Causes of Performance Swings Due to Code Placement in IA

Zia Ansari
zia.ansari@intel.com
Intel Corporation
11/03/16

Agenda

- The purpose of this presentation
- Intel Architecture FE 101
- Let’s look at some example
- So can we do anything about all this?
- Conclusion / Future work

Purpose of This Presentation

- Performance swings not immediately apparent at a high level
- Are my changes “good”?
    - Performance doesn’t match expectations
    - Performance neutral changes caused swings
    - Help when performance results lie to you
        - Evaluation through micro-benchmarking
        - Wrong decisions are made

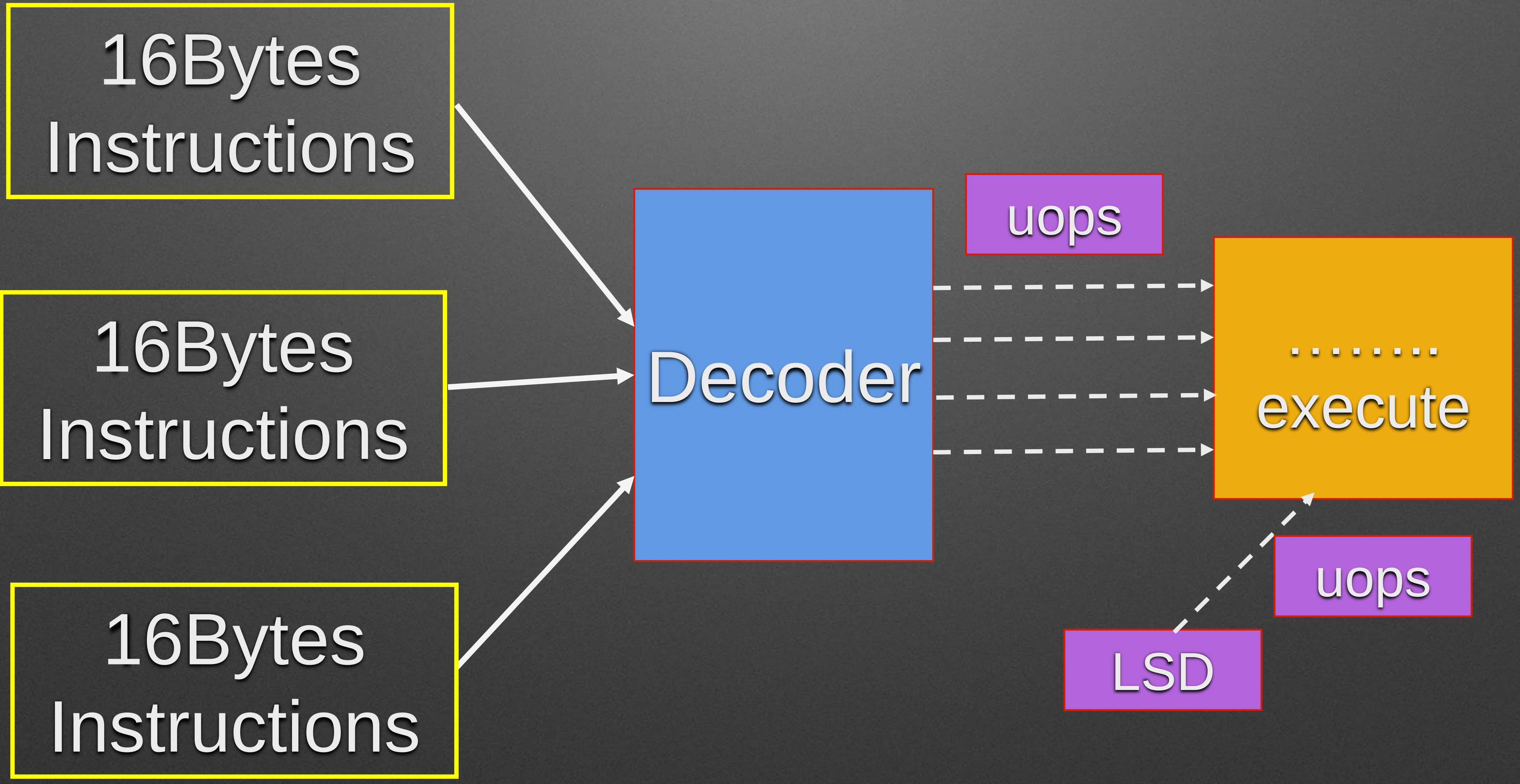
Purpose of This Presentation

- Important to having a better understanding of the architecture
  - Make better optimization decisions
  - Save time on analysis
- May not be able to resolve all the issues, but useful to at least understand

