

6.172  
Performance  
Engineering  
of Software  
Systems



LECTURE 4  
Assembly Language and  
Computer Architecture

Charles E. Leiserson

# Source Code to Execution

Source code fib.c

```
int64_t fib(int64_t n) {  
    if (n < 2) return n;  
    return (fib(n-1) + fib(n-2));  
}
```

Machine code fib

```
01010101 01001000 10001001  
11100101 01010011 01001000  
10000011 11101100 00001000  
10001001 01111101 11110100  
10000011 01111101 11110100  
00000001 01111111 00001000  
10001011 01000101 11110100  
10001001 01000101 11110000  
11101011 00011101 10001011  
01000101 11110100 10001101  
01111000 11111111 11101000  
11011011 11111111 11111111  
11111111 10001001 11000011  
10001011 01000101 11110100  
10001101 01111000 11111110  
11101000 11001110 11111111  
11111111 11111111 00000001  
11000011 10001001 01011101  
11110000 10001011 01000101  
11110000 01001000 10000011  
11000100 00001000 01011011  
11001001 11000011
```

\$ clang fib.c -o fib

Compilation

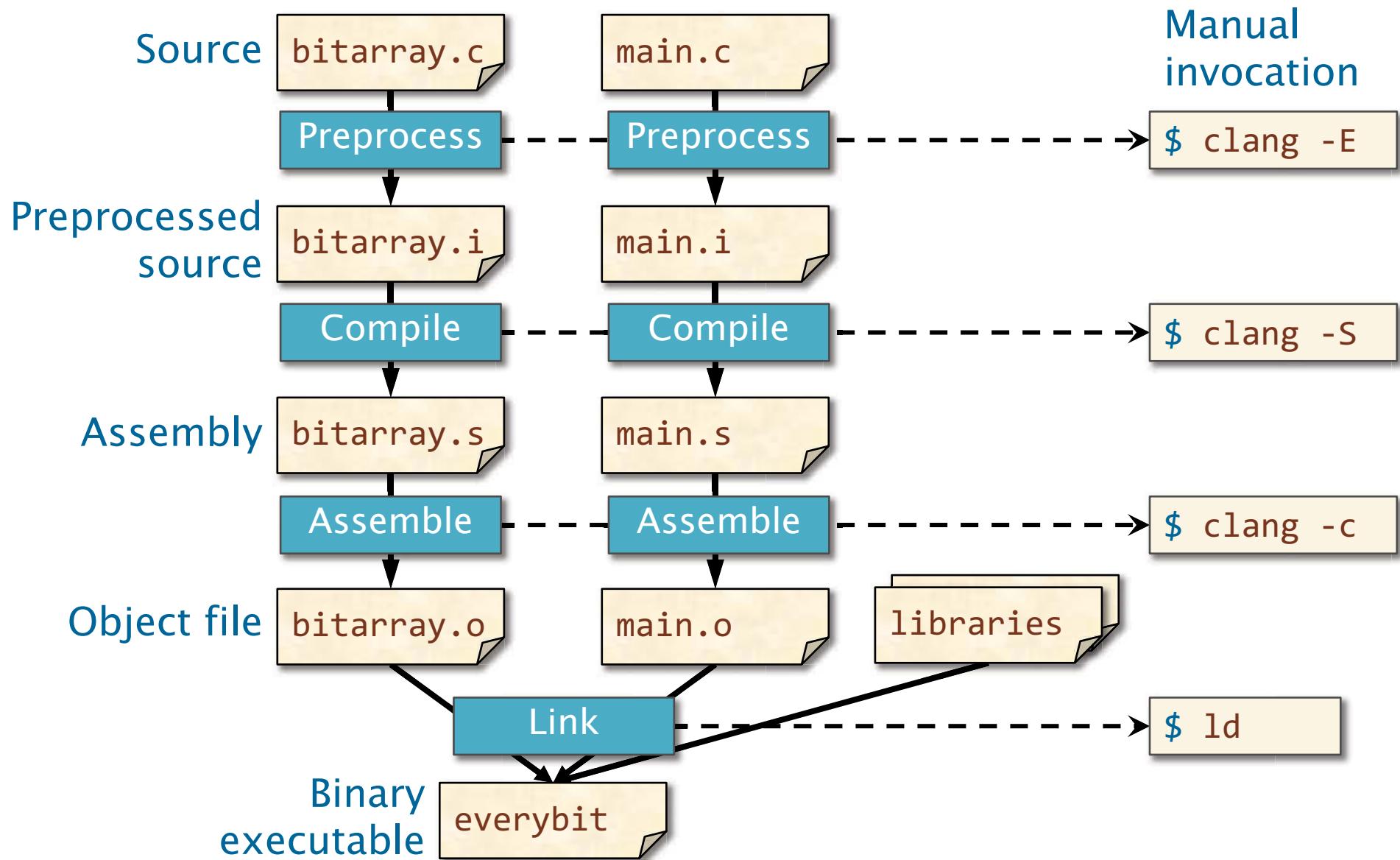
four stages { Preprocessing  
Compiling  
Assembling  
Linking

Hardware interpretation

\$ ./fib

Execution

# The Four Stages of Compilation



# Source Code to Assembly Code

## Source code fib.c

```
int64_t fib(int64_t n) {  
    if (n < 2) return n;  
    return (fib(n-1) + fib(n-2));  
}
```

```
$ clang -O3 fib.c -S
```

## Assembly code fib.s

```
.globl _fib  
.p2align 4, 0x90  
## @fib  
pushq %rbp  
movq %rsp, %rbp  
pushq %r14  
pushq %rbx  
movq %rdi, %rbx  
cmpq $2, %rbx  
jge LBB0_1  
movq %rbx, %rax  
jmp LBB0_3  
  
LBB0_1:  
    leaq -1(%rbx), %rdi  
    callq _fib  
    movq %rax, %r14  
    addq $-2, %rbx  
    movq %rbx, %rdi  
    callq _fib  
    addq %r14, %rax  
  
LBB0_3:  
    popq %rbx  
    popq %r14  
    popq %rbp  
    retq
```

Assembly language provides a convenient symbolic representation of machine code.

See <http://sourceware.org/binutils/docs/as/index.html>.

# Assembly Code to Executable

## Assembly code fib.s

```
_fib:    .p2align 4, 0x90
         ## @fib
         pushq   %rbp
         movq   %rsp, %rbp
         pushq   %r14
         pushq   %rbx
         movq   %rdi, %rbx
         cmpq   $2, %rbx
         jge    LBB0_1
         movq   %rbx, %rax
         jmp    LBB0_3

LBB0_1:
         leaq    -1(%rbx), %rdi
         callq   _fib
         movq   %rax, %r14
         addq   $-2, %rbx
         movq   %rbx, %rdi
         callq   _fib
         addq   %r14, %rax

LBB0_3:
         popq   %rbx
         popq   %r14
         popq   %rbp
         retq
```

## Assembling

```
$ clang fib.s -o fib.o
```

## Machine code

01010101	01001000
10001001	11100101
01010011	01001000
10000011	11101100
00001000	10001001
01111101	11110100
10000011	01111101
11110100	00000001
01111111	00001000
10001011	01000101
11110100	10001001
01000101	11110000
11101011	00011101
10001011	01000101
11110100	10001101
01111000	11111111
11101000	11011011
11111111	11111111
11111111	10001001
11000011	10001011
01000101	11110100
...	

You can edit fib.s and assemble with clang.

# Disassembling

## Source, machine, & assembly

Binary executable  
fib with debug  
symbols (i.e.,  
compiled with -g):

```
$ objdump -S fib
```

```
Disassembly of section __TEXT,__text:  
_fib:  
; int64_t fib(int64_t n) {  
    0: 55                      pushq %rbp  
    1: 48 89 e5                movq %rsp, %rbp  
    4: 41 56                  pushq %r14  
    6: 53                      pushq %rbx  
    7: 48 89 fb                movq %rdi, %rbx  
; if (n < 2) return n;  
    a: 48 83 fb 02            cmpq $2, %rbx  
    e: 7d 05                  jge 5 <_fib+0x15>  
;  
    10: 48 89 d8              movq %rbx, %rax  
    13: eb 1b                  jmp 27 <_fib+0x30>  
; return (fib(n-1) + fib(n-2));  
    15: 48 8d 7b ff            leaq -1(%rbx), %rdi  
    19: e8 e2 ff ff ff        callq -30 <_fib>  
    1e: 49 89 c6              movq %rax, %r14  
    21: 48 83 c3 fe            addq $-2, %rbx  
    25: 48 89 df              movq %rbx, %rdi  
    28: e8 d3 ff ff ff        callq -45 <_fib>  
    2d: 4c 01 f0              addq %r14, %rax  
;  
    30: 5b                      popq %rbx  
    31: 41 5e                  popq %r14  
    33: 5d                      popq %rbp  
    34: c3                      retq
```

# Why Assembly?

## Why bother looking at the assembly of your program?

- The assembly reveals what the compiler did and did not do.
- Bugs can arise at a low level. For example, a bug in the code might only have an effect when compiling at `-O3`. Furthermore, sometimes the compiler is the source of the bug!
- You can modify the assembly by hand, when all else fails.
- **Reverse engineering:** You can decipher what a program does when you only have access to its binary.

# Expectations of Students

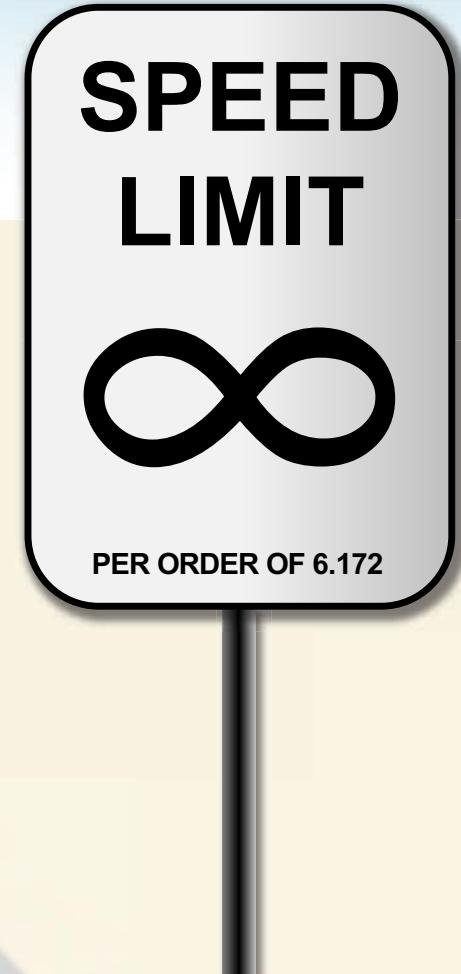
Assembly is **complicated**, and you needn't memorize the manual. Here's what we expect of you:

- Understand how a compiler **implements** C linguistic constructs using x86 instructions. (Lecture 5.)
- Demonstrate a proficiency in **reading** x86 assembly language (with the aid of an architecture manual).
- Understand the **high-level performance implications** of common assembly patterns.
- Be able to make **simple modifications** to the x86 assembly language generated by a compiler.
- Use **compiler intrinsic functions** to use assembly instructions not directly available in C.
- Know how to go about **writing** your own assembly code from scratch if the situation demands it.

# Outline

- x86-64 ISA PRIMER
- FLOATING-POINT AND VECTOR HARDWARE
- OVERVIEW OF COMPUTER ARCHITECTURE

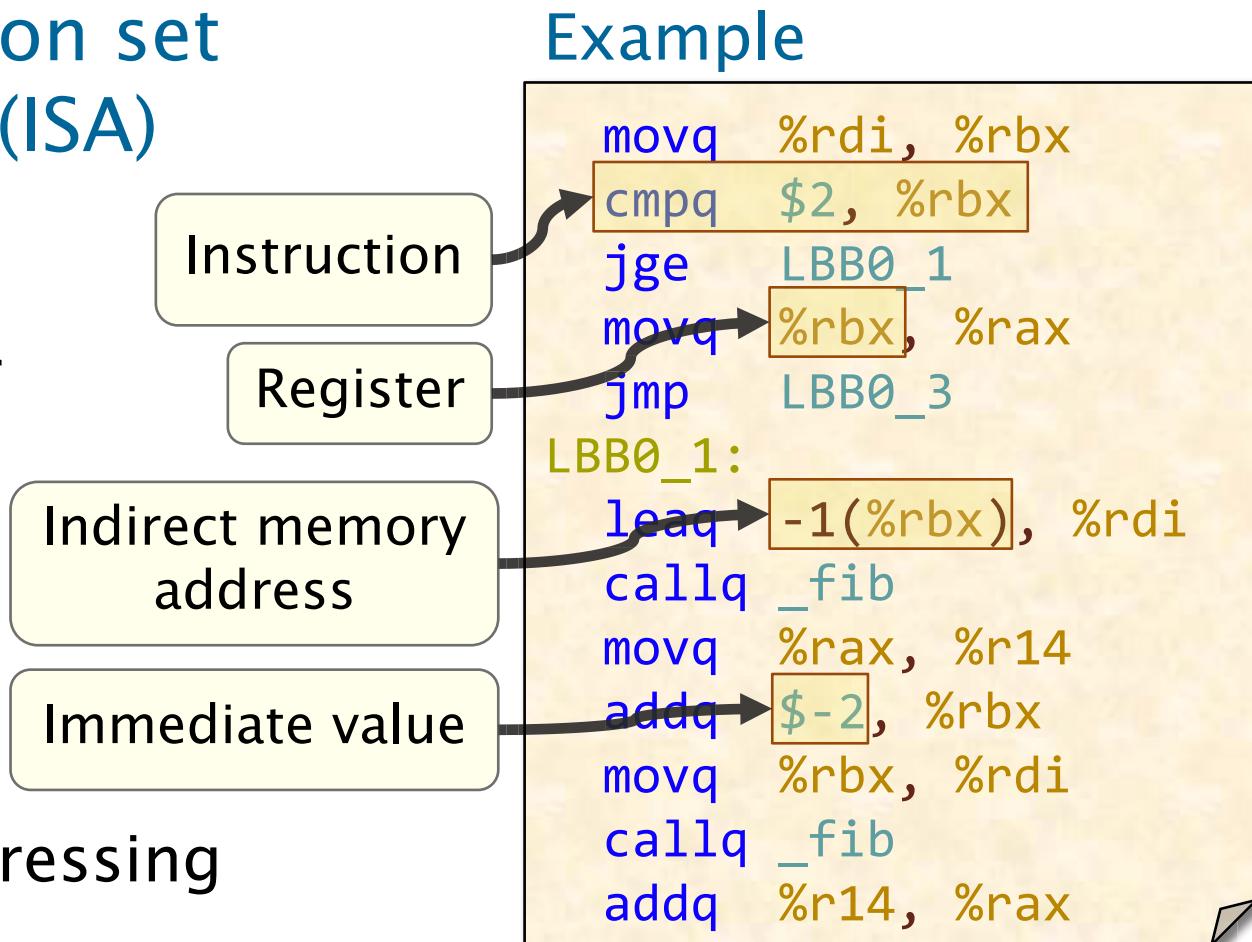
# x86-64 ISA PRIMER



# The Instruction Set Architecture

The instruction set architecture (ISA) specifies the syntax and semantics of assembly.

