# “Precise Exception Semantics in Dynamic Compilation”

Michael Gschwind, Erik Altman
IBM T.J. Watson Research Center

2002 Symposium on Compiler Construction

presented by nick black <nickblack@linux.com> for cs8803dc 2010-02-02

## Motivation

• Synchronous exceptions are bound to instructions and cannot be deferred

• Expose *user-managed state*, violating Bruening's (2004) model of even *extrinsic* compatibility

• Hardware-signaled exceptions mark *our* PC, not guest's

• Optimization changes user-managed state
(DCE, PRE, code sinking...)

“We're gonna need a bigger ROB.”
(actually, a *side table*)

## Difficulties of virtualizing exceptions:

- Determination of guest PC from host notifications
  (only applicable outside *interpretive trap detection)*
  - Dynamic binary translation eliminates bijection!
- Avoid exponential state costs from optimizing

## Program Counter Discovery, Part 1/3

- Easy for interpretation. Either:
  - Trap case is detected by the VM (interpretive detect), *or*
  - Synchronous exception is delivered before PC update
    (or any other externally-visible changes)
- Either way, the source PC is directly available

## Program Counter Discovery, Part 2/3

- Things become more difficult for binary translation!

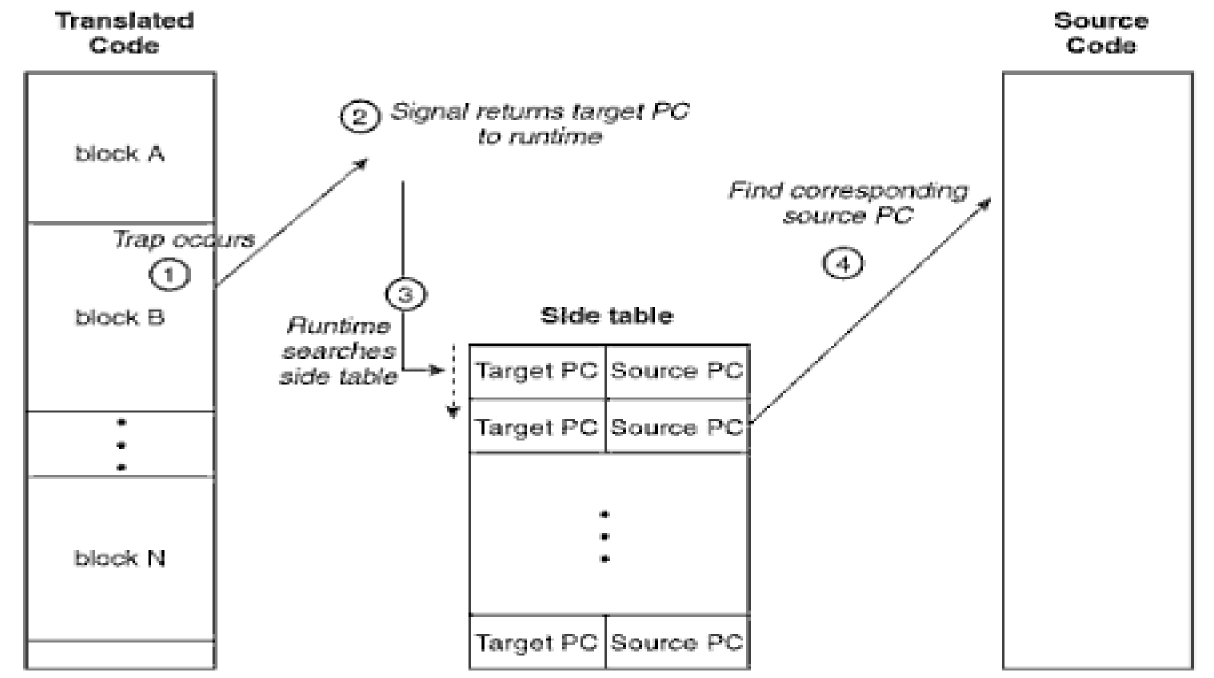

Translated Code
block A
Trap occurs
1
block B
block N
2 Signal returns target PC to runtime
3
Runtime searches side table
Side table
Target PC
Source PC
Find corresponding source PC
4
Source Code

