High Performance CPU Cores

18-613: Foundations of Computer Systems
5th Lecture, **Feb 26, 2019**

Instructor:

Franz Franchetti

Parts based on "15-740 Computer Architecture" by Nathan Beckmann

Midterm Exam: Mechanics

Pittsburgh (Sections A, B, and C)

- **Part of 1x-x13 exam setup**
  WEH Cluster, desktop computer, together with all others
- **Time slot: March 5, 12pm EST**
  we sign you up
- **If you have a conflict you can start a bit earlier/later**
  e-mail me ASAP

Silicon Valley (Section SA)

- **Separate exam on your own laptop**
  SV B23 118, bring charger for your laptop
- **Time Slot (tentative)**
  March 5, 4:30pm EST – 7:30pm EST (max)

Sunday, March 3: Midterm Review (PIT and SV)

Midterm Exam: 613 Extra Question

One extra question

- **Equally weighted with other 213 questions**
  12.5% of exam score

Three sub-questions

- **State of the art in computing**
  Covers main ideas from Lecture 2 (Multiple choice)
- **C Language**
  Tests your knowledge of C (Understand provided C code)
- **Intel SIMD Extensions**
  Fill holes in Intel SSE/AVX program (Instruction semantics provided)

Practice/example exam will be provided as PDF in Canvas

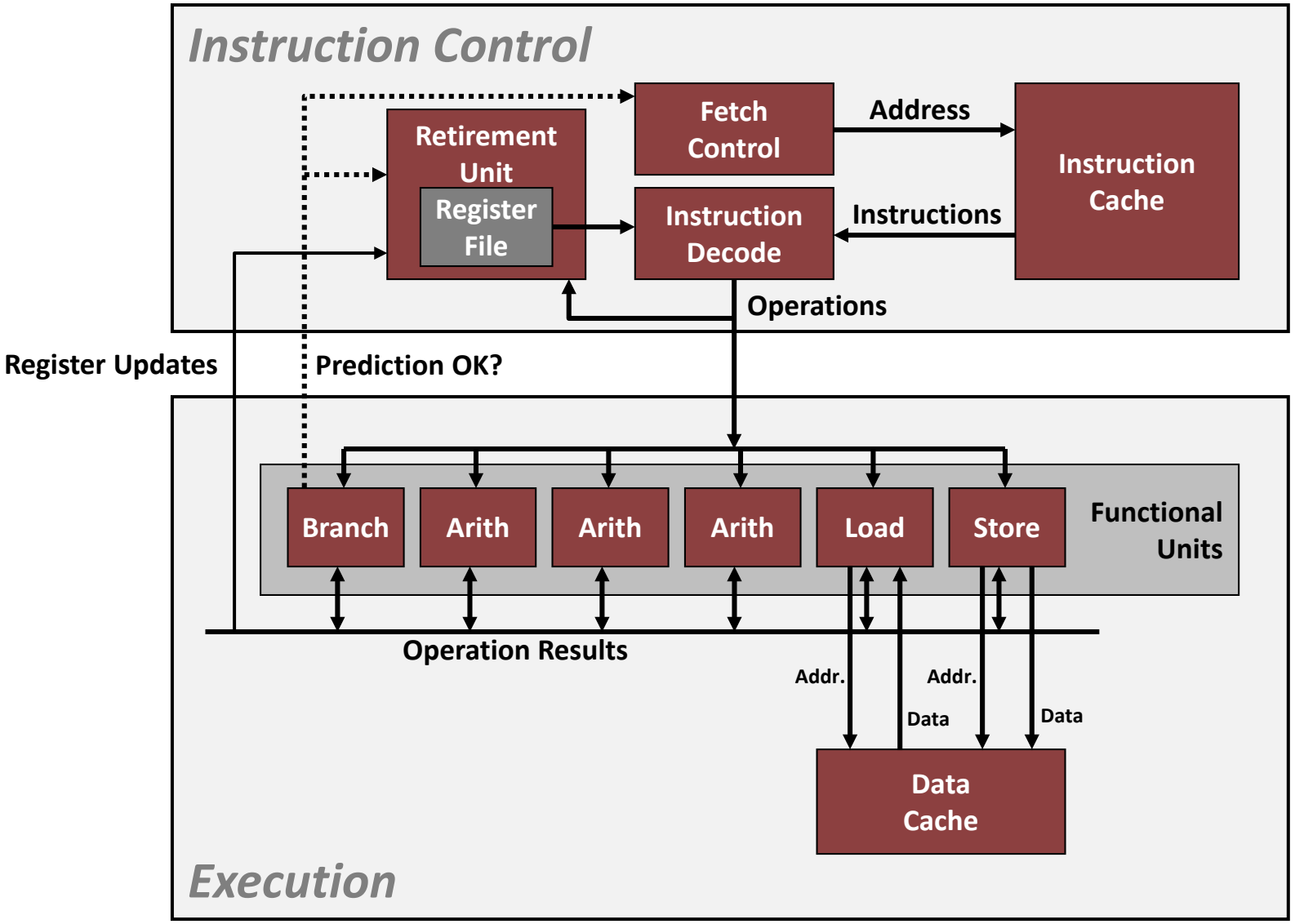
From 213 Lecture 10: Superscalar Processor

- **Definition:** A superscalar processor can issue and execute ***multiple instructions in one cycle***. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- Benefit: without programming effort, superscalar processor can take advantage of the ***instruction level parallelism*** that most programs have

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

Modern CPU Design



Intel Haswell CPU

- 8 Total Functional Units

- **Multiple instructions can execute in parallel**

  2 load, with address computation

  1 store, with address computation

  4 integer

  2 FP multiply

  1 FP add

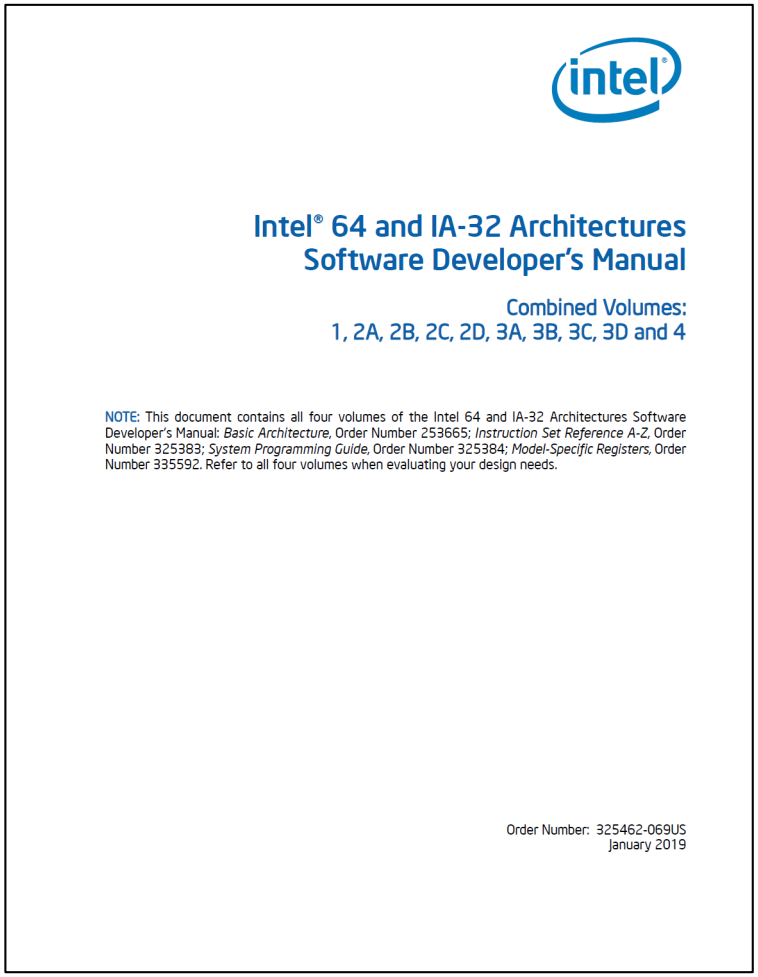
  1 FP divide

- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

Advanced Topics

- **Instruction encoding**
- RISC vs. CISC
- Pipelining
- Out of order execution
- Intel Microarchitecture
- Latencies/Throughput, IACA tool



213 Lecture 5: Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

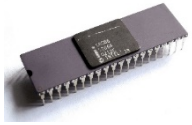
- **Disassembler**

  `objdump -d sum`

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

x86 ISA Evolution In Practice

40 years of x86 binary compatible, but 500x parallelism



1978/1979
8086/8088, 4.77/8 MHz
8087 non-IEEE FPU
x86-16

1985
80386, 12 MHz
80387 IEEE FPU
x86-32

1996
Pentium MMX
166 MHz
x86-32 + MMX

2004
Pentium 4F, 3.6 GHz
Execution disable
x86-64 + SSE3

2011
Sandy Bridge, 3.6 GHz
2-8 cores, HD3000 GPU
x86-64 + AVX

2019
Xeon Platinum 8176F
28 cores, 3.8 GHz
x86-64 + AVX-512

x86 is the abstraction, backwards-compatible ISA extensions necessary

x86 Instruction Format

- Translation of byte-string to assembly instruction
- Instruction decoder needs to interpret the bytes
- A single instruction can be up to 15 bytes (longer -> core dump)

