

Performance Engineering of Software Systems

LECTURE 4 Assembly Language and Computer Architecture

Srini Devadas

September 20, 2022



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

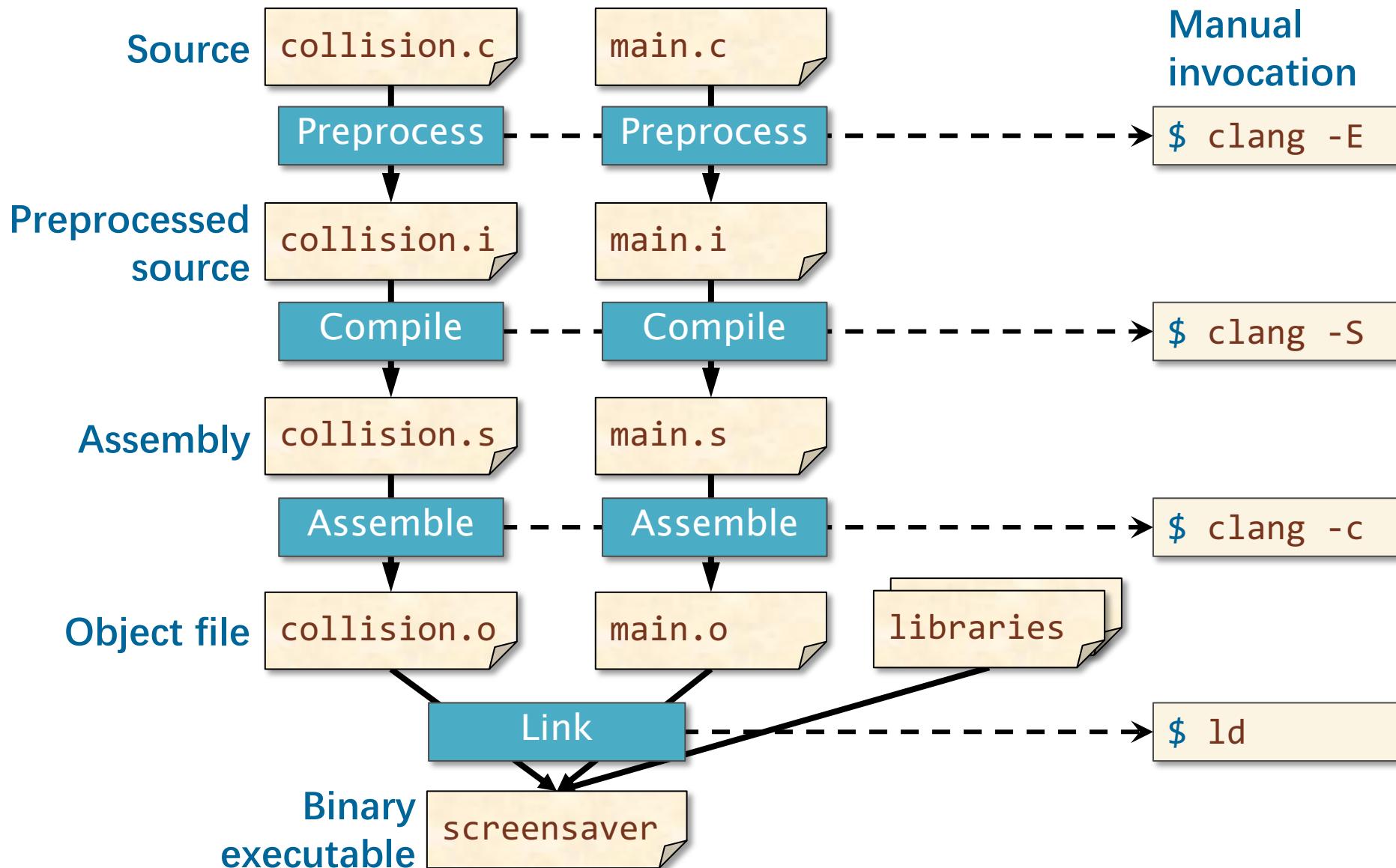
Four stages { Preprocessing
Compiling
Assembling
Linking

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 language
provides a convenient
symbolic representation of
machine code.

Assembly code fib.s

```
.globl _fib  
.p2align ## @fib  
_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
```

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

Assembly Code to Executable

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
```

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
...
...
```

Assembling

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

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**.
 - Sometimes the compiler is the source of the bug!
 - You can write or 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
 - Understand how a compiler implements C language 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
 - Quick introduction
 - Opcodes
 - Operands
- Overview of Modern Computer Architecture
 - Quick Introduction
 - Dealing with Structural Hazards
 - Dealing with Control Hazards
 - Dealing with Data Hazards

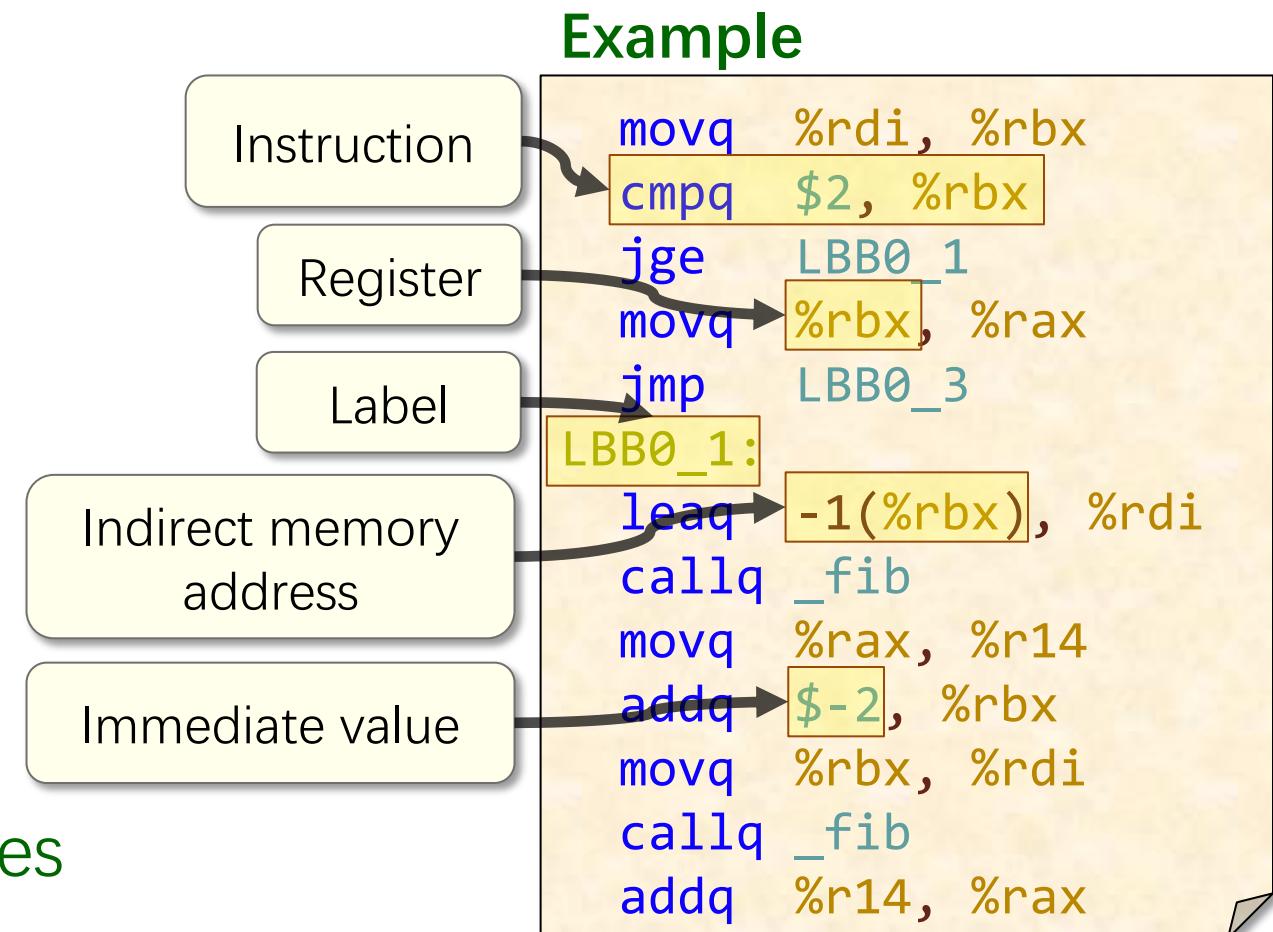
x86-64 ISA PRIMER



The Instruction Set Architecture

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

- Registers
- Instructions
- Data types
- Memory addressing modes



Common x86-64 Registers

Number	Width (bits)	Name(s)	Purpose
16	64	(many)	General-purpose registers (GPRs)
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 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
- For most operations, all operands are sources, and one operand might also be the destination



AT&T Versus Intel Syntax

- What does “ <op> A, B ” mean?