**Figure 3.21** Finding the Trapping Source PC, Given the Target PC. *(1) The trap occurs and (2) the signal handler returns the target PC to the runtime software. The runtime (3) does a search of the side table (4) to find the corresponding source PC that caused the trap.*

## Program Counter Discovery, Part 3/3

- Inefficiencies in Target PC / Source PC map:
  - Pair of address pointers for each translated op
    (could be larger than translated source!)
  - Given target op might correspond to multiple source ops
- Both can be addressed via *translation block* map
  - *Coalesce* various instructions of a translation block
  - *Augment* the translation block map with register maps
  - See Figure 3.22 and Chapter 4 of the textbook

## Code Optimization with Precise Exceptions

- Let us consider the following (contrived) code:

```
ADDPD %xmm0, %xmm1              # SIMD add into xmm0
MOVAPD 0x20(%ebx), %xmm1        # aligned load into xmm1
ADDPD %xmm0, %xmm1              # SIMD add into xmm0
```

- First add could be excised via dead code elimination

- *...unless* there's a page fault at 0x20(%ebx)

## Where else have we seen this issue?

- Out-of-order processor

  *“Implementing Precise Interrupts in Pipelined Processors”*
  James Smith and Andrew Pleszkun, IEEE ToC, 1988

  - Since IBM 360/91, Tomasulo augmented via ROB:
    Reorder buffer feeds results
    Exceptions accounted for at ROB graduation

- Static compilation

  - Reorganization of code around branches needs fixups
  - Exponential state for static CFG amendments
  - With access to source, dynamic compilers fixup just-in-time

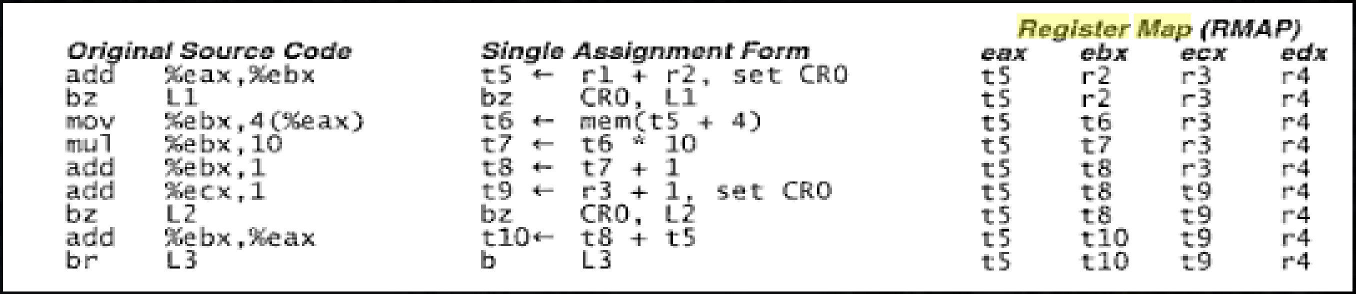
## Low-Cost Recovery from Exceptions

- Retain information regarding modified operations
  - Including preservation of input values
  - Reconstruct state on the fly when (rarely) needed
    - Is this infrequency assumption always valid? What if not?
      - p. 197: “maintain [an unoptimized] translation to the side”
    - Can't address architecture-invisible elements (intrinsic compatibility)
- Aside: suitable for static compilers?
  - Superficial similarity to retaining debugging symbols
  - More like running -g binary in a debugger (due to state)
  - Method likely unfit for static compilation.
  - So it goes.

