x86 Internals for Fun and Profit

Matt Godbolt
matt@godbolt.org
@mattgodbolt

DRW Trading

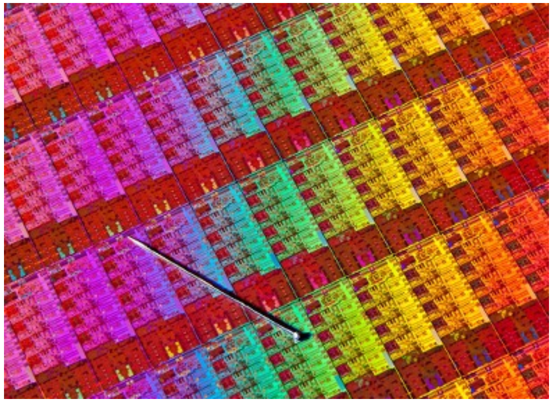


Image credit:
Intel Free Press

Well, mostly fun

- Understanding what's going on helps
  - Can explain unusual behaviour
  - Can lead to new optimization opportunities
- But mostly it's just really interesting!

What's all this then?

- Pipelining
- Branch prediction
- Register renaming
- Out of order execution
- Caching

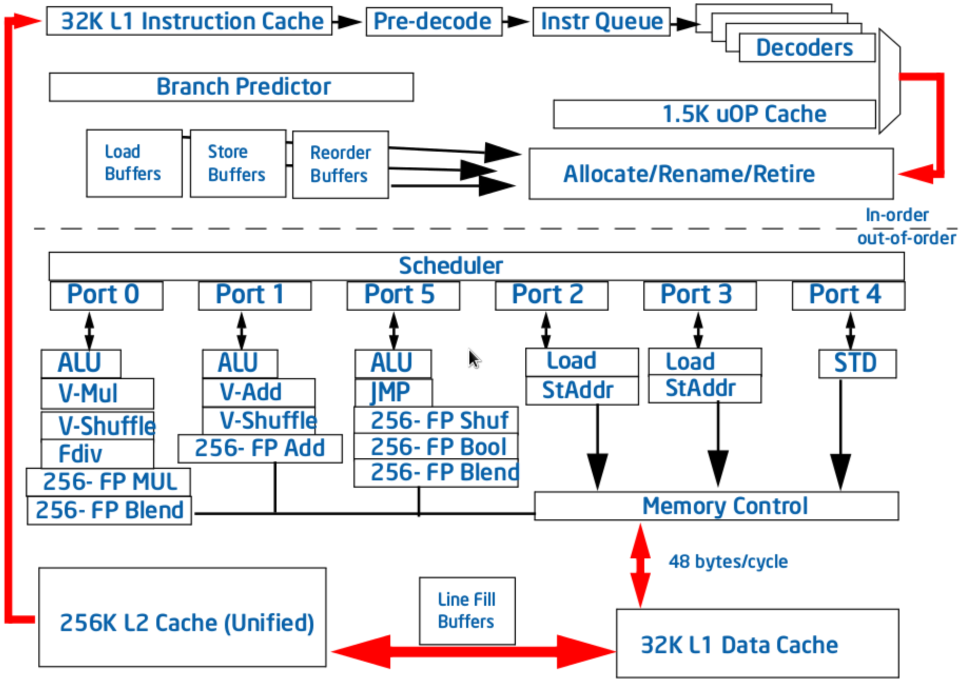


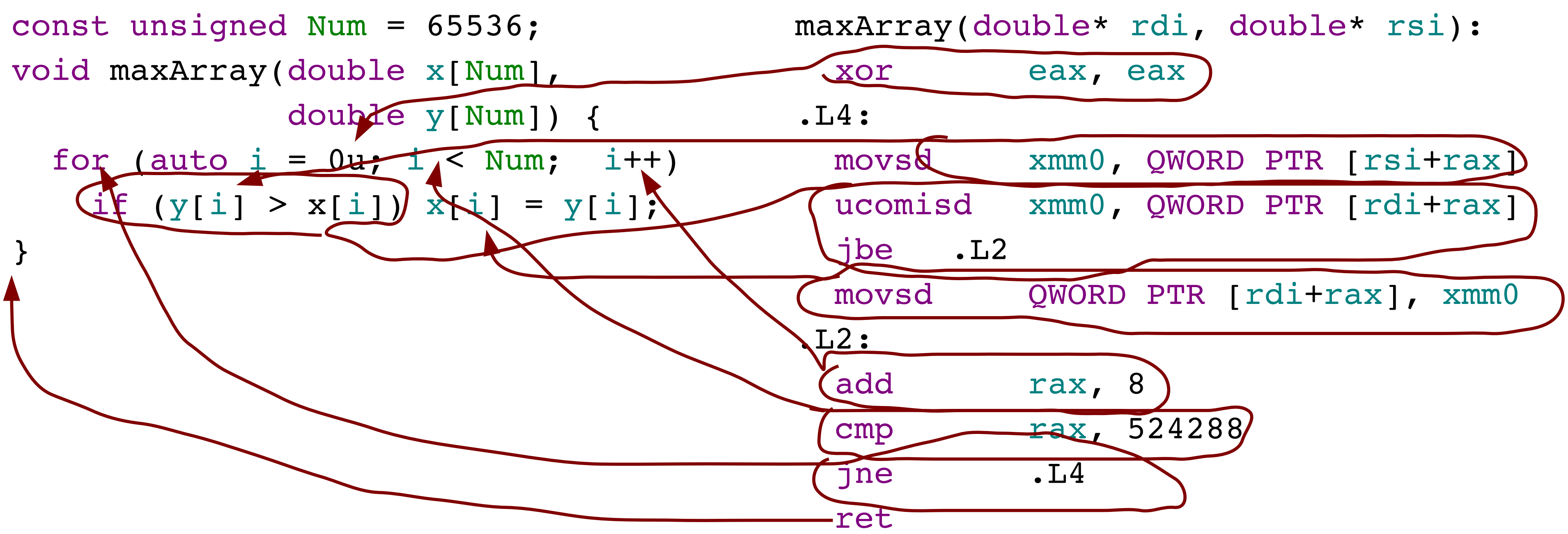
Image credit: Intel

ASM overview

- Intel syntax: `OP dest, source`
- Register operand, e.g.
  - `rax rbx rcx rdx rbp rsp rsi rdi`
    `r8 – r15          xmm0 - xmm15`
  - Partial register e.g. `eax ax ah al`
- Memory operand:
  - `ADDR TYPE mem[reg0 + reg1 * {1,2,4,8}]`
- Constant
- Example:
  - `ADD DWORD PTR array[rbx + 4*rdx], eax`

```
tmp = array[b + d * 4]
tmp = tmp + a
array[b + d * 4] = tmp
```

ASM example

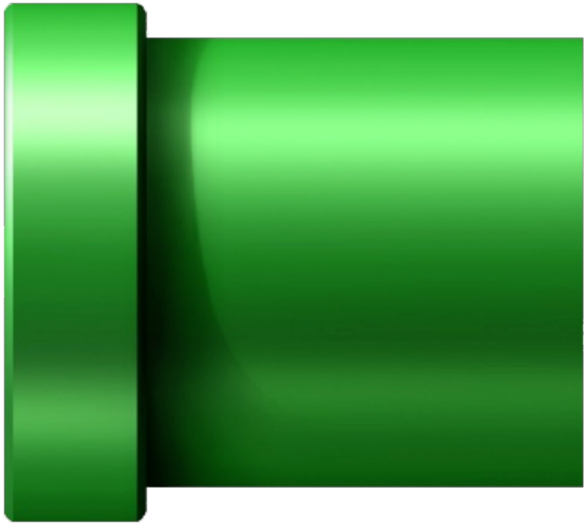


```
const unsigned Num = 65536;
void maxArray(double x[Num],
              double y[Num]) {
  for (auto i = 0u; i < Num;  i++)
    if (y[i] > x[i]) x[i] = y[i];
}
```

```
maxArray(double* rdi, double* rsi):
  xor       eax, eax
.L4:
  movsd     xmm0, QWORD PTR [rsi+rax]
  ucomisd   xmm0, QWORD PTR [rdi+rax]
  jbe   .L2
  movsd     QWORD PTR [rdi+rax], xmm0
.L2:
  add       rax, 8
  cmp       rax, 524288
  jne       .L4
  ret
```