AT&T Syntax $B \text{ <op> } A \rightarrow B$	Intel Syntax $A \leftarrow A \text{ <op> } B$
<code>movl \$1, %eax</code>	<code>mov eax, 1</code>
<code>addl (%ebx,%ecx,0x2), %eax</code>	<code>add eax, [ebx+ecx*2h]</code>
<code>subq 0x20(%rbx), %rax</code>	<code>sub rax, [rbx+20h]</code>

Generated or used by
clang, objdump, perf,
and 6.106 lectures

Used by Intel
documentation

x86-64 ISA PRIMER: OPCODES



Common x86-64 Opcodes

Type of operation		Examples
Data movement	Move	<code>mov</code>
	Conditional move	<code>cmov</code>
	Sign or zero extension	<code>movs, movz</code>
	Stack	<code>push, pop</code>
Arithmetic and logic	Integer arithmetic	<code>add, sub, mul, imul, div, idiv, lea, sal, sar, shl, shr, rol, ror, inc, dec, neg</code>
	Binary logic	<code>and, or, xor, not</code>
	Boolean logic	<code>test, cmp</code>
Control transfer	Unconditional jump	<code>jmp</code>
	Conditional jumps	<code>j<condition></code>
	Subroutines	<code>call, ret</code>

- 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**.

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.

x86-64 ISA PRIMER: OPERANDS



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

Examples

- 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

```
movq $106, %rdi
```

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

```
movq 0x106, %rdi
```

x86-64 Indirect Addressing Modes

The x86-64 ISA also supports **indirect addressing**: specifying a memory address as the result of some simple 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

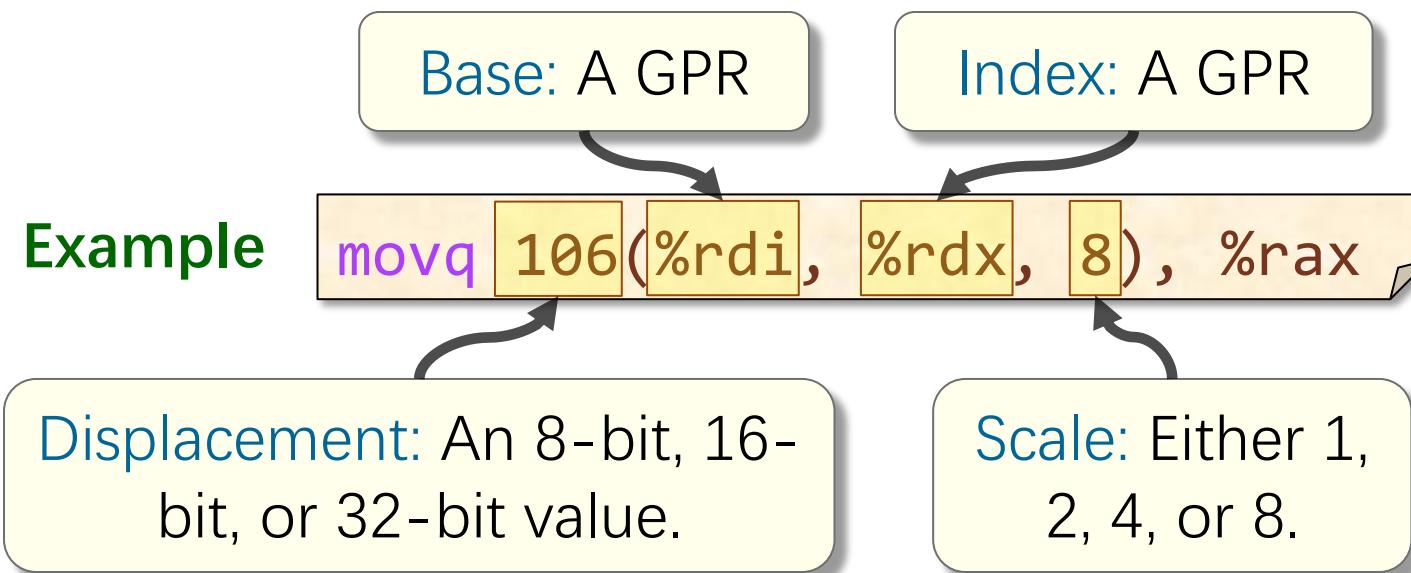
```
movq (%rax), %rdi
```

```
movq 106(%rax), %rdi
```

```
movq 106(%rip), %rdi
```

Base Indexed Scale Displacement

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



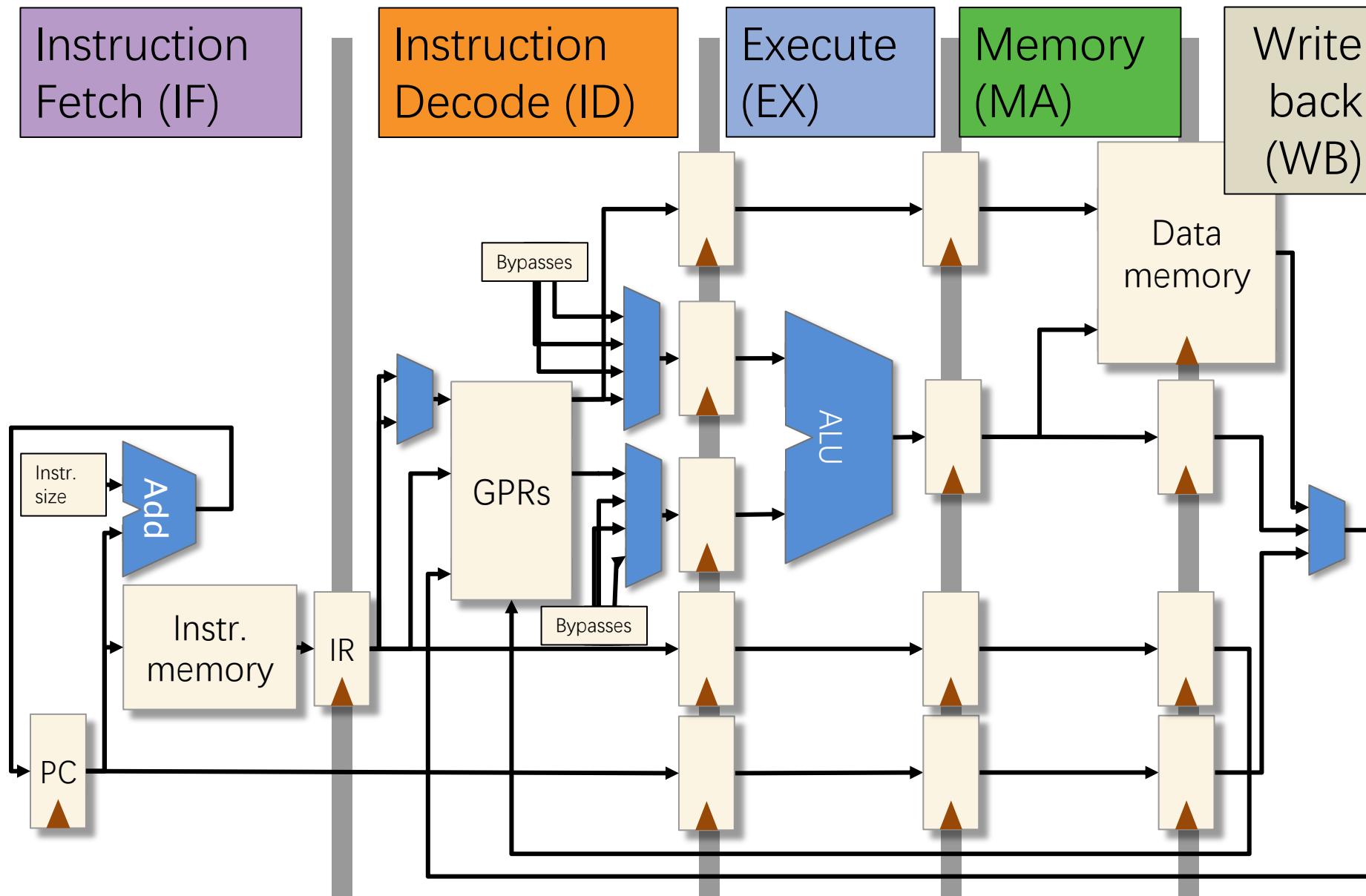
This mode refers to the address: $\text{Base} + \text{Index} * \text{Scale} + \text{Displacement}$