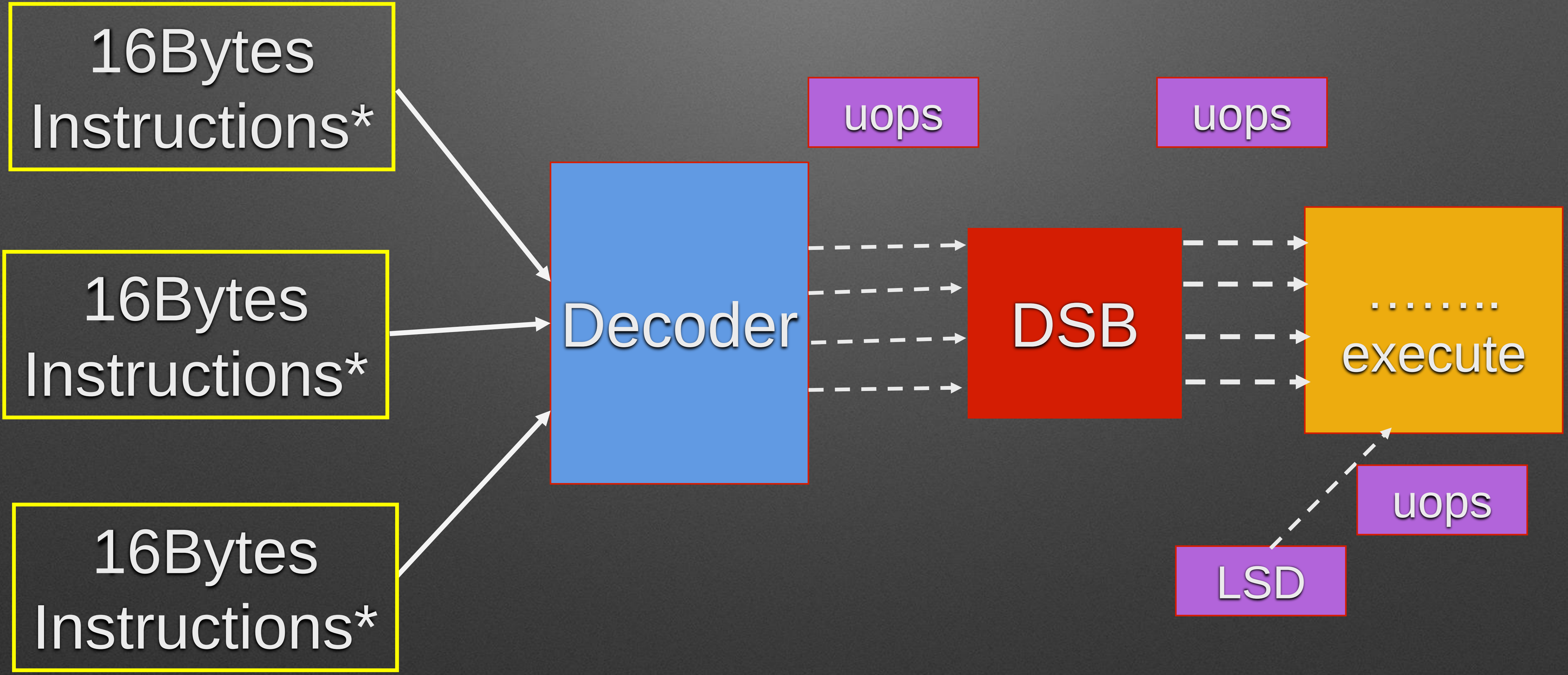
IA Front End 101

- Older Gen Intel Architectures VS. Newer Gen Intel Architectures

Core / NHM / Atom



IVB / SNB / HSW / SKL*



Let’s Look At Some Examples

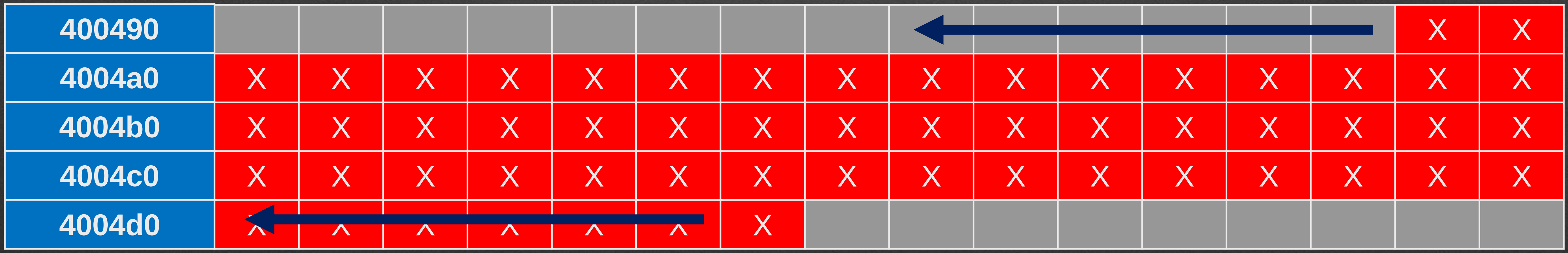
- Core / NHM / Atom decoder alignment
- DSB Throughput Alignment
- DSB Thrashing Alignment
- BPU Alignment

Aligning for 16B Fetch Lines

```
for (ii=0;ii<64;ii++) {
        a[ii] = b[ii] + c[ii];
        b[ii] = c[ii] + a[ii];
        c[ii] = c[ii] - 10;
        total += a[ii] + b[ii] - c[ii];
}
```

```
40049e: mov 0x600be0(%rax),%ecx
4004a4: mov 0x600a40(%rax),%edx
4004aa: add %ecx,%edx
4004ac: lea (%rdx,%rcx,1),%esi
4004af: sub $0xa,%ecx
4004b2: mov %edx,0x6008a0(%rax)
4004b8: mov %ecx,0x600be0(%rax)
4004be: mov %esi,0x600a40(%rax)
4004c4: sub %ecx,%esi
4004c6: add $0x4,%rax
4004ca: lea (%rsi,%rdx,1),%edx
4004cd: add %edx,%edi
4004cf: cmp $0x100,%rax
4004d5: jne 40049e
```

```
400497: mov 0x600be0(%rax),%ecx
40049d: mov 0x600a40(%rax),%edx
4004a3: add %ecx,%edx
4004a5: lea (%rdx,%rcx,1),%esi
4004a8: sub $0xa,%ecx
4004ab: mov %edx,0x6008a0(%rax)
4004b1: mov %ecx,0x600be0(%rax)
4004b7: mov %esi,0x600a40(%rax)
4004bd: sub %ecx,%esi
4004bf: add $0x4,%rax
4004c3: lea (%rsi,%rdx,1),%edx
4004c6: add %edx,%edi
4004c8: cmp $0x100,%rax
4004ce: jne 400497 <main+0x17>
```



400490															X	X
4004a0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4004b0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4004c0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4004d0	X	X	X	X	X	X	X									

20% Speedup (NHM)

… but wait

```
~~for (ii=0;ii<64;ii++) {~~
for (ii=0;ii<65;ii++) {
        a[ii] = b[ii] + c[ii];
        b[ii] = c[ii] + a[ii];
        c[ii] = c[ii] - 10;
        total += a[ii] + b[ii] - c[ii];
}
```