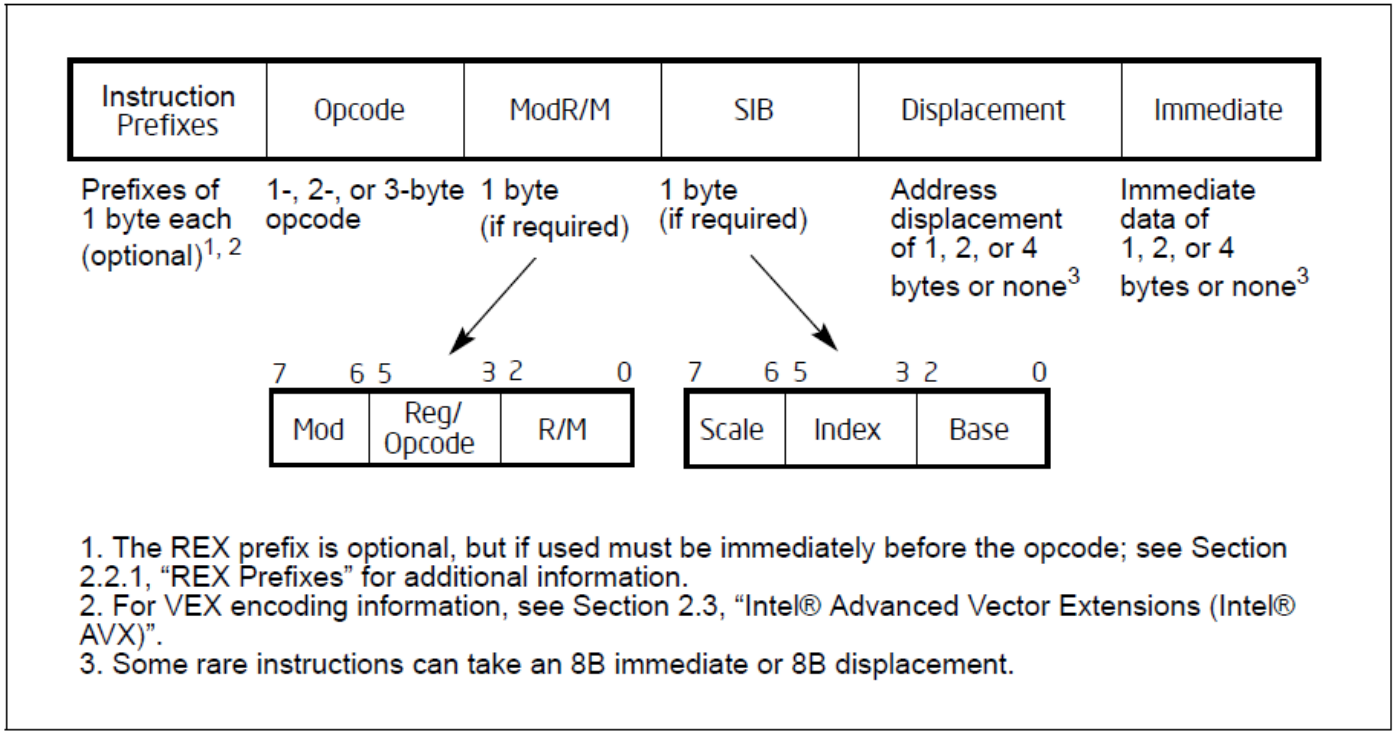
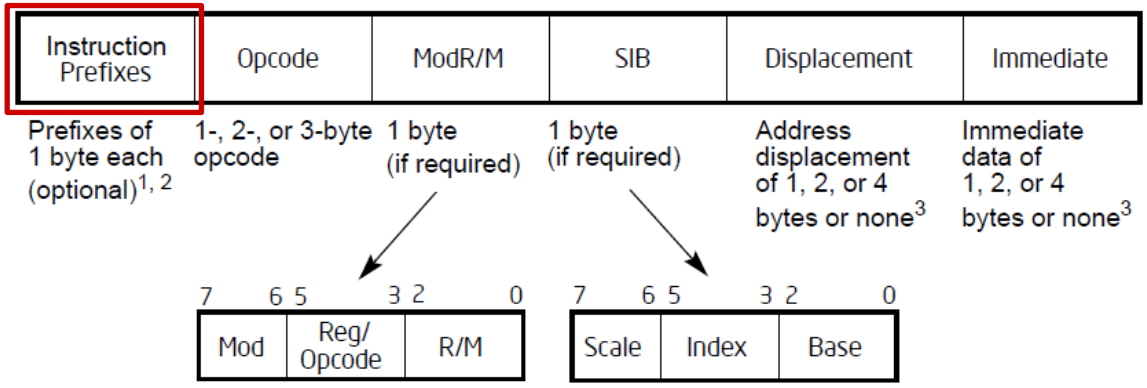


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

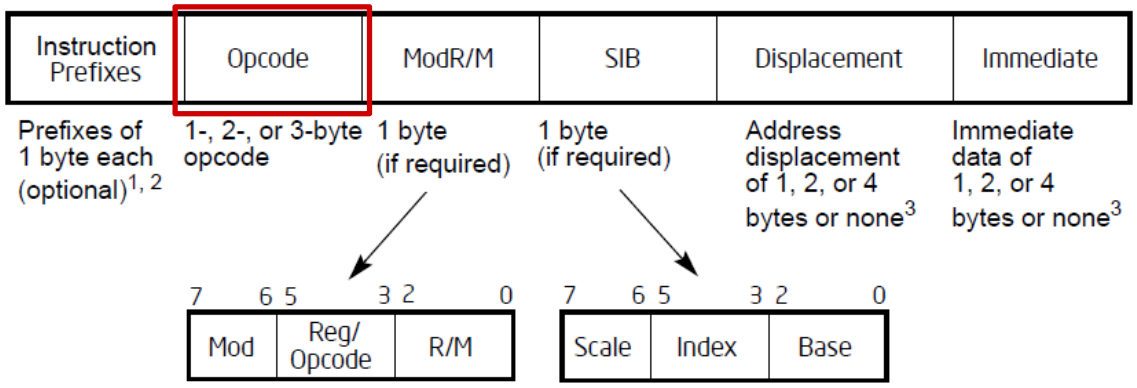
Instruction Format: Prefix

- Group 1: LOCK, REPNE, BND
- Group 2: segment override, branch hints
- Group 3: operand size override
- Group 4: address size override
- 64-bit and SIMD: REX, VEX, EVEX



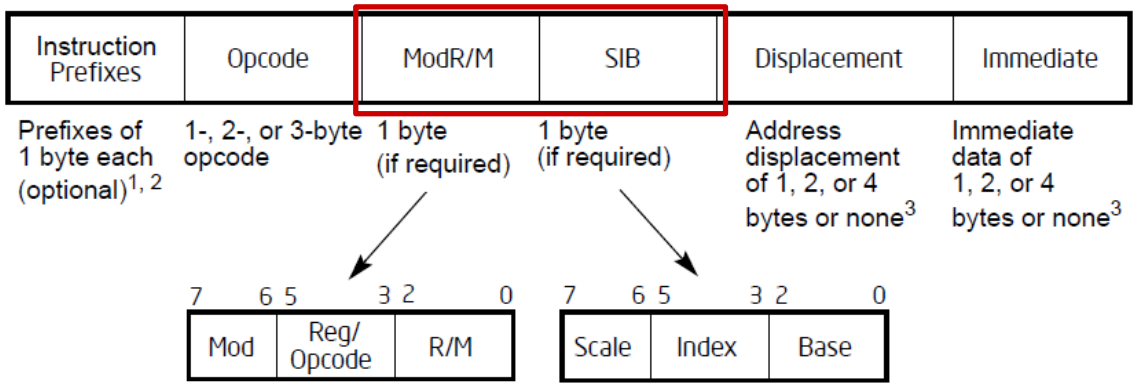
Instruction Format: Opcode

- **One, two or three bytes**
- **Opcode may spill 3 bit into ModR/M**
- **Fields in opcode can define behavior**
  displacement size, register encoding, condition codes, sign extension
- **SIMD instructions require various escape codes**
  this carves out encoding space while keeping instruction length down
- **GDB or disassembler decodes for us**



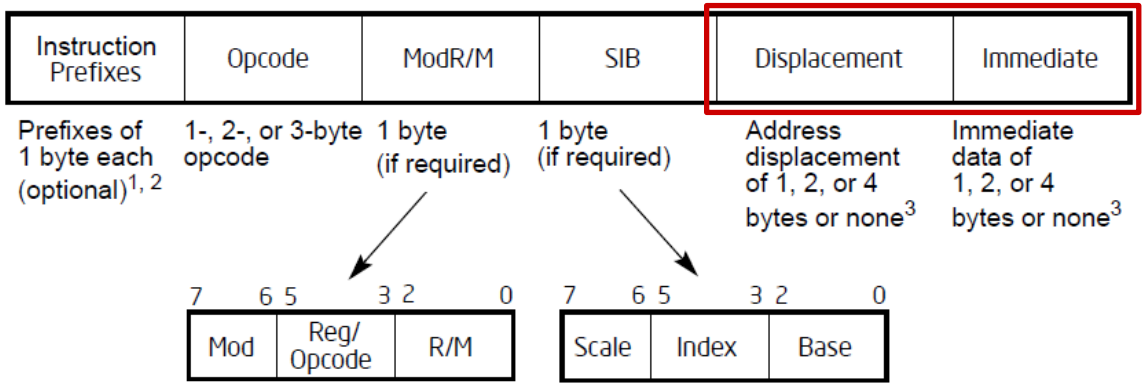
Instruction Format: ModR/M and SIB

- **Mod: Addressing-form specifier**
  32 values: 8 registers and 24 addressing modes
- **Reg/Opcode: used to specify either register or opcode**
  depends on primary Opcode field (previous slide)
- **R/M: register or addressing mode**
  may also be used for opcode information
- **SIB: scale/index/base for certain addressing modes**



Displacement and Immediate

- **Some addressing modes need displacements**
  1, 2 or 4 byte values
- **Immediate**
  Constant values go here: 1, 2, or 4 byte values



What is missing so far?