Trip through the Intel pipeline

- Branch prediction
- Fetch
- Decode
- Rename
- Reorder buffer read
- Reservation station
- Execution
- Reorder buffer write
- Retire



BP	Fetch	Decode	Rename	ROB read	RS	Exec	ROB write	Retire

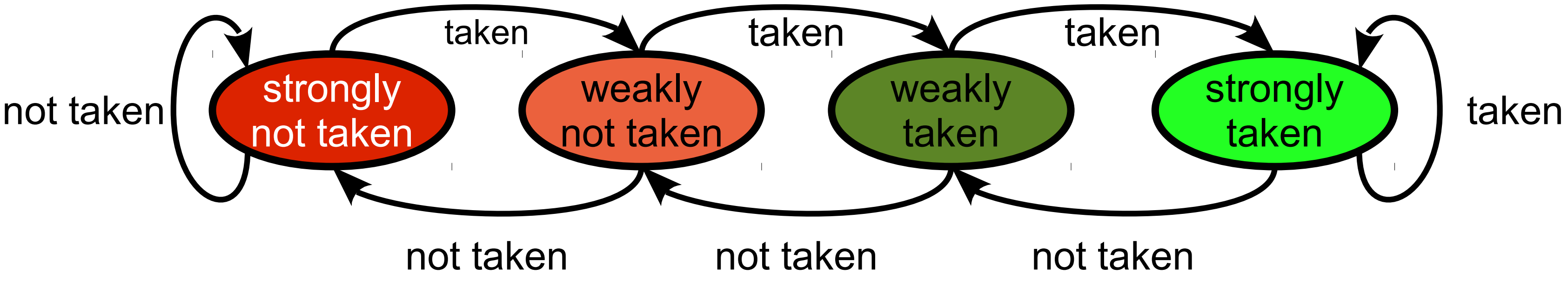
Branch Prediction

- Pipeline is great for overlapping work
- Doesn't deal with feedback loops
- How to handle branches?
  - Informed guess!

BP	Fetch	Decode	Rename	ROB read	RS	Exec	ROB write	Retire

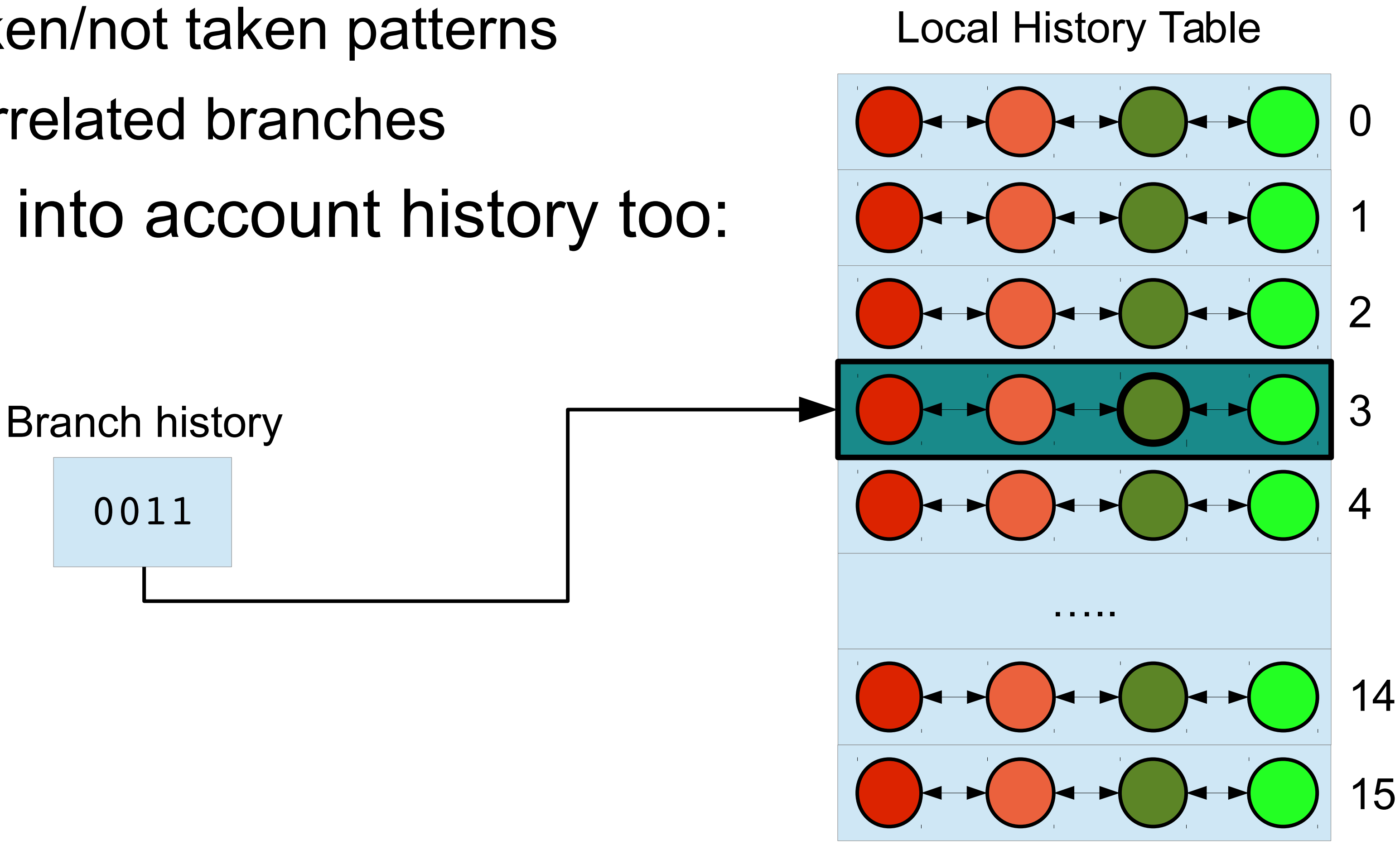
Branch Prediction

- Need to predict:
  - Whether branch is taken (for conditionals)
  - What destination will be (all branches)
- Branch Target Buffer (BTB)
  - Caches destination address
  - Keeps "history" of previous outcomes