- Registers
- Instructions
- Data types
- Memory addressing modes



# x86-64 Registers

Number	Width (bits)	Name(s)	Purpose
16	64	(many)	General-purpose registers
6	16	%ss,%[c-g]s	Segment registers
1	64	RFLAGS	Flags register
1	64	%rip	Instruction pointer register
7	64	%cr[0-4,8], %xcr0	Control registers
8	64	%mm[0-7]	MMX registers
1	32	mxcsr	SSE2 control register
16	128	%xmm[0-15]	XMM registers (for SSE)
	256	%ymm[0-15]	YMM registers (for AVX)
8	80	%st([0-7])	x87 FPU data registers
1	16	x87 CW	x87 FPU control register
1	16	x87 SW	x87 FPU status register
1	48		x87 FPU instruction pointer register
1	48		x87 FPU data operand pointer register
1	16		x87 FPU tag register
1	11		x87 FPU opcode register

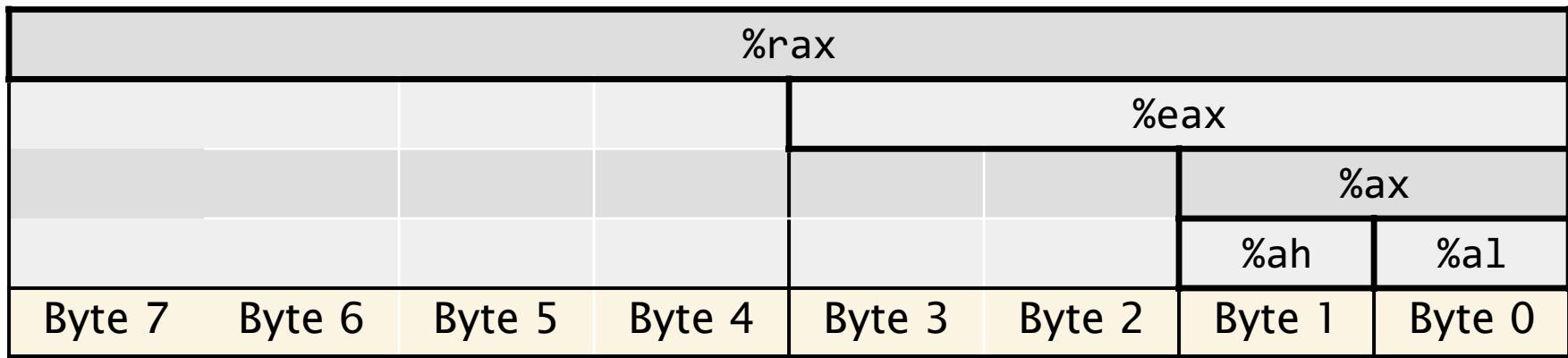
# Common x86-64 Registers

Number	Width (bits)	Name(s)	Purpose
16	64	(many)	General-purpose registers
6	16	%ss,%[c-g]s	Segment registers
1	64	RFLAGS	Flags register
1	64	%rip	Instruction pointer register
7	64	%cr[0-4,8], %xcr0	Control registers
8	64	%mm[0-7]	MMX registers
1	32	mxcsr	SSE2 control register
16	128	%xmm[0-15]	XMM registers (for SSE)
	256	%ymm[0-15]	YMM registers (for AVX)
8	80	%st([0-7])	x87 FPU data registers
1	16	x87 CW	x87 FPU control register
1	16	x87 SW	x87 FPU status register
1	48		x87 FPU instruction pointer register
1	48		x87 FPU data operand pointer register
1	16		x87 FPU tag register
1	11		x87 FPU opcode register

# x86-64 Register Aliasing

The x86-64 general-purpose registers are **aliased**: each has multiple names, which refer to overlapping bytes in the register.

## General-purpose register layout



Only `%rax`, `%rbx`, `%rcx`, and  
`%rdx` have a separate  
register name for this byte.

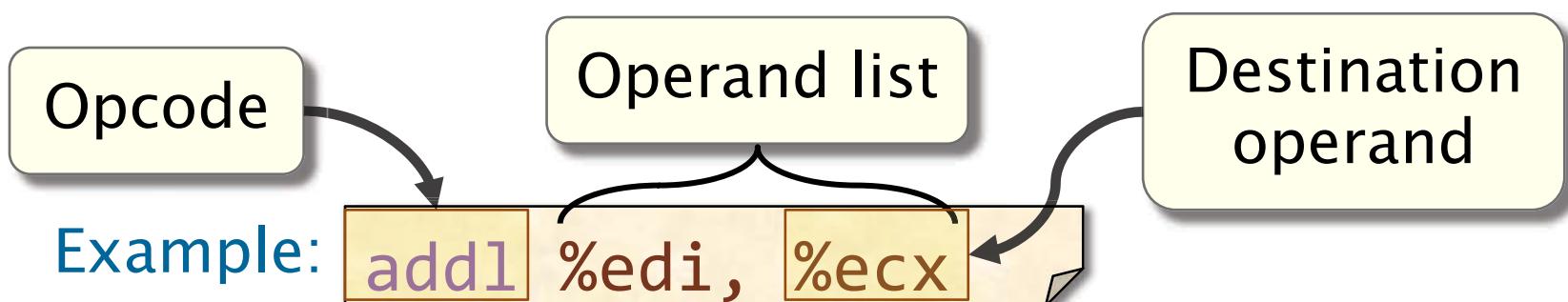
# x86-64 General-Purpose Registers

64-bit name	32-bit name	16-bit name	8-bit name(s)
%rax	%eax	%ax	%ah, %al
	%ecx	%cx	%ch, %cl
	%esi	%si	%sil
	%ebp	%bp	%bpl
	%r8d	%r8w	%r8b
	%r10d	%r10w	%r10b
	%r12d	%r12w	%r12b
	%r14d	%r14w	%r14b

# x86-64 Instruction Format

Format: <opcode> <operand\_list>

- <opcode> is a short mnemonic identifying the type of instruction.
- <operand\_list> is 0, 1, 2, or (rarely) 3 operands, separated by commas.
- Typically, all operands are sources, and one operand might also be the destination.



# AT&T versus Intel Syntax

What does “ $\langle \text{op} \rangle \text{ A, B}$ ” mean?

## AT&T Syntax

$B \leftarrow B \langle \text{op} \rangle A$

## Intel Syntax

$A \leftarrow A \langle \text{op} \rangle B$

`movl $1, %eax`

`mov eax, 1`

`addl (%ebx,%ecx,0x2), %eax`

`add eax, [ebx+ecx*2h]`

`subq 0x20(%rbx), %rax`

`sub rax, [rbx+20h]`

Generated or used by  
`clang`, `objdump`, `perf`,  
6.172 lectures.

Used by Intel  
documentation.



# Common x86-64 Opcodes

Type of operation		Examples
Data movement	Move	mov
	Sign or zero extension	movs, movz
Arithmetic and logic	Integer arithmetic	add, sub, mul, imul, div, idiv, lea, sal, sar, shl, shr, rol, ror, inc, dec, neg
	Boolean logic	test, cmp
	Conditional jumps	j<condition>

**Note:** The subtraction operation  
“`subq %rax, %rbx`” computes  
 $\%rbx = \%rbx - \%rax$ .

# Opcode Suffixes

Opcodes might be augmented with a **suffix** that describes the **data type** of the operation or a **condition code**.

- An opcode for data movement, arithmetic, or logic uses a single-character suffix to indicate the **data type**.
- If the suffix is missing, it can usually be inferred from the sizes of the operand registers.

## Example

`movq -16(%rbp), %rax`

Moving a 64-bit integer.

# x86-64 Data Types

C declaration	C constant	x86-64 size (bytes)	Assembly suffix	x86-64 data type
char	'c'	1	b	Byte
short	172	2	w	Word
int	172	4	l or d	Double word
unsigned int	172U	4	l or d	Double word
long	172L	8	q	Quad word
unsigned long	172UL	8	q	Quad word
char *	"6.172"	8	q	Quad word
float	6.172F	4	s	Single precision
double	6.172	8	d	Double precision
long double	6.172L	16(10)	t	Extended precision

# Opcode Suffixes for Extension

Sign-extension or zero-extension opcodes use two data-type suffixes.

Examples:

Extend with zeros.

`movzb1 %al, %edx`

Move an 8-bit integer  
into a 32-bit integer  
register.

Preserve the sign.

`movslq %eax, %rdx`

Move a 32-bit integer  
into a 64-bit integer  
register.

**Careful!** Results of 32-bit operations are implicitly zero-extended to 64-bit values, unlike the results of 8- and 16-bit operations.

# Conditional Operations

Conditional jumps and conditional moves use a one- or two-character suffix to indicate the condition code.

## Example

```
cmpq $4096, %r14  
jne .LBB1_1
```

The jump should only be taken if the arguments of the previous comparison are **not equal**.

# RFLAGS Register

Bit(s)	Abbreviation	Description
0	CF	Carry
1		<i>Reserved</i>
2	PF	Parity
3		<i>Reserved</i>
4	AF	Adjust
5		<i>Reserved</i>
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12-63		<i>System flags or reserved</i>

Arithmetic and logic operations update **status flags** in the RFLAGS register.

Decrement **%rbx**, and set **ZF** if the result is 0.

Example:

`decq %rbx`

`jne .LBB7_1`

Jump to label **.LBB7\_1** if **ZF** is not set.

# RFLAGS Register

Bit(s)	Abbreviation	Description
0	CF	Carry
1		<i>Reserved</i>
2	PF	Parity
3		<i>Reserved</i>
4	AF	Adjust
5		<i>Reserved</i>
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12–63		<i>System flags or reserved</i>

The last ALU operation generated a carry or borrow out of the most-significant bit.

The result of the last ALU operation was 0.

The last ALU operation produced a value whose sign bit was set.

The last ALU operation resulted in arithmetic overflow.

# Condition Codes

Condition code	Translation	RFLAGS status flags checked
a	if above	CF = 0 and ZF = 0
ae	if above or equal	CF = 0
c	on carry	CF = 1
e	if equal	ZF = 1
ge	if greater or equal	SF = OF
ne	if not equal	ZF = 0
o	on overflow	OF = 1
z	if zero	ZF = 1

**Question:** Why do the condition codes **e** and **ne** check the zero flag?

**Answer:** Hardware typically compares integer operands using subtraction.

# x86-64 Direct Addressing Modes

The operands of an instruction specify values using a variety of **addressing modes**.

- At most one operand may specify a memory address.

## Direct addressing modes

- **Immediate:** Use the specified value.
- **Register:** Use the value in the specified register.
- **Direct memory:** Use the value at the specified memory address.

### Examples

```
movq $172, %rdi
```

```
movq %rcx, %rdi
```

```
movq 0x172, %rdi
```

# x86-64 Indirect Addressing Modes

The x86-64 ISA also supports **indirect addressing**: specifying a memory address by some computation.

- **Register indirect:** The address is stored in the specified register.
- **Register indexed:** The address is a constant offset of the value in the specified register.
- **Instruction-pointer relative:** The address is indexed relative to **%rip**.

## Examples

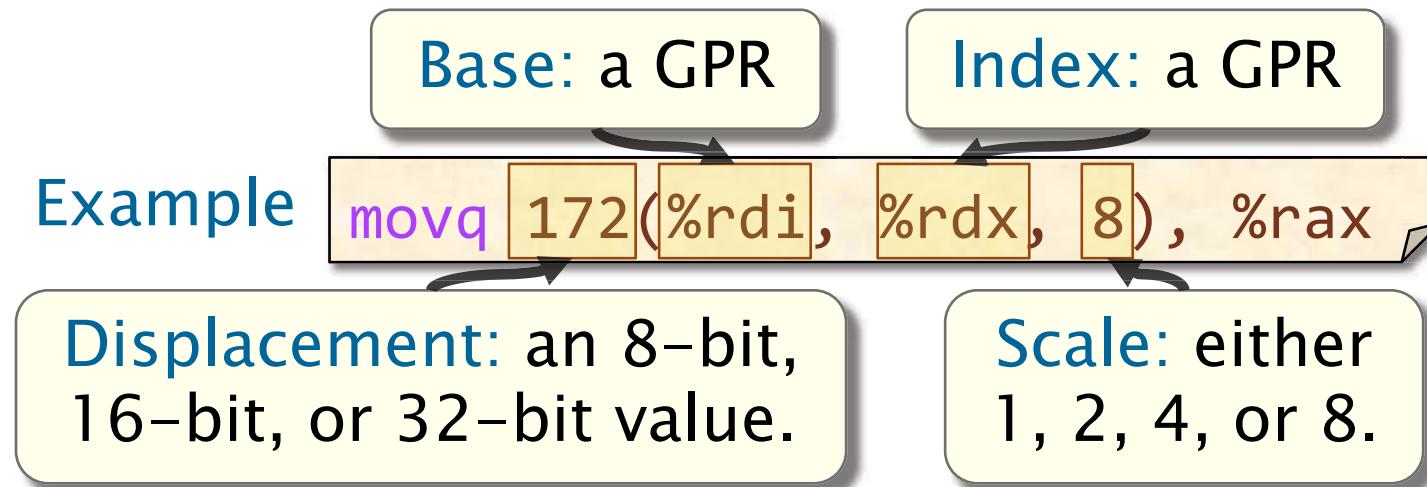
```
movq (%rax), %rdi
```

```
movq 172(%rax), %rdi
```

```
movq 172(%rip), %rdi
```

# Base Indexed Scale Displacement

The most general form of indirect addressing supported by x86-64 is the **base indexed scale displacement** mode.



This mode refers to the address  
**Base + Index<sup>\*</sup>Scale + Displacement.**

If unspecified, **Index** and **Displacement** default to **0**, and **Scale** defaults to **1**.

# Jump Instructions

The x86-64 jump instructions, `jmp` and `j<condition>`, take a **label** as their operand, which identifies a location in the code.

Example from `fib.s`

```
jge LBB0_1  
...  
LBB0_1:  
    leaq -1(%rbx), %rdi
```

- Labels can be **symbols**, **exact addresses**, or **relative addresses**.
- An **indirect jump** takes as its operand an indirect address.

Example from `objdump fib`

```
jge 5 <_fib+0x15>  
...  
15:  
    leaq -1(%rbx), %rdi
```

Example: `jmp *%eax`

# Assembly Idiom 1

The XOR opcode, “`xor A, B`,” computes the bitwise XOR of `A` and `B`.

**Question:** What does the following assembly do?

```
xor %rax, %rax
```

**Answer:** Zeros the register.

# Assembly Idiom 2

The test opcode, “`test A, B`,” computes the bitwise AND of `A` and `B` and discard the result, preserving the RFLAGS register.

## Status flags in RFLAGS

Bit	Abbreviation	Description
0	CF	Carry
2	PF	Parity
4	AF	Adjust
6	ZF	Zero
7	SF	Sign
11	OF	Overflow

**Question:** What does the `test` instruction test for in the following assembly snippets?

```
test %rcx, %rcx  
je 400c0a <mm+0xda>
```

```
test %rax, %rax  
cmovne %rax, %r8
```

**Answer:** Checks to see whether the register is 0.

# Assembly Idiom 3

The x86-64 ISA includes several no-op (no operation) instructions, including “nop,” “nop A,” (no-op with an argument), and “data16.”

**Question:** What does this line of assembly do?

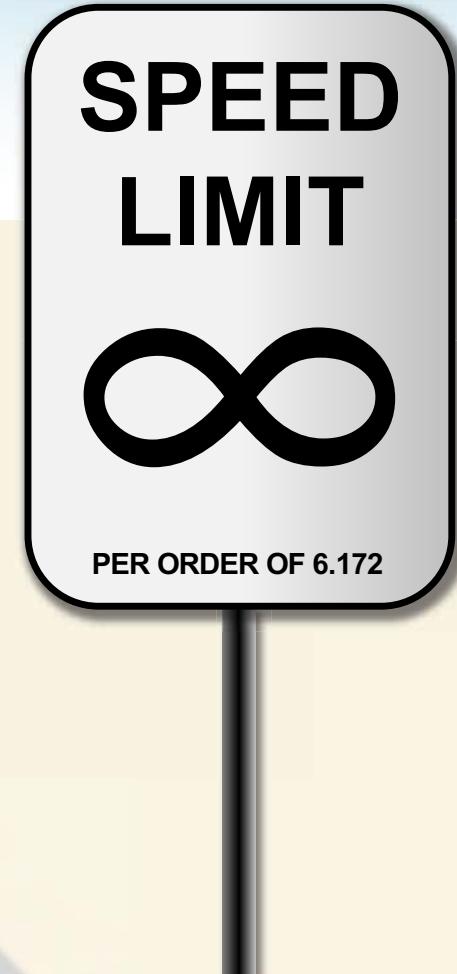
```
data16 data16 data16 nopw %cs:0x0(%rax,%rax,1)
```

**Answer:** Nothing!

**Question:** Why would the compiler generate assembly with these idioms?

**Answer:** Mainly, to optimize instruction memory (e.g., code size, alignment).

# FLOATING-POINT AND VECTOR HARDWARE



# Floating–Point Instruction Sets

Modern x86–64 architectures support **scalar** (i.e., non–vector) floating–point arithmetic via a couple of different instruction sets.

