                   ( 11011.10.0 10.110.sss -------- )

1     FLOW:      TMP0       =     fp_load.Port_2.latency_1(MEM)
2     FLOW:      ST0        =     port_0.latency_4_or_5_or_99(ST0, TMP0)   1
-----------------------

HDR:  "FSUBR m32real":                 ( 11011.00.0 00.101.sss -------- )
HDR:                                   ( 11011.00.0 01.101.sss -------- )
HDR:                                   ( 11011.00.0 10.101.sss -------- )
HDR:  "FSUBR m64real":                 ( 11011.10.0 00.101.sss -------- )
HDR:                                   ( 11011.10.0 01.101.sss -------- )
HDR:                                   ( 11011.10.0 10.101.sss -------- )
HDR:  "FDIVR m32real":                 ( 11011.00.0 00.111.sss -------- )
HDR:                                   ( 11011.00.0 01.111.sss -------- )
HDR:                                   ( 11011.00.0 10.111.sss -------- )
HDR:  "FDIVR m64real":                 ( 11011.10.0 00.111.sss -------- )
HDR:                                   ( 11011.10.0 01.111.sss -------- )
```

```
HDR:                                      ( 11011.10.0 10.111.sss -------- )

1     FLOW:       TMP0        =      fp_load.Port_2.latency_1(MEM)
2     FLOW:       ST0         =      port_0.latency_4_or_5_or_99(TMP0, ST0)   1
-----------------------

HDR:  "FIADD m32int":                     ( 11011.01.0 00.000.sss -------- )
HDR:                                      ( 11011.01.0 01.000.sss -------- )
HDR:                                      ( 11011.01.0 10.000.sss -------- )
HDR:  "FIADD m16int":                     ( 11011.11.0 00.000.sss -------- )
HDR:                                      ( 11011.11.0 01.000.sss -------- )
HDR:                                      ( 11011.11.0 10.000.sss -------- )
HDR:  "FISUB m32int":                     ( 11011.01.0 00.100.sss -------- )
HDR:                                      ( 11011.01.0 01.100.sss -------- )
HDR:                                      ( 11011.01.0 10.100.sss -------- )
HDR:  "FISUB m16int":                     ( 11011.11.0 00.100.sss -------- )
HDR:                                      ( 11011.11.0 01.100.sss -------- )
HDR:                                      ( 11011.11.0 10.100.sss -------- )
HDR:  "FIMUL m32int":                     ( 11011.01.0 00.001.sss -------- )
HDR:                                      ( 11011.01.0 01.001.sss -------- )
HDR:                                      ( 11011.01.0 10.001.sss -------- )
HDR:  "FIMUL m16int":                     ( 11011.11.0 00.001.sss -------- )
HDR:                                      ( 11011.11.0 01.001.sss -------- )
HDR:                                      ( 11011.11.0 10.001.sss -------- )
HDR:  "FIDIV m32int":                     ( 11011.01.0 00.110.sss -------- )
HDR:                                      ( 11011.01.0 01.110.sss -------- )
HDR:                                      ( 11011.01.0 10.110.sss -------- )
HDR:  "FIDIV m16int":                     ( 11011.11.0 00.110.sss -------- )
HDR:                                      ( 11011.11.0 01.110.sss -------- )
HDR:                                      ( 11011.11.0 10.110.sss -------- )

Complex Instruction Found, Data not Presented

HDR:  "FISUBR m32int":                    ( 11011.01.0 00.101.sss -------- )
HDR:                                      ( 11011.01.0 01.101.sss -------- )
HDR:                                      ( 11011.01.0 10.101.sss -------- )
HDR:  "FISUBR m16int":                    ( 11011.11.0 00.101.sss -------- )
HDR:                                      ( 11011.11.0 01.101.sss -------- )
HDR:                                      ( 11011.11.0 10.101.sss -------- )
HDR:  "FIDIVR m32int":                    ( 11011.01.0 00.111.sss -------- )
HDR:                                      ( 11011.01.0 01.111.sss -------- )
HDR:                                      ( 11011.01.0 10.111.sss -------- )
HDR:  "FIDIVR m16int":                    ( 11011.11.0 00.111.sss -------- )
HDR:                                      ( 11011.11.0 01.111.sss -------- )
HDR:                                      ( 11011.11.0 10.111.sss -------- )