Branch Prediction

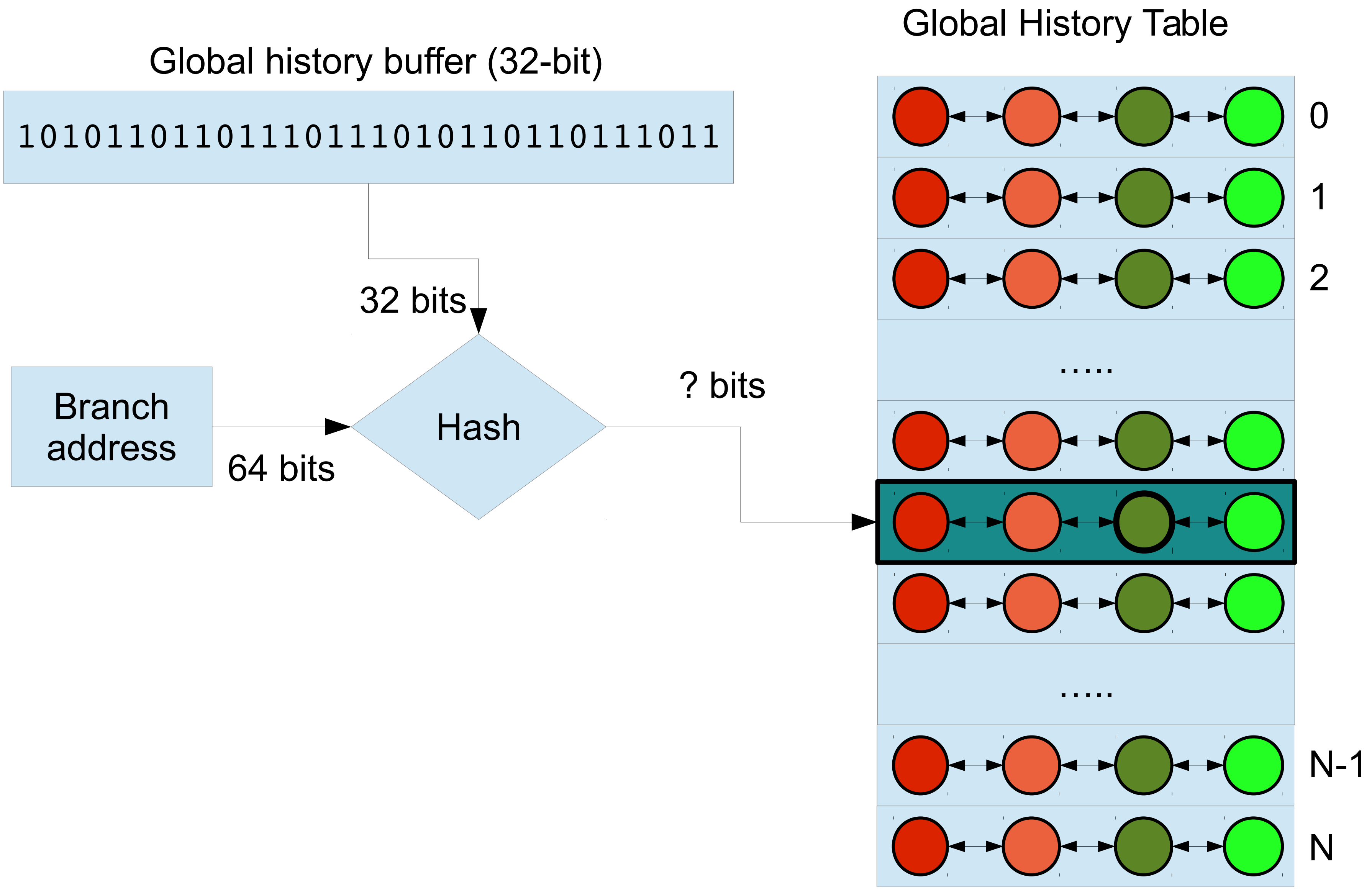
- Doesn't handle
  - taken/not taken patterns
  - correlated branches
- Take into account history too:



Branch Prediction

- Doesn't scale too well
  - n + 2^n*2 bits per BTB entry
- Loop predictors mitigate this
- Sandy Bridge and above use
  - 32 bits of global history
  - Shared history pattern table
  - BTB for destinations

Sandy Bridge Branch Prediction



Global history buffer (32-bit)

10101101101110111010110110111011

32 bits

Branch address

64 bits

Hash

? bits

Global History Table

0

1

2

.....

.....

N-1

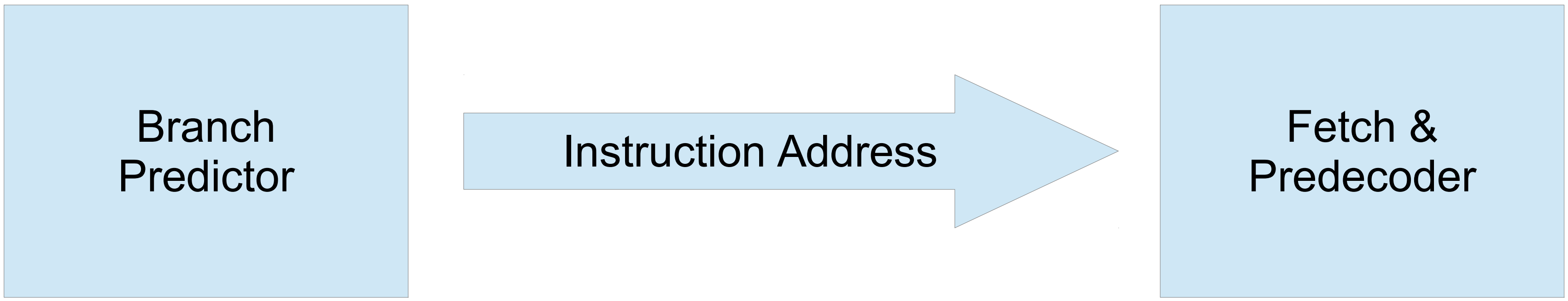
N

Does it matter?

```
def test(array):
  total = num_clipped = clipped_total = 0

  for i in array:
    total += i

    if i < 128:
      num_clipped += 1
      clipped_total += i

  return (total / len(array),
          clipped_total / num_clipped)
```

- Random: 102ns / element
- Sorted: 94ns / element
  - 8% faster!

Branch predictor → Fetcher



Branch Predictor

Instruction Address

Fetch & Predecoder

BP	**Fetch**	Decode	Rename	ROB read	RS	Exec	ROB write	Retire

Fetcher

- Reads 16-byte blocks
- Works out where the instructions are

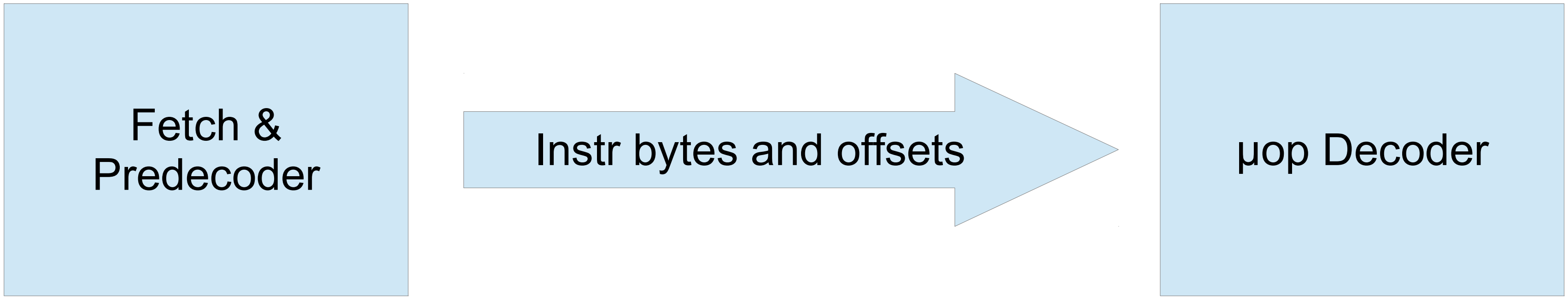
`31 c0 f2 0f 10 04 06 66 0f 2e 04 07 76 05 f2 0f ...`