- The **SSE and AVX instructions** support single–precision and double–precision scalar floating–point arithmetic, i.e., “**float**” and “**double**.”
- The **x87 instructions** support single–, double–, and extended–precision scalar floating–point arithmetic, i.e., “**float**,” “**double**,” and “**long double**.”

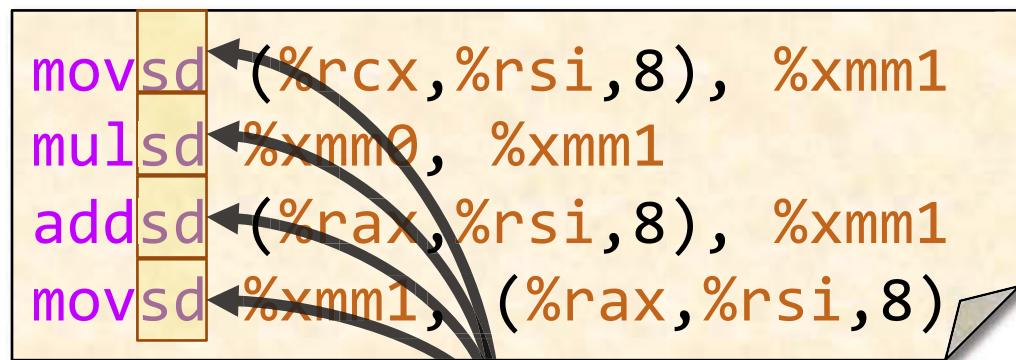
The SSE and AVX instruction sets also include **vector instructions**.

# SSE for Scalar Floating-Point

Compilers prefer to use the SSE instructions over the x87 instructions because SSE instructions are simpler to compile for and to optimize.

- SSE opcodes on floating-point values are similar to x86\_64 opcodes.
- SSE operands use XMM registers and floating-point types.

## Example



Data type is a **double-precision floating-point** value (i.e., a **double**).

# SSE Opcode Suffixes

SSE instructions use two-letter suffixes to encode the data type.

Assembly suffix	Data type
ss	One single-precision floating-point value (float)
sd	One double-precision floating-point value (double)
ps	Vector of single-precision floating-point values
pd	Vector of double-precision floating-point values

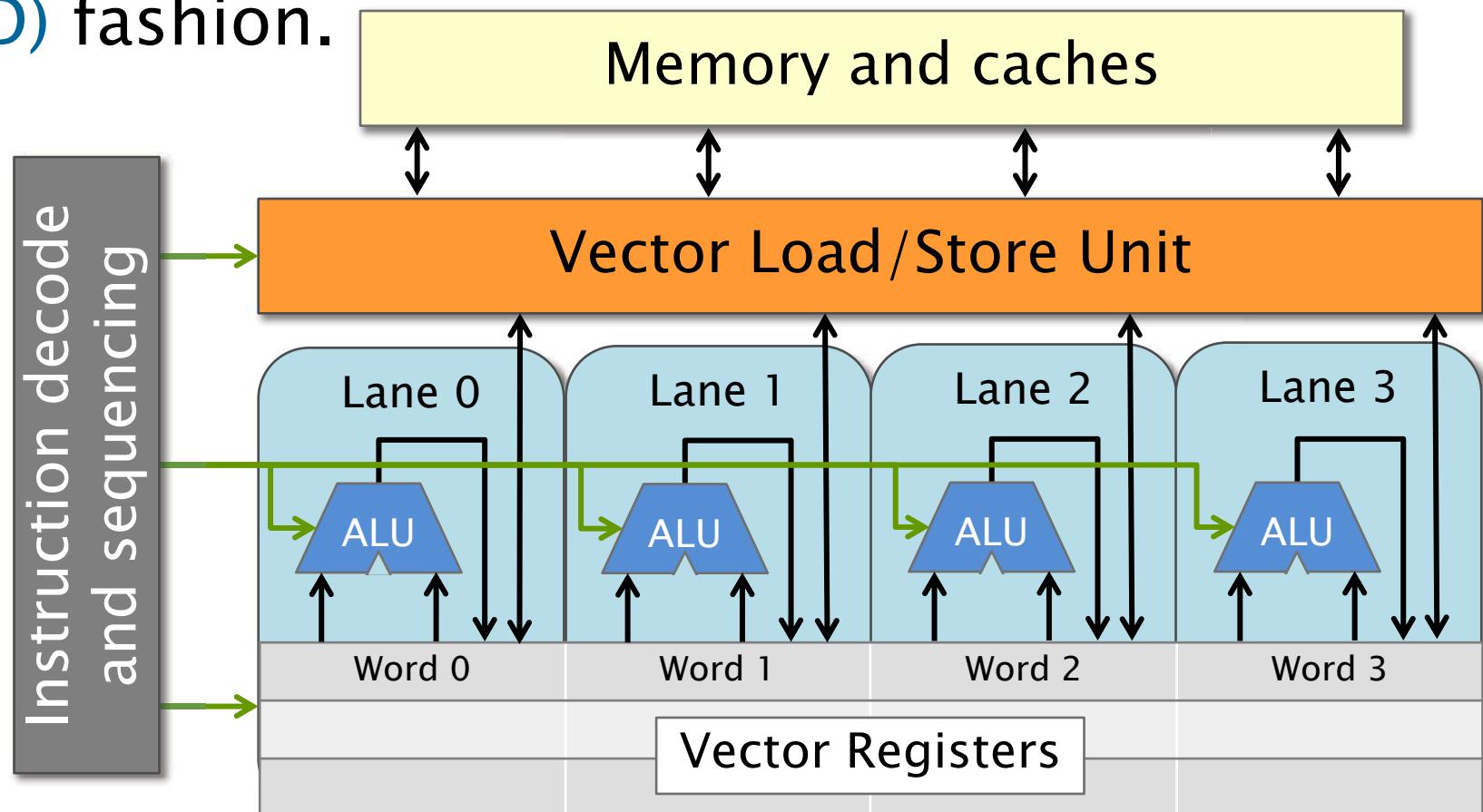


## Mnemonic

- The first letter distinguishes single or packed (i.e., vector).
- The second letter distinguishes single-precision or double-precision.

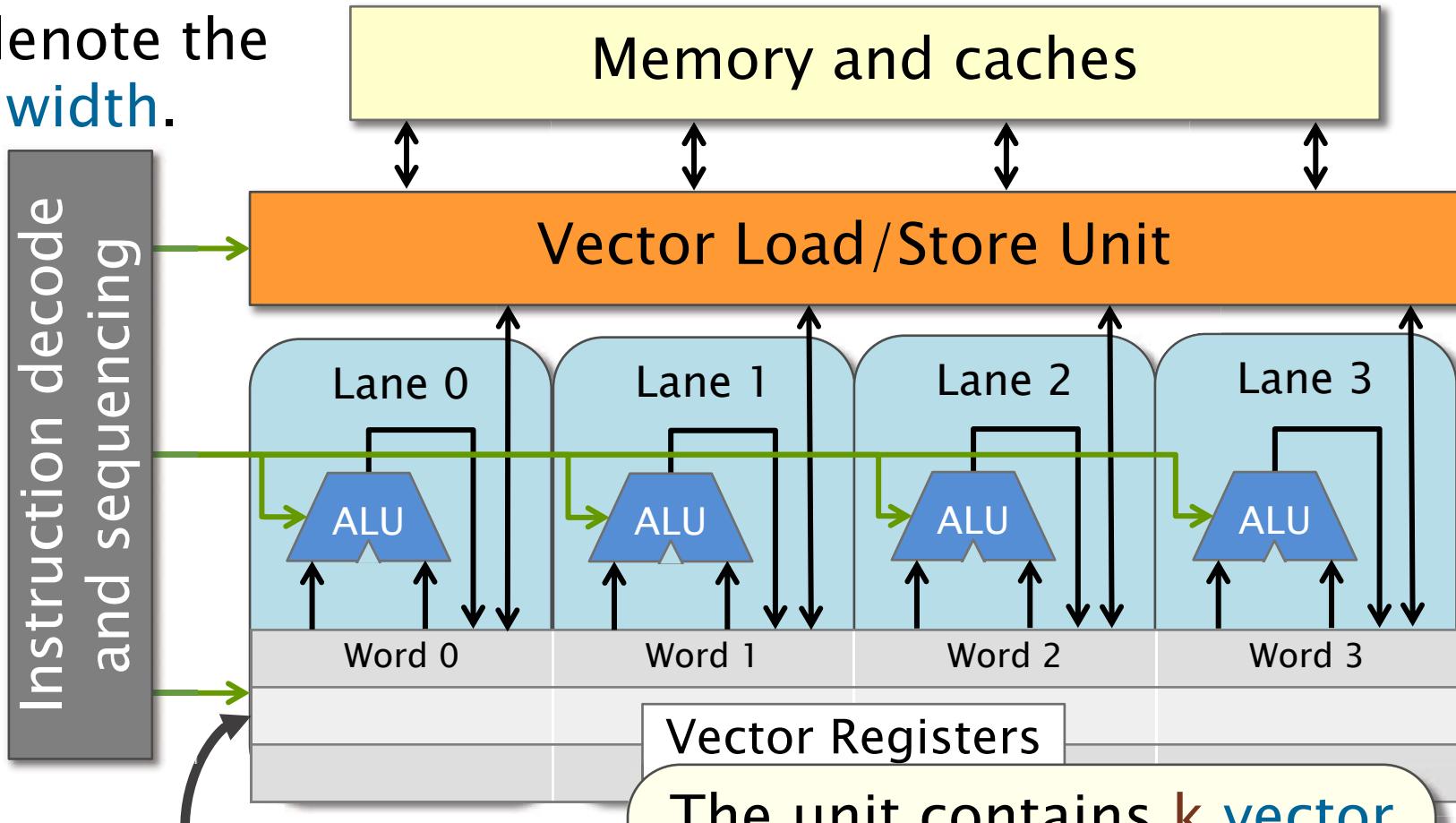
# Vector Hardware

Modern microprocessors often incorporate **vector hardware** to process data in a **single-instruction stream, multiple-data stream (SIMD) fashion**.



# Vector Unit

Let  $k$  denote the vector width.

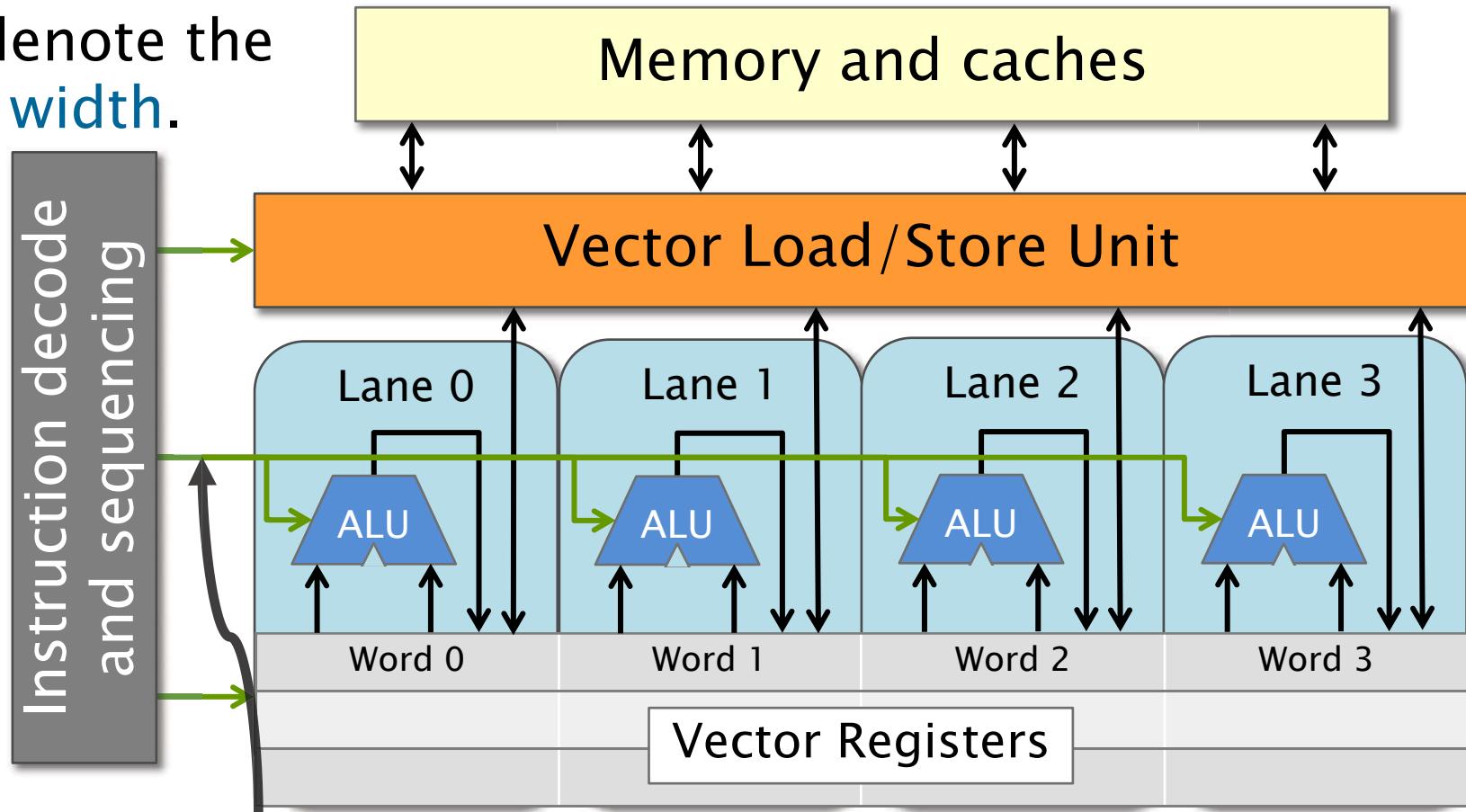


Each **vector register** holds  $k$  scalar integer or floating-point values.

The unit contains  $k$  **vector lanes**, each containing scalar integer or floating-point hardware.

# Vector Unit

Let  $k$  denote the vector width.



All vector lanes operate in **lock-step** and use the **same** instruction and control signals.

# Vector Instructions

Vector instructions generally operate in an elementwise fashion:

- The *i*th element of one vector register can only take part in operations with the *i*th element of other vector registers.
- All lanes perform exactly the same operation on their respective elements of the vector.
- Depending on the architecture, vector memory operands might need to be aligned, meaning their address must be a multiple of the vector width.
- Some architectures support cross-lane operations, such as inserting or extracting subsets of vector elements, permuting (a.k.a., shuffling) the vector, scatter, or gather.

# Vector-Instruction Sets

Modern x86-64 architectures support multiple **vector-instruction sets**.

- Modern SSE instruction sets support vector operations on integer, single-precision, and double-precision floating-point values.
- The AVX instructions support vector operations on single-precision, and double-precision floating-point values.
- The AVX2 instructions add integer-vector operations to the AVX instruction set.
- The AVX-512 (AVX3) instructions increase the register length to 512 bits and provide new vector operations, including popcount. (Not available on Haswell.)

# SSE Versus AVX and AVX2

The AVX and AVX2 instruction sets extend the SSE instruction set in several ways.

- The **SSE instructions** use 128-bit XMM vector registers and operate on at most 2 operands at a time.
- The **AVX instructions** can alternatively use 256-bit YMM vector registers and can operate on 3 operands at a time: two source operands, and one distinct destination operand.

Example AVX instruction

```
vaddpd %ymm0, %ymm1, %ymm2
```

Destination  
operand

# SSE and AVX Vector Opcodes

Many of the SSE and AVX opcodes are similar to traditional x86-64 opcodes, with minor differences.

Example: Opcodes to add 64-bit values

	SSE	AVX/AVX2
Floating-point	addpd	vaddpd
Integer	paddq	vpaddq

The “p” prefix distinguishes an integer vector instruction.