Complex Instruction Found, Data not Presented

HDR:  "FSQRT":                            ( 11011.001 11111010 -------- )

1     FLOW:       ST0         =      fp_sqrt.Port_0.Latency_66(ST0, CONST)
-----------------------

HDR:  "FCOM m32real":                     ( 11011.00.0 00.010.MMM -------- )
HDR:                                      ( 11011.00.0 01.010.MMM -------- )
HDR:                                      ( 11011.00.0 10.010.MMM -------- )
```

```
HDR:  "FCOM m64real":                     ( 11011.10.0 00.010.MMM -------- )
HDR:                                      ( 11011.10.0 01.010.MMM -------- )
HDR:                                      ( 11011.10.0 10.010.MMM -------- )

1     FLOW:       TMP0        =      fp_load.Port_2.latency_1(MEM)
2     FLOW:       sink  =      fp_compare.Port_0.latency_1(ST0, TMP0)
-----------------------

HDR:  "FCOM STi":                         ( 11011.000 11.010.iii -------- )
HDR:  "FCOM2 STi":                        ( 11011.100 11.010.iii -------- )

1     FLOW:       sink        =      fp_compare.Port_0.latency_1(ST0, ST(i))
-----------------------

HDR:  "FCOMP m32real":                    ( 11011.00.0 00.011.MMM -------- )
HDR:                                      ( 11011.00.0 01.011.MMM -------- )
HDR:                                      ( 11011.00.0 10.011.MMM -------- )
HDR:  "FCOMP m64real":                    ( 11011.10.0 00.011.MMM -------- )
HDR:                                      ( 11011.10.0 01.011.MMM -------- )
HDR:                                      ( 11011.10.0 10.011.MMM -------- )

1     FLOW:       TMP0        =      fp_load.Port_2.latency_1(MEM)
2     FLOW:       sink        =      fp_compare.Port_0.latency_1(ST0, TMP0)
-----------------------

HDR:  "FCOMP STi":                        ( 11011.000 11.011.iii -------- )
HDR:  "FCOMP3 STi":                       ( 11011.100 11.011.iii -------- )
HDR:  "FCOMP5 STi":                       ( 11011.110 11.010.iii -------- )

1     FLOW:       sink        =      fp_compare.Port_0.latency_1(ST0, ST(i))
-----------------------

HDR:  "FCOMPP":                           ( 11011.110 1101.1001 -------- )

1     FLOW:       sink        =      fp_compare.Port_0.latency_1(ST0, ST1)
2     FLOW:       sink        =      move.Port_01.latency_1(ST0)
-----------------------

HDR:  "FICOM m32int":                     ( 11011.01.0 00.010.MMM -------- )
HDR:                                      ( 11011.01.0 01.010.MMM -------- )
HDR:                                      ( 11011.01.0 10.010.MMM -------- )
HDR:  "FICOM m16int":                     ( 11011.11.0 00.010.MMM -------- )
HDR:                                      ( 11011.11.0 01.010.MMM -------- )
HDR:                                      ( 11011.11.0 10.010.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "FICOMP m32int":                    ( 11011.01.0 00.011.MMM -------- )
HDR:                                      ( 11011.01.0 01.011.MMM -------- )
HDR:                                      ( 11011.01.0 10.011.MMM -------- )
HDR:  "FICOMP m16int":                    ( 11011.11.0 00.011.MMM -------- )
HDR:                                      ( 11011.11.0 01.011.MMM -------- )
HDR:                                      ( 11011.11.0 10.011.MMM -------- )

Complex Instruction Found, Data not Presented
```

```
HDR:  "FTST":                            ( 11011.001 1110.0100 -------- )