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

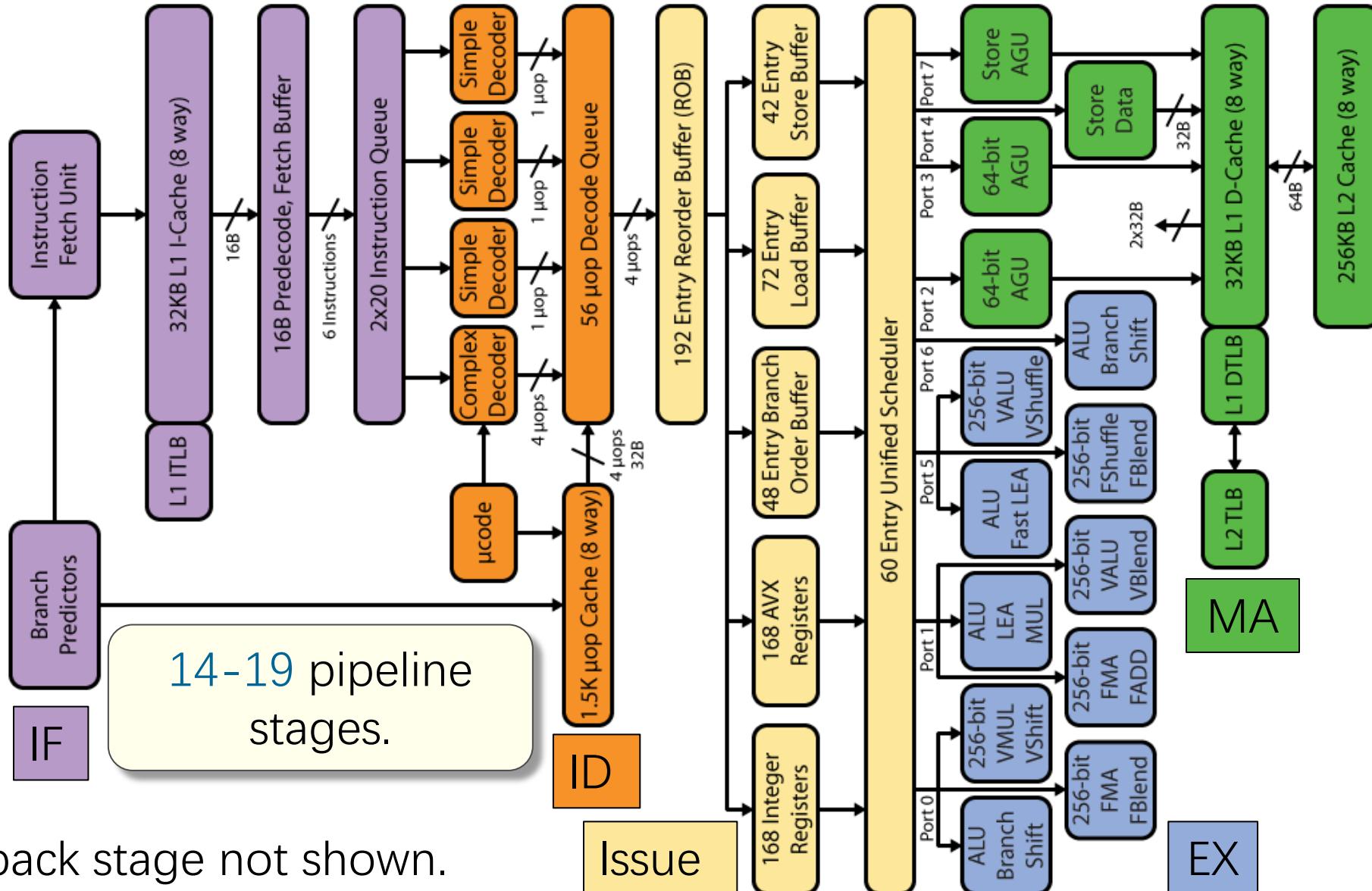
OVERVIEW OF COMPUTER ARCHITECTURE



A Simple 5-Stage Processor



Intel Haswell Microarchitecture



Bridging the Gap

- The rest of this lecture bridges the gap between the simple 5-stage processor and a modern processor core by examining several key design features:
 - Vector hardware
 - Superscalar processing
 - Branch prediction
 - Out-of-order execution

Architectural Improvements

- Broadly speaking, computer architects have historically 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.

Pipelined Instruction Execution

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

Ideal

Pipelined timing

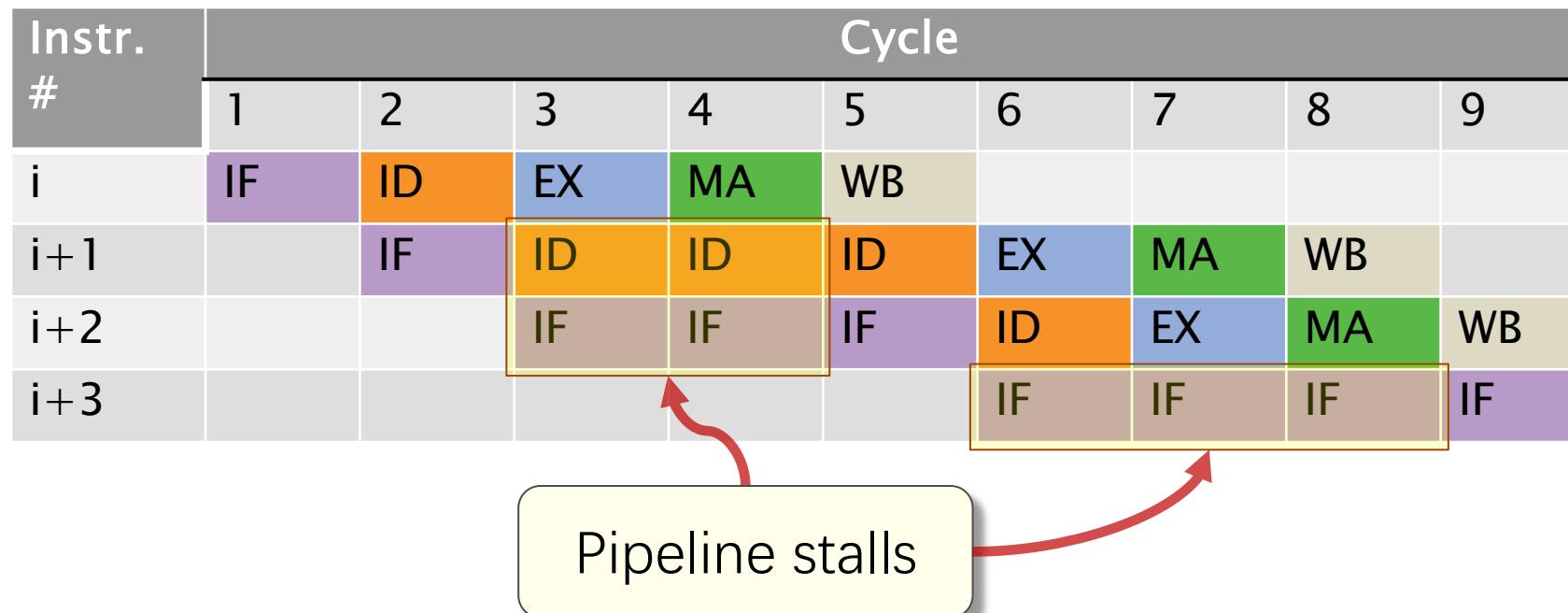
Instr. #	Cycles				
	1	2	3	4	5
i	IF	ID	EX	MA	WB
i+1		IF	ID	EX	MA WB
i+2			IF	ID EX	MA WB
i+3				IF ID EX MA	WB
i+4					IF ID EX MA WB

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

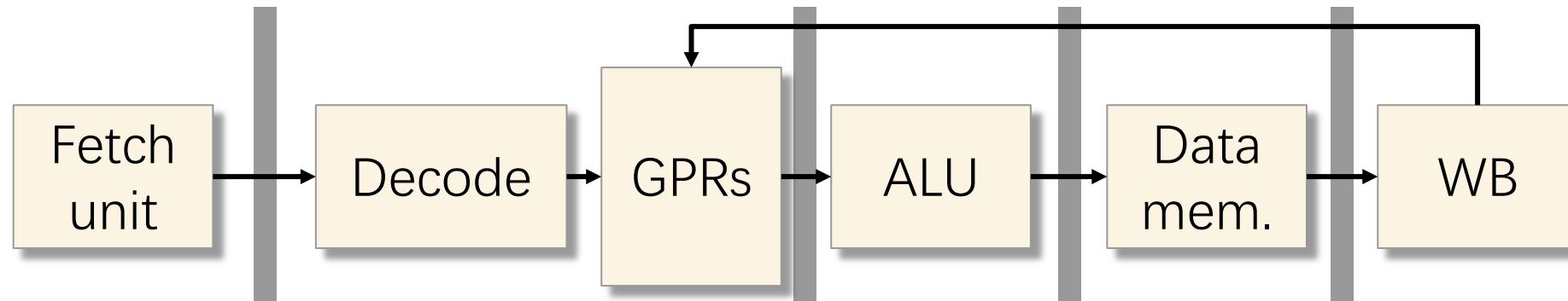
Three types of hazards may prevent an instruction from executing during its designated clock cycle.