- **More than 8 registers**
  REX prefix
- **64-bit immediate**
  Semantics change of `mov` instruction: immediate -> load
- **3-operand AVX, AVX-512**
  VEX and EVEX prefix

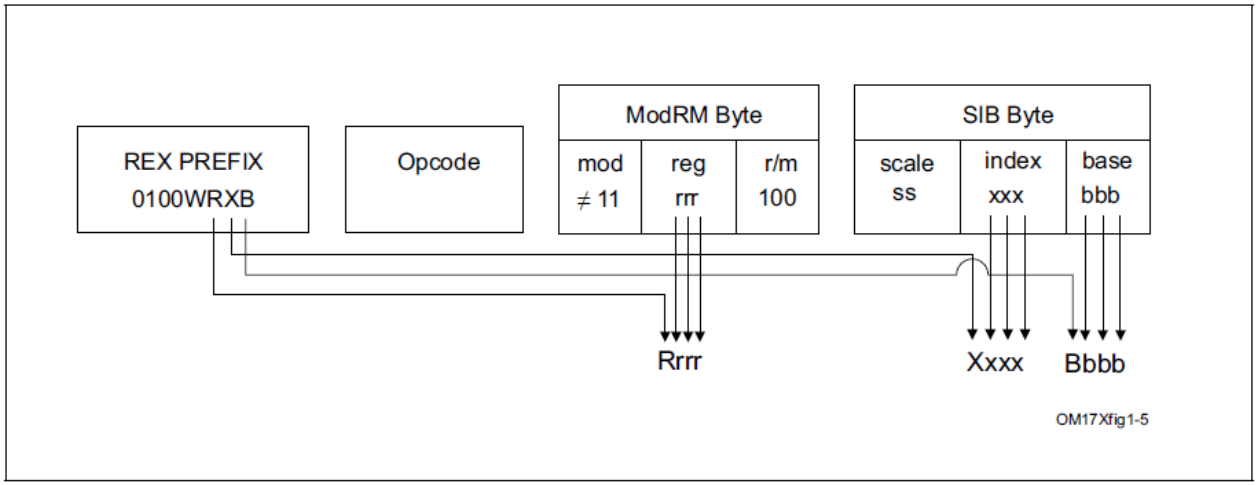


Figure 2-6. Memory Addressing With a SIB Byte

An Example: ADD

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8*, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m8.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
00 /*r*	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8.*
REX + 00 /*r*	ADD *r/m8*, r8**	MR	Valid	N.E.	Add *r8* to *r/m8.*
01 /*r*	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16.*
01 /*r*	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32.*
REX.W + 01 /*r*	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64.*
02 /*r*	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8.*
REX + 02 /*r*	ADD *r8*, r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8.*
03 /*r*	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16.*
03 /*r*	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32.*
REX.W + 03 /*r*	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64.*

NOTES:

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Complete specification: 2 full pages in the instruction manual

Advanced Topics

- Instruction encoding
- **RISC vs. CISC**
- Pipelining
- Out of order execution
- Intel Microarchitecture
- Latencies/Throughput, IACA tool

Background:
Nathan Beckmann: "15-740: Computer Architecture"
https://www.cs.cmu.edu/afs/cs/academic/class/15740-f18/www/

RISC vs. CISC

- **Complex Instruction Set Computing (CISC)**
  - variable length instructions: 1-321 bytes
  - GP registers+special purpose registers+PC+SP+conditions
  - Data: bytes to strings
  - memory-memory instructions
  - special instructions: e.g., crc, polyf, …
- **Reduced Instruction Set Computing (RISC)**
  - fixed length instructions: 4 bytes
  - GP registers + PC
  - load/store with few addressing modes

The RISC Design Tenets

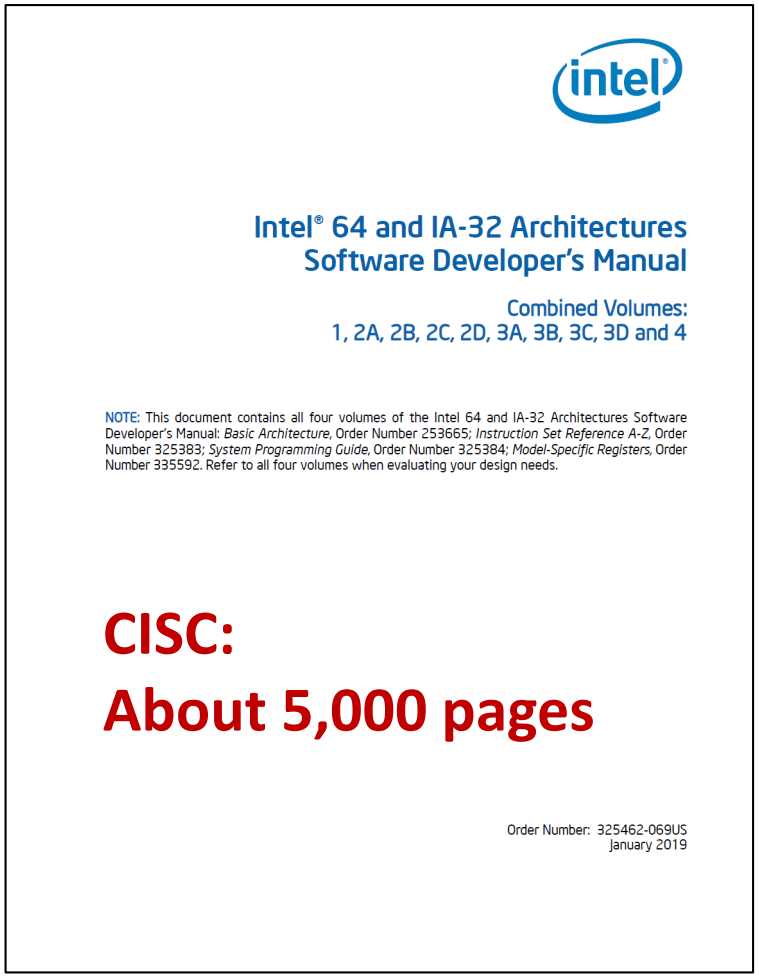
- **Single-cycle execution**
  CISC: many multicycle operations
- **Hardwired (simple) control**
  CISC: microcode for multi-cycle operations
- **Load/store architecture**
  CISC: register-memory and memory-memory
- **Few memory addressing modes**
  CISC: many modes
- **Fixed-length instruction format**
  CISC: many formats and lengths
- **Reliance on compiler optimizations**
  CISC: hand assemble to get good performance
- **Many registers (compilers can use them effectively)**
  CISC: few registers

Schools of ISA Design and Performance

- **Complex instruction set computer (CISC)**
  - Complex instructions -> lots of work per instruction -> fewer instructions per program
  - But… more cycles per instruction & longer clock period
  - Modern μarch gets around most of this!
- **Reduced instruction set computer (RISC)**
  - Fine-grain instructions -> less work per instruction -> more instructions per program
  - But… lower cycles per instruction & shorter clock period
  - Heavy reliance on compiler to "do the right thing"

$$CPU\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

Complexity: x86 vs. RISC-V



intel

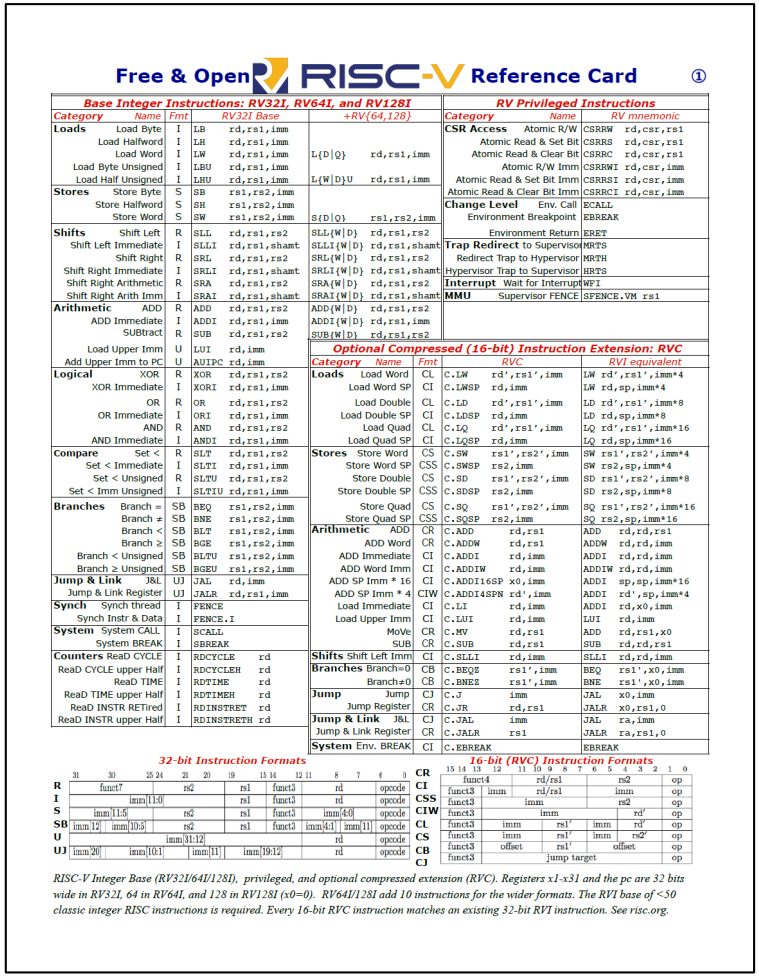
Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325462-069US
January 2019

CISC:
About 5,000 pages



Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd',rs1',imm	LD rd',rs1',imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd',rs1',imm	LQ rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1',rs2',imm	SD rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,imm
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

32-bit Instruction Formats

16-bit (RVC) Instruction Formats

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

RISC: 2 pages

Intel's x86 Trick: RISC Inside

1993: Intel wanted "out-of-order execution" in Pentium Pro

- Hard to do with a coarse grain ISA like x86

Solution? Translate x86 to RISC micro-ops internally (μops)

```
push $eax →      store $eax, -4($esp)
                 addi $esp,$esp,-4
```

- Processor maintains **x86 ISA externally for compatibility**
- But executes **RISC μISA internally for implementability**
- Given translator, x86 almost as easy to implement as RISC
  - *Intel implemented "out-of-order" before any RISC company!*
  - "OoO" also helps x86 more (because ISA limits compiler)
- **Different μops for different designs**
  - Not part of the ISA specification ➔ Implementation flexibility

Potential Micro-op Scheme

Most instructions are a single micro-op

- Add, xor, compare, branch, etc.
- Loads example: `mov -4(%rax), %ebx`
- Stores example: `mov %ebx, -4(%rax)`

Each memory access adds a micro-op

- `addl -4(%rax), %ebx` is two micro-ops (`load`, `add`)
- `addl %ebx, -4(%rax)` is three micro-ops (`load`, `add`, `store`)