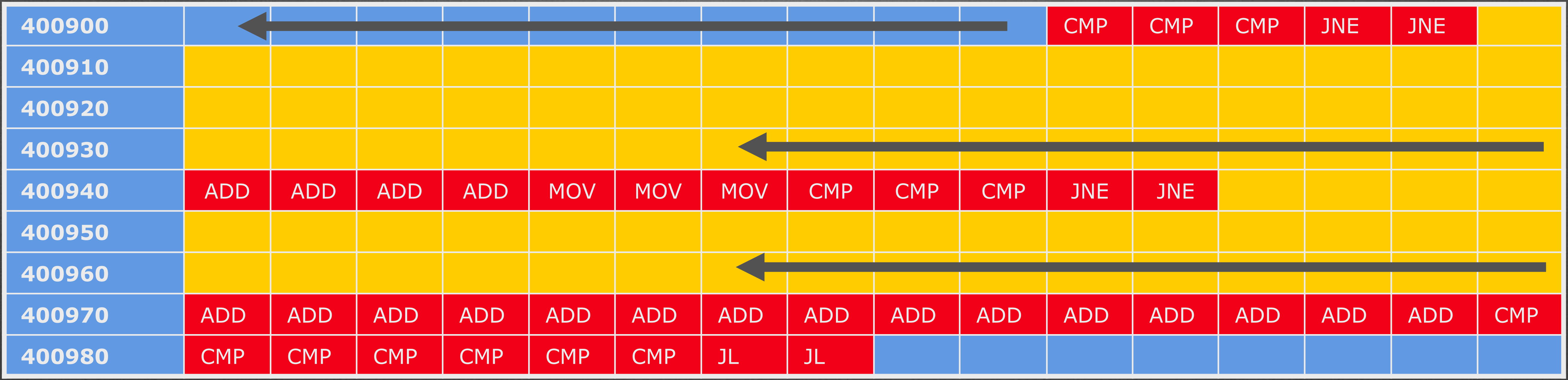
- Aligned case 9% slower
- Misaligned case on par with aligned case
- Why?

- LSD firing, delivering uops from cache
- Speeds up FE, but costs mispredict
- As iterations go up, penalty lessens, and alignment doesn’t matter anymore.

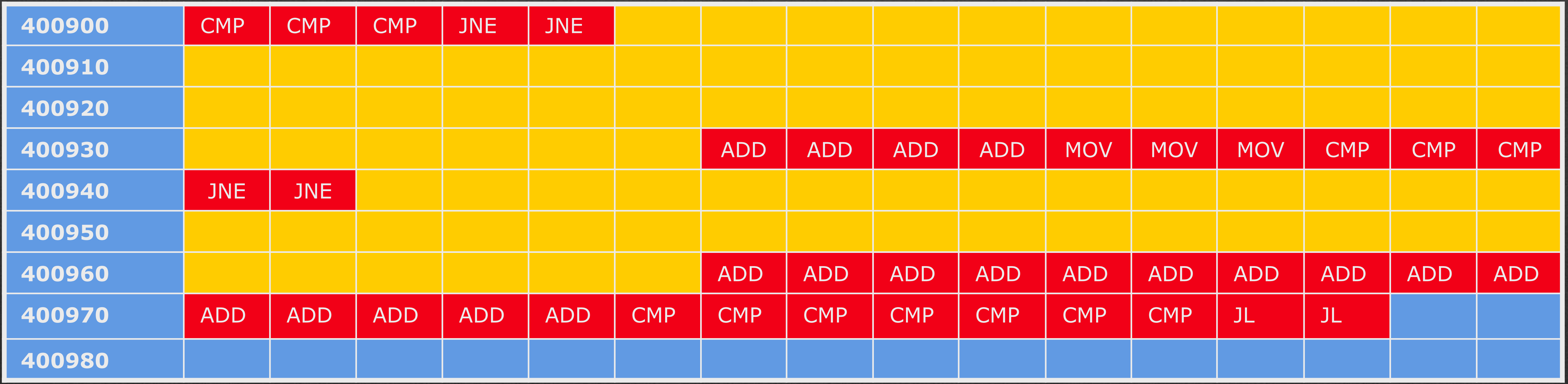
Why not just always align?

- Costs code size
- Can cost performance if executed
- With branches, becomes a gamble

4 16B chunks -> 5 16B chunks
~80% Slowdown on Core/NHM



400900											CMP	CMP	CMP	JNE	JNE	
400910																
400920																
400930																
400940	ADD	ADD	ADD	ADD	MOV	MOV	MOV	CMP	CMP	CMP	JNE	JNE				
400950																
400960																
400970	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	CMP
400980	CMP	CMP	CMP	CMP	CMP	CMP	JL	JL								