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

OVERVIEW OF COMPUTER ARCHITECTURE: DEALING WITH STRUCTURAL HAZARDS



Reexamining the 5-Stage Processor



Observation: The 5-stage processor uses a single ALU to perform all arithmetic operations in the program.

Problem: Complex Operations

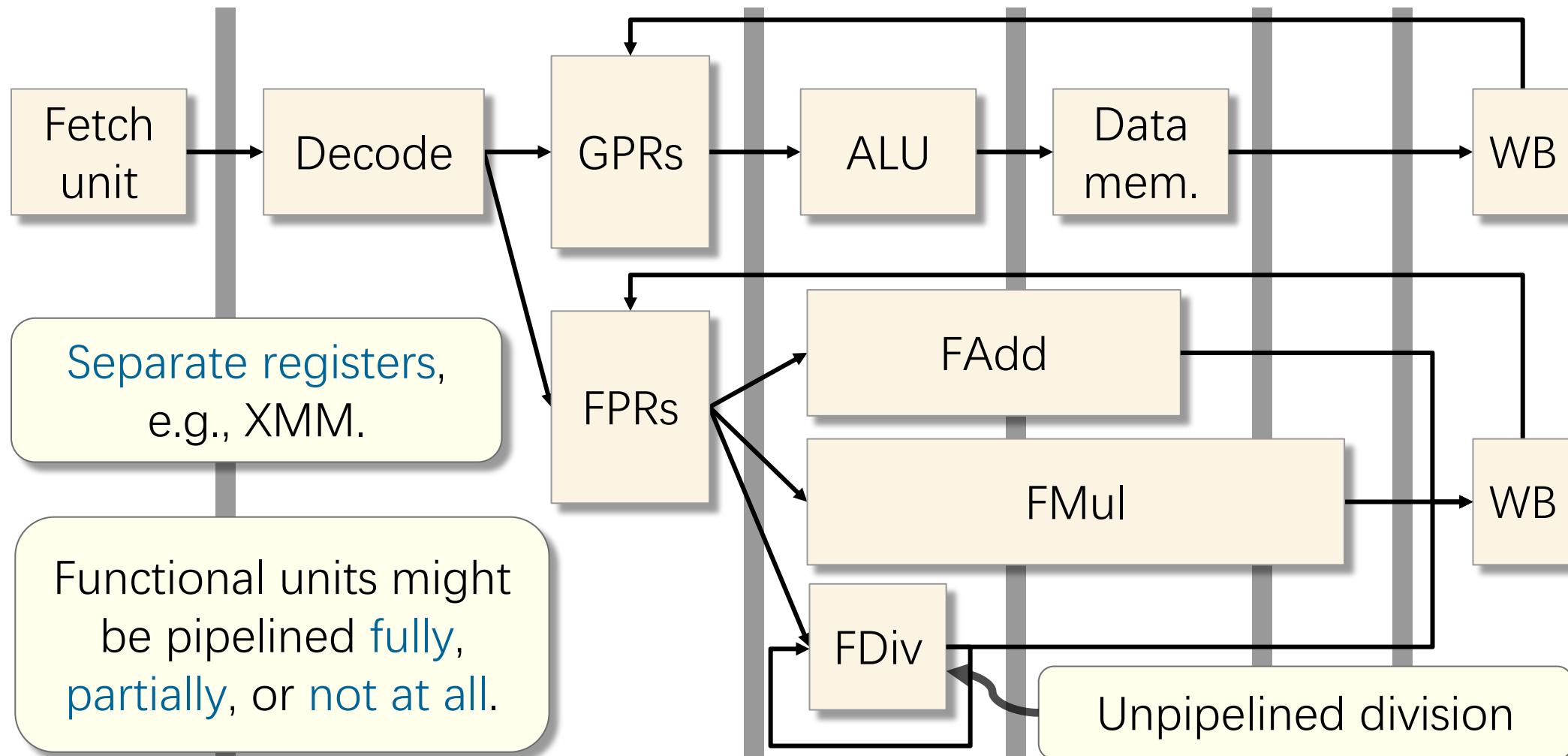
Some arithmetic operations are **complex** to implement in hardware and have **long latencies**.

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, vfmasd	5

How can hardware accommodate these complex operations?