Function call (CALL) – 4 uops

- Get program counter, store program counter to stack, adjust stack pointer, unconditional jump to function start

Return from function (RET) – 3 uops

- Adjust stack pointer, load return address from stack, jump register

Again, just a basic idea, micro-ops are specific to each chip

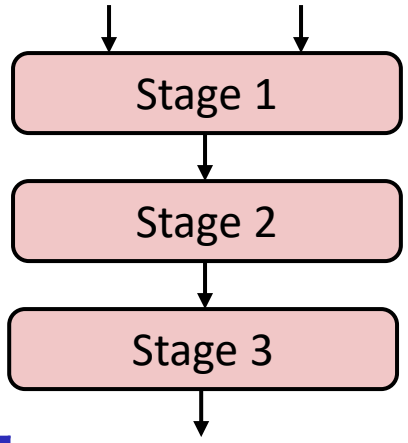
Advanced Topics

- Instruction encoding
- RISC vs. CISC
- **Pipelining**
- Out of order execution
- Intel Microarchitecture
- Latencies/Throughput, IACA tool

Background:
Nathan Beckmann: "15-740: Computer Architecture"
https://www.cs.cmu.edu/afs/cs/academic/class/15740-f18/www/

From 213 Lecture 10: Pipelining

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

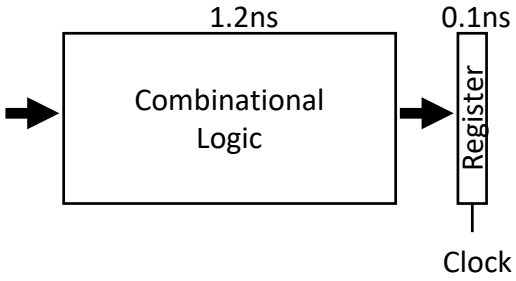


	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

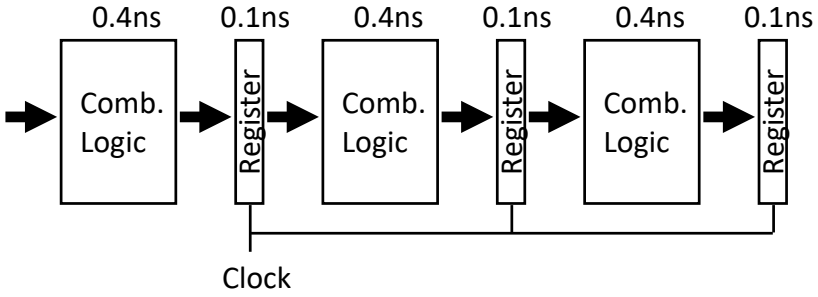
Pipelining: A Closer Look

Unpipelined System



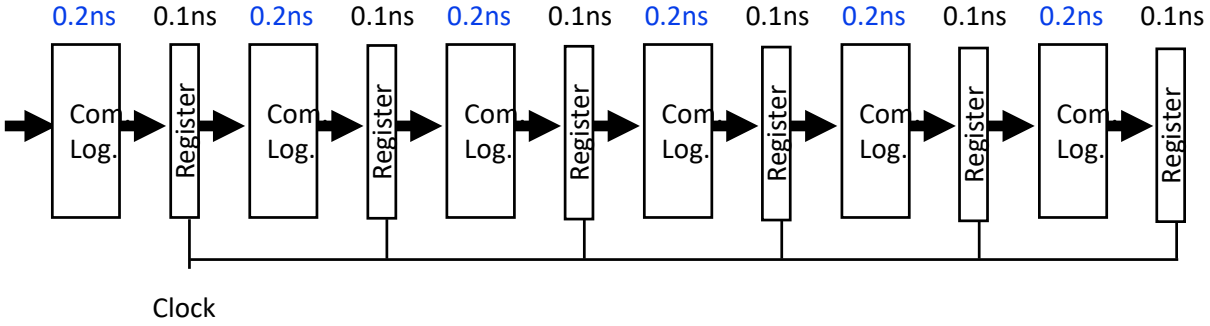
Delay = 1.3 ns
Throughput = 0.77 GHz

3-Stage Pipeline



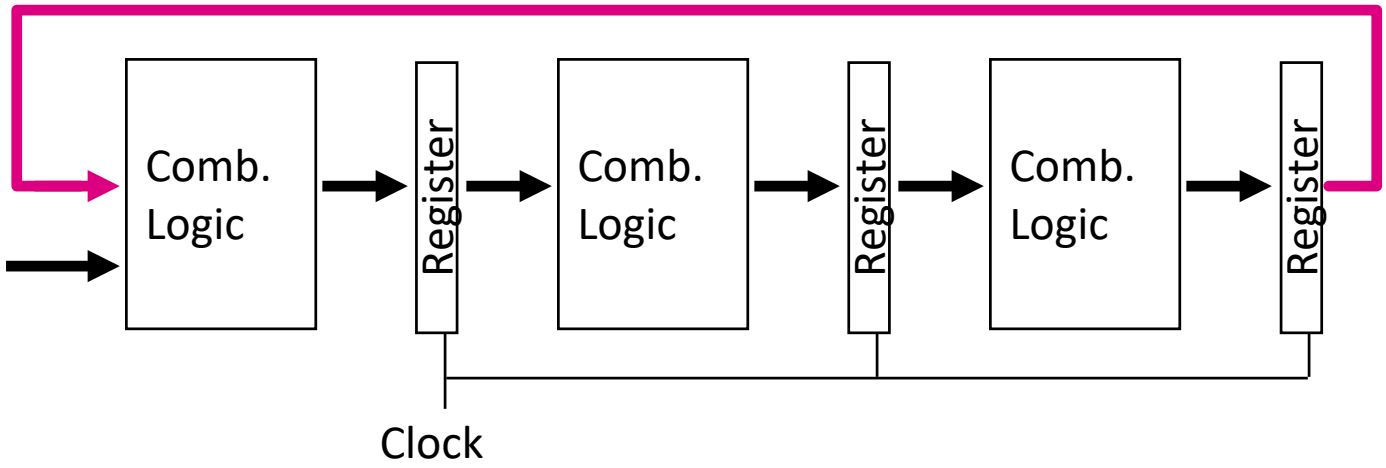
Delay = 1.5 ns
Throughput = 2.0 GHz

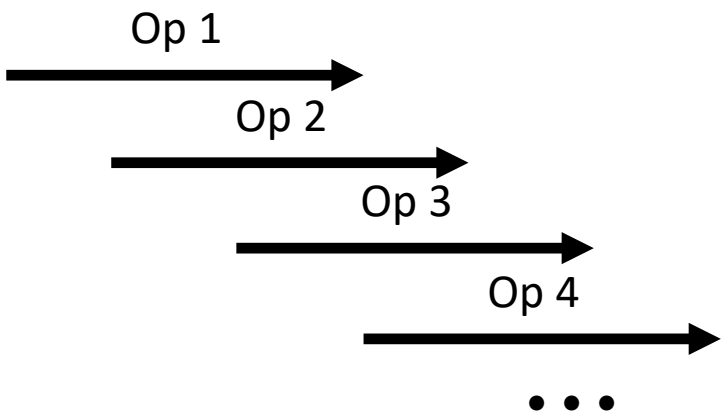
Deep Pipeline



Delay = 1.8ns
Throughput = 3.33GHz

Limitation: Sequential Dependences





- Op4 gets result from Op3!
- *Pipeline Hazard* ➔ *Extra delay*

Intel Pipelines

Year	Micro-architecture	Pipeline stages	max. Clock	Tech process
1989	486 (80486)	3	100 MHz	1000 nm
1993	P5 (Pentium)	5	300 MHz	600 nm
1995	P6 (Pentium Pro; later Pentium II)	14 (17 with load & store/retire)	450 MHz	350 nm
1999	P6 (Pentium III) (Copper Mine)	12 (15 with load & store/retire)	1400 MHz	250 nm
2000	NetBurst (Pentium 4) (Willamette)	20 unified with branch prediction	2000 MHz	180 nm
2002	NetBurst (Pentium 4), (Northwood, Gallatin)		3466 MHz	130 nm
2003	Pentium M	10 (12 with fetch/retire)	2133 MHz	130 nm
2004	NetBurst (Pentium 4) (Prescott)	31 unified with branch prediction	3800 MHz	90 nm
2006	Intel Core	12 (14 with fetch/retire)	3000 MHz	65 nm
2007	Penryn		3333 MHz	45 nm
2008	Nehalem	20 unified (14 without miss prediction)	3600 MHz	
	Bonnell	16 (20 with prediction miss)	2100 MHz	
2010	Westmere	20 unified (14 without miss prediction)	3730 MHz	32 nm
2011	*Saltwell*	16 (20 with prediction miss)	2130 MHz	
	Sandy Bridge	14 (16 with fetch/retire)	4000 MHz	
2012	Ivy Bridge		4100 MHz	22 nm
2013	*Silvermont*	14-17 (16-19 with fetch/retire)	2670 MHz	
	Haswell	14 (16 with fetch/retire)	4400 MHz	
2014	Broadwell		3700 MHz	14 nm
2015	*Airmont*	14-17 (16-19 with fetch/retire)	2640 MHz	
	Skylake	14 (16 with fetch/retire)	4200 MHz	
2016	*Goldmont*	20 unified with branch prediction	2600 MHz	
	Kaby Lake	14 (16 with fetch/retire)	4500 MHz	
2017	Coffee Lake		5000 MHz	
	Goldmont Plus	? 20 unified with branch prediction ?	2800 MHz	
2018	Cannon Lake	14 (16 with fetch/retire)	3200 MHz	10 nm
	Whiskey Lake		4600 MHz	14 nm

https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures

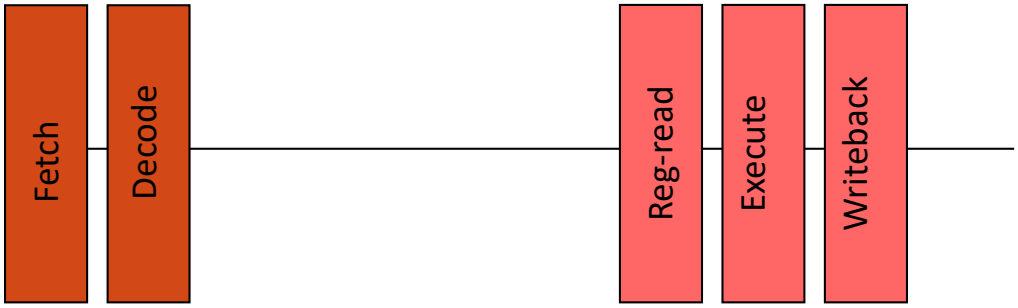
Advanced Topics

- Instruction encoding
- RISC vs. CISC
- Pipelining
- **Out of order execution**
- Intel Microarchitecture
- Latencies/Throughput, IACA tool