31 c0 **f2** 0f 10 04 06 **66** 0f 2e 04 07 **76** 05 **f2** 0f ...

```
31 c0                  xor eax, eax
f2 0f 10 04 06         movsd xmm0, QWORD PTR [rsi+rax]
66 0f 2e 04 07         ucomisd xmm0, QWORD PTR [rdi+rax]
76 05                  jbe skip
```

Image Credit: Sini Merikallio [CC-BY-SA-2.0]

Fetcher → Decoder



BP	Fetch	**Decode**	Rename	ROB read	RS	Exec	ROB write	Retire

Decode

Image credit:
Magnus Manske

- Generate µops for each instruction
- Handles up to 4 instructions/cycle
- CISC → internal RISC
- Micro-fusion
- Macro-fusion
- µop cache
  - short-circuits pipeline
  - 1536 entries

Decode example

```
maxArray(double*, double*):
  xor       eax, eax                     eax = 0
.L4:
  movsd     xmm0, QWORD PTR [rsi+rax]    xmm0 = rd64(rsi + rax)
  ucomisd   xmm0, QWORD PTR [rdi+rax]    tmp = rd64(rdi + rax)
                                               compare(xmm0, tmp)
  jbe    .L2                             if (be) goto L2
  movsd     QWORD PTR [rdi+rax], xmm0    wr64(rdi + rax, xmm0)
.L2:
  add       rax, 8                       rax = rax + 8
  cmp       rax, 524288                  comp(rax, 524288); if (ne) goto L4
  jne       .L4
  ret                                    rsp = rsp + 8
                                               goto rd64(rsp -8)
```

Multiple uops

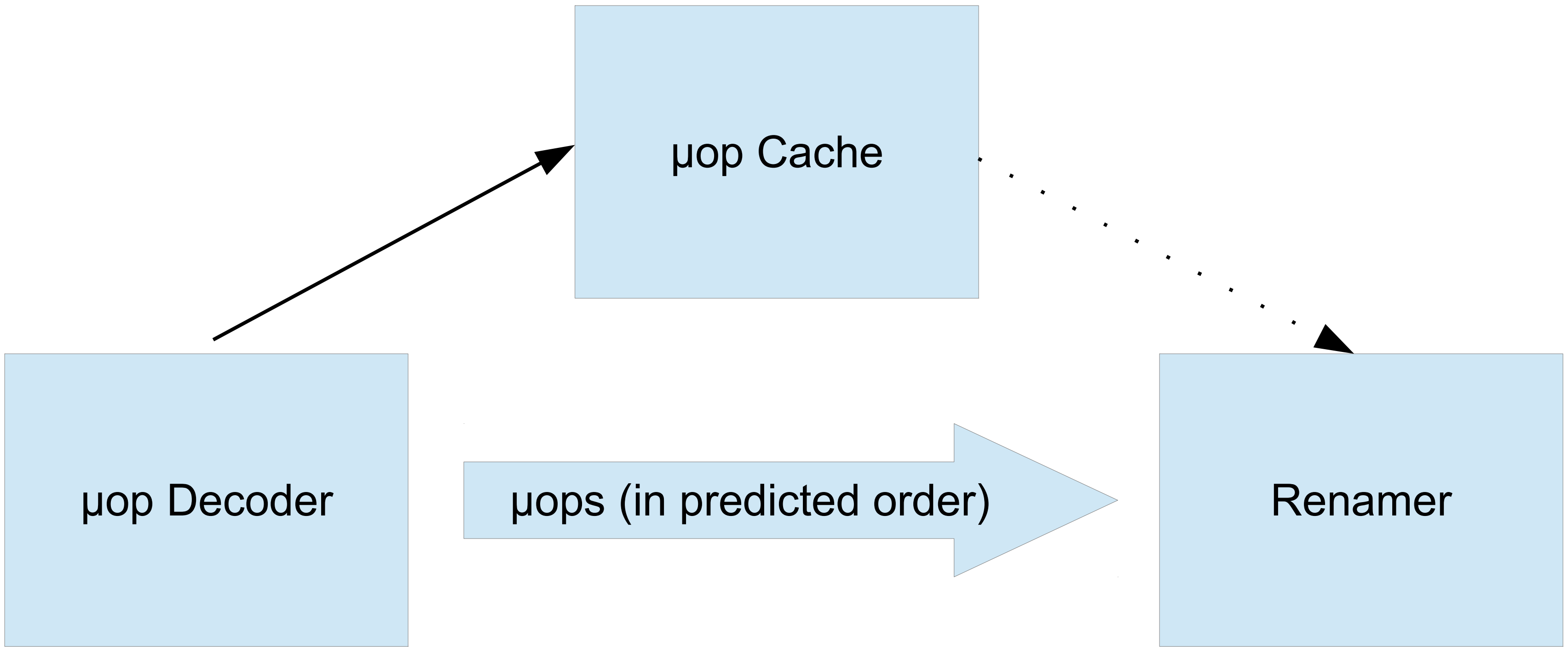
Macro-fusion

But this isn't quite what happens

Something more like...

```
Addr | Micro operations
0x00 | eax = 0
0x08 | xmm0 = rd64(rsi + rax)
0x0d | tmp = rd64(rdi + rax)
0x0d |       comp(xmm0, tmp)
0x12 | if (be) goto 0x19 ; predicted taken
0x19 | rax = rax + 8
0x1d | comp(rax, 524288); if (ne) goto 0x08 ; predicted taken
0x08 | xmm0 = rd64(rsi + rax)
0x0d | tmp = rd64(rdi + rax)
0x0d |       comp(xmm0, tmp)
0x12 | if (be) goto 0x19 ; predicted not taken
0x14 | wr64(rdi+rax, xmm0)
0x19 | rax = rax + 8
0x1d | comp(rax, 524288); if (ne) goto 0x08 ; predicted taken
...
```

Decoder → Renamer



µop Cache

µop Decoder

µops (in predicted order)

Renamer

BP	Fetch	Decode	**Rename**	ROB read	RS	Exec	ROB write	Retire

Renaming

- 16 x86 architectural registers
  - The ones that can be encoded
- Separate independent instruction flows
  - Unlock more parallelism!
- 100+ “registers” on-chip
- Map architectural registers to these
  - On-the-fly dependency analysis

Image Credit:
Like_the_Grand_Canyon
[CC-BY-2.0]

Renaming (example)

```c
extern int globalA;
extern int globalB;

void inc(int x, int y) {

  globalA += x;

  globalB += y;

}
```

```asm
mov eax, globalA

add edi, eax

mov globalA, edi


mov eax, globalB

add esi, eax

mov globalB, esi

ret
```

Renamed

```
eax = rd32(globalA)
edi = edi + eax
wr32(globalA, edi)

eax = rd32(globalB)
esi = esi + eax
wr32(globalB, esi)
```

```
eax_1 = rd32(globalA)
edi_2 = edi_1 + eax_1
wr32(globalA, edi_2)

eax_2 = rd32(globalB)
esi_2 = esi_1 + eax_2
wr32(globalB, esi_2)
```