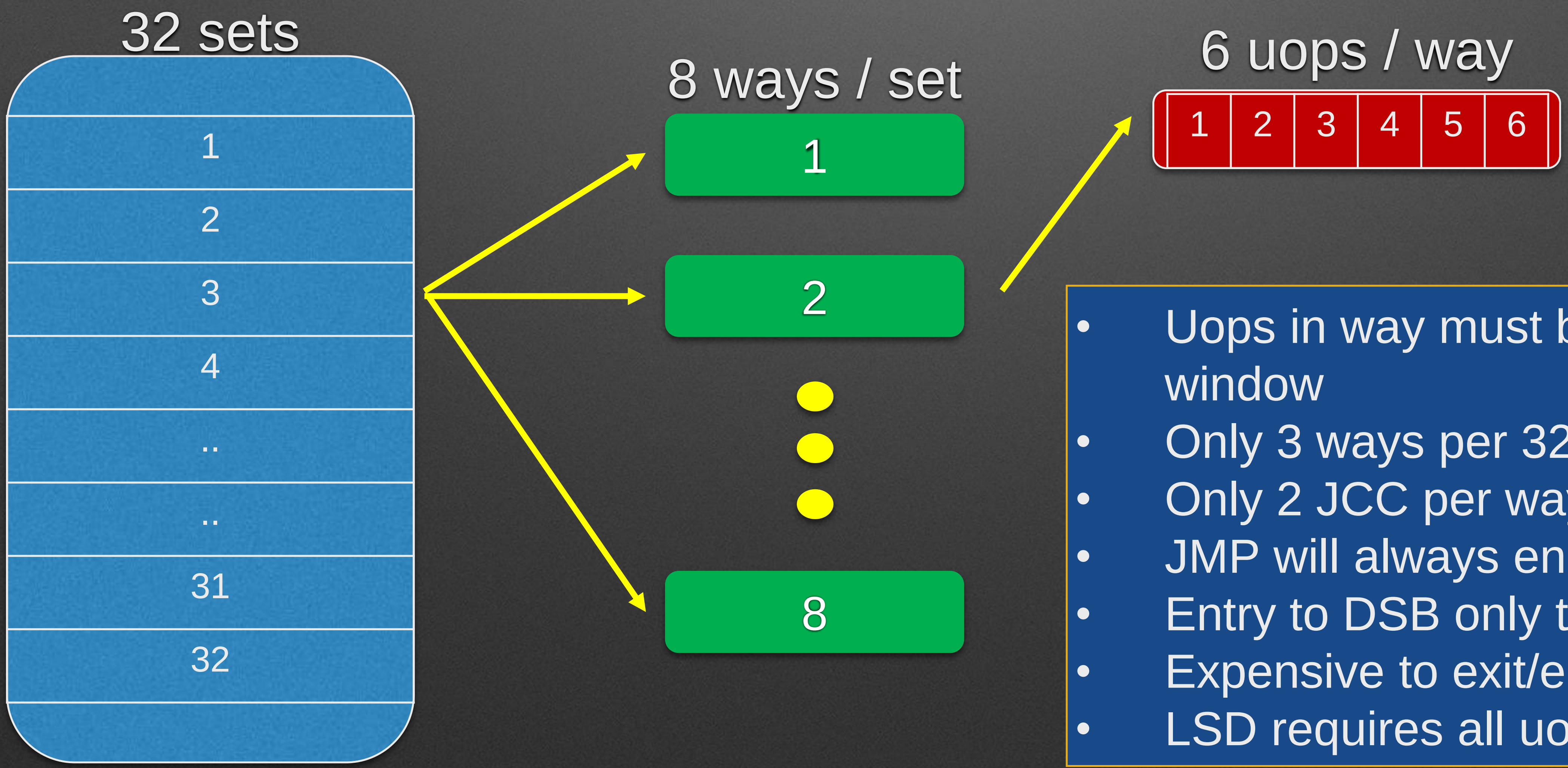
400900	CMP	CMP	CMP	JNE	JNE											
400910																
400920																
400930							ADD	ADD	ADD	ADD	MOV	MOV	MOV	CMP	CMP	CMP
400940	JNE	JNE														
400950																
400960							ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD
400970	ADD	ADD	ADD	ADD	ADD	CMP	CMP	CMP	CMP	CMP	CMP	CMP	JL	JL		
400980																

Breaking the Instruction Bottleneck

- - Fetching 16B of instructions at a time can be limiting
  - movups 0x80(%r15,%rax,8),%xmm0 : 9 Bytes!
  - Decoder restrictions, power, etc...
- LSD helps by replaying uops, but is very limited
  - Has a small window of instructions, within a loop only
  - Assumes “endless loop” (no prediction)
- Ideally, we’d like to cache arbitrary uops for replay
  - Decoded Stream Buffer (DSB)

Decoded Stream Buffer (DSB)

- DSB is a cache for uops that have been decoded.
- Extends the FE window to 32B to increase throughput.
- Saves power and lowers mispredict costs.



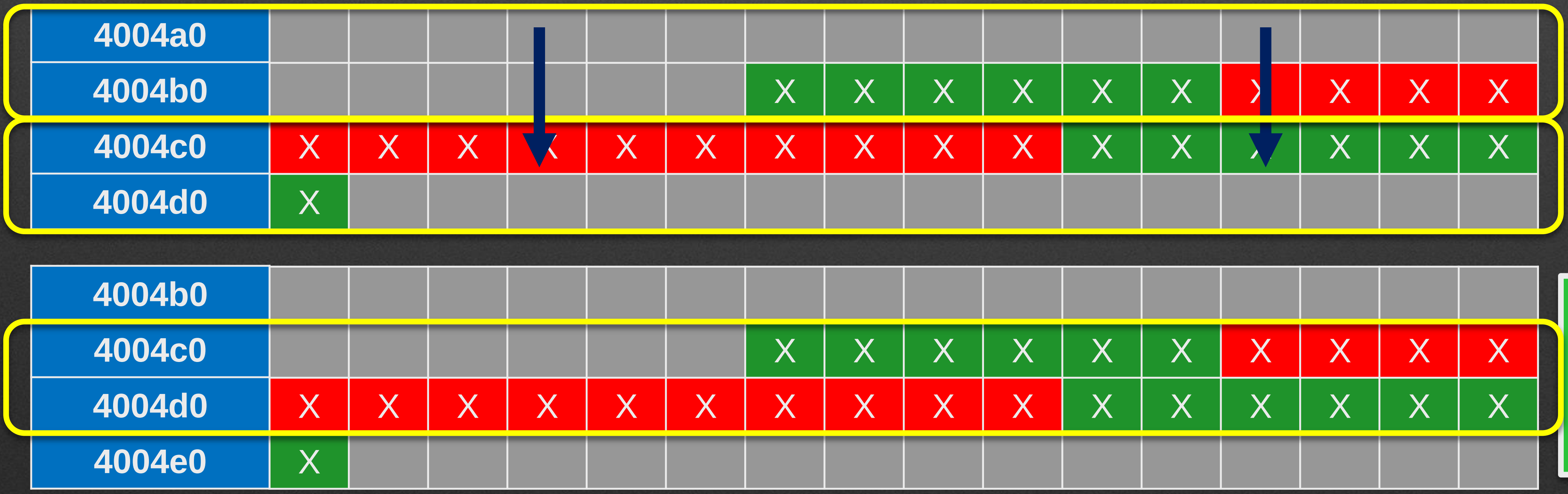
- Uops in way must be in 32B aligned window
- Only 3 ways per 32B windows
- Only 2 JCC per way
- JMP will always end a way
- Entry to DSB only through branch
- Expensive to exit/enter frequently
- LSD requires all uops to be in DSB