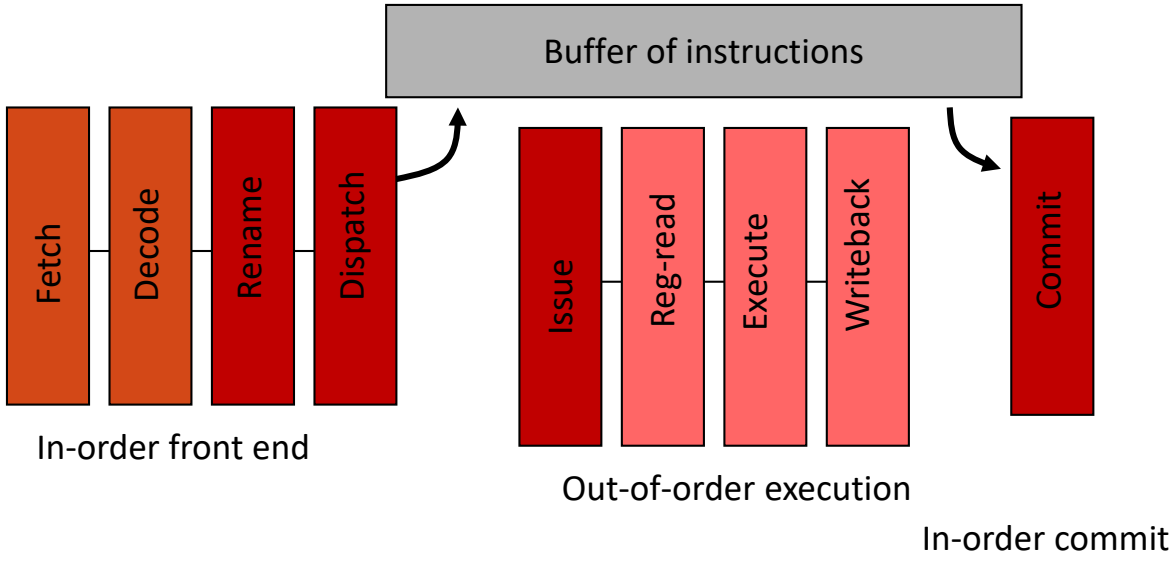
Background:
Nathan Beckmann: "15-740: Computer Architecture"
https://www.cs.cmu.edu/afs/cs/academic/class/15740-f18/www/

Out-of-Order Execution

In-order pipeline

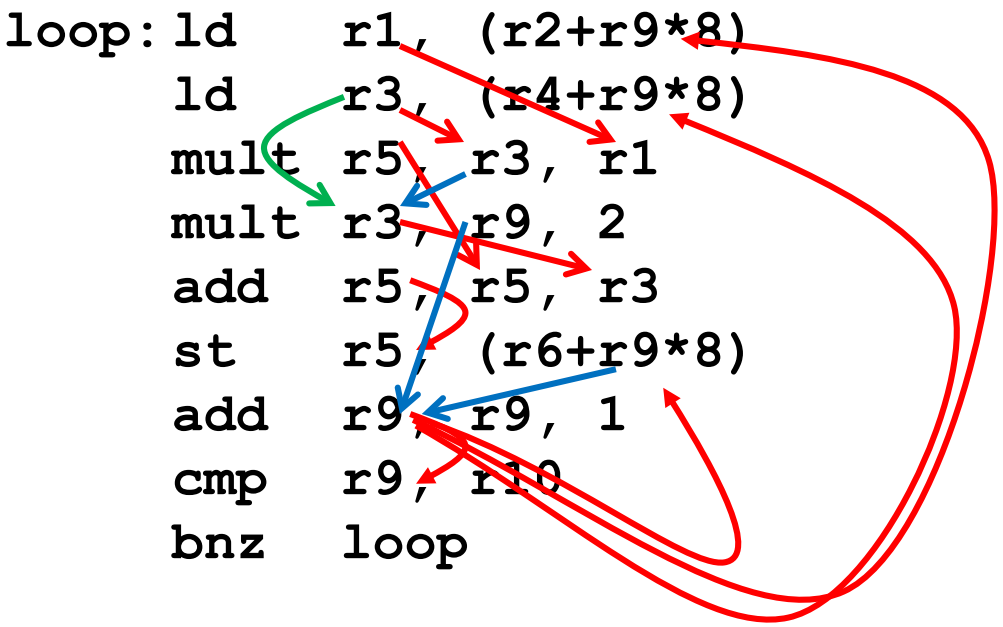


Out-of-order pipeline



Problem: Dependencies

- **Data Dependencies**
  - **RAW:** Read after write. (true dependence)
  - **WAR:** write after read (false dependence)
  - **WAW:** write after write (false dependence)



Alpha ASM Syntax:
`inst dst, src`

Out-of-order execution: execute as soon as RAW allows

Register Renaming

- "Architected" vs "Physical" registers – level of indirection
  - Names: `r1`, `r2`, `r3`
  - Locations: `p1`, `p2`, `p3`, `p4`, `p5`, `p6`, `p7`
  - Original mapping: `r1`→`p1`, `r2`→`p2`, `r3`→`p3`, `p4`–`p7` are "available"

MapTable

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

FreeList

p4,p5,p6,p7
p5,p6,p7
p6,p7
p7

Original insns

```
add r2,r3→r1
sub r2,r1→r3
mul r2,r3→r3
div r1,4→r1
```

Renamed insns

```
add p2,p3→p4
sub p2,p4→p5
mul p2,p5→p6
div p4,4→p7
```

  - Renaming – conceptually write each register once
    - + Removes false dependences
    - + Leaves true dependences intact!
  - When to reuse a physical register? After overwriting is done

Register renaming: resolves false dependencies

Out Of Order: Dynamic Scheduling

```
add p2,p3→p4
sub p2,p4→p5
mul p2,p5→p6
div p4,4→p7
```

Ready Table

Time

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

```
add p2,p3→p4
sub p2,p4→p5 and div p4,4→p7
mul p2,p5→p6
```

- Instructions fetch/decoded/renamed into *re-order buffer (ROB)*
- Check which instructions are ready and execute earliest ready instruction

Dynamic Scheduling: instruction level parallelism and throughput

Branch Prediction

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **Cannot resolve until outcome determined by branch/integer unit**