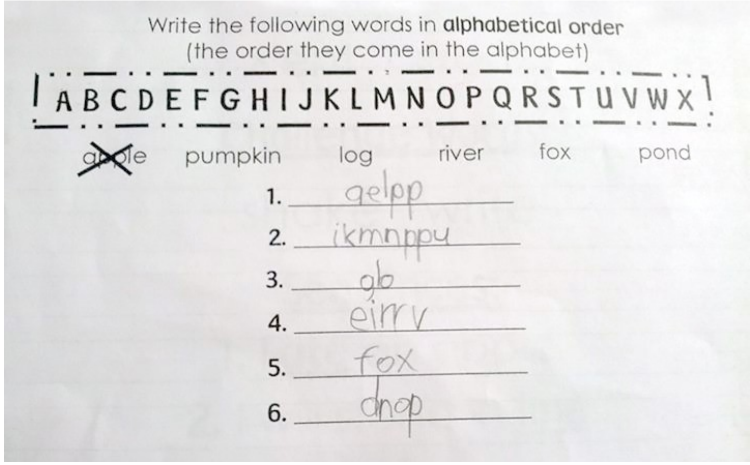
Renaming

- Register Alias Table
  - Tracks current version of each register
  - Maps into Reorder Buffer or PRF
- Understands dependency breaking idioms

  -
    ```
    XOR EAX, EAX
    SUB EAX, EAX
    ```
- Can eliminate moves
  - Ivy Bridge and newer

Reorder Buffer

- Holds state of in-progress μops
- Snoops output of completing μops
- Fetches available inputs
  - From permanent registers
- μops remain in buffer until retired



Write the following words in **alphabetical order**
(the order they come in the alphabet)

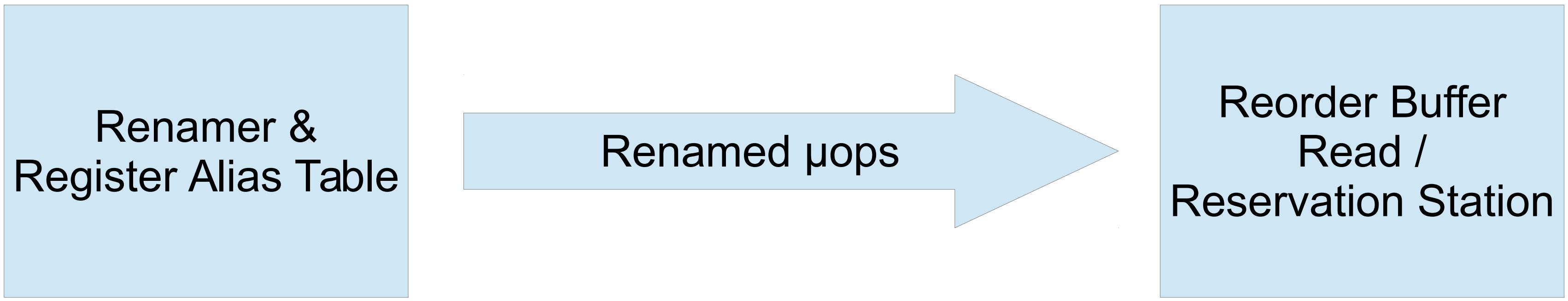
A B C D E F G H I J K L M N O P Q R S T U V W X

~~apple~~ pumpkin log river fox pond

1. aelpp
2. ikmnppu
3. glo
4. eirrv
5. fox
6. dnop

Image credit: B-93.7
Grand Rapids, Michigan

Renamer → Scheduler



Renamer & Register Alias Table

Renamed µops

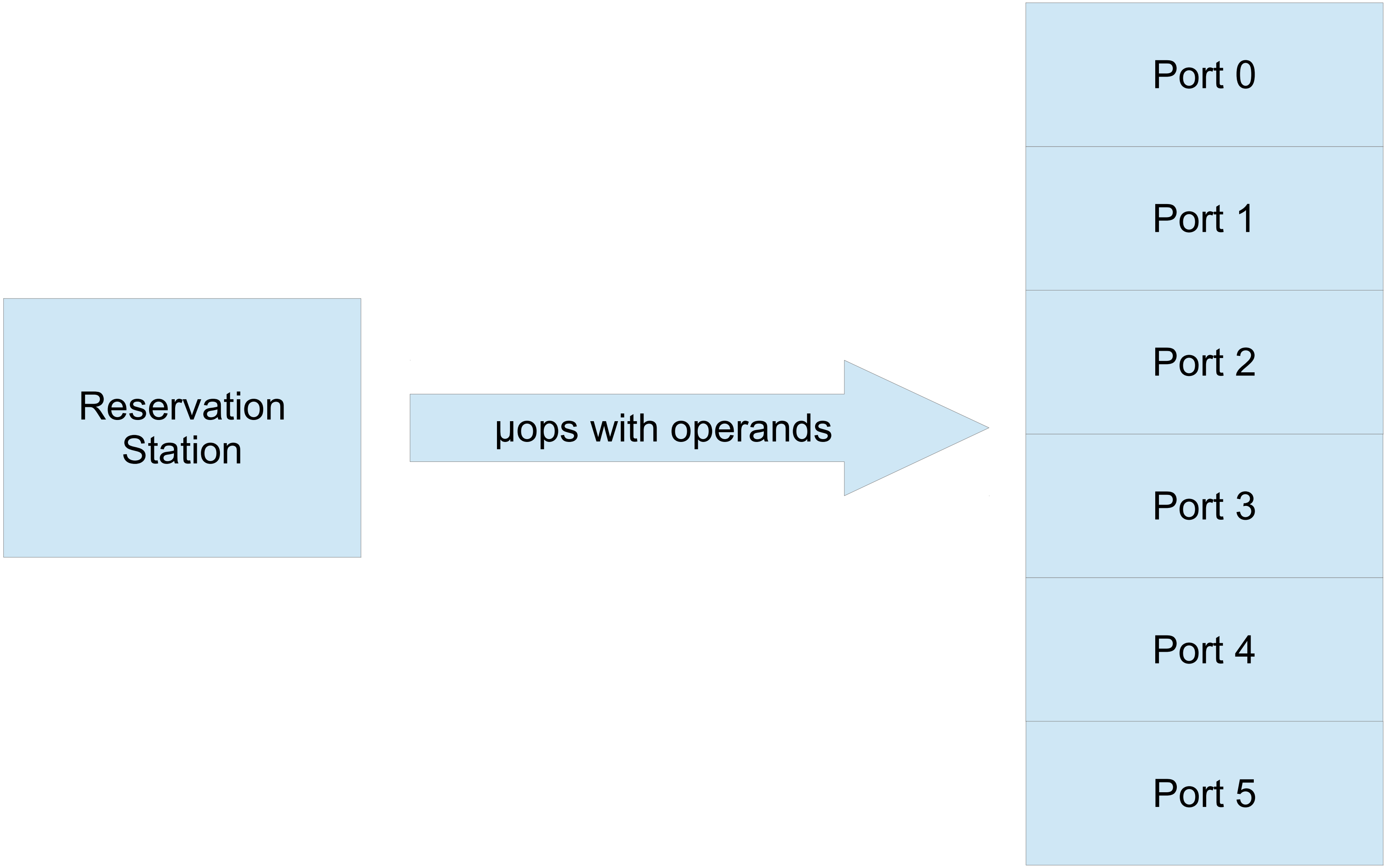
Reorder Buffer Read / Reservation Station

BP	Fetch	Decode	Rename	**ROB read**	**RS**	Exec	ROB write	Retire

Reservation Station

- Connected to 6 execution ports
- Each port can only process subset of µops
- µops queued until inputs ready

RS → Execution Ports



Reservation Station

μops with operands

Port 0

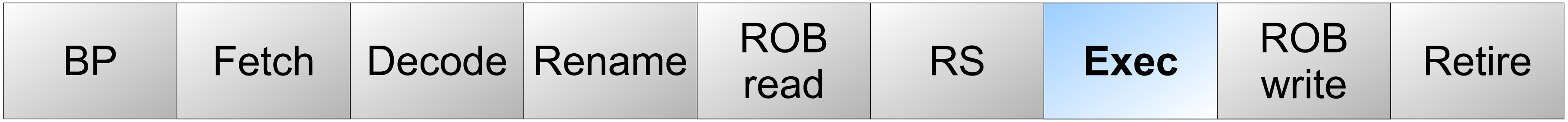
Port 1

Port 2

Port 3

Port 4

Port 5



BP

Fetch

Decode

Rename

ROB read

RS

Exec

ROB write

Retire

Execution!

- Finally, something actually happens!

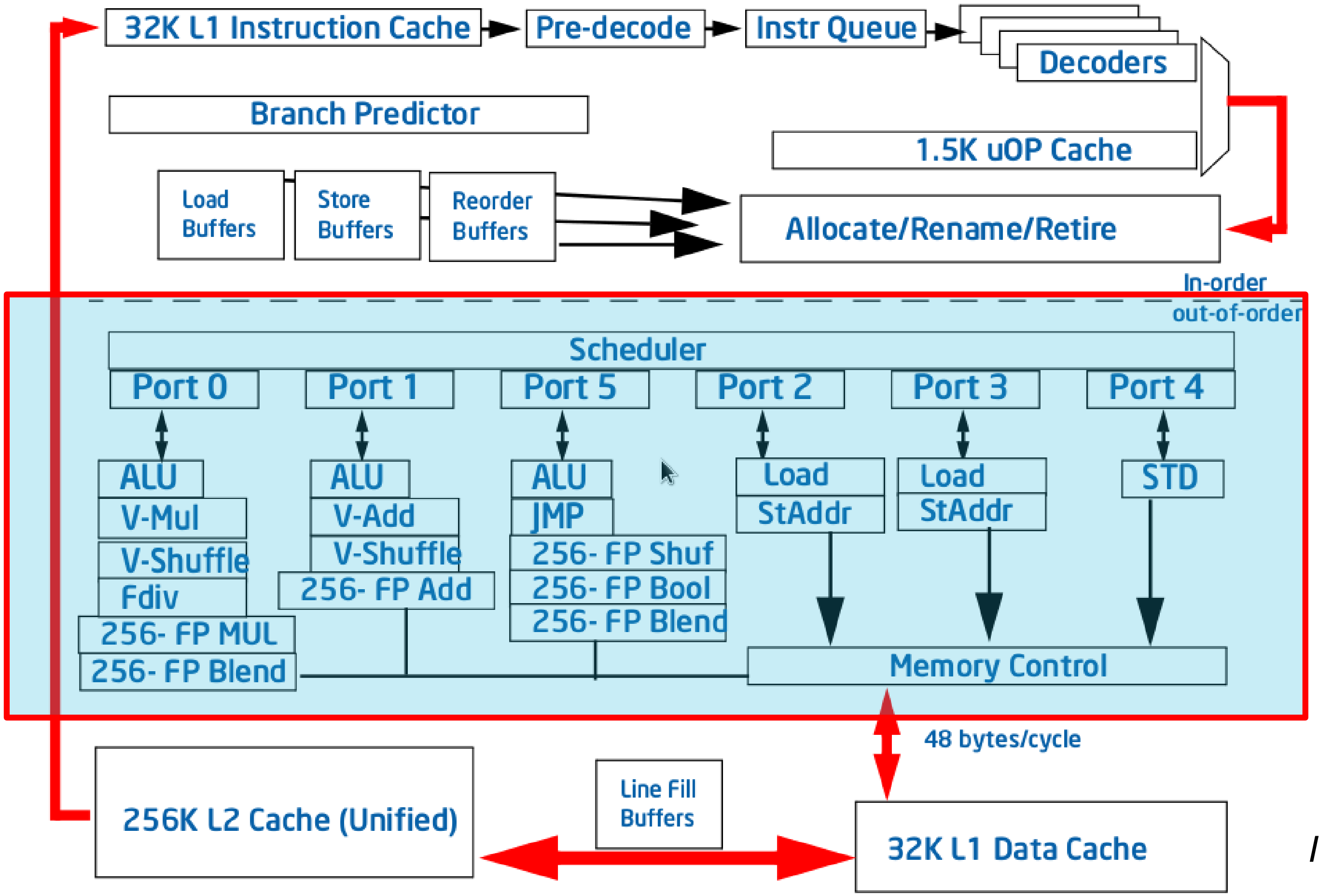
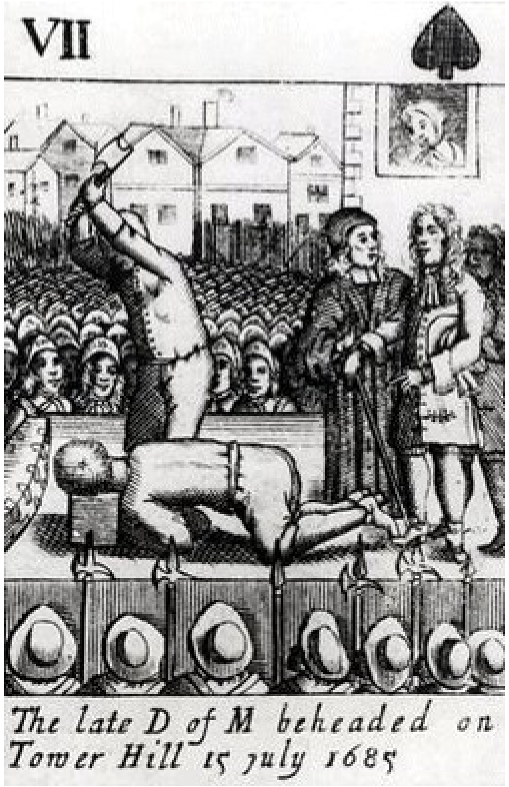


Image credit: Intel

Execution

- 6 execution units
  - 3 general purpose
  - 2 load