Aligning for 32B DSB Lines

```
for (i = 0; i < n; i++) {
  for (ii = 0; ii < m; ii++) {
    if (ii == 0) {
      x++;
    }
    if (ii > 0) {
      x+=2;
    }
  }
}
```

```
4004b6: test %esi,%esi
4004b8: jle  4004ca
4004ba: xor  %eax,%eax
   4004bc: test %eax,%eax
   4004be: je   4004da
   4004c0: add  $0x2,%edx
   4004c3: add  $0x1,%eax
   4004c6: cmp  %esi,%eax
   4004c8: jne  4004bc
4004ca: add  $0x1,%ecx
4004cd: cmp  %edi,%ecx
4004cf: jne  4004b6
```

```
4004c6: test %esi,%esi
4004c8: jle  4004da
4004ca: xor  %eax,%eax
   4004cc: test %eax,%eax
   4004ce: je   4004ea
   4004d0: add  $0x2,%edx
   4004d3: add  $0x1,%eax
   4004d6: cmp  %esi,%eax
   4004d8: jne  4004cc
4004da: add  $0x1,%ecx
4004dd: cmp  %edi,%ecx
4004df: jne  4004c6
```

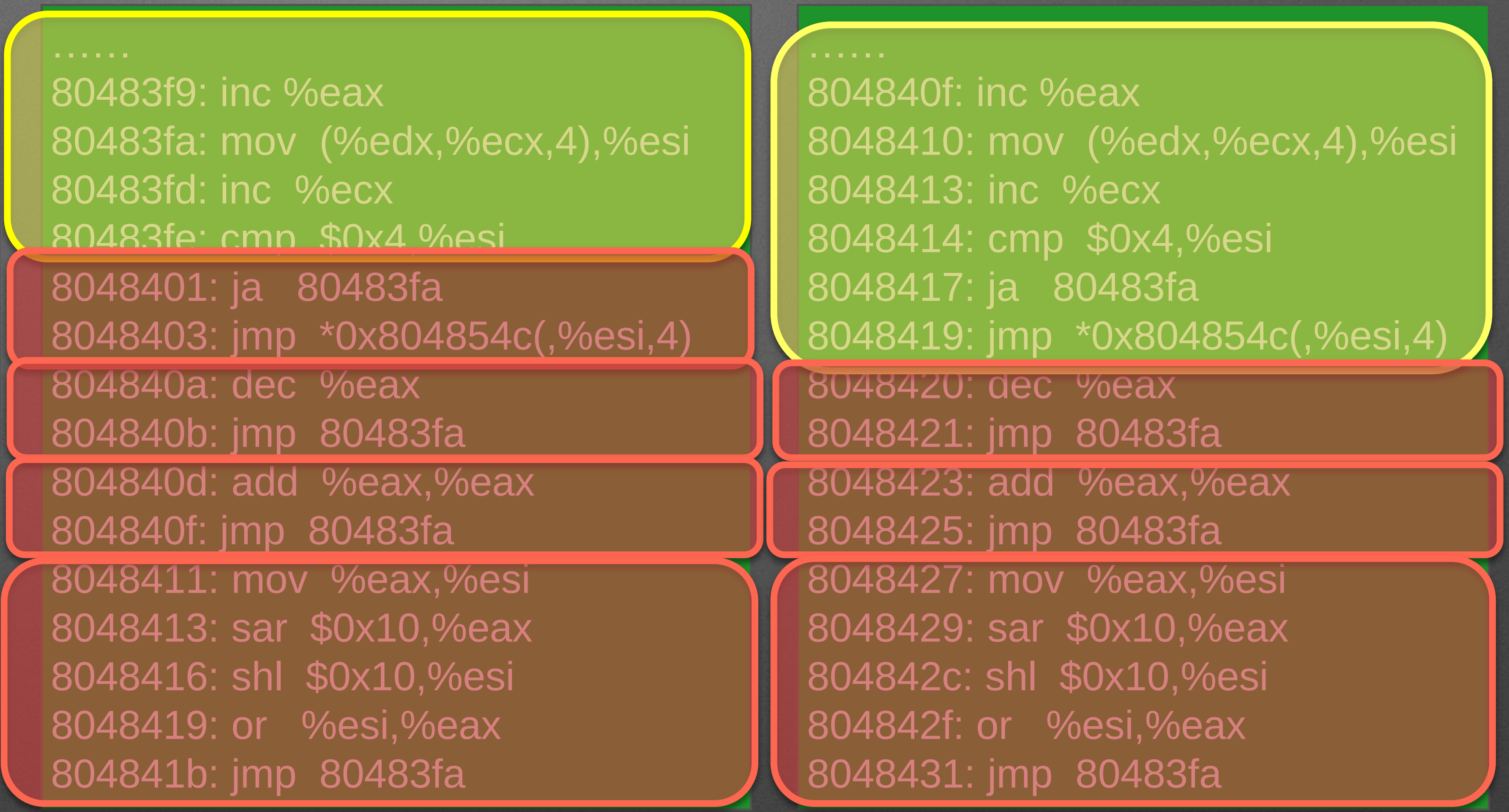


30% Speedup
SNB/IVB/HSW

DSB Thrashing

```
int foo(int *DATA, int n)
{
  int i = 0;
  int result = 0;
  while (1) {
    switch (DATA[i++]) {
        case 0: return result;
        case 1: result++; break;
        case 2: result--; break;
        case 3: result <<=1; break;
        case 4: result = (result << 16) |
                                 (result >> 16); break;
    }
  }
}
```