## A Scheduling Algorithm, Part 1/2 *(ibid.* §3.5.2)

"Out-of-Order Execution Tech. for RT Binary Translators"
Bich Le, 8th Conference on ASPLOS, 1998

- Assume or force single-assign IR (SSA, CPS...)
- Derive the register map:



*Original Source Code*

```
add     %eax,%ebx
bz      L1
mov     %ebx,4(%eax)
mul     %ebx,10
add     %ebx,1
add     %ecx,1
bz      L2
add     %ebx,%eax
br      L3
```

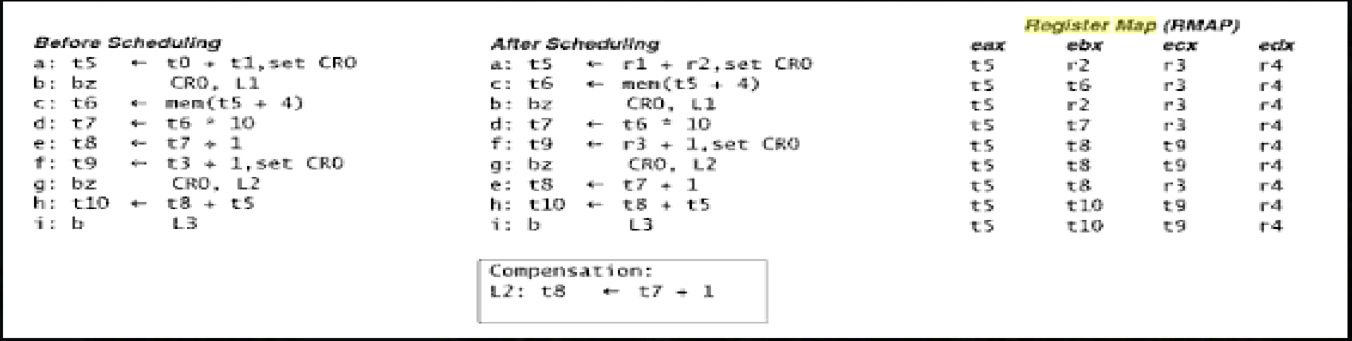
*Single Assignment Form*

```
t5 ←   r1 + r2, set CR0
bz     CR0, L1
t6 ←   mem(t5 + 4)
t7 ←   t6 * 10
t8 ←   t7 + 1
t9 ←   r3 + 1, set CR0
bz     CR0, L2
t10←   t8 + t5
b      L3
```

Register Map (RMAP)

| *eax* | *ebx* | *ecx* | *edx* |
|---|---|---|---|
| t5 | r2 | r3 | r4 |
| t5 | r2 | r3 | r4 |
| t5 | t6 | r3 | r4 |
| t5 | t7 | r3 | r4 |
| t5 | t8 | r3 | r4 |
| t5 | t8 | t9 | r4 |
| t5 | t8 | t9 | r4 |
| t5 | t10 | t9 | r4 |
| t5 | t10 | t9 | r4 |

- Reorder the code:



*Before Scheduling*

```
a: t5   ←  t0 + t1,set CR0
b: bz       CR0, L1
c: t6   ←  mem(t5 + 4)
d: t7   ←  t6 * 10
e: t8   ←  t7 + 1
f: t9   ←  t3 + 1,set CR0
g: bz       CR0, L2
h: t10  ←  t8 + t5
i: b        L3
```

*After Scheduling*

```
a: t5   ←  r1 + r2,set CR0
c: t6   ←  mem(t5 + 4)
b: bz       CR0, L1
d: t7   ←  t6 * 10
f: t9   ←  r3 + 1,set CR0
g: bz       CR0, L2
e: t8   ←  t7 + 1
h: t10  ←  t8 + t5
i: b        L3
```