  - 1 store
- Most are pipelined
- Issue rates
  - Up to 3/cycle for simple ops
  - FP multiplies 1/cycle



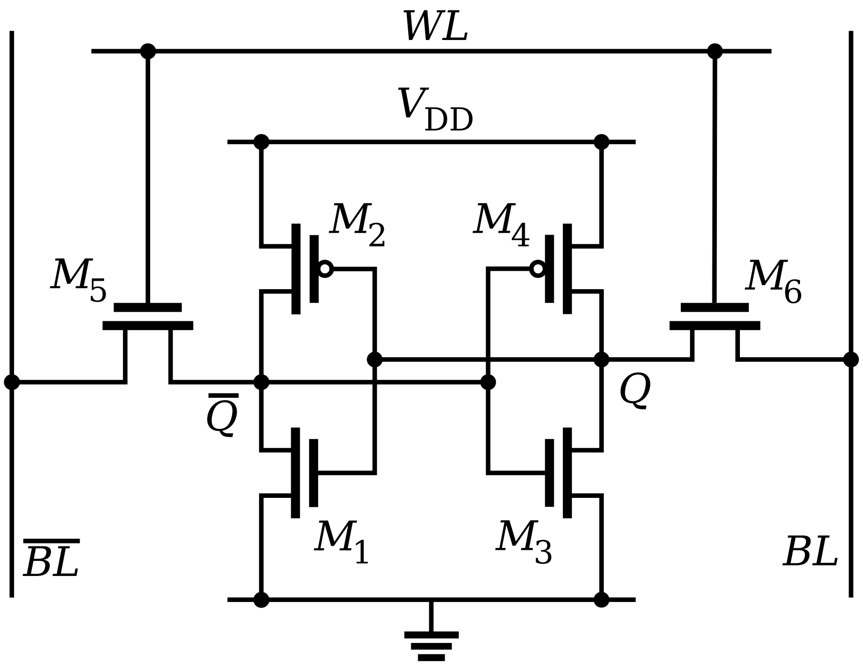
Execution

- Dependency chain latency
  - Logic ops/moves: 1
  - Integer multiply: ~3
  - FP multiply: ~5
  - FP sqrt: 10-24
  - 64-bit integer divide/remainder: 25-84
    - Not pipelined!
  - Memory access 3-250+

Wait a second!

3 - 250+ cycles for a memory access?

SRAM vs DRAM



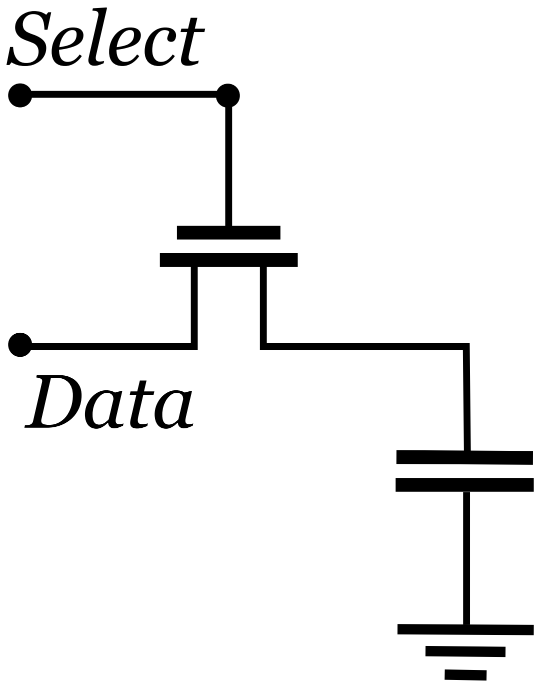
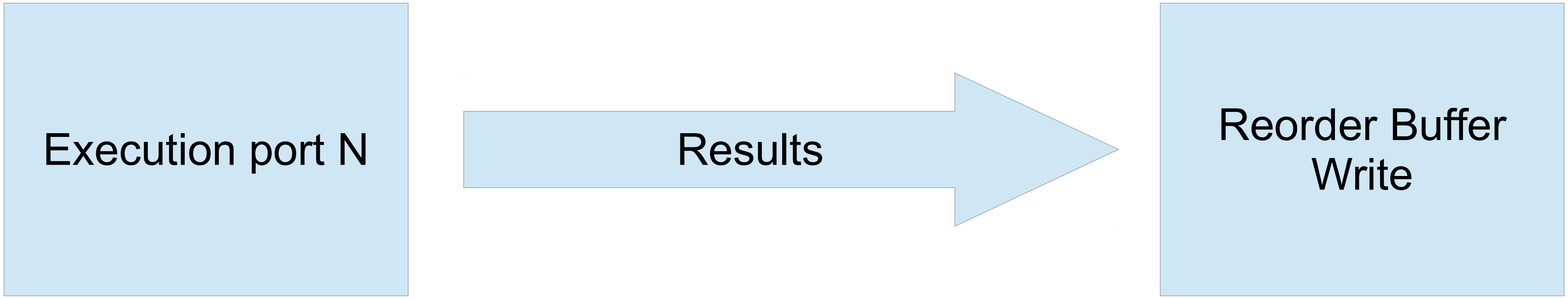


Image source: Wikipedia

Timings and sizes

- Approximate timings for Sandy Bridge
- L1 32KB ~3 cycles
- L2 256KB ~ 8 cycles
- L3 10-20MB ~ 35 cycles
- Main memory ~ 250 cycles

Execution → ROB Write

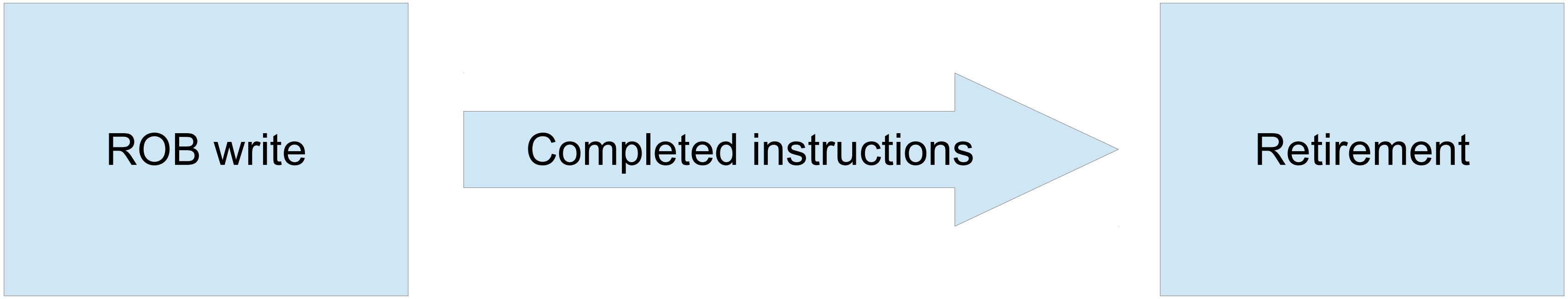


BP	Fetch	Decode	Rename	ROB read	RS	Exec	**ROB write**	Retire

Reorder Buffer Write

- Results written
  - Unblocks waiting operations
- Store forwarding
  - Speculative – can mispredict
- Pass completed µops to retirement

ROB Write → Retirement



ROB write

Completed instructions

Retirement

BP	Fetch	Decode	Rename	ROB read	RS	Exec	ROB write	**Retire**

Retire

- Instructions complete in program order
- Results written to permanent register file
- Exceptions
- Branch mispredictions
- Haswell STM
  - Maybe (Skylake or later?)

Conclusions

- A lot goes on under the hood!

Any questions?

Resources

- Intel's docs
- Agner Fog's info: http://www.agner.org/assem/
- GCC Explorer: http://gcc.godbolt.org/
- http://instlatx64.atw.hu/
- perf
- likwid

Other topics

If I haven't already run ludicrously over time...

ILP Example