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```

Branch Misprediction Invalidation

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Assume vector length = 100

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

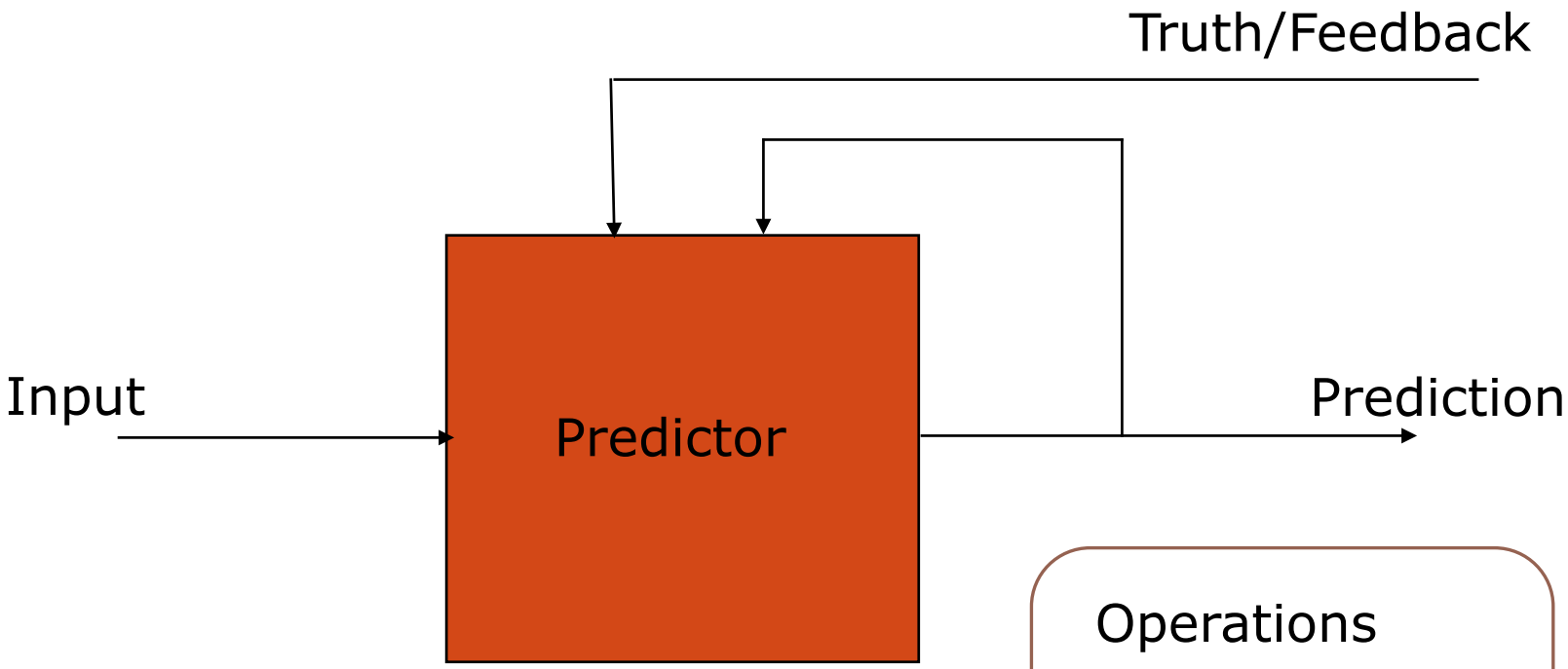
i = 100

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Invalidate

Branch Predictor



Prediction as a feedback control process

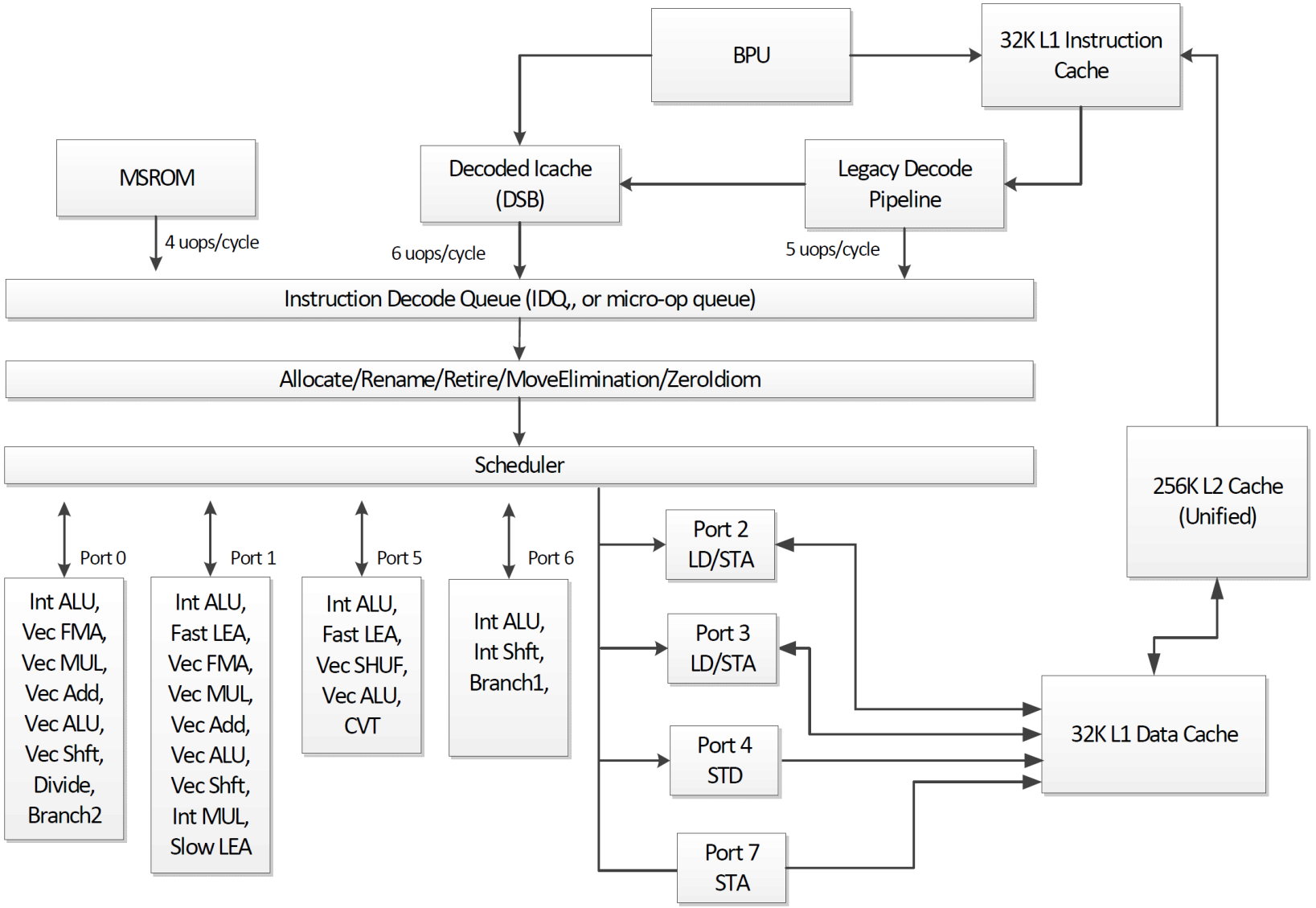
Operations

- Predict
- Update

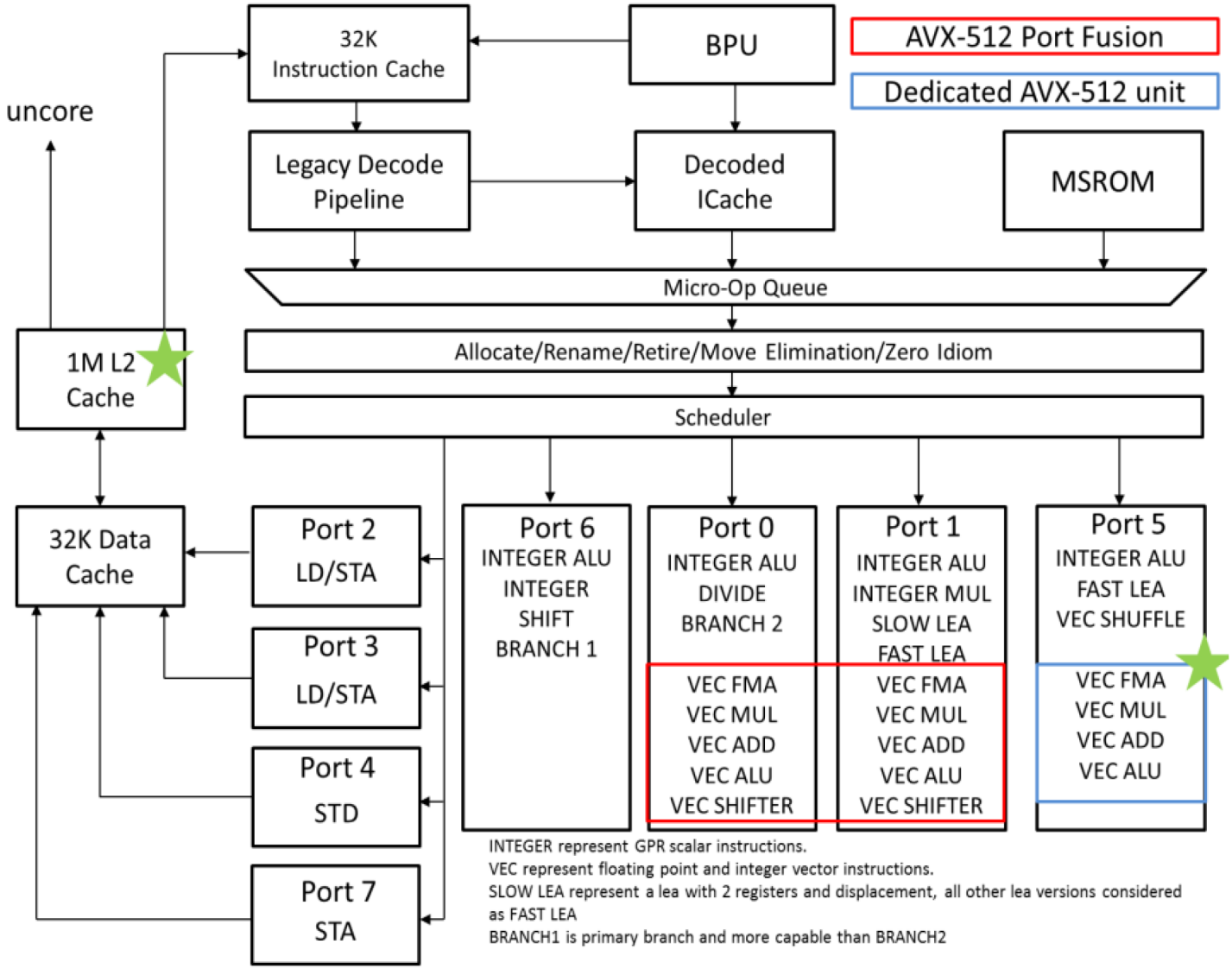
Advanced Topics

- Instruction encoding
- RISC vs. CISC
- Pipelining
- Out of order execution
- **Intel Microarchitecture**
- Latencies/Throughput, IACA tool