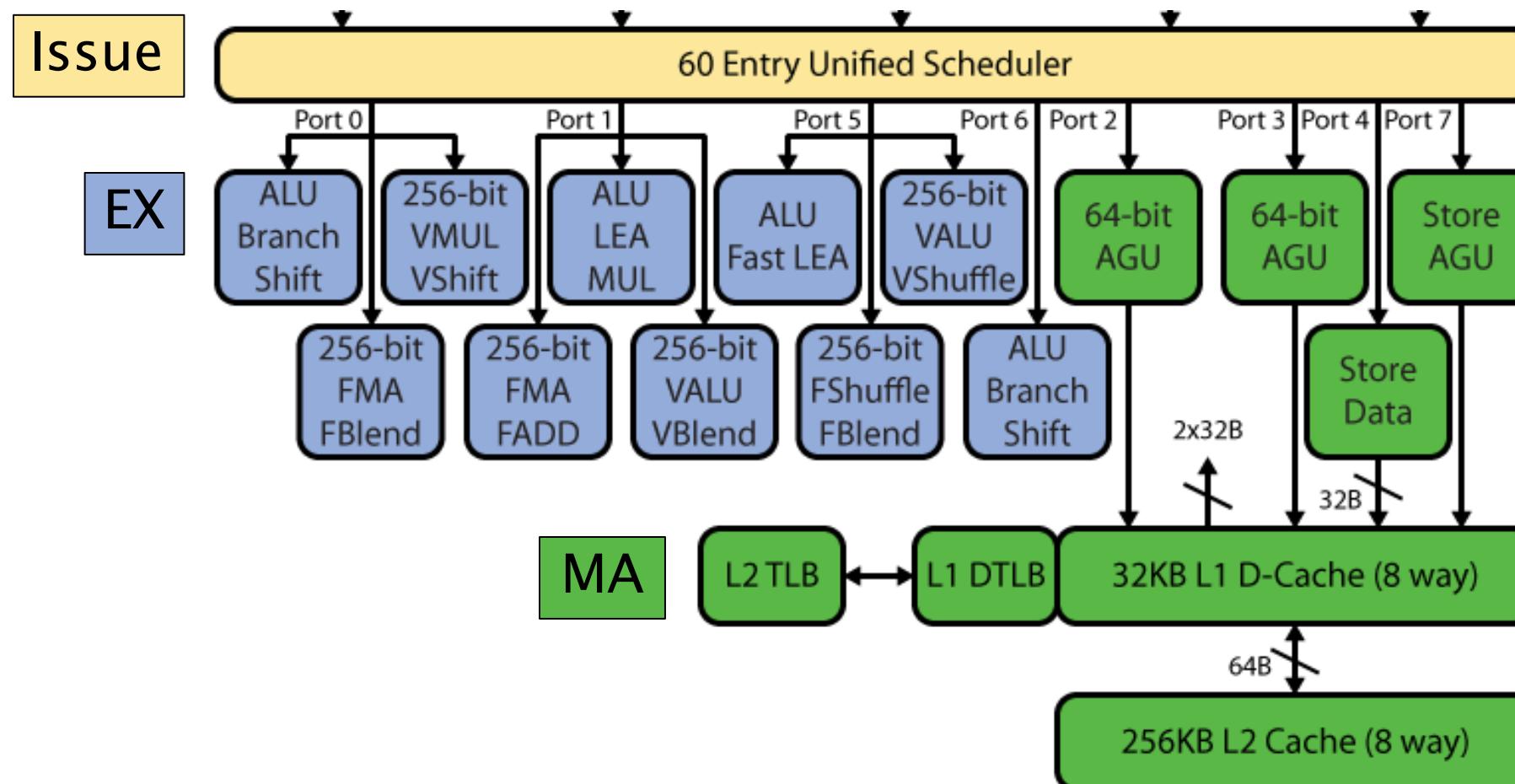
Solution: Complex Pipelining

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



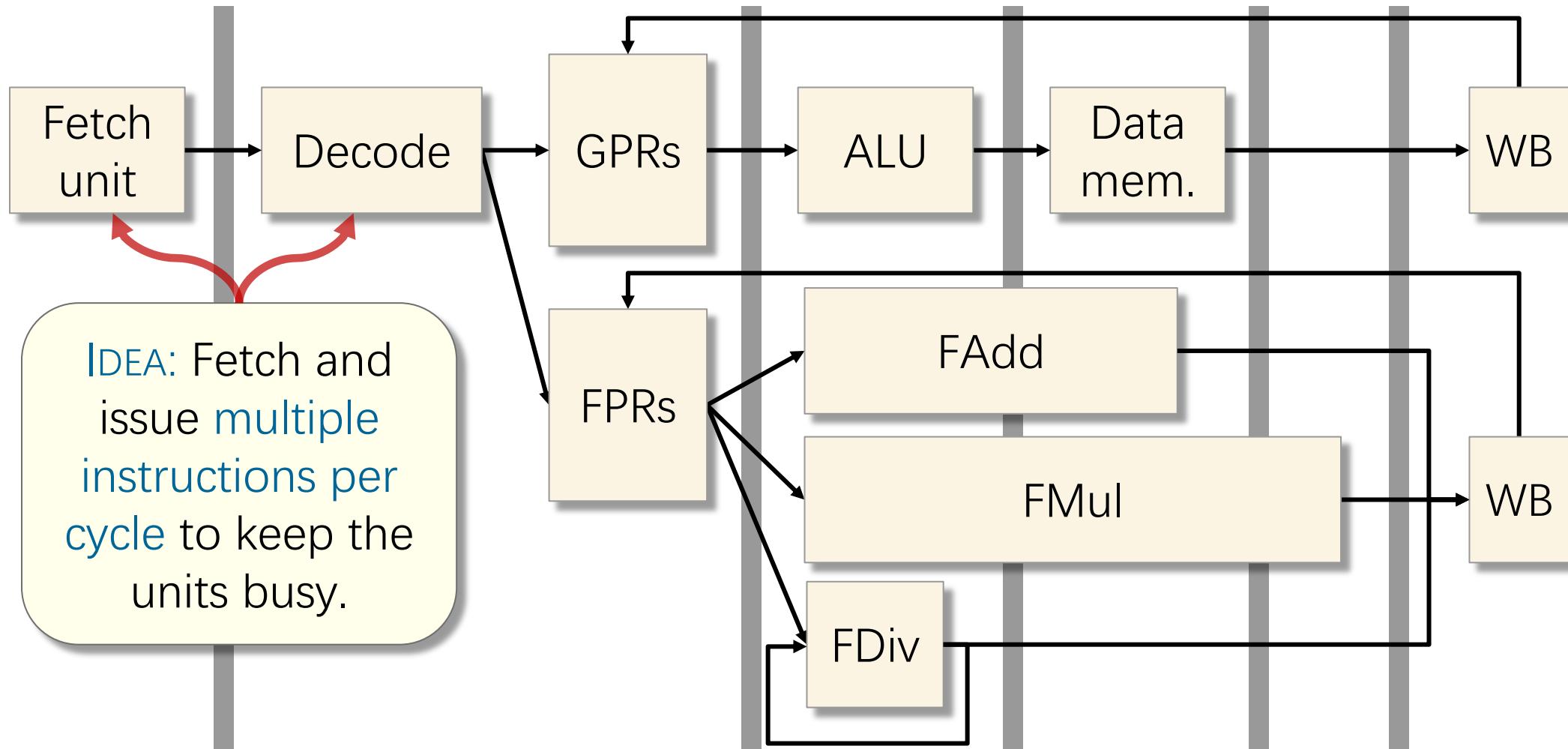
Intel Haswell Functional Units

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**?

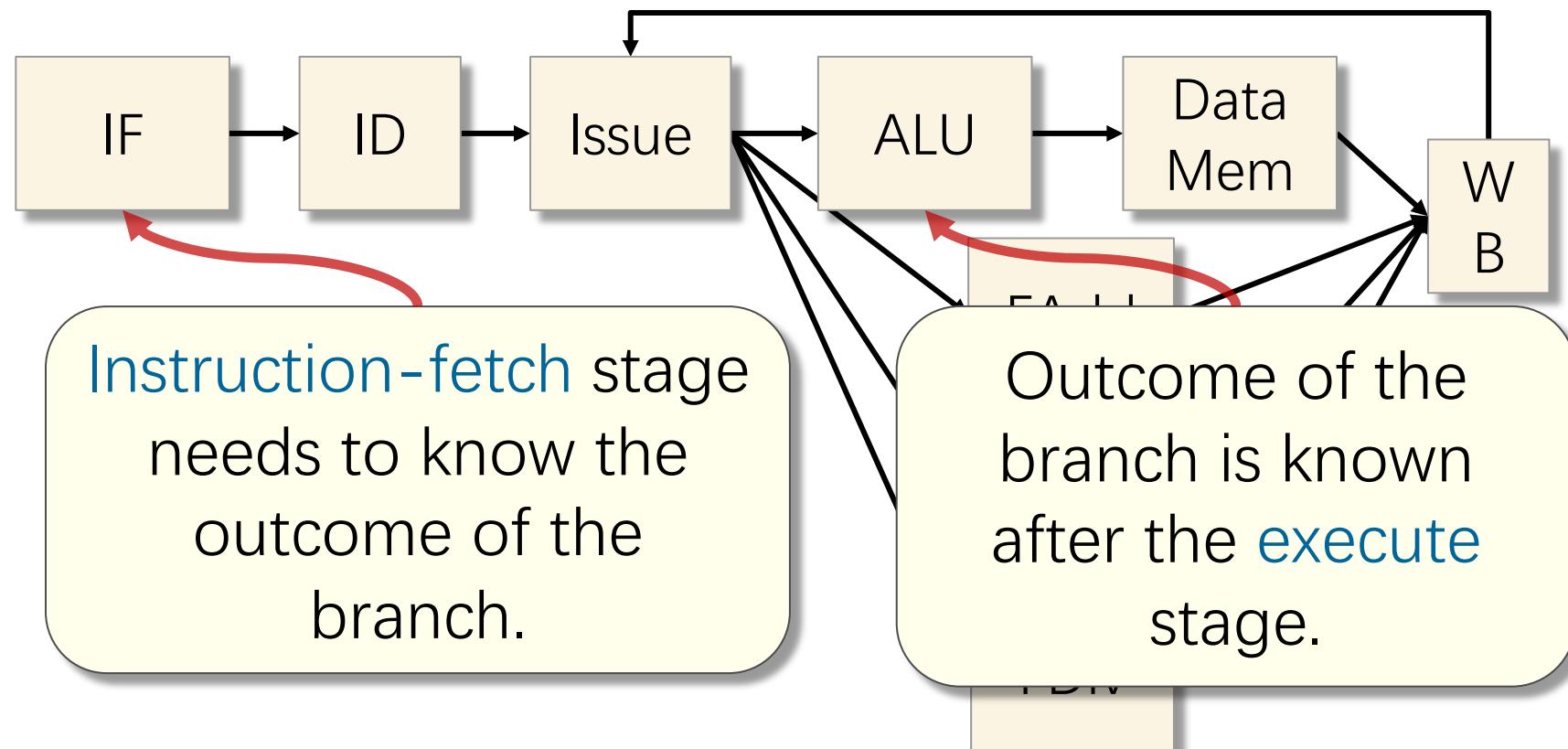


OVERVIEW OF COMPUTER ARCHITECTURE: DEALING WITH CONTROL HAZARDS



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**.