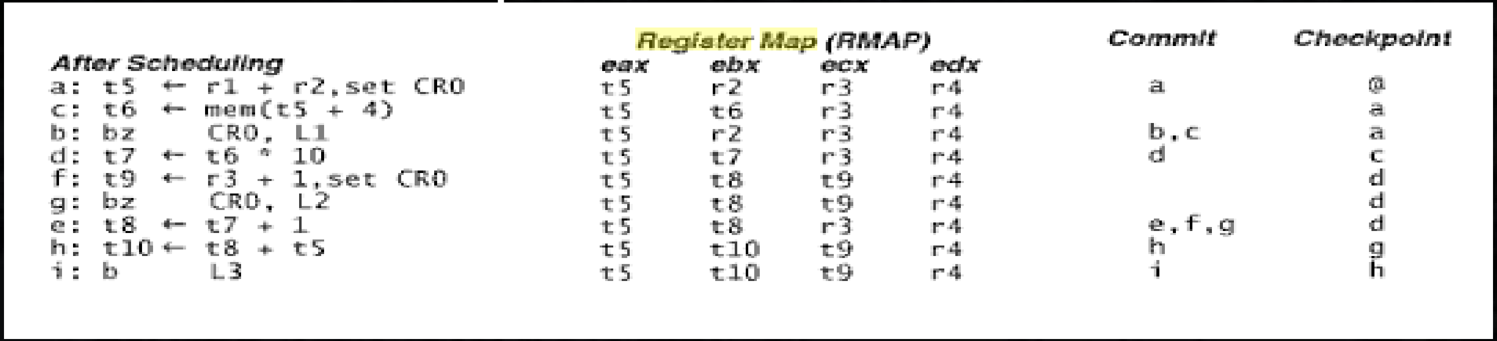
```
Compensation:
L2: t8    ←  t7 + 1
```

Register Map (RMAP)

| *eax* | *ebx* | *ecx* | *edx* |
|---|---|---|---|
| t5 | r2 | r3 | r4 |
| t5 | t6 | r3 | r4 |
| t5 | r2 | r3 | r4 |
| t5 | t7 | r3 | r4 |
| t5 | t8 | t9 | r4 |
| t5 | t8 | t9 | r4 |
| t5 | t8 | r3 | r4 |
| t5 | t10 | t9 | r4 |
| t5 | t10 | t9 | r4 |

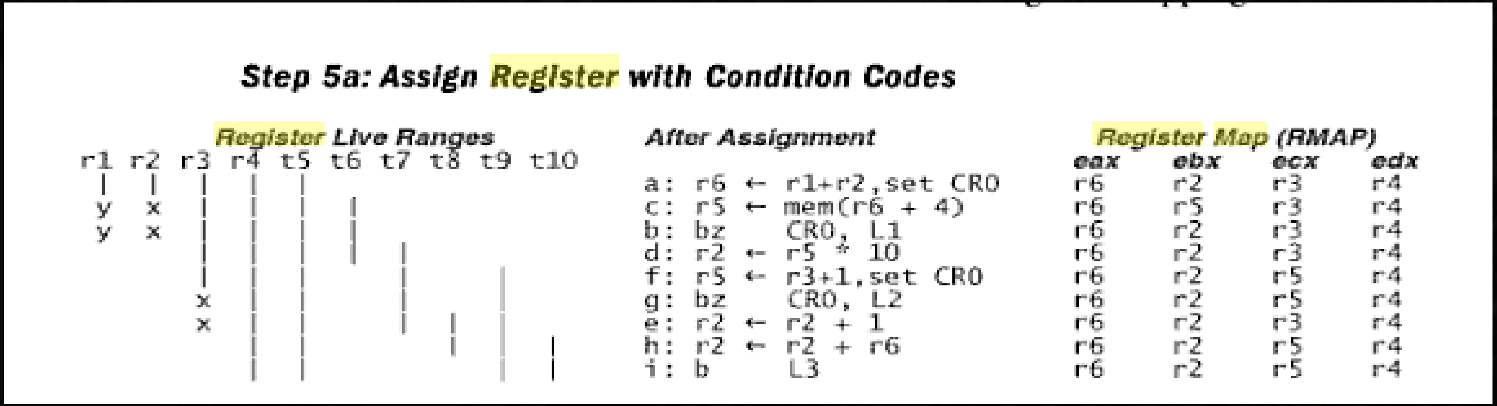
## A Scheduling Algorithm, Part 2/2

- Determine checkpoints:



| After Scheduling | Register Map (RMAP) eax | ebx | ecx | edx | Commit | Checkpoint |
|---|---|---|---|---|---|---|
| a: t5 ← r1 + r2,set CR0 | t5 | r2 | r3 | r4 | a | @ |
| c: t6 ← mem(t5 + 4) | t5 | t6 | r3 | r4 | | a |
| b: bz CR0, L1 | t5 | r2 | r3 | r4 | b,c | a |
| d: t7 ← t6 * 10 | t5 | t7 | r3 | r4 | d | c |
| f: t9 ← r3 + 1,set CR0 | t5 | t8 | t9 | r4 | | d |
| g: bz CR0, L2 | t5 | t8 | t9 | r4 | | d |
| e: t8 ← t7 + 1 | t5 | t8 | r3 | r4 | e,f,g | d |
| h: t10 ← t8 + t5 | t5 | t10 | t9 | r4 | h | g |
| i: b L3 | t5 | t10 | t9 | r4 | i | h |

- Assign registers, paying attention to flag register(s)



**Step 5a: Assign Register with Condition Codes**

*Register Live Ranges*

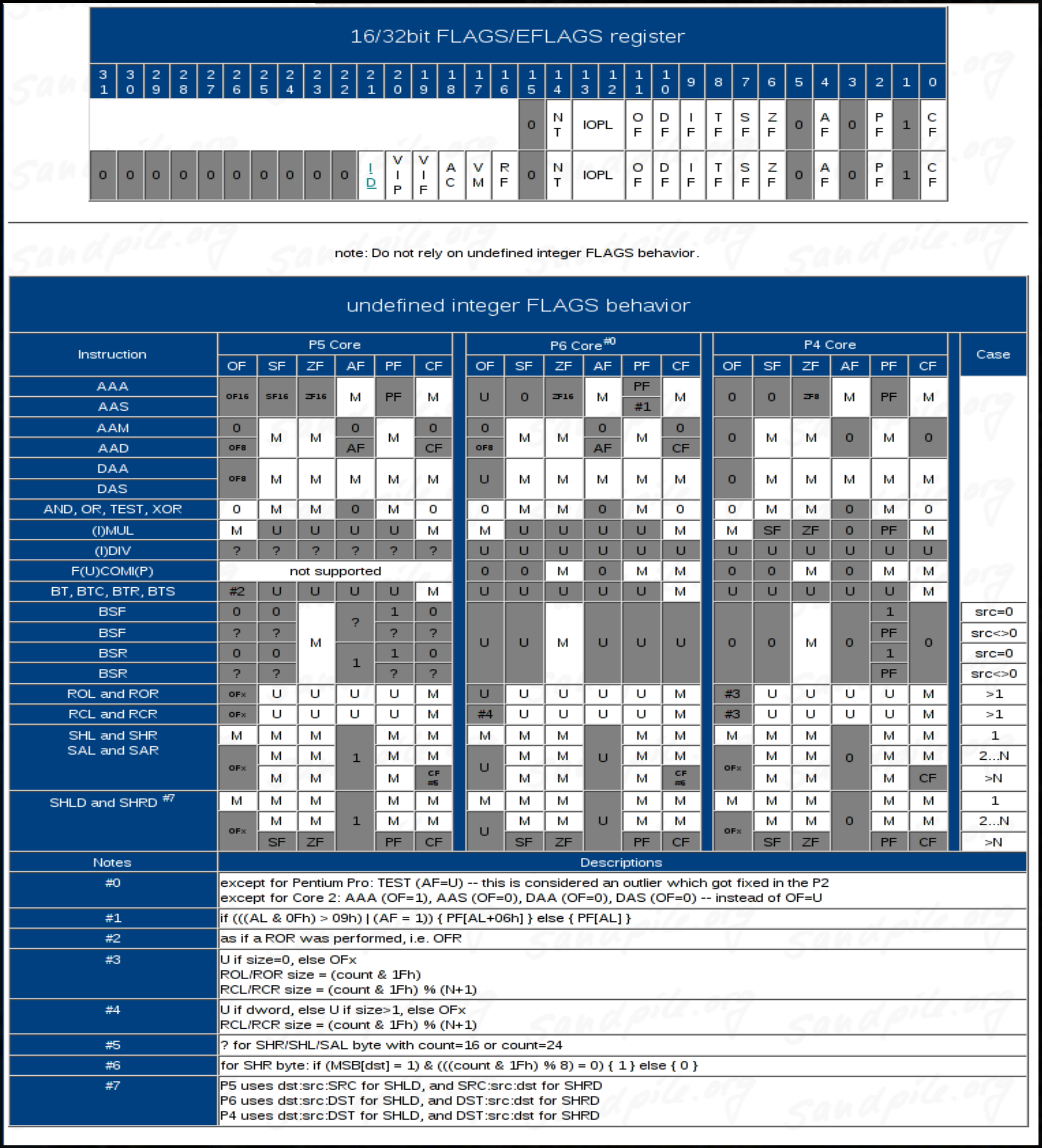
```
r1 r2 r3 r4 t5 t6 t7 t8 t9 t10
 |  |  |  |  |
 y  x  |  |  |  |
 y  x  |  |  |  |
       |  |  |  |  |
       |  |  |     |     |
       x  |  |     |     |
       x  |  |     |  |  |
          |  |        |  |  |
          |  |           |  |
```