```
float mul6(float a[6]) {
  return a[0] * a[1]
    * a[2] * a[3]
    * a[4] * a[5];
}
```

```
movss  xmm0, [rdi]
mulss  xmm0, [rdi+4]
mulss  xmm0, [rdi+8]
mulss  xmm0, [rdi+12]
mulss  xmm0, [rdi+16]
mulss  xmm0, [rdi+20]
```

9 cycles

```
float mul6(float a[6]) {
  return (a[0] * a[1])
    * (a[2] * a[3])
    * (a[4] * a[5]);
}
```

```
movss  xmm0, [rdi]
movss  xmm1, [rdi+8]
mulss  xmm0, [rdi+4]
mulss  xmm1, [rdi+12]
mulss  xmm0, xmm1
movss  xmm1, [rdi+16]
mulss  xmm1, [rdi+20]
mulss  xmm0, xmm1
```

3 cycles

(Back of envelope calculation gives ~28 vs ~21 cycles)

Hyperthreading

- Each HT thread has
  - Architectural register file
  - Loop buffer
- Fetch/Decode shared on alternate cycle
- Everything else shared competitively
  - L1 cache
  - µop cache
  - ROB/RAT/RS
  - Execution resources
  - Etc

ASM example revisited

```
// Compile with -O3 -std=c++11
//   -march=corei7-avx
//   -falign-loops=16
#define ALN64(X) \
  (double*)__builtin_assume_aligned(X, 64)

void maxArray(double* __restrict x,
              double* __restrict y) {
    x = ALN64(x);
    y = ALN64(y);
    for (auto i = 0; i < 65536; i++) {
        x[i] = (y[i] > x[i]) ? y[i] : x[i];
    }
}
```