1     FLOW:       sink        =      fp_compare.Port_0.latency_1(ST0, CONST)
-----------------------

HDR:  "FUCOM STi":                       ( 11011.101 11.100.iii -------- )

1     FLOW:       sink        =      fp_compare.Port_0.latency_1(ST0, ST(i))
-----------------------

HDR:  "FUCOMP STi":                      ( 11011.101 11.101.iii -------- )

1     FLOW:       sink        =      fp_compare.Port_0.latency_1(ST0, ST(i))
-----------------------

HDR:  "FUCOMPP":                         ( 11011.010 1110.1001 -------- )

1     FLOW:       sink        =      fp_compare.Port_0.latency_1(ST0, ST1)
2     FLOW:       sink        =      move.Port_01.latency_1(ST0)
-----------------------

HDR:  "FXAM":                            ( 11011.001 1110.0101 -------- )

1     FLOW:       sink        =      fp_examine.Port_0.Latency_3(ST0)
-----------------------

HDR:  "FCOMI STi":                       ( 11011.011 11.110.iii -------- )

1     FLOW:       sink        = fp_compare.Port_(not found).latency_1(ST0, ST(i))
-----------------------

HDR:  "FCOMIP STi":                      ( 11011.111 11.110.iii -------- )

1     FLOW:       sink        = fp_compare.Port_(not found).latency_1(ST0, ST(i))
-----------------------

HDR:  "FUCOMI STi":                      ( 11011.011 11.101.iii -------- )

1     FLOW: sink              = fp_compare.Port_(not found).latency_1(ST0, ST(i))
-----------------------

HDR:  "FUCOMIP STi":                     ( 11011.111 11.101.iii -------- )

1     FLOW:       sink        = fp_compare.Port_(not found).latency_1(ST0, ST(i))
-----------------------

HDR:  "FLDZ":                            ( 11011001 11101.110 -------- )

1     FLOW:       ST7         =      fp_move.Port_0.latency_1(CONST)
-----------------------

HDR:  "FLD1":                            ( 11011001 11101.000 -------- )

1     FLOW:       TMP0        =      freadrom.Port_(not found).latency_1(CONST)
2     FLOW:       ST7         =      fp_move.Port_0.latency_1(TMP0)      1
-----------------------
```

```
HDR:  "FLDPI":                           ( 11011001 11101.011 -------- )

1     FLOW:       TMP0        =      freadrom.Port_(not found).latency_1(CONST)
2     FLOW:       ST7         =      fp_normalize.Port_0.Latency_3(TMP0)
-----------------------

HDR:  "FLDL2T":                          ( 11011001 11101.001 -------- )


1     FLOW:       TMP0        =      freadrom.Port_(not found).latency_1(CONST)
2     FLOW:       ST7         =      fp_normalize.Port_0.Latency_3(TMP0)
-----------------------

HDR:  "FLDL2E":                          ( 11011001 11101.010 -------- )


1     FLOW:       TMP0        =      freadrom.Port_(not found).latency_1(CONST)
2     FLOW:       ST7         =      fp_normalize.Port_0.Latency_3(TMP0)
-----------------------

HDR:  "FLDLG2":                          ( 11011001 11101.100 -------- )


1     FLOW: TMP0              =      freadrom.Port_(not found).latency_1(CONST)
2     FLOW: ST7               =      fp_normalize.Port_0.Latency_3(TMP0)
-----------------------

HDR:  "FLDLN2":                          ( 11011001 11101.101 -------- )


1     FLOW: TMP0              =      freadrom.Port_(not found).latency_1(CONST)
2     FLOW: ST7               =      fp_normalize.Port_0.Latency_3(TMP0)
-----------------------

HDR:  "FNINIT":                          ( 11011.011 11100011 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FNSTCW m2byte":                   ( 11011.001 00.111.MMM -------- )
HDR:                                     ( 11011.001 01.111.MMM -------- )
HDR:                                     ( 11011.001 10.111.MMM -------- )