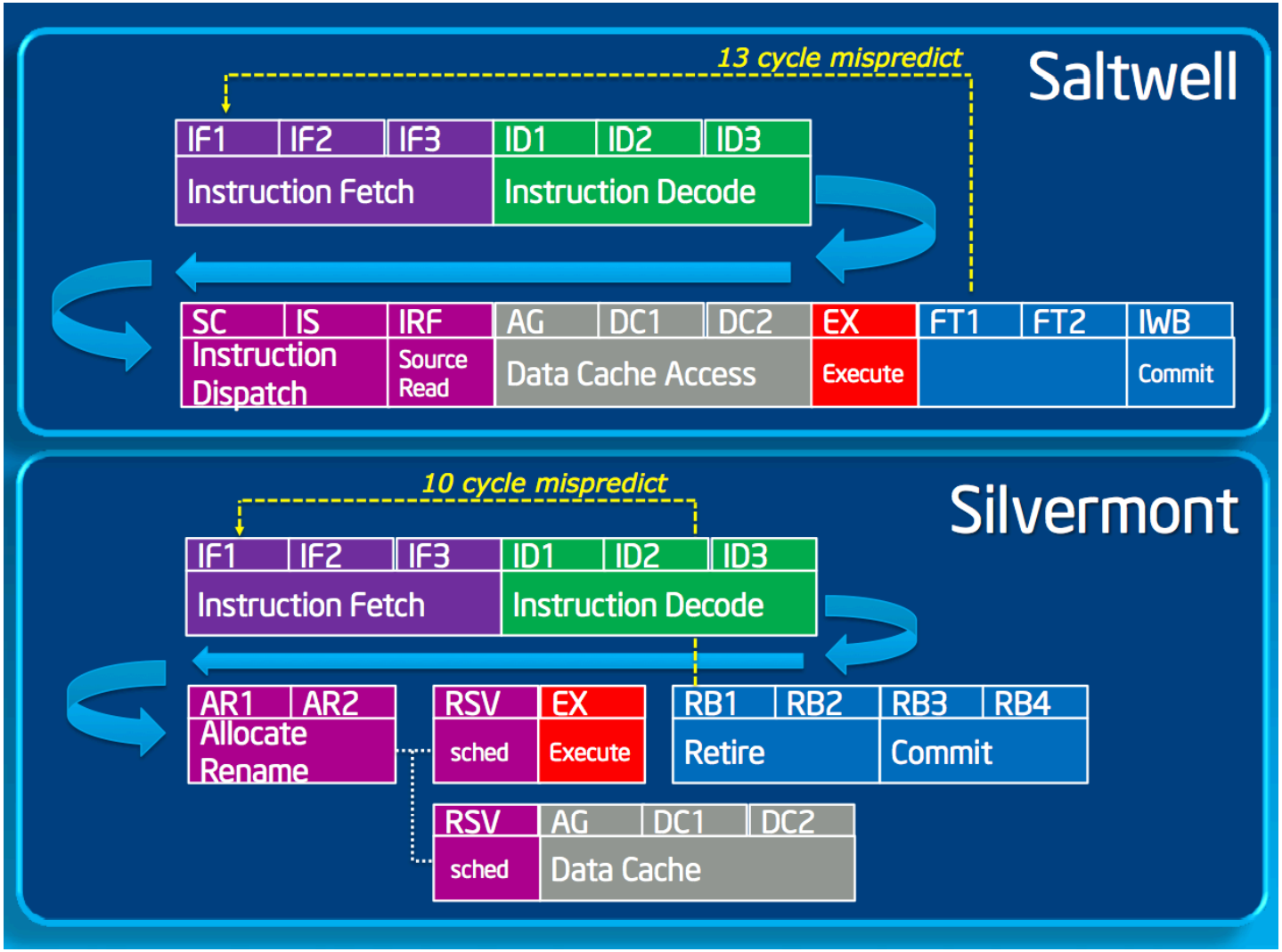
Intel Skylake



Intel Skylake Server: AVX-512



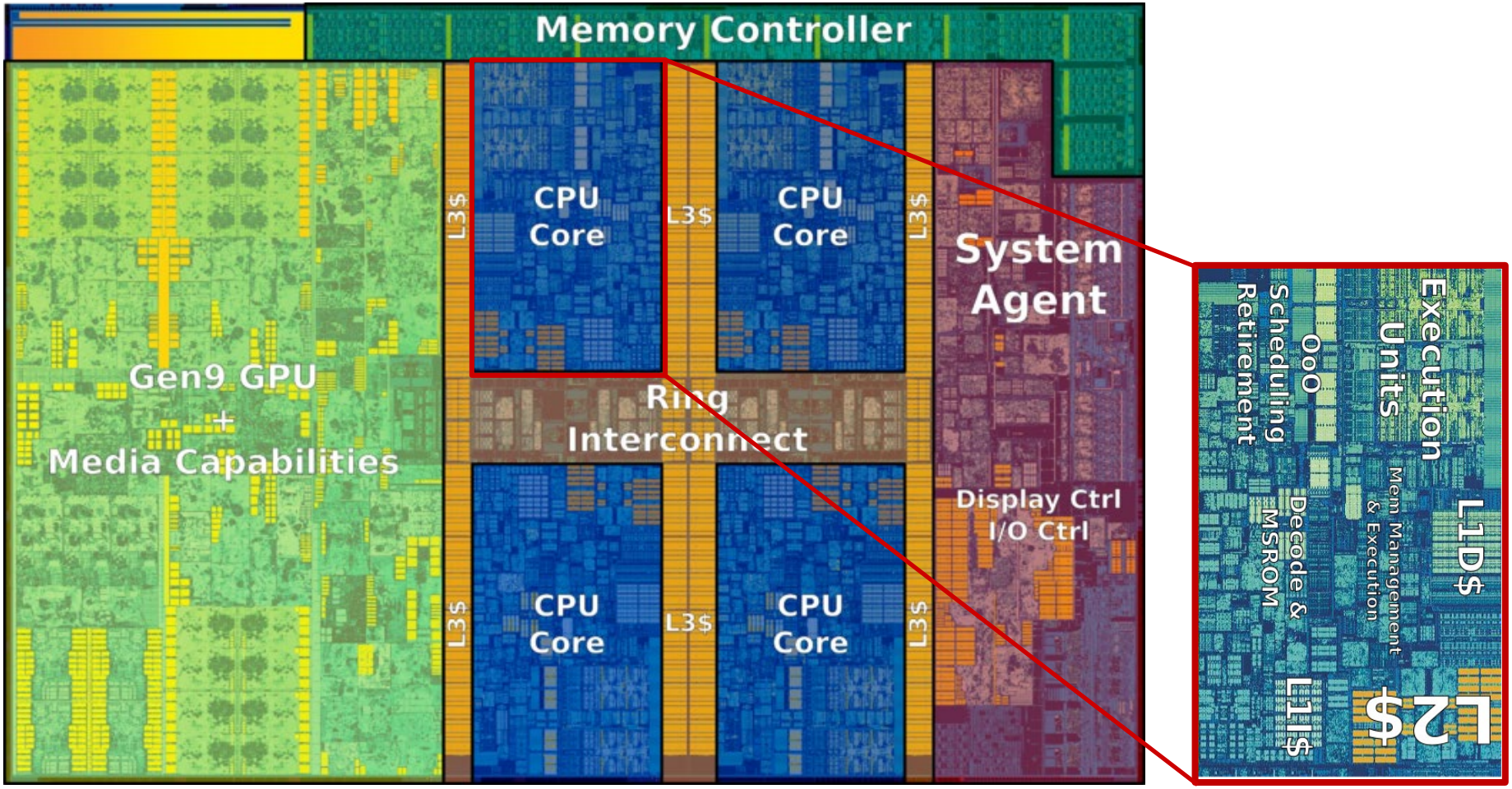
Intel Pipeline Example



Intel Microarchitectural Features

- **Superscalar out of order engine**
  8 functional units (port 0—7)
- **Hyperthreading**
  2 threads per core, 2 register files but only one set of functional units
- **Zero idioms, move elimination**
  recognize and optimize common simple operations
- **Decoded Icache**
  Store decoded micro-ops
- **Store-to-load forwarding**
  Data reused before stored to cache
- **Loop stream detector**
  detect loops post-decode (around 28 uops, 8 branches)
- **Instruction ROM**
  microcode (uop sequences) for less frequent instructions
- **Micro-fusion, macro-fusion and uop unlamination**
  combine or split micro-ops for efficiency and throughput