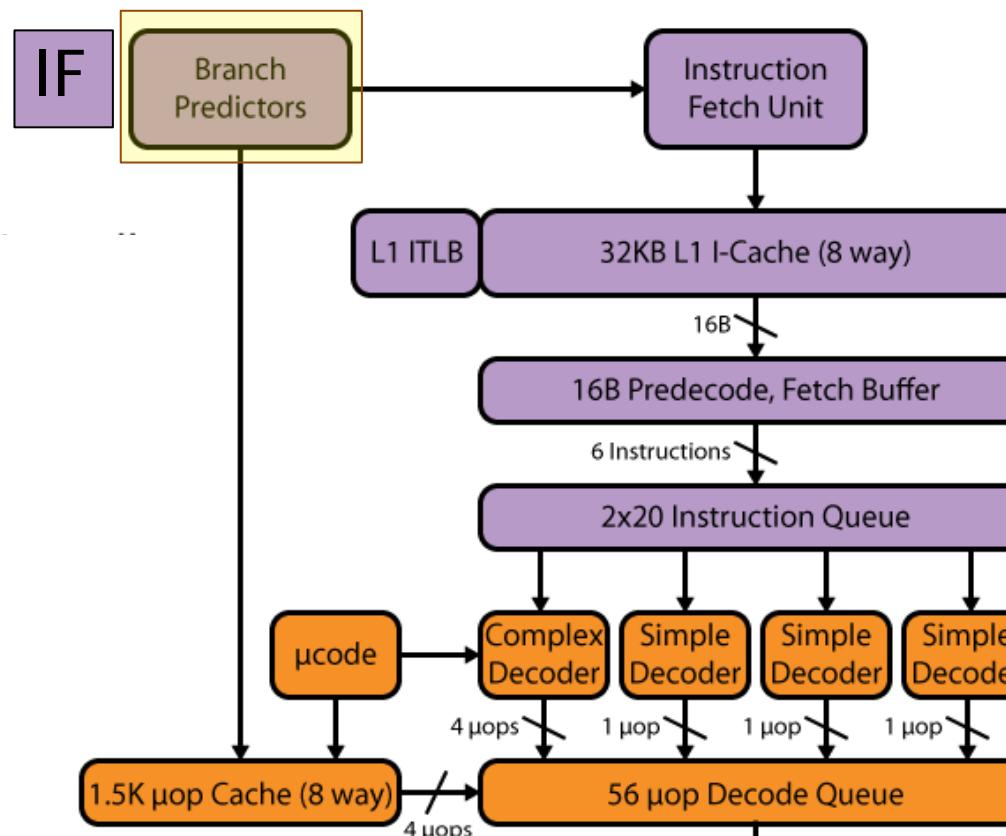
If it is later discovered that the branch was **taken**, then **undo** the speculative computation.

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	Strongly taken
1	Weakly taken
0	Weakly not taken
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.

OVERVIEW OF COMPUTER ARCHITECTURE: DEALING WITH DATA HAZARDS

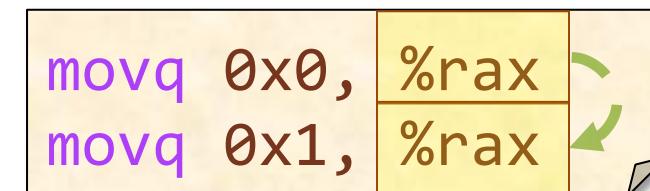
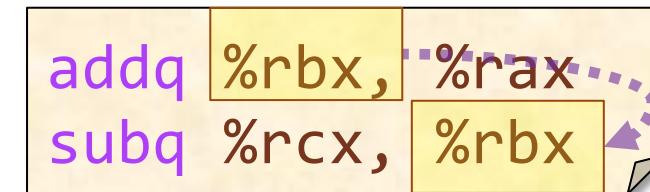
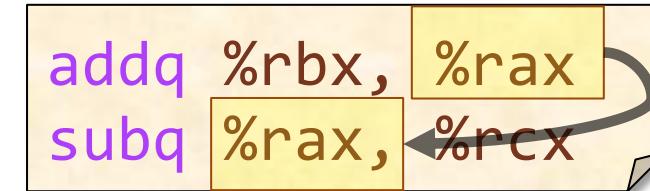


Sources of Data Hazards

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.

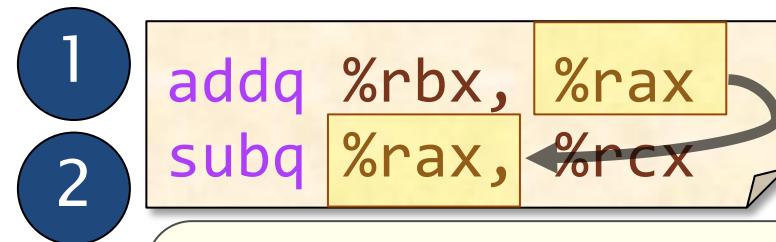
Examples



Bypassing

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

Example



Without bypassing

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

Stall waiting for `%rax` to be written back to a register.

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			

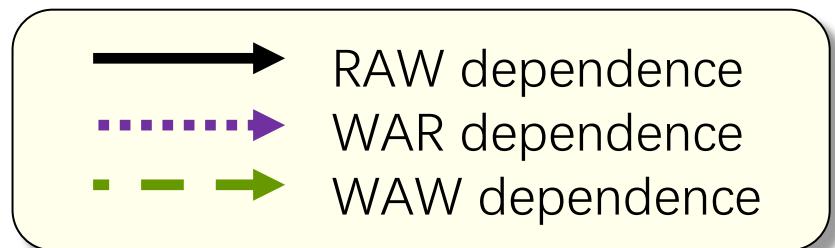
Stall eliminated.

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	1
6	mulsd %xmm1, %xmm0	3



Simplifying assumptions

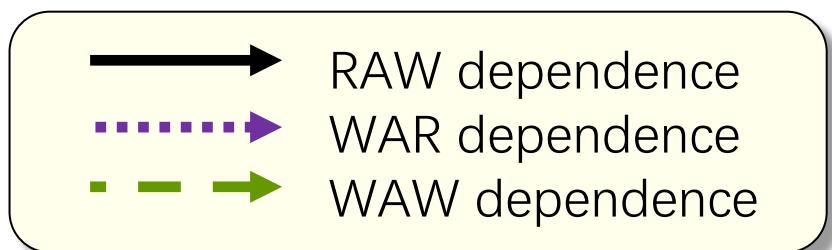
- The issue stage issues 1 operation per cycle.
- The processor has plenty of functional units for all operations.
- We'll ignore the non-execute stages of the pipeline.
- Latencies are chosen to simplify the example.

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	1
6	mulsd %xmm1, %xmm0	3



Simplifying assumptions

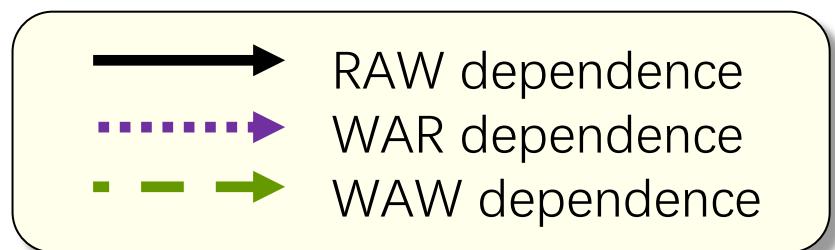
- The issue stage issues 1 operation per cycle.
- The processor has plenty of functional units for all operations.
- We'll ignore the non-execute stages of the pipeline.
- Latencies are chosen to simplify the example.

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	1
6	mulsd %xmm1, %xmm0	3



Simplifying assumptions

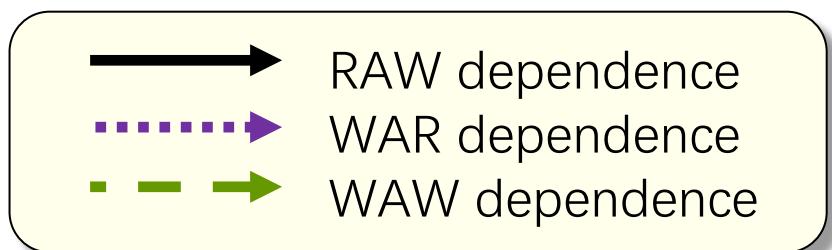
- The issue stage issues 1 operation per cycle.
- The processor has plenty of functional units for all operations.
- We'll ignore the non-execute stages of the pipeline.
- Latencies are chosen to simplify the example.