1     FLOW:       TMP0        =      Port_(not found).latency_1(CONST, CONST)
2     FLOW:       sink        =      store_data.Port_4.latency_1(TMP0)    1
3     FLOW:       sink        =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "FNSTENV m14/28byte":              ( 11011.001 00.110.MMM -------- )
HDR:                                     ( 11011.001 01.110.MMM -------- )
HDR:                                     ( 11011.001 10.110.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "FNSAVE m94/108byte":              ( 11011.101 00.110.MMM -------- )
HDR:                                     ( 11011.101 01.110.MMM -------- )
```

```
HDR:                                    ( 11011.101 10.110.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "FLDCW m2byte":                   ( 11011.001 00.101.MMM -------- )
HDR:                                    ( 11011.001 01.101.MMM -------- )
HDR:                                    ( 11011.001 10.101.MMM -------- )

1     FLOW:      TMP0       =      load.Port_2.latency_1(MEM)
2     FLOW:      sink       =      Port_(not found).latency_1(CONST, TMP0)
3     FLOW:      sink       =      move.Port_01.latency_1(CONST)
-----------------------

HDR:  "FRSTOR m14/28byte":              ( 11011.001 00.100.MMM -------- )
HDR:                                    ( 11011.001 01.100.MMM -------- )
HDR:                                    ( 11011.001 10.100.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "FRSTOR m94/108byte":             ( 11011.101 00.100.MMM -------- )
HDR:                                    ( 11011.101 01.100.MMM -------- )
HDR:                                    ( 11011.101 10.100.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "FNCLEX ":                        ( 11011.011 11100010 -------- )

1     FLOW:      sink       =      move.Port_01.latency_1(CONST)
2     FLOW:      FSW        =      and.Port_01.latency_1(FSW, CONST)
3     FLOW:      sink       =      move.Port_01.latency_1(CONST)
-----------------------

HDR:  "FNSTSW m2byte":                  ( 11011.101 00.111.MMM -------- )
HDR:                                    ( 11011.101 01.111.MMM -------- )
HDR:                                    ( 11011.101 10.111.MMM -------- )

1     FLOW:      TMP0       =      Port_0.latency_1(FCC, FSW)
2     FLOW:      sink       =      store_data.Port_4.latency_1(TMP0)
3     FLOW:      sink       =      store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "FNSTSW AX":                      ( 11011.111 11100000 -------- )

1     FLOW:      TMP0       =      Port_0.latency_1(FCC, FSW)
2     FLOW:      TMP1       =      shr.Port_0.latency_1(EAX, CONST)
3     FLOW:      EAX        =      Port_0.latency_1(TMP1, TMP0)
-----------------------

HDR:  "FDISI":                          ( 11011.011 11100001 -------- )

1     FLOW: sink            =      move.Port_01.latency_1(CONST)
-----------------------

HDR:  "FENI":                           ( 11011.011 11100000 -------- )

1     FLOW: sink            =      move.Port_01.latency_1(CONST)
```

```
-----------------------

HDR:  "FSETPM":                         ( 11011.011 11100100 -------- )

1     FLOW: sink             =      move.Port_01.latency_1(CONST)
-----------------------

HDR:  "FINCSTP":                        ( 11011.001 1111011.1 -------- )

1     FLOW: sink             =      fp_move.Port_0.latency_1(CONST)
-----------------------

HDR:  "FDECSTP":                        ( 11011.001 1111011.0 -------- )

1     FLOW:       sink       =      fp_move.Port_0.latency_1(CONST)
-----------------------

HDR:  "FFREE ST( i)":                   ( 11011.101 )

1     FLOW: sink             =      fp_move.Port_0.latency_1(ST(i))
-----------------------

HDR:  "FFREEP ST( i)":                  ( 11011.111 )

1     FLOW:       sink       =      fp_move.Port_0.latency_1(ST(i))
2     FLOW:       sink       =      fp_move.Port_0.latency_1(ST0)
-----------------------

HDR:  "FNOP":                           ( 11011.001 11010000 -------- )