| After Assignment | Register Map (RMAP) eax | ebx | ecx | edx |
|---|---|---|---|---|
| a: r6 ← r1+r2,set CR0 | r6 | r2 | r3 | r4 |
| c: r5 ← mem(r6 + 4) | r6 | r5 | r3 | r4 |
| b: bz CR0, L1 | r6 | r2 | r3 | r4 |
| d: r2 ← r5 * 10 | r6 | r2 | r3 | r4 |
| f: r5 ← r3+1,set CR0 | r6 | r2 | r5 | r4 |
| g: bz CR0, L2 | r6 | r2 | r5 | r4 |
| e: r2 ← r2 + 1 | r6 | r2 | r3 | r4 |
| h: r2 ← r2 + r6 | r6 | r2 | r5 | r4 |
| i: b L3 | r6 | r2 | r5 | r4 |

- Add compensation code and *run* that sumbitch

## EFLAGS

- Delicious!
- Lots of messy details



16/32bit FLAGS/EFLAGS register

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | 0 | NT | IOPL | | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | 1 | CF |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ID | VIP | VIF | AC | VM | RF | 0 | NT | IOPL | | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | 1 | CF |

note: Do not rely on undefined integer FLAGS behavior.

undefined integer FLAGS behavior

| Instruction | P5 Core | | | | | | P6 Core#0 | | | | | | P4 Core | | | | | | Case |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | OF | SF | ZF | AF | PF | CF | OF | SF | ZF | AF | PF | CF | OF | SF | ZF | AF | PF | CF | |
| AAA | OF16 | SF16 | ZF16 | M | PF | M | U | 0 | ZF16 | M | PF | M | 0 | 0 | ZF8 | M | PF | M | |
| AAS | | | | | | | | | | | #1 | | | | | | | | |
| AAM | 0 | M | M | 0 | M | 0 | 0 | M | M | 0 | M | 0 | 0 | M | M | 0 | M | 0 | |
| AAD | OF8 | | | AF | | CF | OF8 | | | AF | | CF | | | | | | | |
| DAA | OF8 | M | M | M | M | M | U | M | M | M | M | M | 0 | M | M | M | M | M | |
| DAS | | | | | | | | | | | | | | | | | | | |
| AND, OR, TEST, XOR | 0 | M | M | 0 | M | 0 | 0 | M | M | 0 | M | 0 | 0 | M | M | 0 | M | 0 | |
| (I)MUL | M | U | U | U | U | M | M | U | U | U | U | M | M | SF | ZF | 0 | PF | M | |
| (I)DIV | ? | ? | ? | ? | ? | ? | U | U | U | U | U | U | U | U | U | U | U | U | |
| F(U)COMI(P) | not supported | | | | | | 0 | 0 | M | 0 | M | M | 0 | 0 | M | 0 | M | M | |
| BT, BTC, BTR, BTS | #2 | U | U | U | U | M | U | U | U | U | U | M | U | U | U | U | U | M | |
| BSF | 0 | 0 | M | ? | 1 | 0 | U | U | M | U | U | U | 0 | 0 | M | 0 | 1 | 0 | src=0 |