PR5615



```
……
80483f9: inc %eax
80483fa: mov  (%edx,%ecx,4),%esi
80483fd: inc  %ecx
80483fe: cmp  $0x4,%esi
8048401: ja   80483fa
8048403: jmp  *0x804854c(,%esi,4)
804840a: dec  %eax
804840b: jmp  80483fa
804840d: add  %eax,%eax
804840f: jmp  80483fa
8048411: mov  %eax,%esi
8048413: sar  $0x10,%eax
8048416: shl  $0x10,%esi
8048419: or   %esi,%eax
804841b: jmp  80483fa
```

```
……
804840f: inc %eax
8048410: mov  (%edx,%ecx,4),%esi
8048413: inc  %ecx
8048414: cmp  $0x4,%esi
8048417: ja   80483fa
8048419: jmp  *0x804854c(,%esi,4)
8048420: dec  %eax
8048421: jmp  80483fa
8048423: add  %eax,%eax
8048425: jmp  80483fa
8048427: mov  %eax,%esi
8048429: sar  $0x10,%eax
804842c: shl  $0x10,%esi
804842f: or   %esi,%eax
8048431: jmp  80483fa
```

- Uops in way must be in 32B aligned window
- JMP will always end a way
- Only 3 ways per 32B windows
- Only 2 JCC per way
- Entry to DSB only through branch
- LSD requires all uops to be in DSB
- Expensive to exit/enter frequently

DSB2MITE_SWITCHES.COUNT
312M vs 37M

LSD.CYCLES_ACTIVE
32K vs 1B

30% Speedup **SNB/IVB/HSW**

Teaser . . .

```
8048452: inc %eax
8048453: mov (%edx,%ecx,4),%esi
8048456: inc %ecx
8048457: cmp $0x4,%esi
804845a: ja  8048453
804845c: jmp *0x804851c(,%esi,4)
8048463: dec %eax
8048464: jmp 8048453
8048466: add %eax,%eax
8048468: jmp 8048453
804846a: mov %eax,%esi
804846c: sar $0x10,%eax
804846f: shl $0x10,%esi
8048472: or  %esi,%eax
8048474: jmp 8048453
```

- Exact same code
- Different alignment
- > 5x slower

Aligning for Branch Prediction

```
int foo(int i, int m, int p, int q, int *p1, int *p2)
{
  if (i+*p1 == p || i == q || i-*p2 > m) {
    y++;
    x++;
    if (i == q) {
      x += y;
    }
  }

  return 0;
}
```

```
400500: mov  (%r8),%eax
400503: add  %edi,%eax
400505: cmp  %edx,%eax
400507: jne  40054b
40050d: mov  0x200b25(%rip),%eax
400513: inc  %eax
400515: mov  %eax,0x200b1d(%rip)
40051b: mov  0x200b13(%rip),%edx
400521: inc  %edx
400523: mov  %edx,0x200b0b(%rip)
400529: cmp  %ecx,%edi
40052b: jne  400560
400531: add  %e
400533: mov  %e
400539: jmpq 40
```

```
40054b: nop
40054c: cmp  %ecx,%edi
40054e: je   40050d
400554: mov  %edi,%eax
400556: sub  (%r9),%eax
400559: cmp  %esi,%eax
40055b: jg   40050d
400561: xor %eax, %eax
```

BR_MISP_RETIRED.ALL_BRANCHES
300M vs 150M

30% Speedup SNB/IVB/HSW

- Avoid putting two conditional branch instructions in a loop so that both have the same branch target address and, at the same time, belong to (i.e. have their last bytes' addresses within) the same 16-byte aligned code block.

Identifying Potential Issues

- Understand the architecture / Read the optimization manual
- If your perf swings “don’t make sense”
  - Compare before / after hardware counters
    - Branch mispredicts
    - Delivery : Fetch? LSD? DSB? Switch counts?
- Come up with potential theories, and try adding nops
- If all else fails, ask Intel

Current / Future Work

- Do we really need alignment on all loops and branch targets? Why 16B?
  - Architectures becoming less alignment sensitive
  - Spec2k -O2 is 2.72% smaller w/o alignment with flat performance
  - Maybe make them more limited (no branchy loops)