1     FLOW:       sink       =      fp_move.Port_0.latency_1(CONST)
-----------------------

HDR:  "FWAIT":                          ( 10011011 -------- -------- )

1     FLOW:       sink       =      move.Port_01.latency_1(CONST)
2     FLOW:       sink  =      move.Port_01.latency_1(CONST)
-----------------------

HDR:  "FILD m16int":                    ( 11011.11.1 00.000.MMM -------- )
HDR:                                    ( 11011.11.1 01.000.MMM -------- )
HDR:                                    ( 11011.11.1 10.000.MMM -------- )
HDR:  "FILD m32int":                    ( 11011.01.1 00.000.MMM -------- )
HDR:                                    ( 11011.01.1 01.000.MMM -------- )
HDR:                                    ( 11011.01.1 10.000.MMM -------- )
HDR:  "FILD m64int":                    ( 11011.11.1 00.101.MMM -------- )
HDR:                                    ( 11011.11.1 01.101.MMM -------- )
HDR:                                    ( 11011.11.1 10.101.MMM -------- )

1     FLOW:       TMP1       =      fp_load.Port_2.latency_1(MEM)
2     FLOW:       TMP0       =      freadrom.Port_(not found).latency_1(CONST)
3     FLOW:       TMP2       =      Port_0.latency_1(TMP1)
4     FLOW:       ST7        =      fp_convert.Port_0.Latency_3(TMP0, TMP2)
-----------------------

HDR:  "FLD m32real":                    ( 11011.00.1 00.000.MMM -------- )
```

```
HDR:                                      ( 11011.00.1 01.000.MMM -------- )
HDR:                                      ( 11011.00.1 10.000.MMM -------- )
HDR:  "FLD m64real":                      ( 11011.10.1 00.000.MMM -------- )
HDR:                                      ( 11011.10.1 01.000.MMM -------- )
HDR:                                      ( 11011.10.1 10.000.MMM -------- )

1     FLOW:       ST7         =      fp_load.Port_2.latency_1(MEM)
-----------------------

HDR:  "FLD m80real":                      ( 11011.01.1 00.101.MMM -------- )
HDR:                                      ( 11011.01.1 01.101.MMM -------- )
HDR:                                      ( 11011.01.1 10.101.MMM -------- )

1     FLOW:       TMP2        =      fp_load.Port_2.latency_1(MEM)
2     FLOW:       TMP0        =      load_ea.Port_0.latency_1(base_BBB)
3     FLOW:       TMP3        =      fp_load.Port_2.latency_1TMP0)
4     FLOW:       ST7         =      merge.Port_0.latency_1(TMP3, TMP2)
-----------------------

HDR:  "FLD STi":                          ( 11011.001 11000.iii -------- )

1     FLOW:       ST7         =      fp_move.Port_0.latency_1(ST(i))
-----------------------

HDR:  "FIST m16int":                      ( 11011.11.1 00.010.MMM -------- )
HDR:                                      ( 11011.11.1 01.010.MMM -------- )
HDR:                                      ( 11011.11.1 10.010.MMM -------- )
HDR:  "FIST m32int":                      ( 11011.01.1 00.010.MMM -------- )
HDR:                                      ( 11011.01.1 01.010.MMM -------- )
HDR:                                      ( 11011.01.1 10.010.MMM -------- )

1     FLOW:       TMP0        =      freadrom.Port_(not found).latency_1(CONST)
2     FLOW:       TMP1        =      fp_convert.Port_0.Latency_3(TMP0, ST0)
3     FLOW:       sink        =      fp_store_data.Port_4.latency_1(TMP1)
4     FLOW:       sink        =      fp_store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "FISTP m16int":                     ( 11011.11.1 00.011.MMM -------- )
HDR:                                      ( 11011.11.1 01.011.MMM -------- )
HDR:                                      ( 11011.11.1 10.011.MMM -------- )
HDR:  "FISTP m32int":                     ( 11011.01.1 00.011.MMM -------- )
HDR:                                      ( 11011.01.1 01.011.MMM -------- )
HDR:                                      ( 11011.01.1 10.011.MMM -------- )
HDR:  "FISTP m64int":                     ( 11011.11.1 00.111.MMM -------- )
HDR:                                      ( 11011.11.1 01.111.MMM -------- )
HDR:                                      ( 11011.11.1 10.111.MMM -------- )