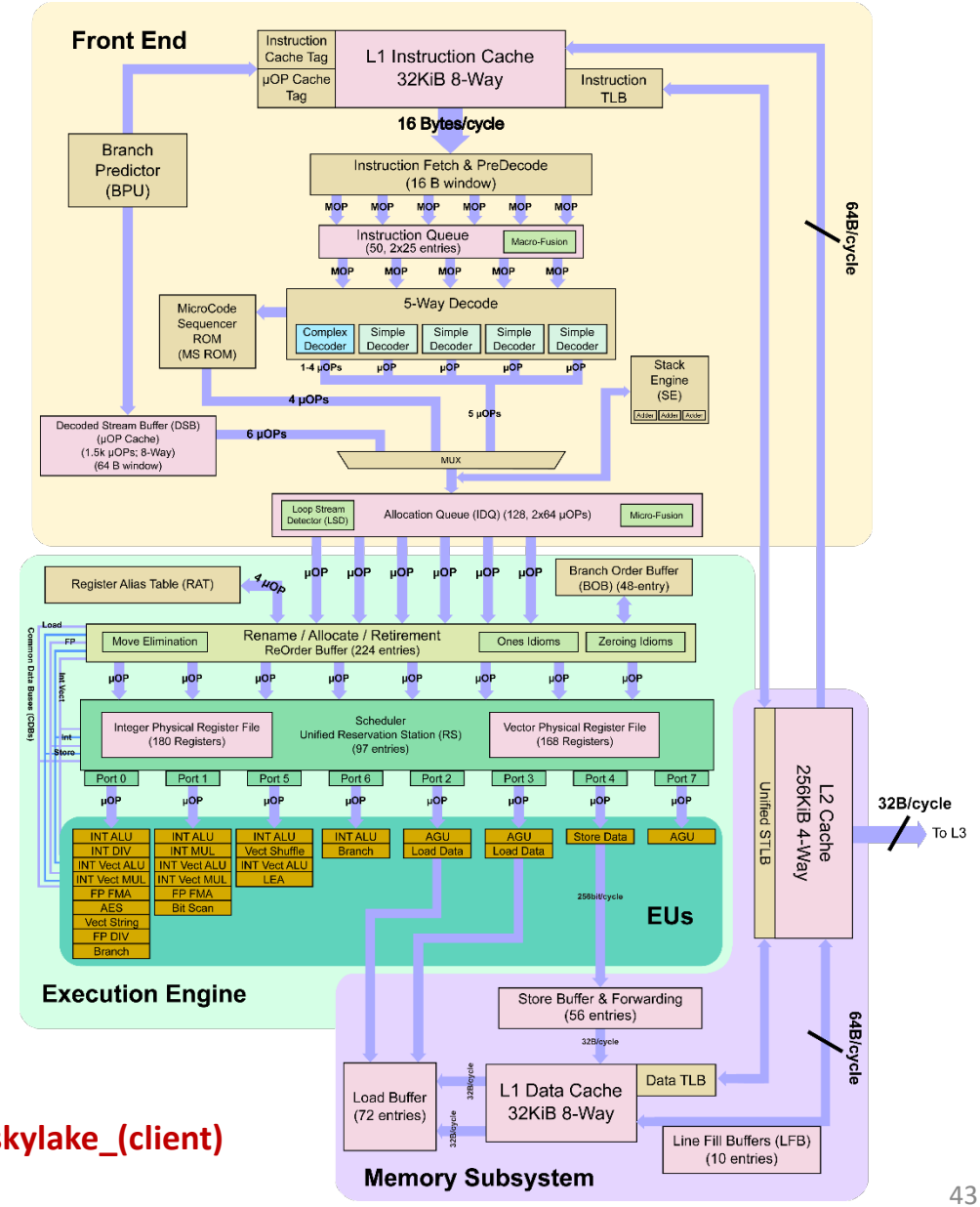
Skylake Desktop CPU Die Shot



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

Skylake By The Numbers

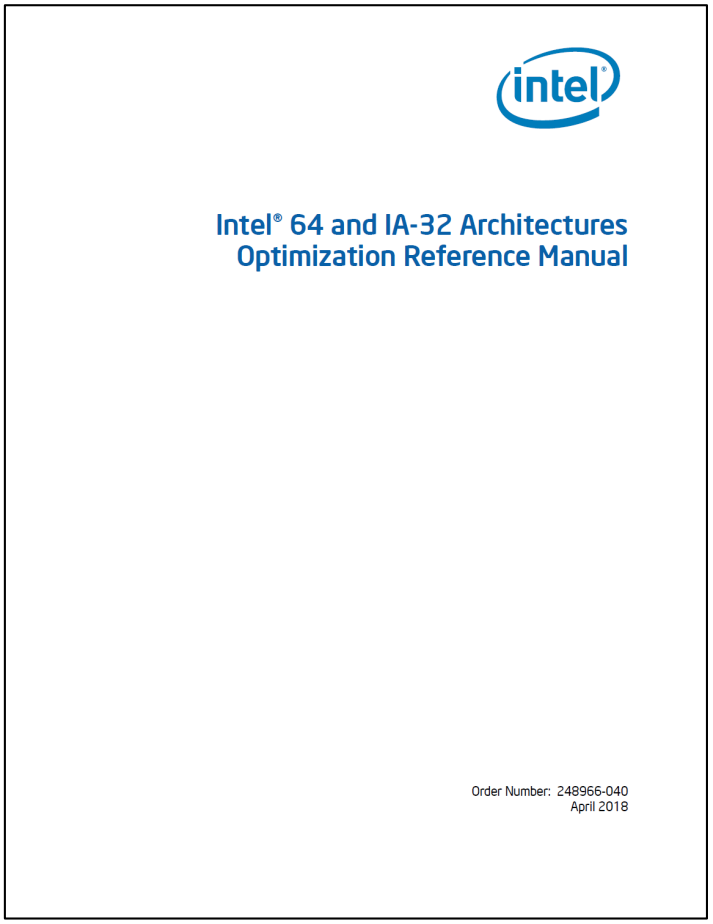
- **Decoder/Frontend**
  5 uop/cycle decoder
  1.5k uop cache (6uop/cycle)
- **OOO execution engine**
  224 entry reorder buffer,
  about 350 uop
  97 entry scheduler
  48 entry branch order buffer
- **Physical register file**
  180 integer register
  168 vector registers
- **Buffers**
  store buffer: 56 entries
  load buffer: 72 entries
  Line fill buffer: 10 entries
- **Power gating**
  10,000 cycles to wake up AVX2



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

Advanced Topics

- Instruction encoding
- RISC vs. CISC
- Pipelining
- Out of order execution
- Intel Microarchitecture
- **Latencies/Throughput, IACA tool**



Instruction Latency/Throughput Table

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E
VADDPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VADDSUBPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VANDNPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VANDPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VBLENDPD/PS ymm1, ymm2, ymm3, imm	1	1	1	1	0.33	0.33	0.33	0.5
VBLENDVPD/PS ymm1, ymm2, ymm3, ymm	1	2	2	1	1	2	2	1
VCMPPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1

DisplayFamily_DisplayModel	Recent Intel Microarchitectures
06_4EH, 06_5EH	Skylake microarchitecture
06_3DH, 06_47H, 06_56H	Broadwell microarchitecture
06_3CH, 06_45H, 06_46H, 06_3FH	Haswell microarchitecture
06_3AH, 06_3EH	Ivy Bridge microarchitecture
06_2AH, 06_2DH	Sandy Bridge microarchitecture
06_25H, 06_2CH, 06_2FH	Intel microarchitecture Westmere
06_1AH, 06_1EH, 06_1FH, 06_2EH	Intel microarchitecture Nehalem
06_17H, 06_1DH	Enhanced Intel Core microarchitecture
06_0FH	Intel Core microarchitecture

Cache Sizes/Latencies/Throughput

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L1 Data Cache Unit (DCU)	Size [KB]	32	32
	Latency [cycles]	4-6	4-6
	Max bandwidth [bytes/cycles]	96	192
	Sustained bandwidth [bytes/cycles]	93	133
	Associativity [ways]	8	8
L2 Mid-level Cache (MLC)	Size [KB]	256	1024 (1MB)
	Latency [cycles]	12	14
	Max bandwidth [bytes/cycles]	32	64
	Sustained bandwidth [bytes/cycles]	25	52
	Associativity [ways]	8	16
L3 Last-level Cache (LLC)	Size [MB]	Up to 2.5 per core	up to 1.375[1] per core
	Latency [cycles]	50-60	50-70
	Max bandwidth [bytes/cycles]	16	16
	Sustained bandwidth [bytes/cycles]	14	15

Intel Architecture Code Analyzer (IACA)

```
Intel(R) Architecture Code Analyzer Version - 2.2 build:356c3b8 (Tue, 13 Dec 2016 16:25:20 +0200)
Analyzed File -
Binary Format - 64Bit
Architecture  - BDW
Analysis Type - Throughput

Throughput Analysis Report
--------------------------
Block Throughput: 5.00 Cycles       Throughput Bottleneck: Port0, Port1, Port5

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |  6   |  7   |
---------------------------------------------------------------------------------------
| Cycles | 5.0    0.0  | 5.0  | 0.0    0.0  | 0.0    0.0  | 0.0  | 5.0  | 0.0  | 0.0  |
---------------------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256/AVX512 instruction, dozens of cycles penalty is expected
X - instruction not supported, was not accounted in Analysis

| Num Of |                    Ports pressure in cycles                     |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |  6  |  7  |    |
---------------------------------------------------------------------------------
|   1    | 1.0       |     |           |           |     |     |     |     | CP | vfmadd231ps ymm0, ymm0, ymm0
|   1    |           | 1.0 |           |           |     |     |     |     | CP | vfmadd231ps ymm1, ymm1, ymm1
|   1    | 1.0       |     |           |           |     |     |     |     | CP | vfmadd231ps ymm2, ymm2, ymm2
|   1    |           | 1.0 |           |           |     |     |     |     | CP | vfmadd231ps ymm3, ymm3, ymm3
|   1    | 1.0       |     |           |           |     |     |     |     | CP | vfmadd231ps ymm4, ymm4, ymm4
|   1    |           | 1.0 |           |           |     |     |     |     | CP | vfmadd231ps ymm5, ymm5, ymm5
|   1    | 1.0       |     |           |           |     |     |     |     | CP | vfmadd231ps ymm6, ymm6, ymm6
|   1    |           | 1.0 |           |           |     |     |     |     | CP | vfmadd231ps ymm7, ymm7, ymm7
|   1    | 1.0       |     |           |           |     |     |     |     | CP | vfmadd231ps ymm8, ymm8, ymm8
|   1    |           | 1.0 |           |           |     |     |     |     | CP | vfmadd231ps ymm9, ymm9, ymm9
|   1    |           |     |           |           |     | 1.0 |     |     | CP | vpaddd ymm10, ymm10, ymm10
|   1    |           |     |           |           |     | 1.0 |     |     | CP | vpaddd ymm11, ymm11, ymm11
|   1    |           |     |           |           |     | 1.0 |     |     | CP | vpaddd ymm12, ymm12, ymm12
|   1    |           |     |           |           |     | 1.0 |     |     | CP | vpaddd ymm13, ymm13, ymm13
|   1    |           |     |           |           |     | 1.0 |     |     | CP | vpaddd ymm14, ymm14, ymm14
Total Num Of Uops: 15
```

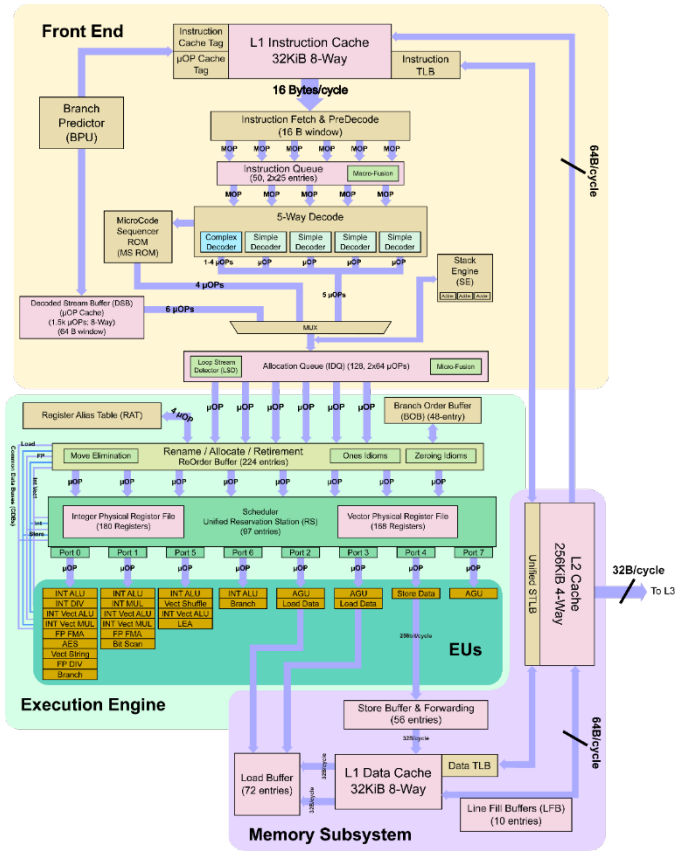
Summary

What you think is happening

```
0000000000400595 <sumstore>:
  400595:   53                   push   %rbx
  400596:   48 89 d3             mov    %rdx,%rbx
  400599:   e8 f2 ff ff ff       callq  400590 <plus>
  40059e:   48 89 03             mov    %rax,(%rbx)
  4005a1:   5b                   pop    %rbx
  4005a2:   c3                   retq
```

Instructions are executed one by one

What really happens



Only way to know: timing of code