The “v” prefix distinguishes the AVX/AVX2 instruction.

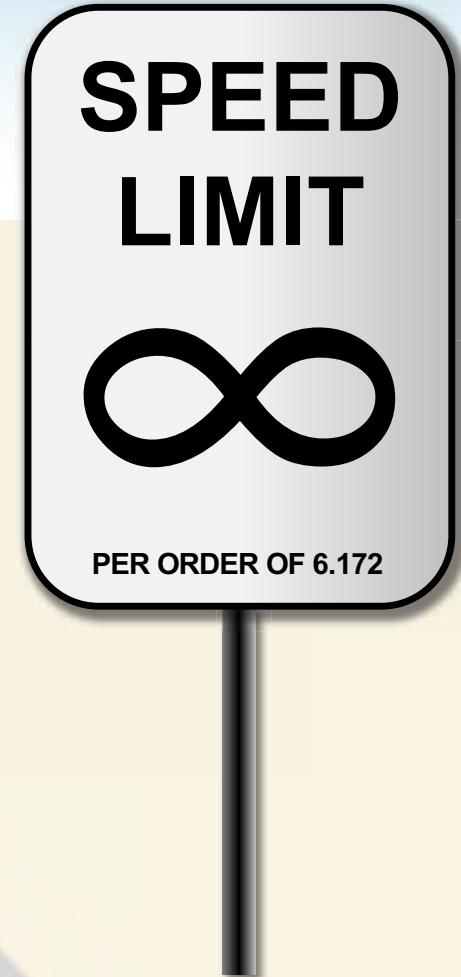
# Vector-Register Aliasing

Like the general-purpose registers, the XMM and YMM vector registers are **aliased**.

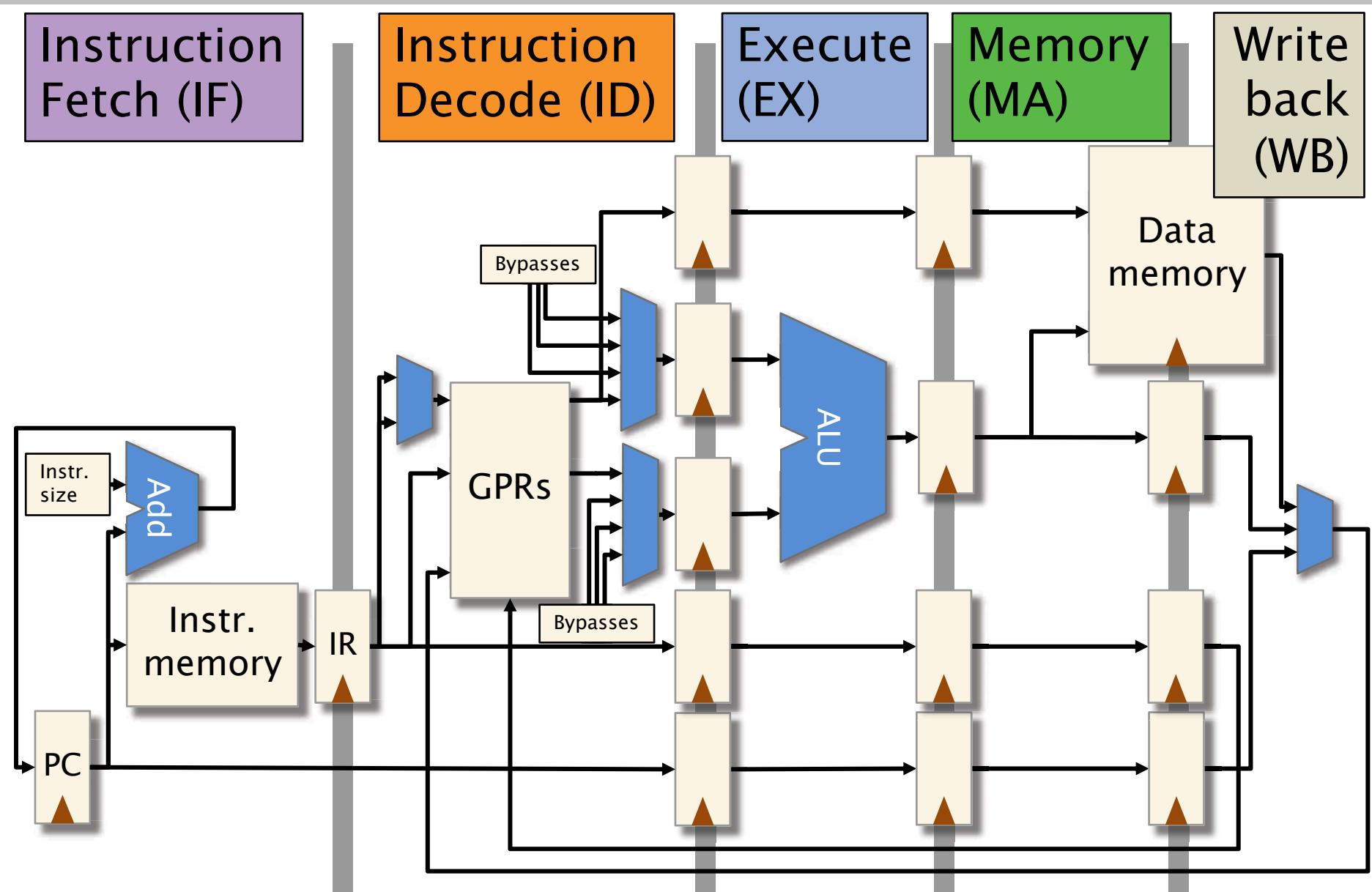
## XMM/YMM vector-register layout

%ymm0		
		%xmm0
High 128 bits		Low 128 bits

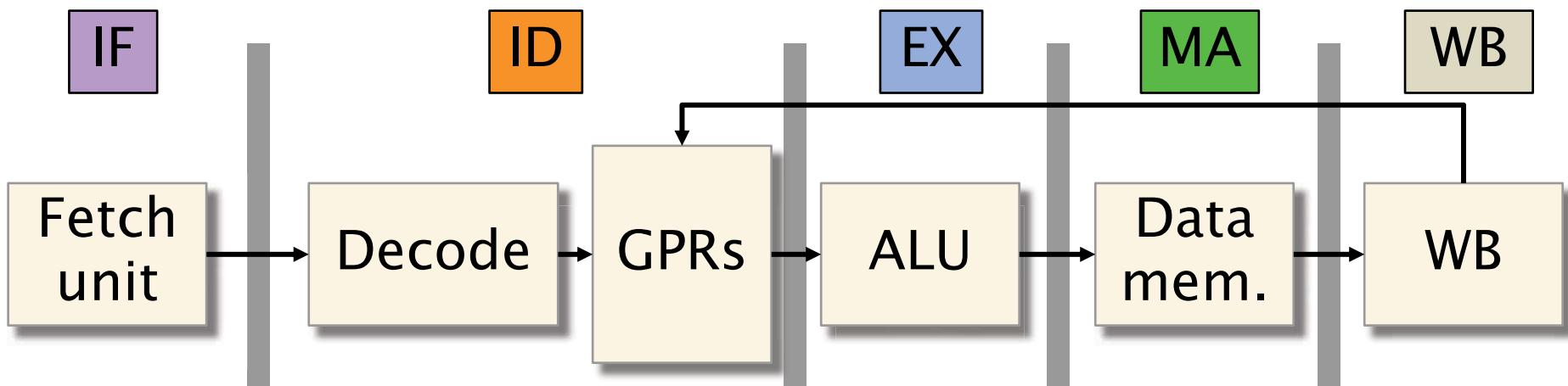
# OVERVIEW OF COMPUTER ARCHITECTURE



# A Simple 5-Stage Processor



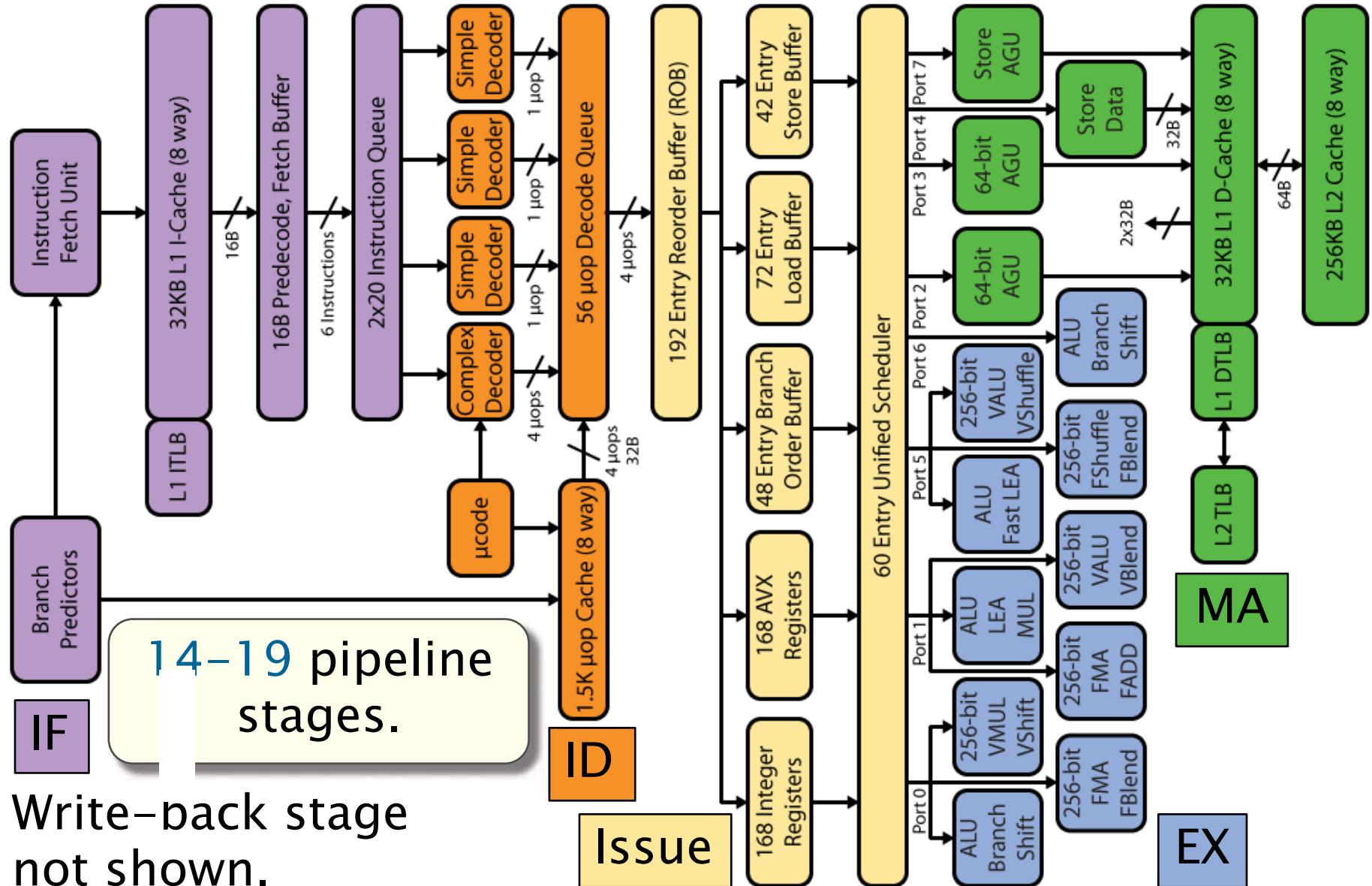
# Block Diagram of 5-Stage Processor



Each instruction is executed through 5 stages:

1. **Instruction fetch (IF):** Read instruction from memory.
2. **Instruction decode (ID):** Determine which units to use to execute the instruction, and extract the register arguments.
3. **Execute (EX):** Perform ALU operations.
4. **Memory (MA):** Read/write data memory.
5. **Write back (WB):** Store result into registers.

# Intel Haswell Microarchitecture



# Bridging the Gap

This lecture bridges the gap between the simple 5-stage processor and a modern processor core by examining several design features:

- Vector hardware
- Superscalar processing
- Out-of-order execution
- Branch prediction

# Architectural Improvements

Historically, computer architects have aimed to improve processor performance by two means:

- Exploit **parallelism** by executing multiple instructions simultaneously.
  - Examples: instruction-level parallelism (ILP), vectorization, multicore.
- Exploit **locality** to minimize data movement.
  - Example: caching.

This lecture: ILP and  
vectorization

# Pipelined Instruction Execution

Processor hardware exploits **instruction-level parallelism** by finding opportunities to execute multiple instructions simultaneously in different pipeline stages.

## Ideal pipelined timing

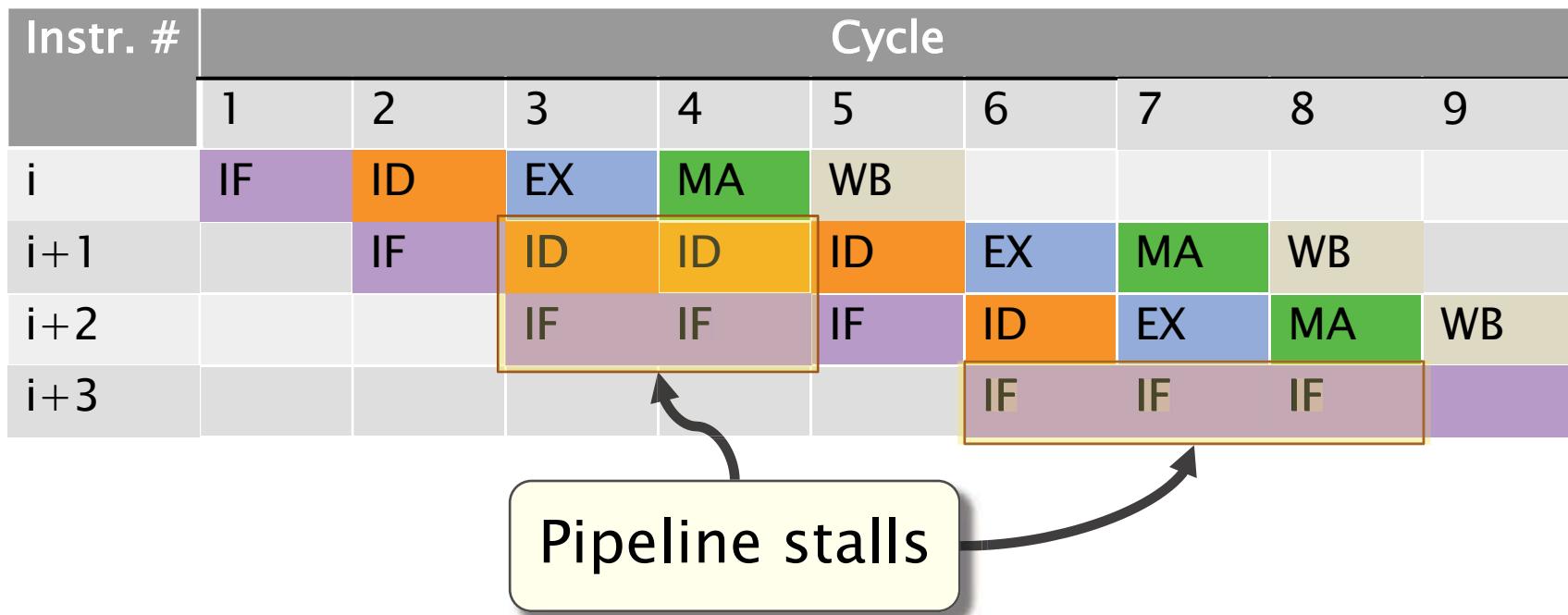
Instr. #	Cycle				
	1	2	3	4	5
i	IF	ID	EX	MA	WB
i+1		IF	ID	EX	MA
i+2			IF	ID	EX
i+3				IF	ID
i+4					IF

Each pipeline stage is executing a different instruction.

Pipelining improves processor throughput.

# Pipelined Execution in Practice

In practice, various issues can prevent an instruction from executing during its designated cycle, causing the processor pipeline to **stall**.