1     FLOW:       TMP0        =      freadrom.Port_(not found).latency_1(CONST)
2     FLOW:       TMP1        =      fp_convert.Port_0.Latency_3(TMP0, ST0)
3     FLOW:       sink        =      fp_store_data.Port_4.latency_1(ST0,TMP1)
4     FLOW:       sink        =      fp_store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "FST m32real":                      ( 11011.00.1 00.010.MMM -------- )
HDR:                                      ( 11011.00.1 01.010.MMM -------- )
HDR:                                      ( 11011.00.1 10.010.MMM -------- )
```

```
HDR:  "FST m64real":                       ( 11011.10.1 00.010.MMM -------- )
HDR:                                       ( 11011.10.1 01.010.MMM -------- )
HDR:                                       ( 11011.10.1 10.010.MMM -------- )

1     FLOW:       sink        =      fp_store_data.Port_4.latency_1(ST0)
2     FLOW:       sink        =      fp_store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "FST STi":                           ( 11011.101 11010.iii -------- )

1     FLOW:       ST(i)       =      fp_move.Port_0.latency_1(ST0)
-----------------------

HDR:  "FSTP m32real":                      ( 11011.00.1 00.011.MMM -------- )
HDR:                                       ( 11011.00.1 01.011.MMM -------- )
HDR:                                       ( 11011.00.1 10.011.MMM -------- )
HDR:  "FSTP m64real":                      ( 11011.10.1 00.011.MMM -------- )
HDR:                                       ( 11011.10.1 01.011.MMM -------- )
HDR:                                       ( 11011.10.1 10.011.MMM -------- )

1     FLOW:       sink        =      fp_store_data.Port_4.latency_1(ST0)
2     FLOW:       sink        =      fp_store_address.Port_3.latency_1(MEM)
-----------------------

HDR:  "FSTP m80real":                      ( 11011.011 00.111.MMM -------- )
HDR:                                       ( 11011.011 01.111.MMM -------- )
HDR:                                       ( 11011.011 10.111.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "FSTP STi":                          ( 11011.101 11011.iii -------- )
HDR:  "FSTP8 STi":                         ( 11011.111 11010.iii -------- )
HDR:  "FSTP9 STi":                         ( 11011.111 11011.iii -------- )

1     FLOW:       ST(i)       =      fp_move.Port_0.latency_1(ST0)
-----------------------

HDR:  "FSTP1 STi":                         ( 11011.001 11011.iii -------- )

1     FLOW:       ST(i)       =      fp_move.Port_0.latency_1(ST0)

-----------------------

HDR:  "FXCH STi":                          ( 11011.001 11001.iii -------- )
HDR:  "FXCH4 STi":                         ( 11011.101 11001.iii -------- )
HDR:  "FXCH7 STi":                         ( 11011.111 11001.iii -------- )

1     FLOW:       sink        = fp_exchange.Port_(none*).latency_1(ST0, ST(i))

* This uop takes no bandwidth and is not dispatched to any port.
----------------------

HDR:  "FCMOVB STi":                        ( 11011.010 11000.iii -------- )


1     FLOW:  MP0              =      merge.Port_0.latency_1(ArithFLAGS, ST(i))
```

```
2     FLOW:       ST0         =      fp_select.Port_0.latency_1(TMP0, ST0)
-----------------------

HDR:  "FCMOVE STi":                      ( 11011.010 11001.iii -------- )

1     FLOW:       TMP0        =      merge.Port_0.latency_1(ArithFLAGS, ST(i))
2     FLOW:       ST0         =      fp_select.Port_0.latency_1(TMP0, ST0)
-----------------------