```
maxArray(double*, double*):
  xor     eax, eax
.L2:
  vmovapd ymm0, YMMWORD PTR [rsi+rax]
  vmaxpd  ymm0, ymm0, YMMWORD PTR [rdi+rax]
  vmovapd YMMWORD PTR [rdi+rax], ymm0
  add     rax, 32
  cmp     rax, 524288
  jne     .L2
  vzeroupper
  ret
```

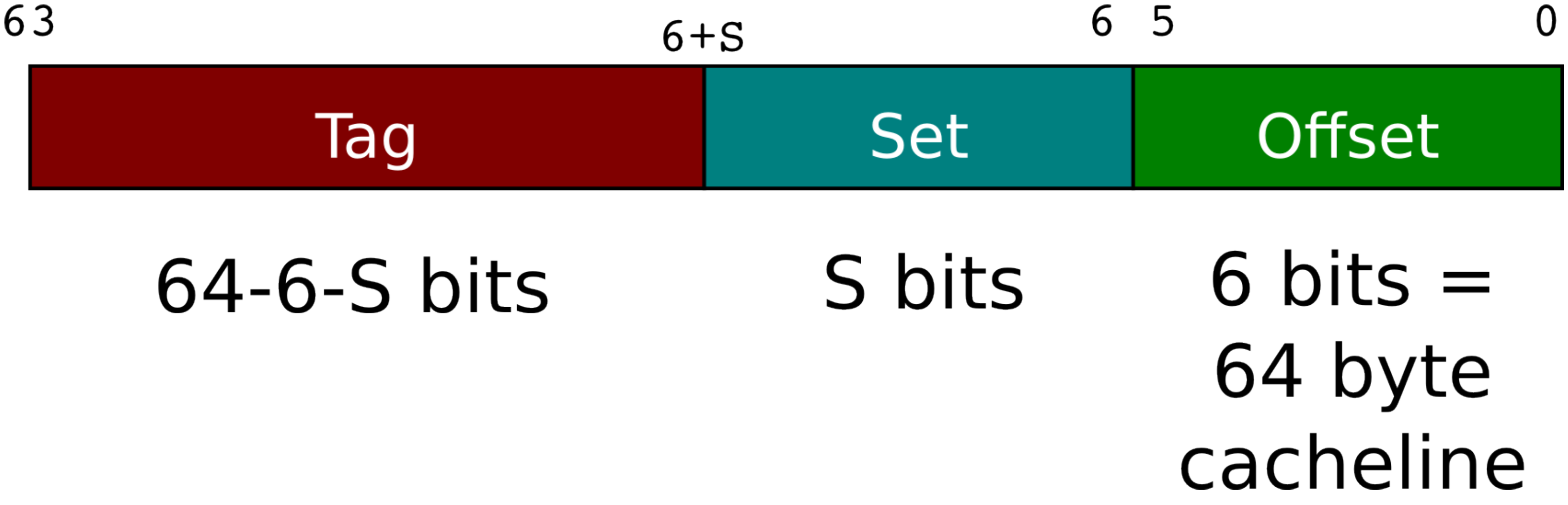
Original algorithm: 40.2μs
Optimized algorithm: 30.1μs

Caching

- Static RAM is small and expensive
- Dynamic RAM is cheap, large, slow
- Use Static RAM as cache for slower DRAM
- Multiple layers of cache

Finding a cache entry

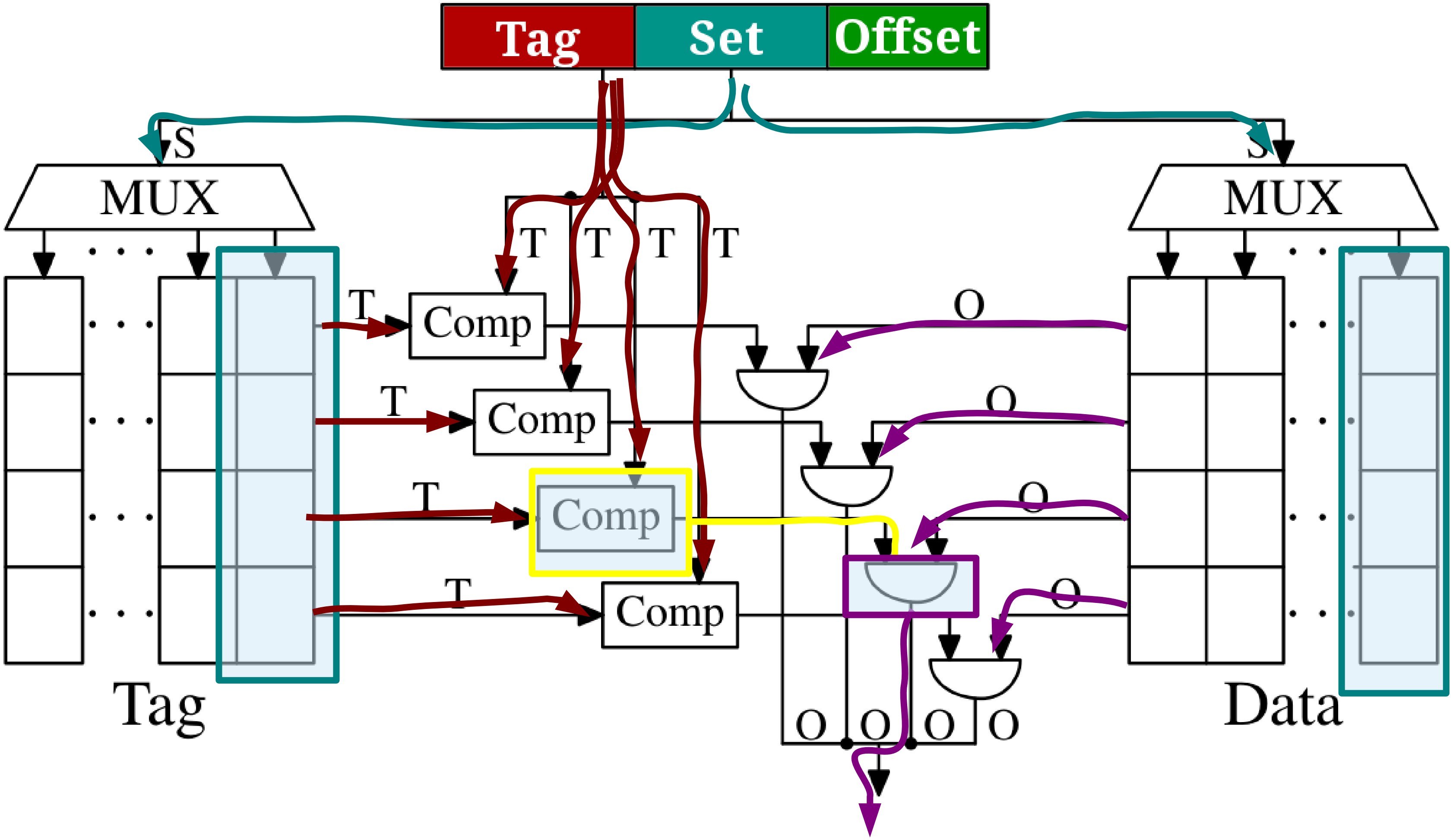
- Organise data in “lines” of 64 bytes
  - Bottom 6 bits of address index into this
- Use some bits to choose a “set”
  - 5 bits for L1, 11 bits for L2, ~13 bits for L3



Finding a cache entry

- Search for cache line within set
  - L1: 8-way, L2: 8-way, L3: 12-way
- Remaining bits (“Tag”) used to find within set
- Why this way?

Caching


Tag
Set
Offset
S
MUX
S
MUX
T
T
T
T
T
Comp
T
Comp
T
Comp
T
Comp
O
O
O
O
O
O
O
O
Tag
Data