# Sources of Pipeline Stalls

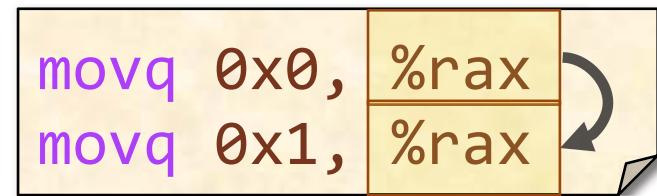
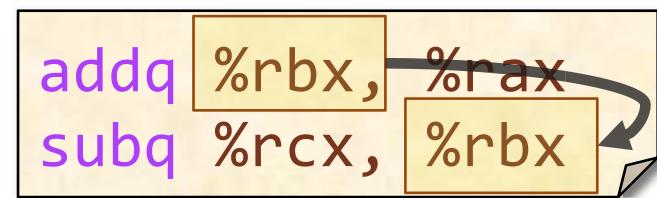
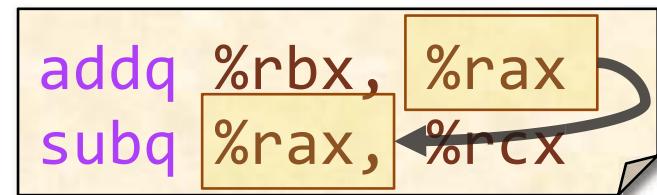
Three types of **hazards** may prevent an instruction from executing during its designated clock cycle.

- **Structural hazard:** Two instructions attempt to use the same functional unit at the same time.
- **Data hazard:** An instruction depends on the result of a prior instruction in the pipeline.
- **Control hazard:** Fetching and decoding the next instruction to execute is delayed by a decision about control flow (i.e., a conditional jump).

# Sources of Data Hazards

An instruction **i** can create a data hazard with a later instruction **j** due to a **dependence** between **i** and **j**.

- **True dependence (RAW):**  
Instruction **i** writes a location that instruction **j** reads.
- **Anti-dependence (WAR):**  
Instruction **i** reads a location that instruction **j** writes.
- **Output-dependence (WAW):** Both instructions **i** and **j** write to the same location.



# Complex Operations

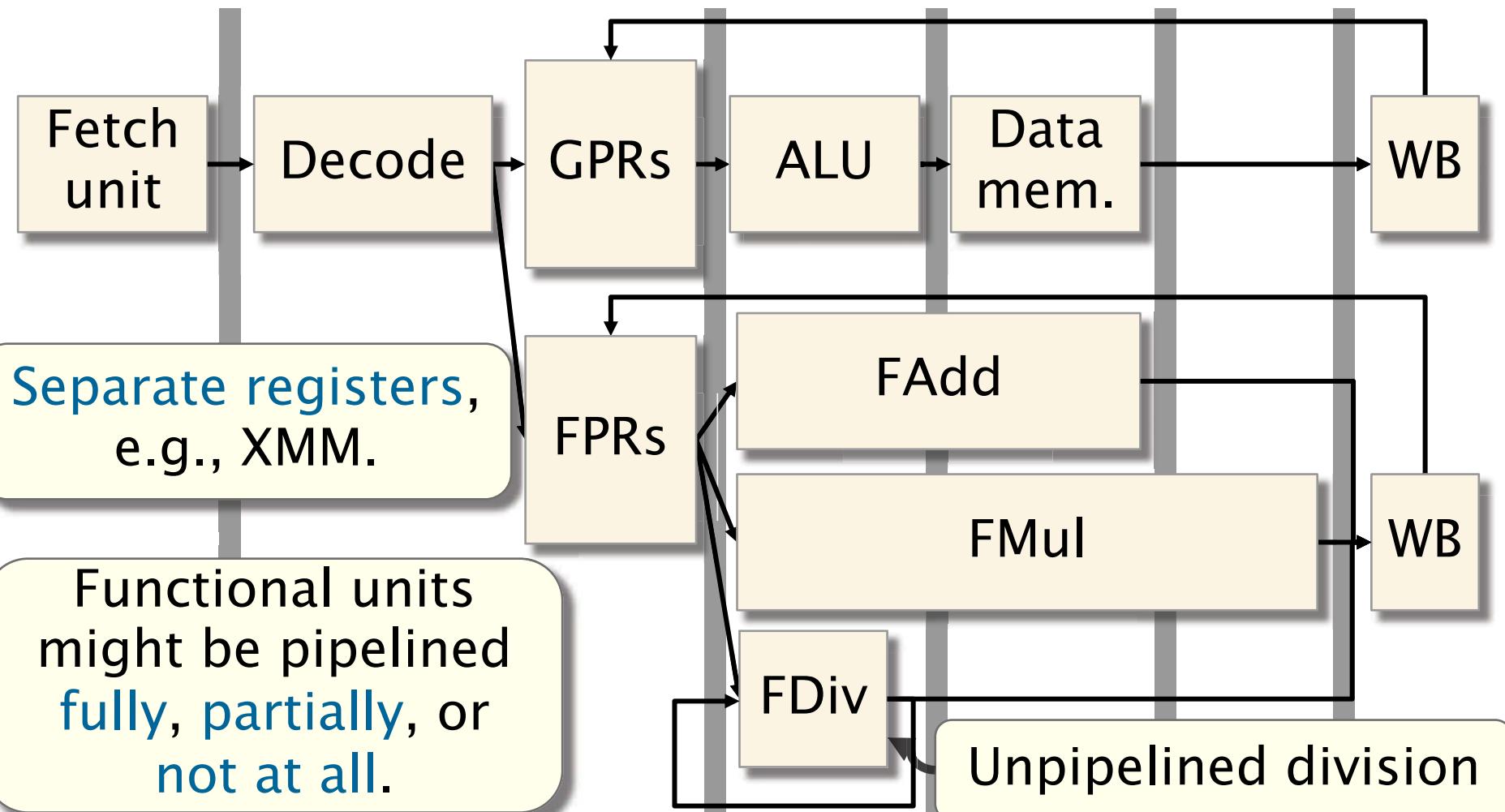
Some arithmetic operations are **complex** to implement in hardware and have **long latencies**.

Operations	Example x86-64 opcodes	Latency (cycles)
Most integer arithmetic, logic, shift	add, sub, and, or, xor, sar, sal, lea...	1
Integer multiply	mul, imul	3
Integer division	div, idiv	variable
Floating-point add	addss, addsd	3
Floating-point multiply	mulss, mulsd	5
Floating-point divide	divss, divsd	variable
Floating-point fma	vfmadd, vfmaddsd	5

How can hardware accommodate these complex operations?

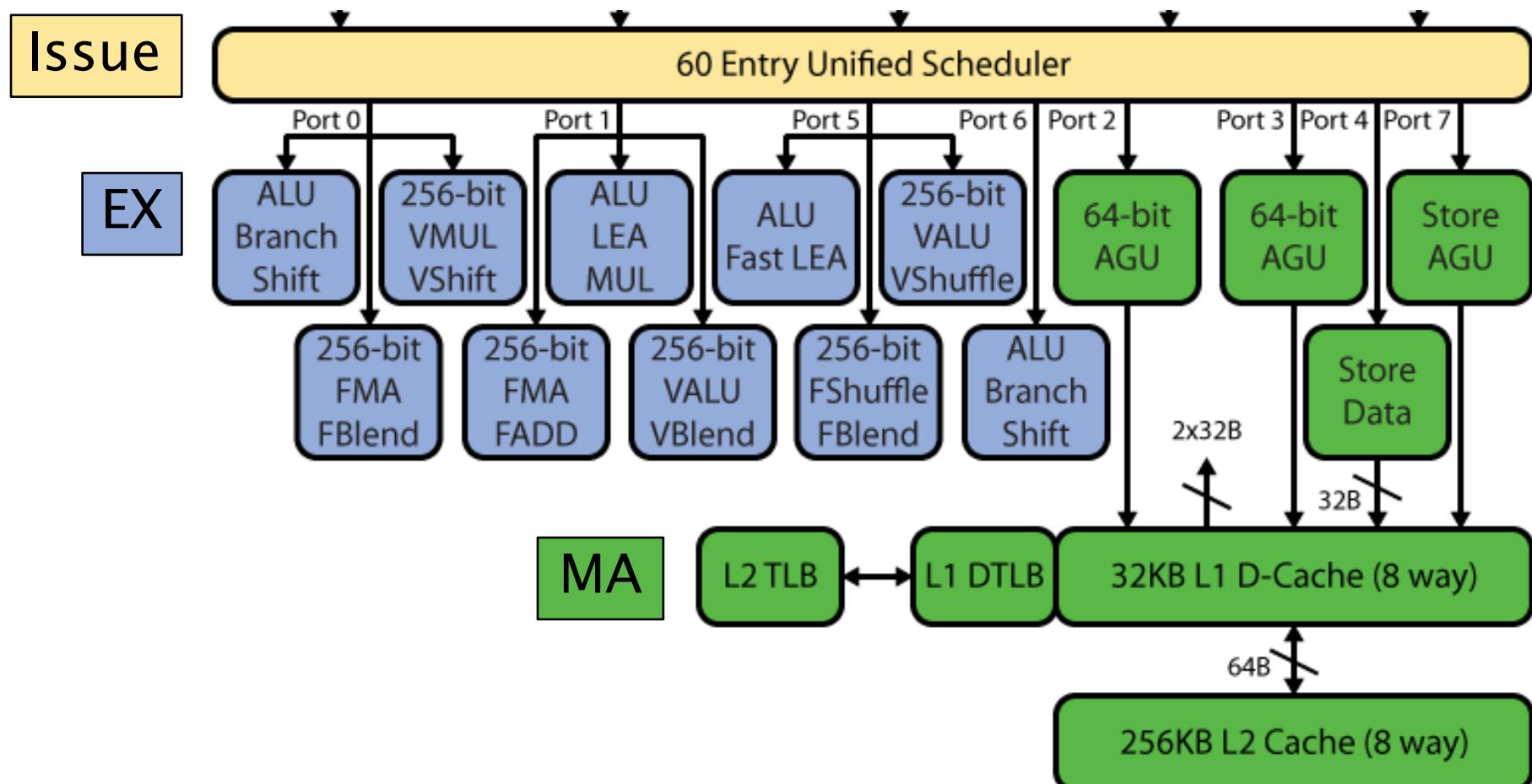
# Complex Pipelining

Idea: Use **separate functional units** for complex operations, such as floating-point arithmetic.



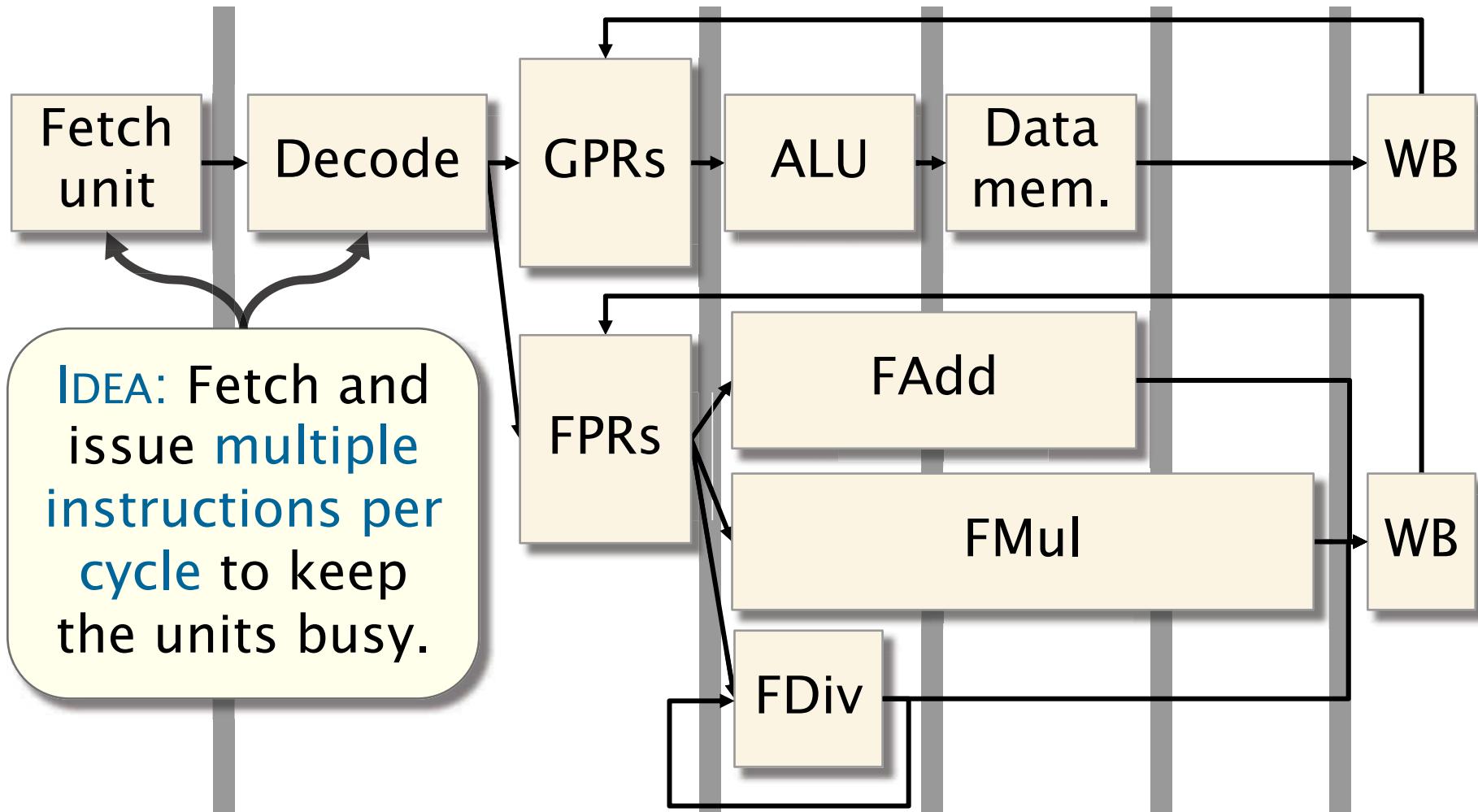
# Intel Haswell Functional Units

Haswell uses a suite of integer, vector, and floating-point functional units, distributed among 8 different ports.



# From Complex to Superscalar

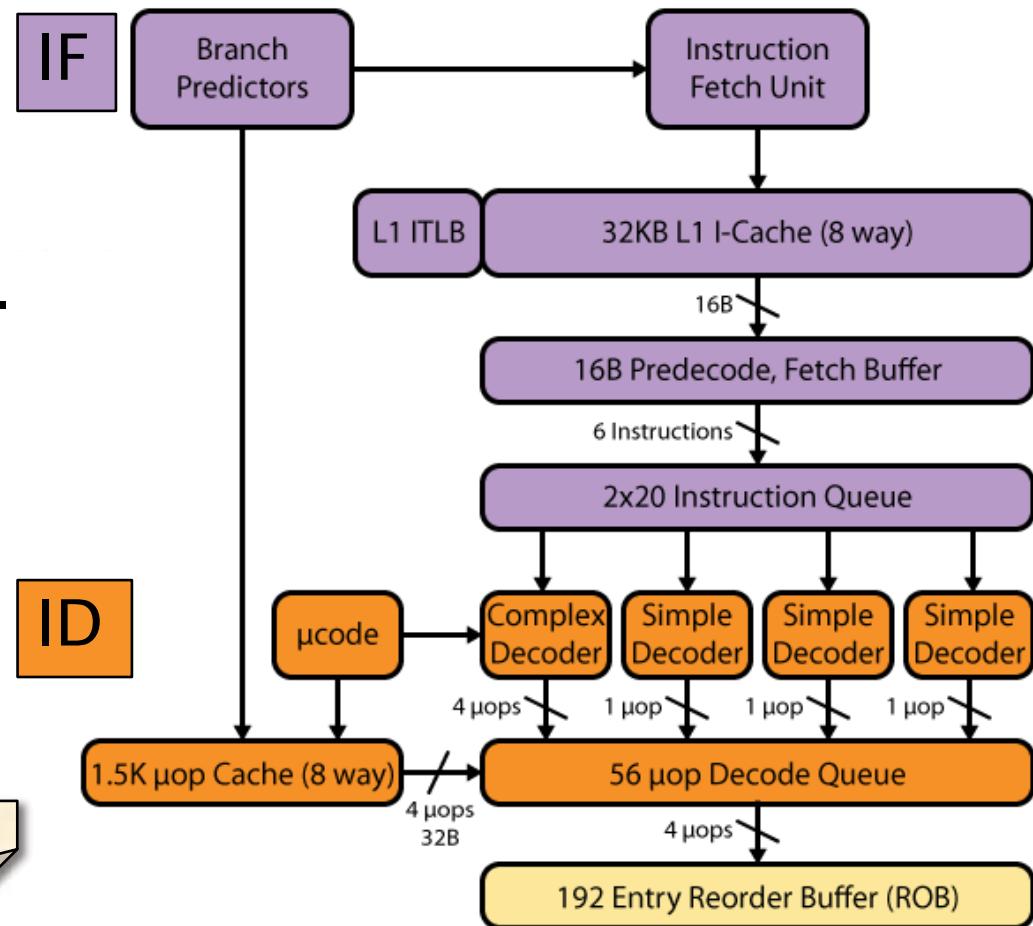
Given these additional functional units, how can the processor further exploit ILP?