HDR:  "FCMOVBE STi":                     ( 11011.010 11010.iii -------- )

1     FLOW:       TMP0        =      merge.Port_0.latency_1(ArithFLAGS, ST(i))
2     FLOW:       ST0         =      fp_select.Port_0.latency_1(TMP0, ST0)
-----------------------

HDR:  "FCMOVU STi":                      ( 11011.010 11011.iii -------- )

1     FLOW:       TMP0        =      merge.Port_0.latency_1(ArithFLAGS, ST(i))
2     FLOW:       ST0         =      fp_select.Port_0.latency_1(TMP0, ST0)
-----------------------

HDR:  "FCMOVNB STi":                     ( 11011.011 11000.iii -------- )

1     FLOW:       TMP0        =      merge.Port_0.latency_1(ArithFLAGS, ST(i))
2     FLOW:       ST0         =      fp_select.Port_0.latency_1(TMP0, ST0)
-----------------------

HDR:  "FCMOVNE STi":                     ( 11011.011 11001.iii -------- )

1     FLOW:       TMP0        =      merge.Port_0.latency_1(ArithFLAGS, ST(i))
2     FLOW:       ST0         =      fp_select.Port_0.latency_1(TMP0, ST0)
-----------------------

HDR:  "FCMOVNBE STi":                    ( 11011.011 11010.iii -------- )

1     FLOW:       TMP0        =      merge.Port_0.latency_1(ArithFLAGS, ST(i))
2     FLOW:       ST0         =      fp_select.Port_0.latency_1(TMP0, ST0)
-----------------------

HDR:  "FCMOVNU STi":                     ( 11011.011 11011.iii -------- )

1     FLOW:       TMP0        =      merge.Port_0.latency_1(ArithFLAGS, ST(i))
2     FLOW:       ST0         =      fp_select.Port_0.latency_1(TMP0, ST0)
-----------------------

HDR:  "FABS":                            ( 11011.001 11100001 -------- )

1     FLOW:       ST0         =      fp_xor_sign.Port_0.latency_1(ST0, ST0)
-----------------------

HDR:  "FCHS":                            ( 11011.001 11100000 -------- )

1     FLOW:       TMP0        =      fp_move.Port_0.latency_1(ST0)
2     FLOW:       TMP1        =      freadrom.Port_(not found).latency_1(CONST)
3     FLOW: ST0   =      fp_xor.Port_0.latency_1(TMP0, TMP1)
-----------------------
```

```
HDR:  "FPREM":                          ( 11011.001 11111000 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FPREM1":                         ( 11011.001 11110101 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FBLD m80dec":                    ( 11011.111 00.100.MMM -------- )
HDR:                                    ( 11011.111 01.100.MMM -------- )
HDR:                                    ( 11011.111 10.100.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "FBSTP m80dec":                   ( 11011.111 00.110.MMM -------- )
HDR:                                    ( 11011.111 01.110.MMM -------- )
HDR:                                    ( 11011.111 10.110.MMM -------- )

Complex Instruction Found, Data not Presented

HDR:  "FRNDINT":                        ( 11011.001 11111100 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FSCALE":                         ( 11011.001 11111101 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FXTRACT":                        ( 11011.001 11110100 -------- )

Complex Instruction Found, Data not Presented

HDR:  "F2XM1":                          ( 11011.001 11110000 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FCOS":                           ( 11011.001 11111111 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FPATAN":                         ( 11011.001 11110011 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FPTAN":                          ( 11011.001 11110010 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FSIN":                           ( 11011.001 11111110 -------- )

Complex Instruction Found, Data not Presented

HDR:  "FSINCOS":                        ( 11011.001 11111011 -------- )

Complex Instruction Found, Data not Presented
```

```
HDR:  "FYL2X":                           ( 11011.001 11110001 -------- )

Complex Instruction Found, Data not Presented

HDR:   "FYL2XP1":                        ( 11011.001 11111001 -------- )

Complex Instruction Found, Data not Presented
```