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	1
6	mulsd %xmm1, %xmm0	3

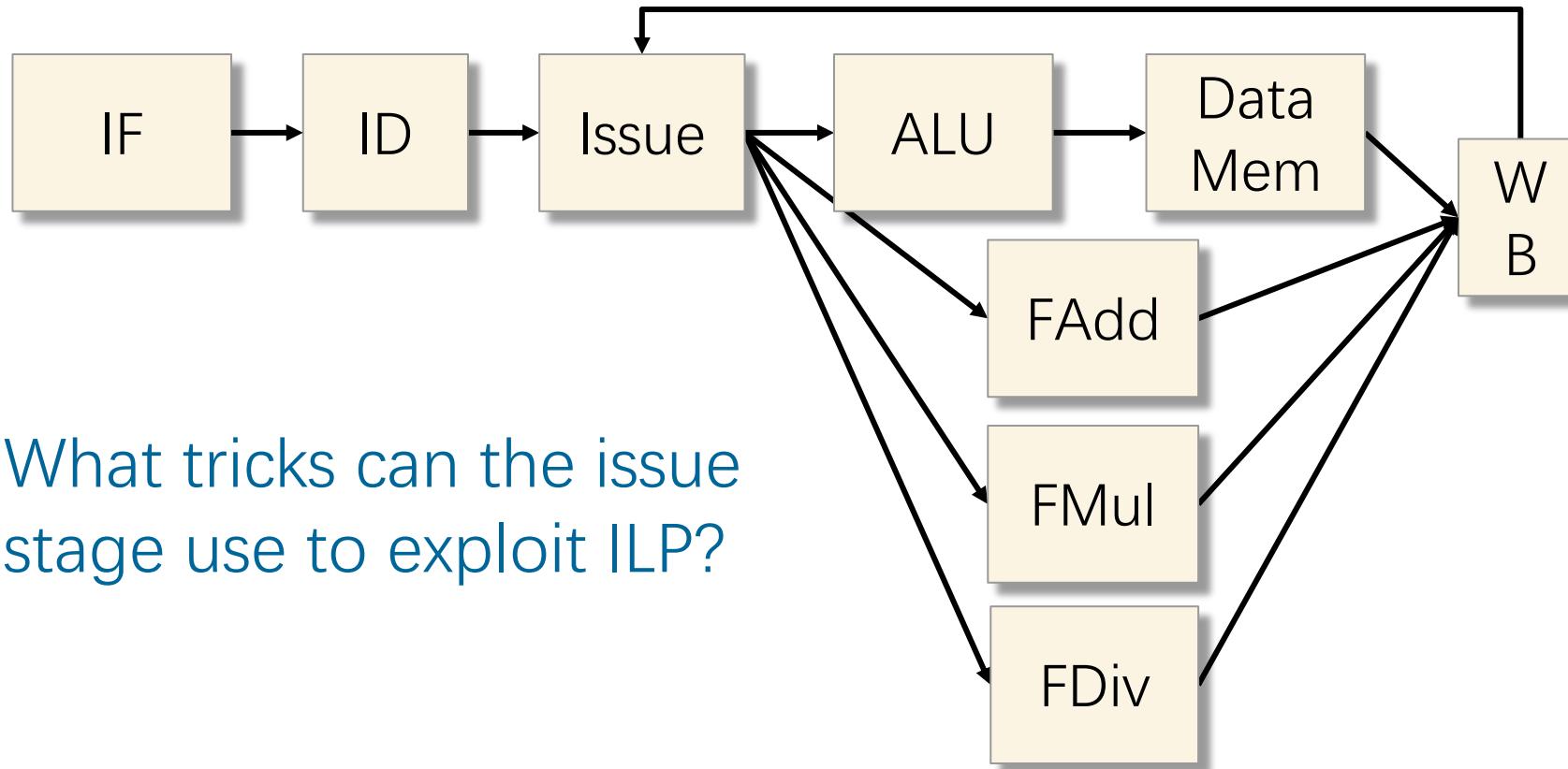


Simplifying assumptions

- The issue stage issues 1 operation per cycle.
- The processor has plenty of functional units for all operations.
- We'll ignore the non-execute stages of the pipeline.
- Latencies are chosen to simplify the example.

Block Diagram of a Superscalar Pipeline

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



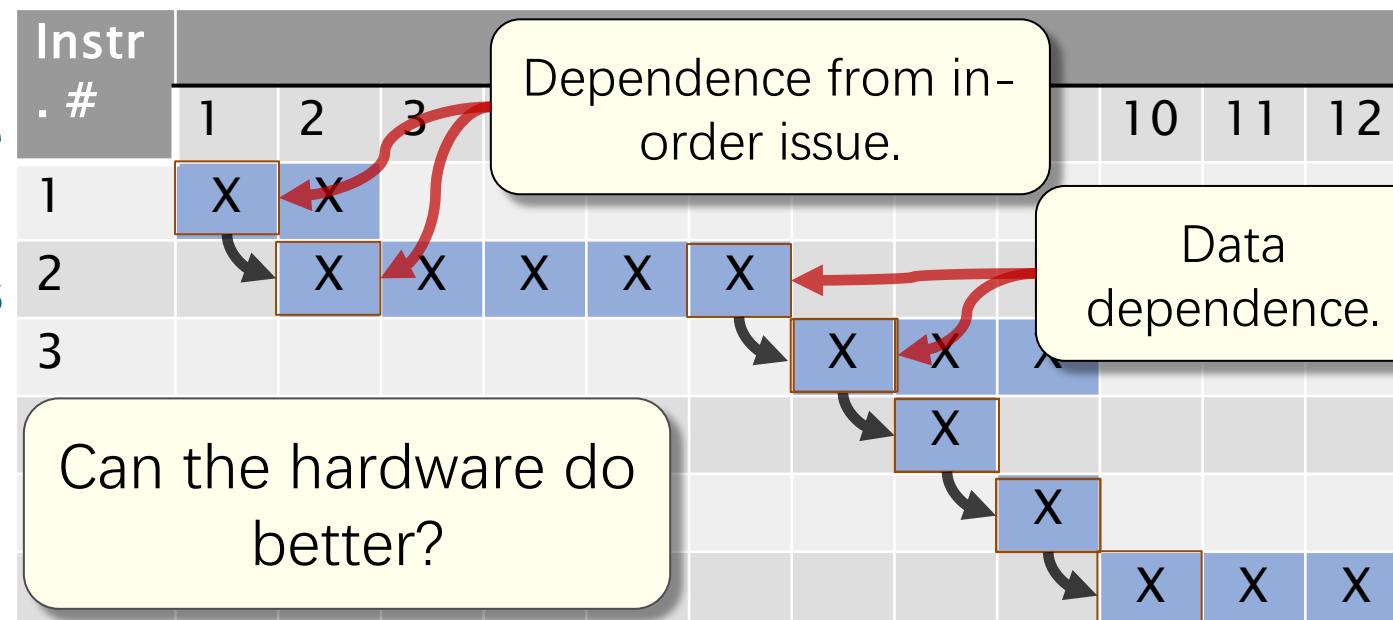
What tricks can the issue stage use to exploit ILP?