# Intel Haswell Fetch and Decode

Haswell break up x86-64 instructions into simpler operations, called **micro-ops**.

- The fetch and decode stages can emit **4** micro-ops per cycle to the rest of the pipeline.
- The fetch and decode stages implement **optimizations** on micro-op processing, including special cases for common patterns, e.g., **xor %rax, %rax**

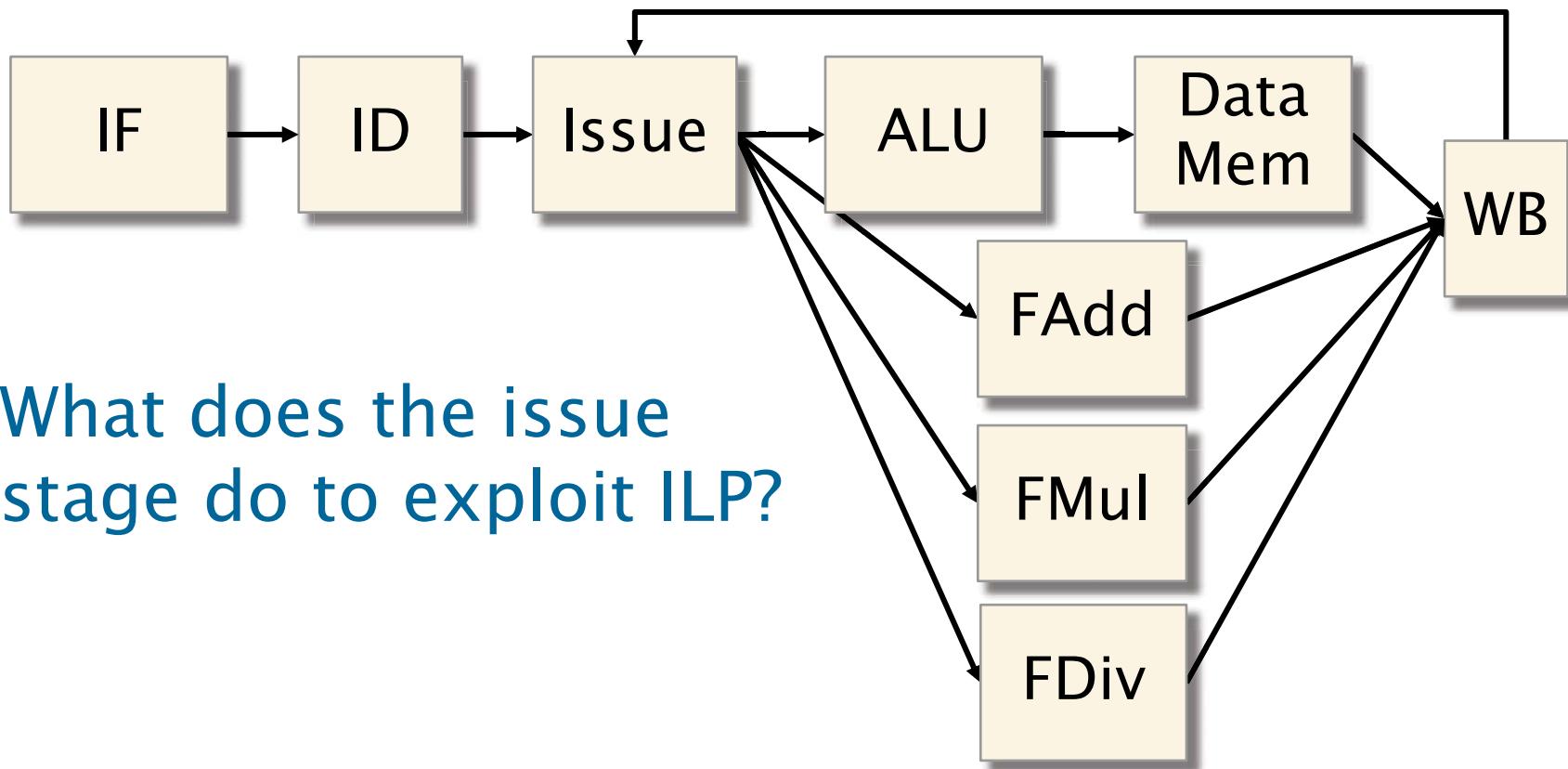


# Bridging the Gap

- Vector hardware
- Superscalar processing
- Out-of-order execution
- Branch prediction

# Block Diagram of a Superscalar Pipeline

The **issue** stage in the pipeline manages the functional units and handles scheduling of instructions.



What does the issue stage do to exploit ILP?

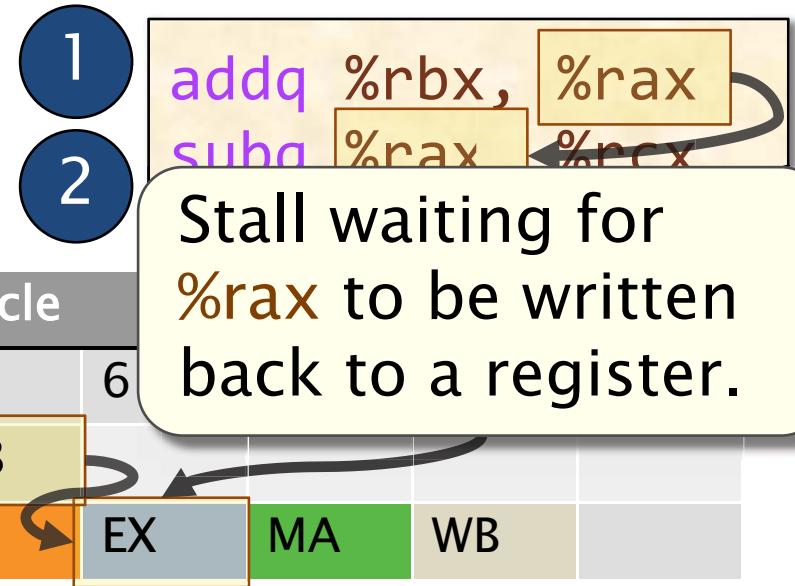
# Bypassing

Bypassing allows an instruction to read its arguments before they've been stored in a GPR.

## Without bypassing

Instr. #	Cycle					
	1	2	3	4	5	6
1	IF	ID	EX	MA	WB	
2		IF	ID	ID	ID	EX

## Example



## With bypassing

Instr. #	Cycle								
	1	2	3	4	5	6	7	8	9
1	IF	ID	EX	MA	WB				
2		IF	ID	EX	MA	WB			

Stall eliminated.

# Data Dependencies: Example

What else can the hardware do to exploit ILP?  
Let's consider a larger code example with more data dependencies.

## Example instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	
	mulsd %xmm1, %xmm0	

## Simplifying assumptions

Latencies chosen to simplify example.

RAW dependence

WAR dependence

The issue stage issues one operation per cycle.

- The processor has plenty of functional units for all operations.

- We'll ignore the non-execute stages of the pipeline.

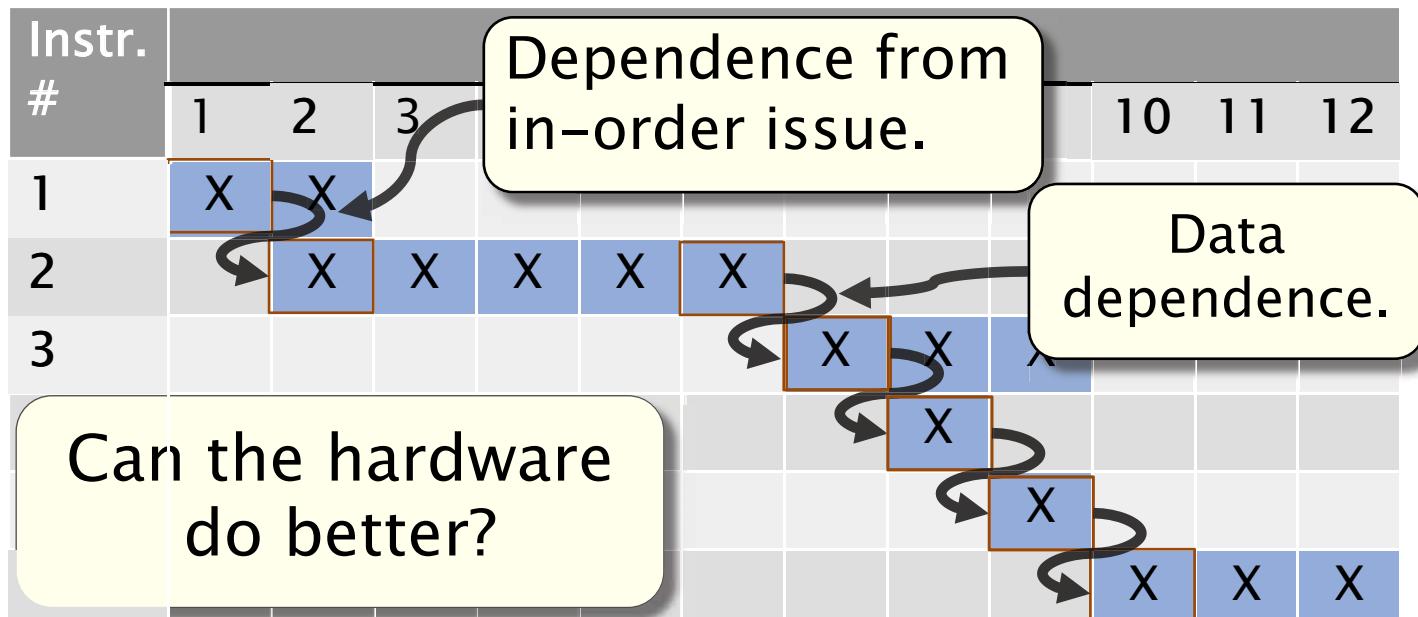
# Example: In-Order Issue

## Instruction stream

If the hardware must issue all instructions **in order**, how long does execution take?

Instruction timing for use of functional units

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

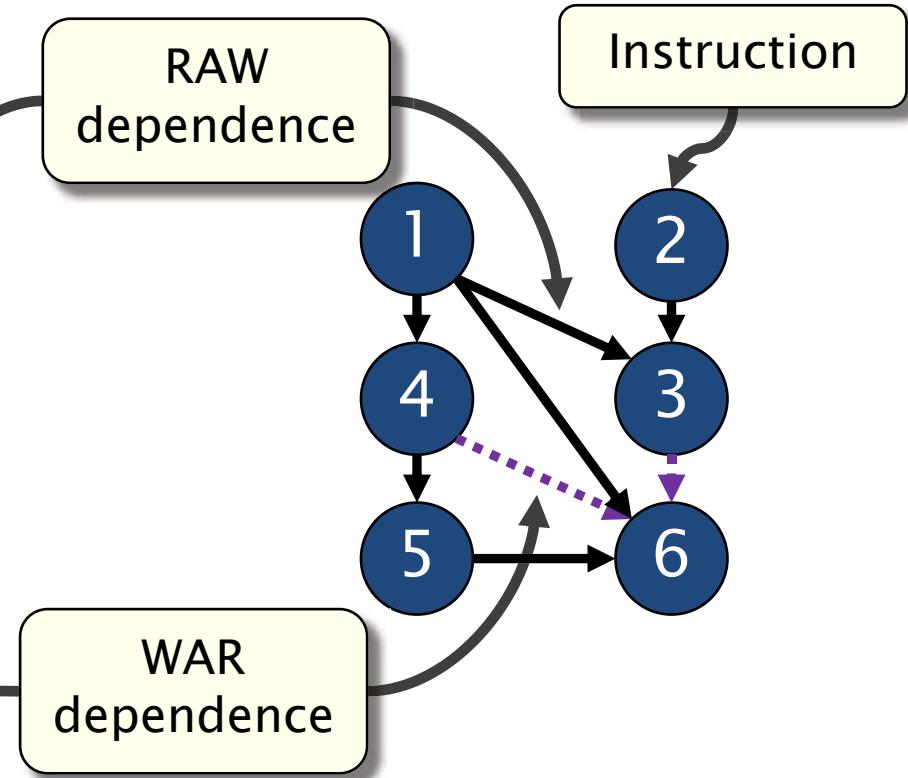


# Data-Flow Graphs

We can model the data dependencies between instructions as a **data-flow graph**.

## Example instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3



# In-Order Issue, Revisited

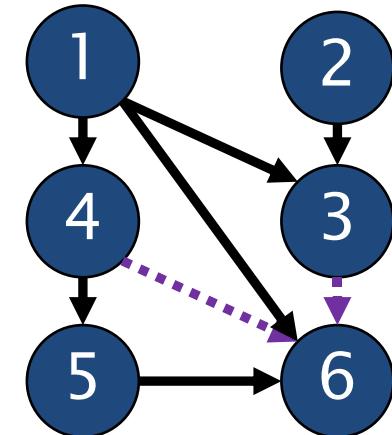
## Instruction stream

False dependence!

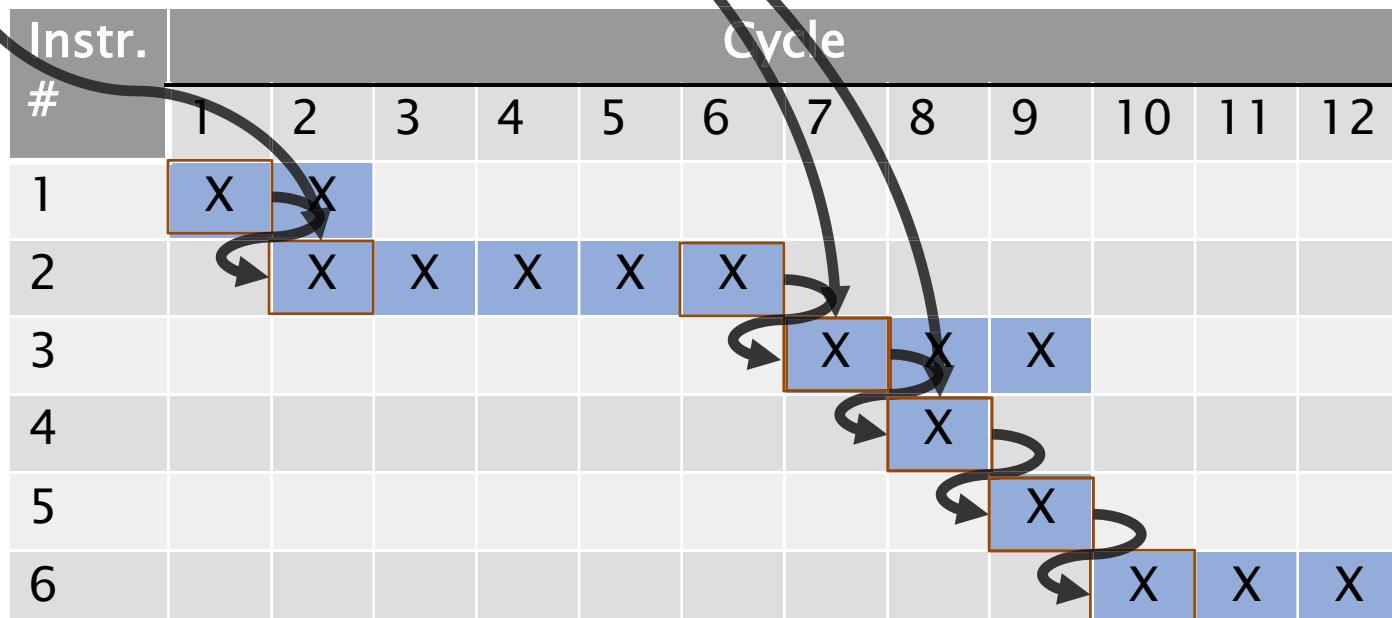
Instruction 4  
doesn't depend on  
instructions 2 or 3!