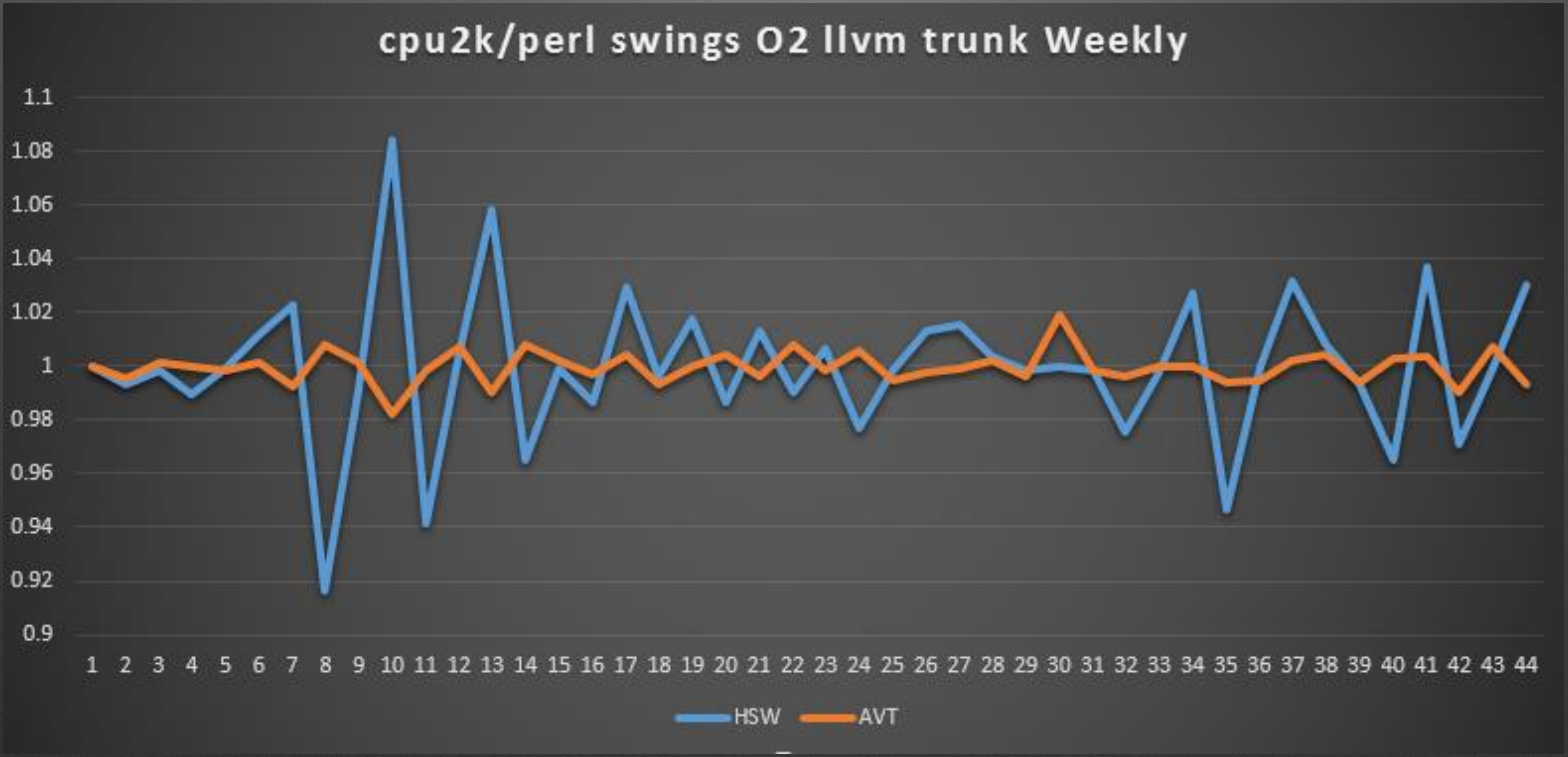
- Better heuristics to catch some subtle cases
  - Space branches in same 32B window to same target
  - Space jmp/jcc to not thrash DSB
  - etc.

- Omer Paparo Bivas at Intel is currently working on experimenting with this and a late “nop” pass

Questions?

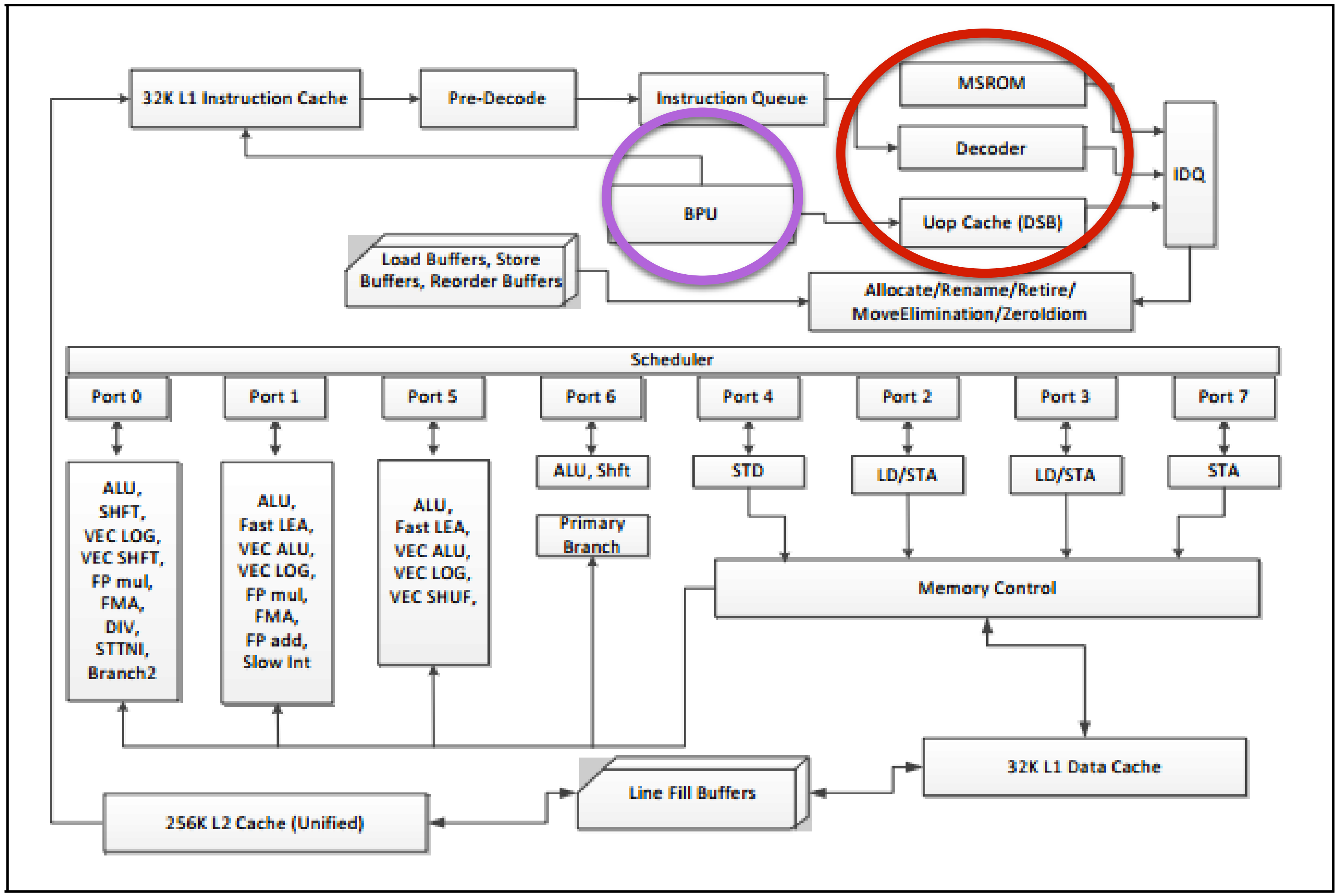
Backup

“Oh, it’s just Perl”