| BSF | ? | ? | | | ? | ? | | | | | | | | | | | PF | | src<>0 |
| BSR | 0 | 0 | | 1 | 1 | 0 | | | | | | | | | | | 1 | | src=0 |
| BSR | ? | ? | | | ? | ? | | | | | | | | | | | PF | | src<>0 |
| ROL and ROR | OFx | U | U | U | U | M | U | U | U | U | U | M | #3 | U | U | U | U | M | >1 |
| RCL and RCR | OFx | U | U | U | U | M | #4 | U | U | U | U | M | #3 | U | U | U | U | M | >1 |
| SHL and SHR SAL and SAR | M | M | M | 1 | M | M | M | M | M | U | M | M | M | M | M | 0 | M | M | 1 |
| | OFx | M | M | | M | M | U | M | M | | M | M | OFx | M | M | | M | M | 2...N |
| | | M | M | | M | CF #5 | | M | M | | M | CF #6 | | M | M | | M | CF | >N |
| SHLD and SHRD #7 | M | M | M | 1 | M | M | M | M | M | U | M | M | M | M | M | 0 | M | M | 1 |
| | OFx | M | M | | M | M | U | M | M | | M | M | OFx | M | M | | M | M | 2...N |
| | | SF | ZF | | PF | CF | | SF | ZF | | PF | CF | | SF | ZF | | PF | CF | >N |

| Notes | Descriptions |
|---|---|
| #0 | except for Pentium Pro: TEST (AF=U) -- this is considered an outlier which got fixed in the P2<br>except for Core 2: AAA (OF=1), AAS (OF=0), DAA (OF=0), DAS (OF=0) -- instead of OF=U |
| #1 | if (((AL & 0Fh) > 09h) \| (AF = 1)) { PF[AL+06h] } else { PF[AL] } |
| #2 | as if a ROR was performed, i.e. OFR |
| #3 | U if size=0, else OFx<br>ROL/ROR size = (count & 1Fh)<br>RCL/RCR size = (count & 1Fh) % (N+1) |
| #4 | U if dword, else U if size>1, else OFx<br>RCL/RCR size = (count & 1Fh) % (N+1) |
| #5 | ? for SHR/SHL/SAL byte with count=16 or count=24 |
| #6 | for SHR byte: if (MSB[dst] = 1) & (((count & 1Fh) % 8) = 0) { 1 } else { 0 } |
| #7 | P5 uses dst:src:SRC for SHLD, and SRC:src:dst for SHRD<br>P6 uses dst:src:DST for SHLD, and DST:src:dst for SHRD<br>P4 uses dst:src:DST for SHLD, and DST:src:dst for SHRD |

(stolen from sandpile.org)

## Other Optimizations

- Code sinking
  - Repair note inserted in original op slot
  - No extra input value preservation necessary

- Unspeculation (PRE)
  - DCE along redundant paths, code sinking where needed

- Constant propagation
- Constant folding
- Commoning

- Elimination of ISA-specific condition code updates

- Patch repair code into eliminated code via flag bit!

  (Mmm, I especially enjoyed that one)