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	(%rax), %xmm2	5
3	(%rax), %xmm2	3
4	(%rax), %xmm1	1
5	(%rax), %xmm1	1
6	mulsd %xmm1, %xmm0	3

## Data-flow graph



## Instruction timing for use of functional units



# Example: Out-of-Order Execution

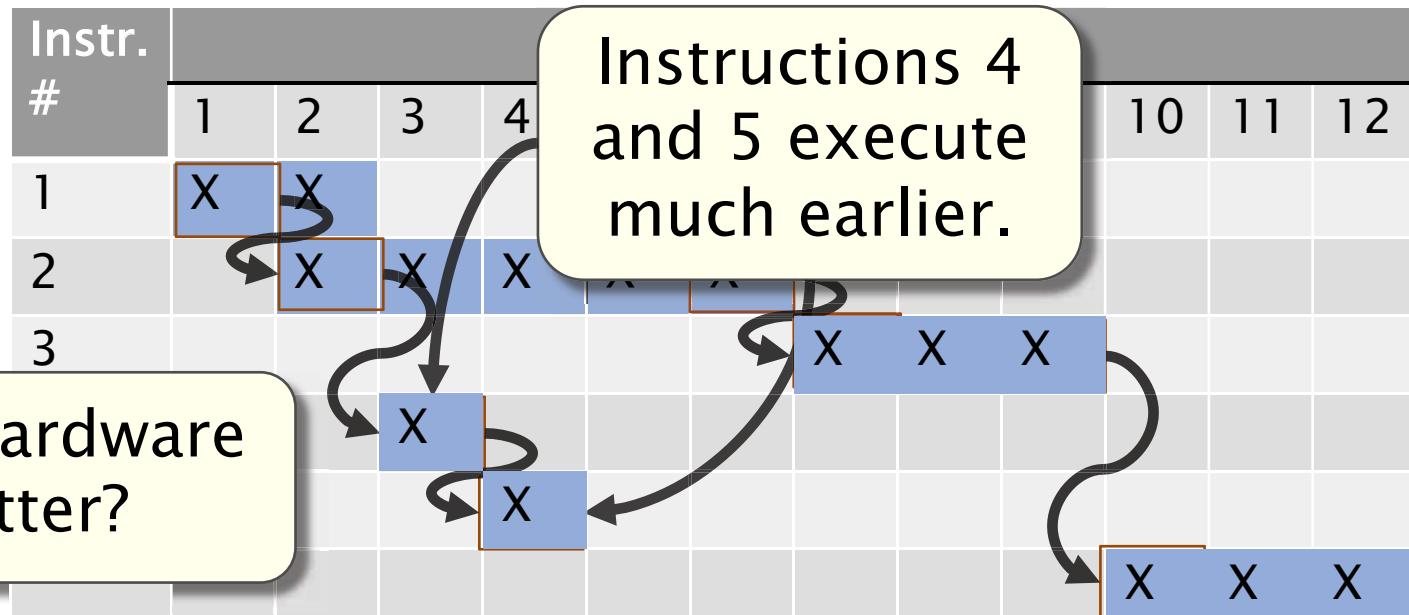
Idea: Let the hardware issue an instruction as soon as its data dependencies are satisfied.

## Instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

Instruction timing for use of functional units

Can the hardware do better?

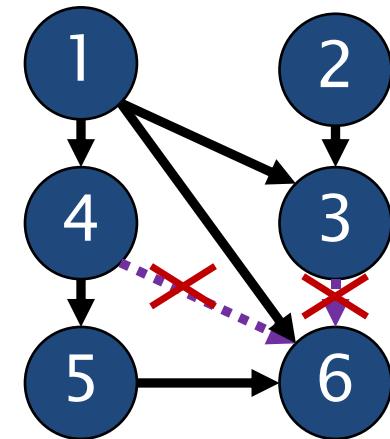


# Eliminating Name Dependencies

Instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

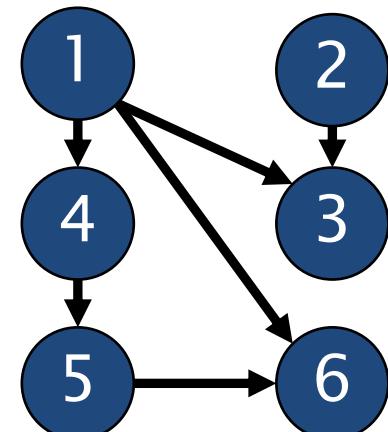
Data-flow graph



Idea: If the **name** of the **destination register** could be changed, then the WAR dependencies could be **eliminated**.

Instruction 6 no longer depends on long latency operations!

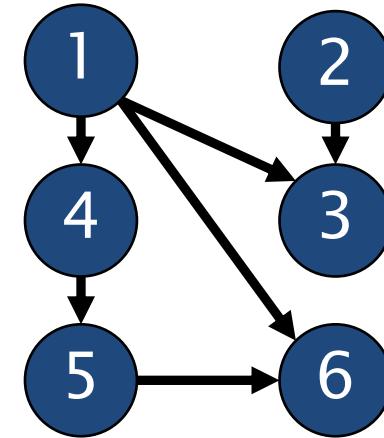
New data-flow graph



# Effect of Eliminating Name Deps.

Idea: Let the hardware change destination register names on the fly.

New data-flow graph



Instruction timing for use of functional units

Instr. #	1	2	3								
1	x	x									
2		x	x	x	x	x	x				
3							x	x	x		
4				x							
5					x						
6						x	x	x			

Combined with out-of-instructions 4, 5, and 6 can all execute earlier.

# Removing Data Dependencies

The processor mitigates the performance loss of data hazards using two techniques.

- **Register renaming** removes WAR and WAW dependencies.
- **Out-of-order execution** reduces the performance lost due to RAW dependencies.

# On-the-Fly Register Renaming

How does hardware overcome WAR and WAW dependencies?

Idea: Architecture implements many more physical registers than registers specified by the ISA.

Renaming table

ISA Reg.	Data
xmm0	Preg7
xmm1	Preg4
xmm2	Preg2
xmm3	

List of free physical registers

Preg3	Preg8	Preg1	...	Preg0
-------	-------	-------	-----	-------

Maintains a mapping from ISA registers to physical registers.



# Dynamic Instruction Reordering

The issue stage tracks the data dependencies between instructions dynamically using a circular buffer, called a **reorder buffer (ROB)**.

## Sketch of a Reorder buffer

Tag	Instr. #	OP	Source 1	Source 2	Dest.
t1	1	movsd			Preg7
t2	2	movsd			Preg2
t3	3	mulsd	t1	t2	
t4					
t5					
t6					
t7					

Actual ROB hardware is more complex.

# Dynamic Reordering and Renaming: Example

Instruction stream

$\%rip \rightarrow$

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

Initial state:  
Instructions 2  
and 3 are  
executing.

Reorder buffer

Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	Preg4
xmm2	t2
xmm3	

Tag	Instr. #	OP	Source 1	Source 2	Dest.
t1	1	movsd			Preg7
t2	2	movsd			Preg2
t3					
t4					
t5					
t6					
t7					

# Dynamic Reordering and Renaming: Example

Instruction stream

$\%rip \rightarrow$

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

Step: Decode instruction 3.

Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	Preg4
xmm2	t2
xmm3	

Reorder buffer

Tag	Instr. #	OP	Source 1	Source 2	Dest.
t1	1	movsd			Preg7
t2	2	movsd			Preg2
t3	3	mulsd	t1	t2	
t4					
t5					
t6					
t7					

# Dynamic Reordering and Renaming: Example

Instruction stream

%rip →

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
		3

Update mapping in renaming table.

Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	Preg4
xmm2	t3
xmm3	

Step: Decode instruction 3.

Reorder buffer

Reg	OP	Source 1	Source 2	Dest.
t1	1	movsd		Preg7
t2	2	movsd		Preg2
t3	3	mulsd	t1	t2
t4				
t5				
t6				
t7				

# Dynamic Reordering and Renaming: Example

Instruction stream

%rip →

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

Step: Decode instruction 4.

Reorder buffer

Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	Preg4
xmm2	t3
xmm3	

Tag	Instr. #	OP	Source 1	Source 2	Dest.
t1	1	movsd			Preg7
t2	2	movsd			Preg2
t3	3	mulsd	t1	t2	
t4	4	addsd	t1	Preg4	
t5					
t6					
t7					

# Dynamic Reordering and Renaming: Example

Instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
		3

%rip →

Update mapping in renaming table.

Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	t4
xmm2	t3
xmm3	

Step: Decode instruction 4.

Reorder buffer

Reg	Op	Source 1	Source 2	Dest.
t1	1	movsd		Preg7
t2	2	movsd		Preg2
t3	3	mulsd	t1	t2
t4	4	addsd	t1	Preg4
t5				
t6				
t7				

# Dynamic Reordering and Renaming: Example

Instruction stream

%rip →

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6		3

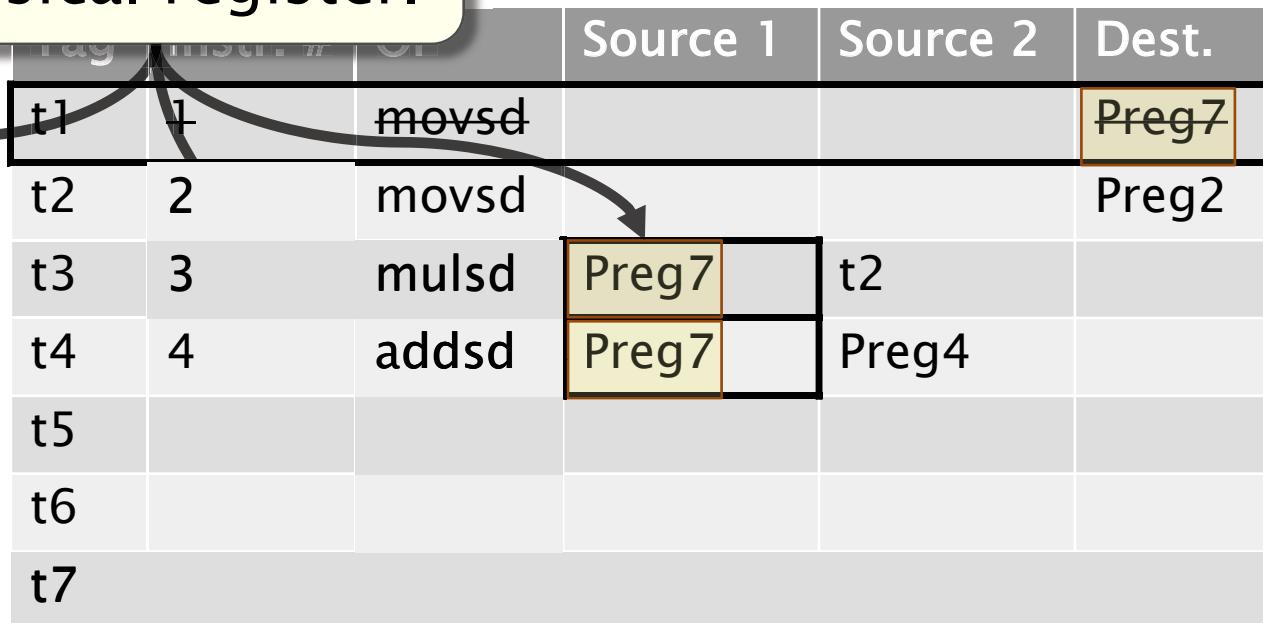
Replace tag with physical register.

Renaming table

ISA Reg.	Data
xmm0	Preg7
xmm1	t4
xmm2	t3
xmm3	

Step: Instruction 1 finishes.

Reorder buffer



# Dynamic Reordering and Renaming: Example

Instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1

%rip →

Instruction 4 is ready to execute **out of order**.

Step: Instruction 1 finishes.

Get the next available physical register from the free list.

Renaming table

ISA Reg.	Data
xmm0	Preg7
xmm1	t4
xmm2	t3
xmm3	

Reg	Index	OP	Source 1	Source 2	Dest.
t1	1	movsd			Preg7
t2	2	movsd			Preg2
t3	3	mulsd	Preg7	t2	
t4	4	addsd	Preg7	Preg4	Preg3
t5					
t6					
t7					

# Dynamic Reordering and Renaming: Example

Instruction stream

%rip →

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

Step: Decode instruction 5.

Reorder buffer

Renaming table

ISA Reg.	Data
xmm0	Preg7
xmm1	t4
xmm2	t3
xmm3	

Tag	Instr. #	OP	Source 1	Source 2	Dest.
t1	1	movsd			Preg7
t2	2	movsd			Preg2
t3	3	mulsd	Preg7	t2	
t4	4	addsd	Preg7	Preg4	Preg3
t5	5	addsd	t4	t4	
t6					
t7					

# Dynamic Reordering and Renaming: Example

Instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
%rip → 5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

Renaming

ISA Reg.	Data
xmm0	Preg7
xmm1	t5
xmm2	t3
xmm3	

Update mapping in renaming table.

Step: Decode instruction 5.

Reorder buffer

	OP	Source 1	Source 2	Dest.
	movsd			Preg7
t2	2	movsd		Preg2
t3	3	mulsd	Preg7	t2
t4	4	addsd	Preg7	Preg4
t5	5	addsd	t4	Preg3
t6				
t7				

# Dynamic Reordering and Renaming: Example

Instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	

%rip →

Renaming table

ISA Reg.	Data
xmm0	Preg7
xmm1	t4
xmm2	t3
xmm3	

Tag	Instr. #	OP			
t1	1	mo			
t2	2	mo			
t3	3	mulsd	Preg7	t2	
t4	4	addsd	Preg7	Preg4	Preg3
t5	5	addsd	t4	t4	
t6	6	mulsd	t5	Preg7	
t7					

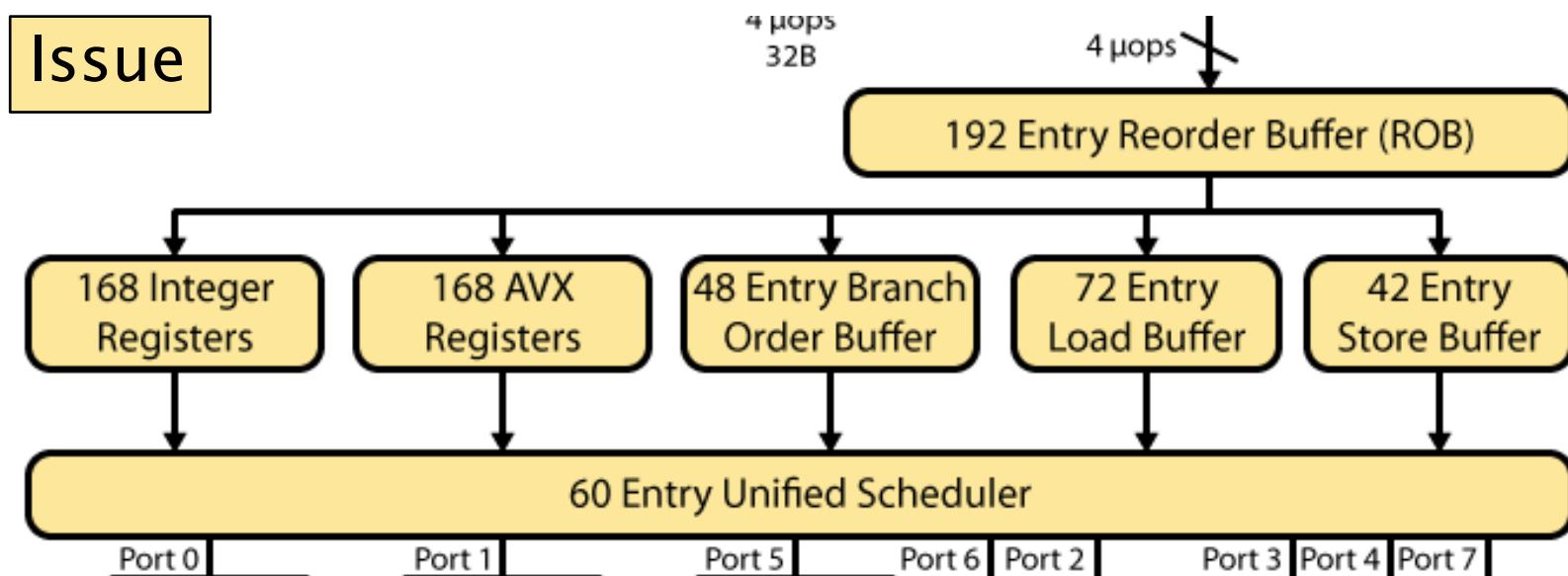
Step: Decode instruction 6.