IVB / SNB / HSW / SKL

- Fetch / decode / feed to DSB / read out of DSB -> execute / read out of DSB (hopefully) -> execute …



32K L1 Instruction Cache
Pre-Decode
Instruction Queue
MSROM
Decoder
Uop Cache (DSB)
IDQ
BPU
Load Buffers, Store Buffers, Reorder Buffers
Allocate/Rename/Retire/ MoveElimination/ZeroIdiom
Scheduler
Port 0
Port 1
Port 5
Port 6
Port 4
Port 2
Port 3
Port 7
ALU, SHFT, VEC LOG, VEC SHFT, FP mul, FMA, DIV, STTNI, Branch2
ALU, Fast LEA, VEC ALU, VEC LOG, FP mul, FMA, FP add, Slow Int
ALU, Fast LEA, VEC ALU, VEC LOG, VEC SHUF,
ALU, Shft
Primary Branch
STD
LD/STA
LD/STA
STA
Memory Control
32K L1 Data Cache
Line Fill Buffers
256K L2 Cache (Unified)

Figure 2-2. CPU Core Pipeline Functionality of the Haswell Microarchitecture

Core / NHM / Atom

- Fetch 16B aligned window of instructions -> decode -> execute -> fetch -> decode -> execute ..

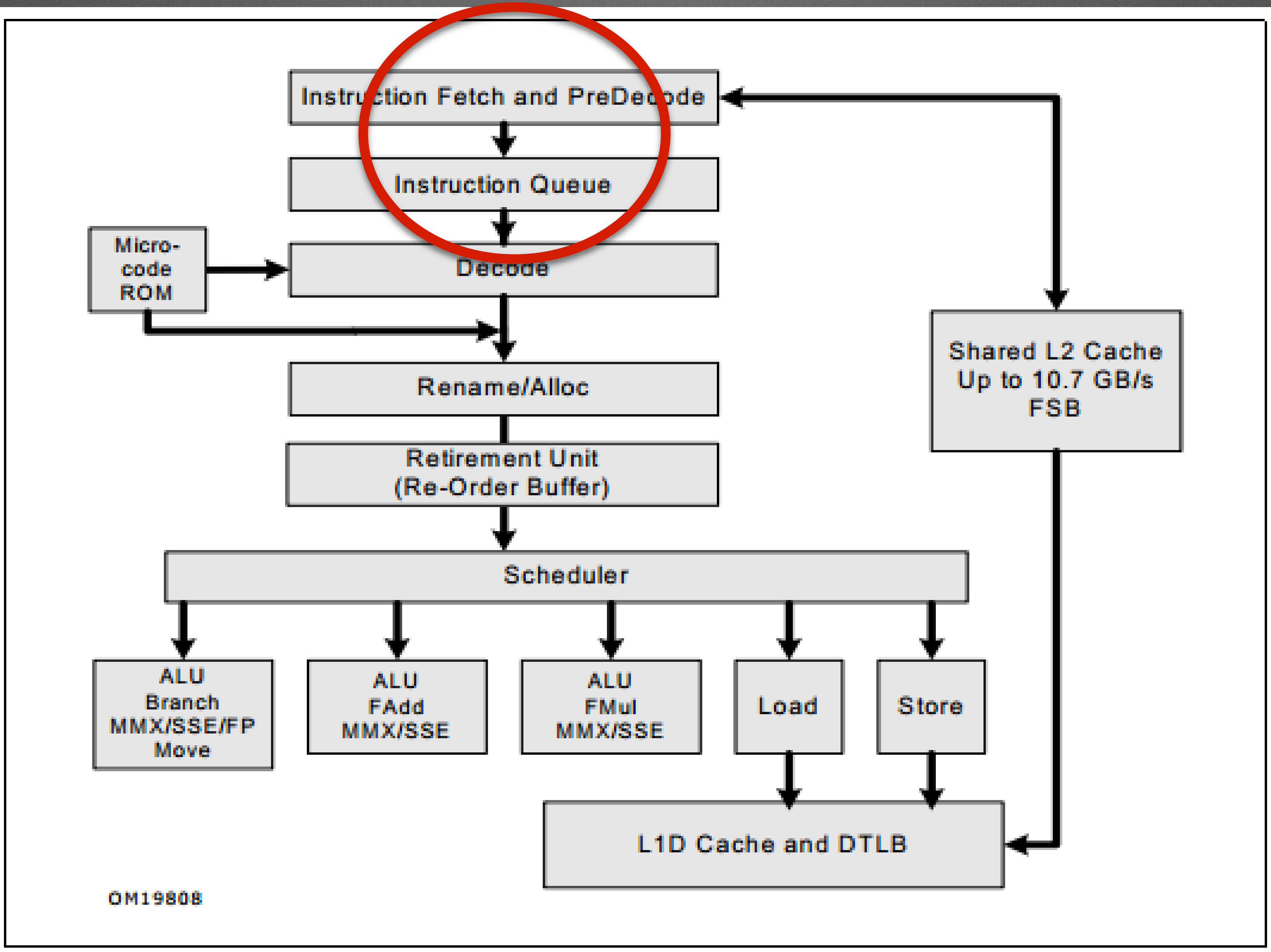

Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
ALU Branch MMX/SSE/FP Move
ALU FAdd MMX/SSE
ALU FMul MMX/SSE
Load
Store
L1D Cache and DTLB
Shared L2 Cache Up to 10.7 GB/s FSB
OM19808

Figure 2-6. Intel Core Microarchitecture Pipeline Functionality