Renaming: Destination register will be chosen from free list when instruction is issued.

# Haswell Renaming and Reordering

Haswell uses a reorder buffer and separate register files for integer and vector/floating-point registers.

- Haswell implements the ISA's **16** distinct integer registers with **168** physical registers. The same ratio holds independently for the AVX registers.
- Conversions between integer and floating-point or vector values involve **data movement** on chip.



# Summary of Reordering and Renaming

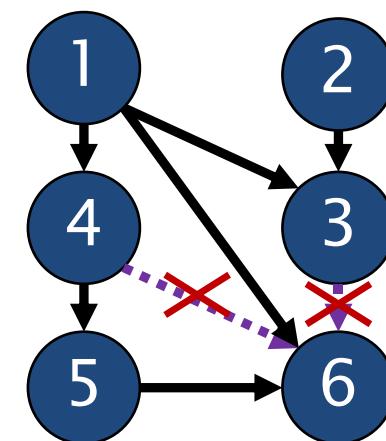
Summary: Hardware renaming and reordering are **effective** in practice.

- Despite the apparent dependencies in the assembly code, typically, only **true dependencies** affect performance.
- Dependencies can be modeled using **data-flow graphs**.

Instruction stream

#	Instruction	Latency
1	movsd (%rax), %xmm0	2
2	movsd (%rbx), %xmm2	5
3	mulsd %xmm0, %xmm2	3
4	addsd %xmm0, %xmm1	1
5	addsd %xmm1, %xmm1	1
6	mulsd %xmm1, %xmm0	3

Data-flow graph

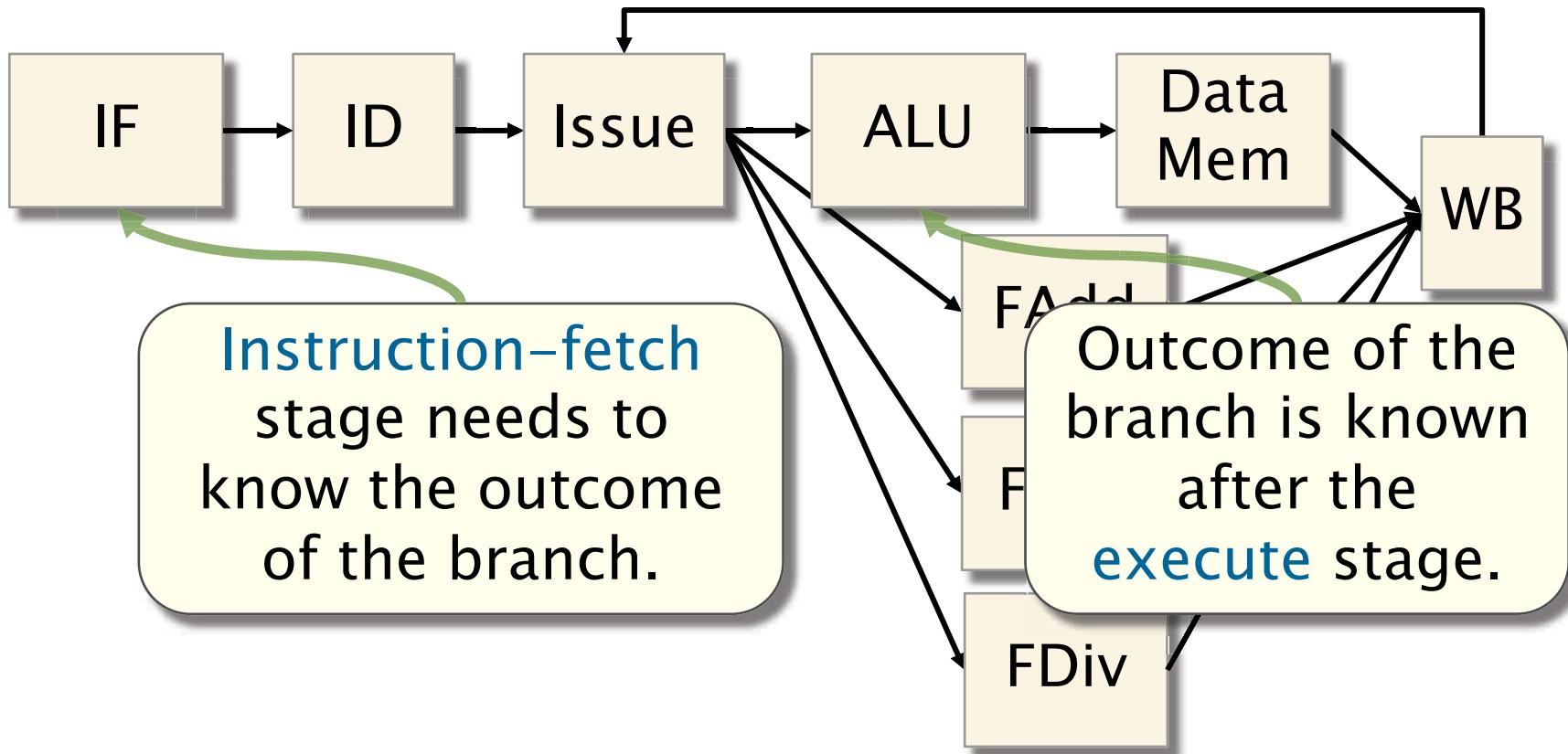


# Bridging the Gap

- Vector hardware
- Superscalar processing
- Out-of-order execution
- Branch prediction

# Control Hazards

What happens if the processor encounters a conditional jump, a.k.a., a **branch**?



# Speculative Execution

To handle a control hazard, the processor either **stalls** at the branch or **speculatively** executes past it.

## Example

%rip →

```
cmpq    %r14, %rbp
jae     .LBB9_3
movq    %rbx, %rdi
movq    %rbp, %rsi
callq   bitarray_get
```

When a branch is encountered, assume it's not taken, and keep executing normally.

# Speculative Execution

To handle a control hazard, the processor either **stalls** at the branch or **speculatively** executes past it.

## Example

%rip →

```
cmpq    %r14, %rbp  
jae     .LBB9_3  
movq    %rbx, %rdi  
movq    %rbp, %rsi  
callq   bitarray_get
```

When a branch is encountered, assume it's **not taken**, and keep executing normally.

**Problem:** The effect on throughput of undoing computation is just like **stalling**.

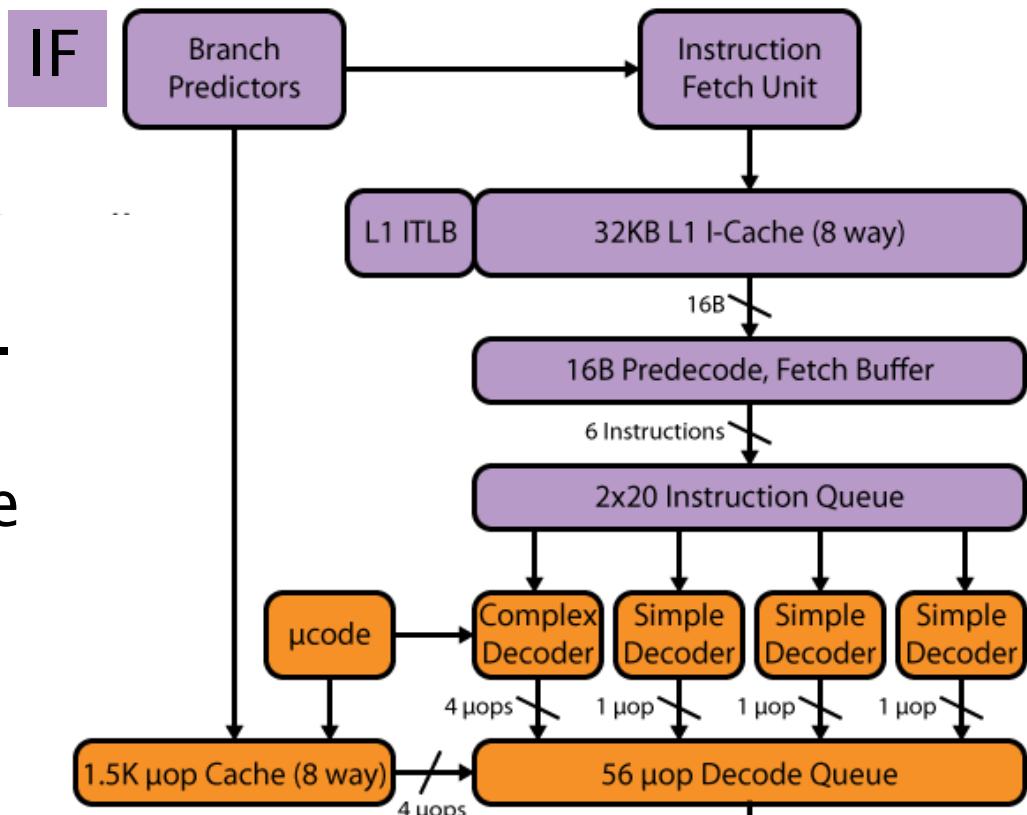
If it is later found that the branch was **taken**, then **undo** the speculative

On Haswell, a mispredicted branch costs **15–20** cycles.

# Supporting Speculative Execution

Modern processors use **branch predictors** to increase the effectiveness of speculative execution.

- The fetch stage dedicates hardware to predicting the outcomes of branches.
- Modern branch predictors are accurate over **95%** of the time.



# Simple Branch Prediction

Idea: Hardware maintains a table mapping addresses of branch instructions to **predictions** of their outcomes.

Instruction address	Prediction
0x400c0c	00
0x400c1b	00
0x400c47	10
0x400cad	10
0x400cd0	00
0x400ced	00
0x400e37	01
0x400e4e	01
0x400e5c	11
0x400e61	10
0x400e72	10

A prediction is encoded as a 2-bit **saturating counter**.

Encoding	Meaning
1 1	Strongly taken
1 0	Weakly taken
0 1	Weakly not taken
0 0	Strongly not taken

A prediction counter is updated based on the actual outcome of the associated branch:

- Taken → increase counter.
- Not taken → decrease counter.

# Branch Prediction with Histories

More-complex branch predictors incorporate **history** information: a record of the outcomes of **k** recent branches, for a small constant **k**.

- A **global history** records the outcomes of the most-recently executed branches on the chip.
- A **local history** records the most recent outcomes of a particular branch instruction.

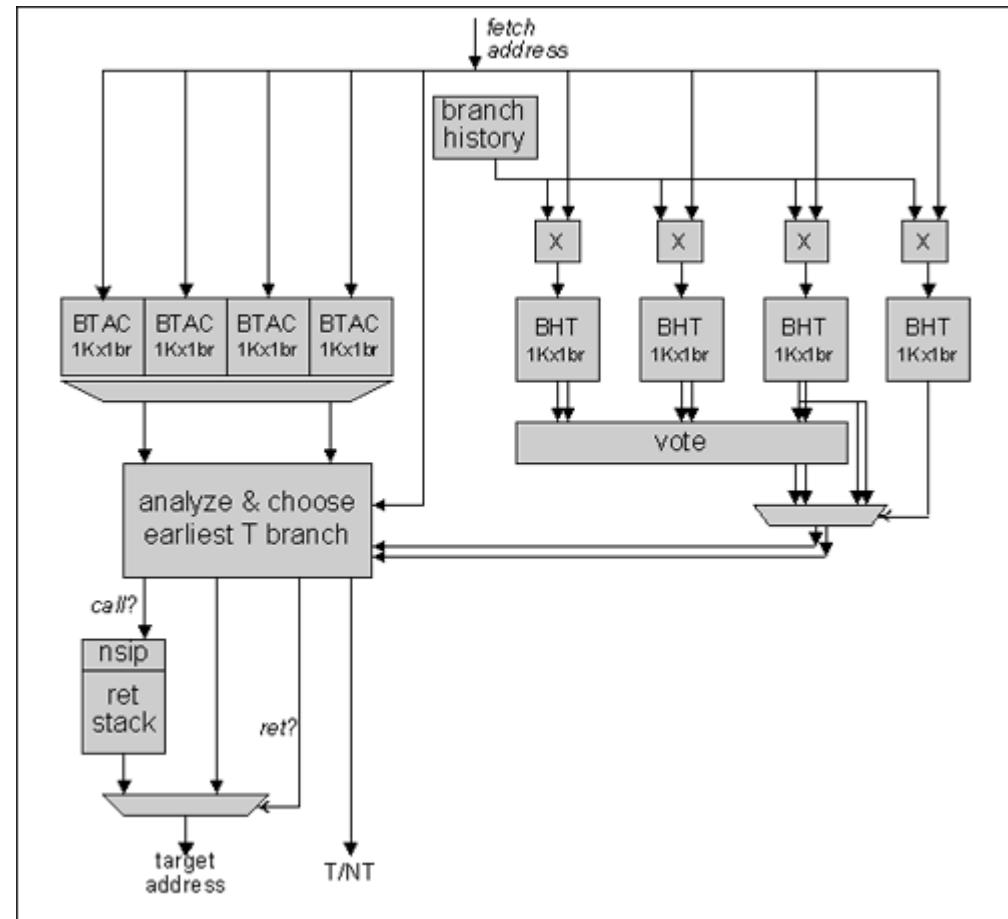
History information can be incorporated into the branch predictor a variety of ways.

- Map the history **directly** to a prediction.
- **Combine** the history with the instruction address (e.g., using XOR) and map the result to a prediction.
- Try **multiple strategies** and **vote** on which result to use at the end.

# Intel Branch Predictor

Not much is publicly known about the construction of the branch predictor in Haswell.

- A **branch-target buffer (BTB)** is used to predict the destinations of **indirect branches**.
- Previous Intel processors have used **two-level predictors**, **loop predictors**, and **hybrid schemes**.
- The branch predictor was redesigned in Haswell....



© [Via Technologies](#). All rights reserved. This content is excluded from our Creative Commons license.

For more information, see <https://ocw.mit.edu/help/faq-fair-use/>

# Summary: Dealing with Hazards

The processor uses several strategies to deal with hazards at runtime:

- **Stalling:** Freeze earlier pipeline stages.
- **Bypassing:** Route the data as soon as it is calculated to an earlier pipeline stage.
- **Out-of-order execution:** Execute a later instruction before an earlier one.
- **Register renaming:** Remove a dependence by changing its register operands.
- **Speculation:** Guess the outcome of the dependence, and restart the calculation only if guess is incorrect.

# Further Reading

- Intel Corporation. Intel 64 and IA-32 Architectures Software Developer Manuals. 2017.  
<https://software.intel.com/en-us/articles/intel-sdm>
- Agner Fog. The Microarchitecture of Intel and AMD CPUs. 2017.  
<http://www.agner.org/optimize/microarchitecture.pdf>
- Intel Corporation. Intel Intrinsics Guide. 2017.