Example: In-Order Issue

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

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

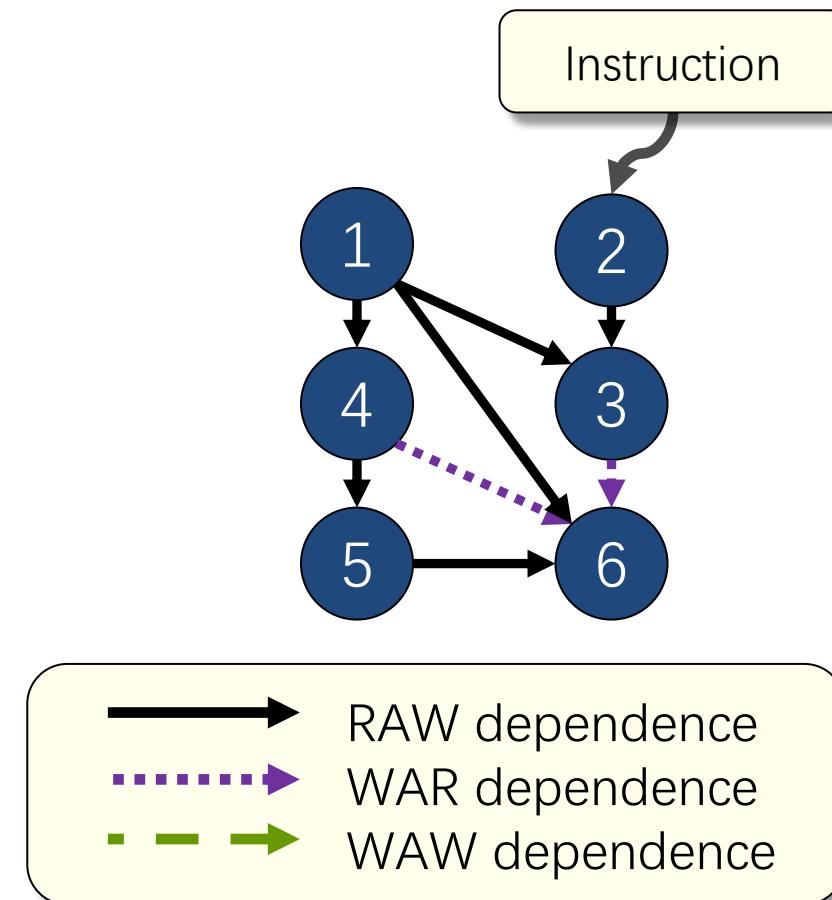


Data-Flow Graphs

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

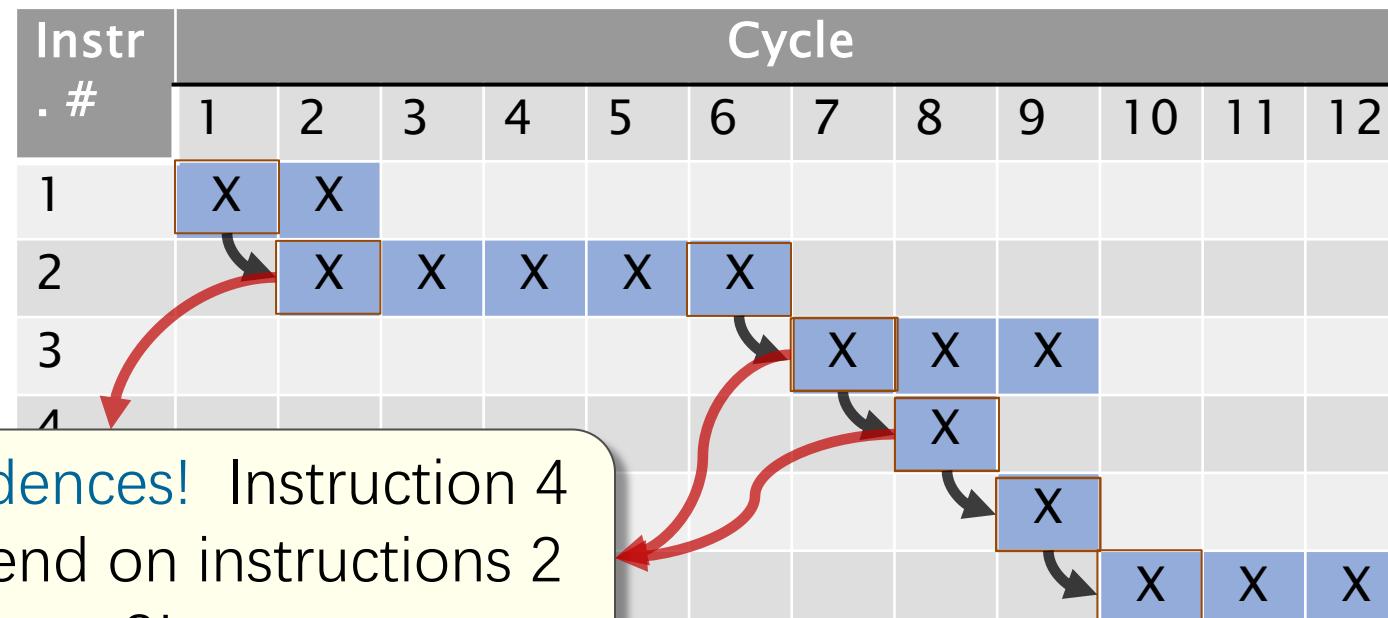


In-Order Issue, Revisited

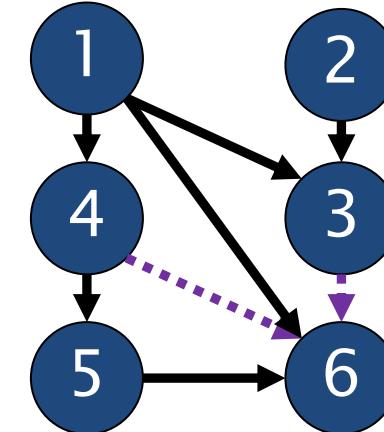
Instruction stream

#	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



Data-flow graph



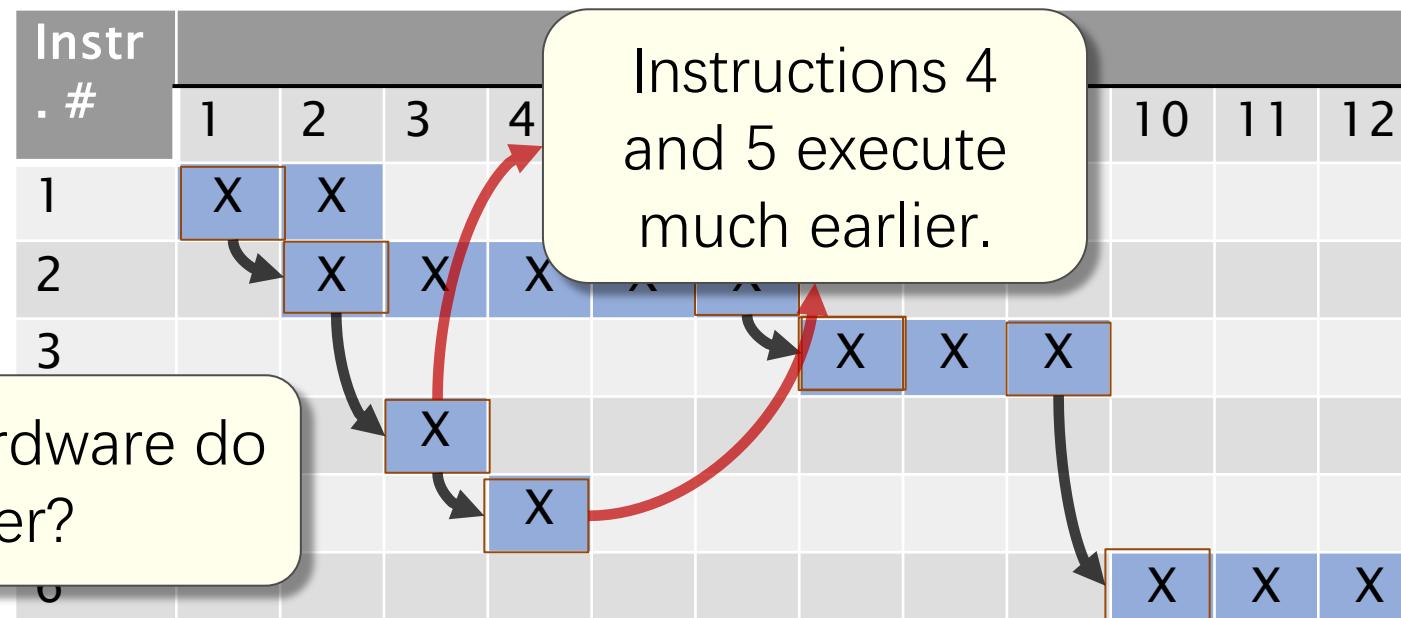
Example: Out-of-Order Execution

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

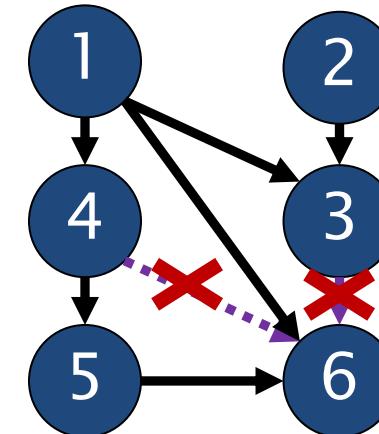
Instruction stream

#	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

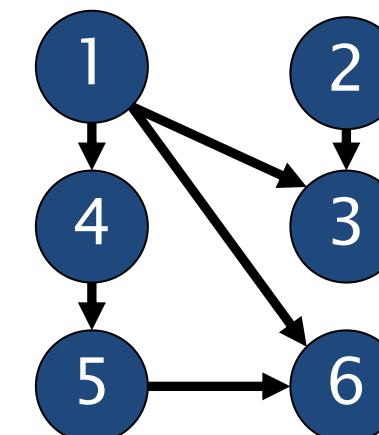
Idea: If the name of the destination register could be changed, then the WAR dependencies could be eliminated.

Instruction 6 no longer depends on long latency operations!

Data-flow graph



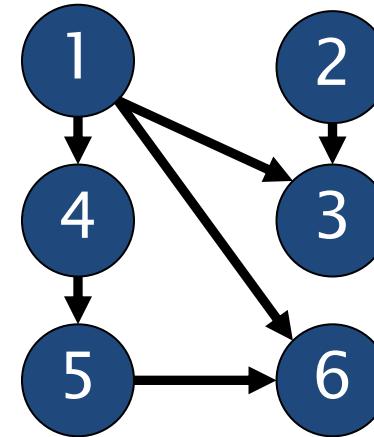
New data-flow graph



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	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311	1312	1313	1314	1315	131

Removing Data Dependencies

- The processor mitigates the performance loss of data hazards using two techniques.
 - **Register renaming** removes WAR and WAW dependencies. (Also see optional slides.)
 - **Out-of-order execution** reduces the performance lost due to RAW dependencies.

Summary: Dealing with Hazards

- A 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. 2019.
<https://software.intel.com/en-us/articles/intel-sdm>
- Agner Fog. The Microarchitecture of Intel and AMD CPUs. 2019. <http://www.agner.org/optimize/microarchitecture.pdf>
- Intel Corporation. Intel Intrinsics Guide. 2019.
<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

OPTIONAL SLIDES REGISTER RENAMING



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 →

#	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.

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					
t4					
t5					
t6					
t7					

Dynamic Reordering and Renaming: Example

Instruction
stream

`%rip` →

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

Step: Decode
instruction 3.

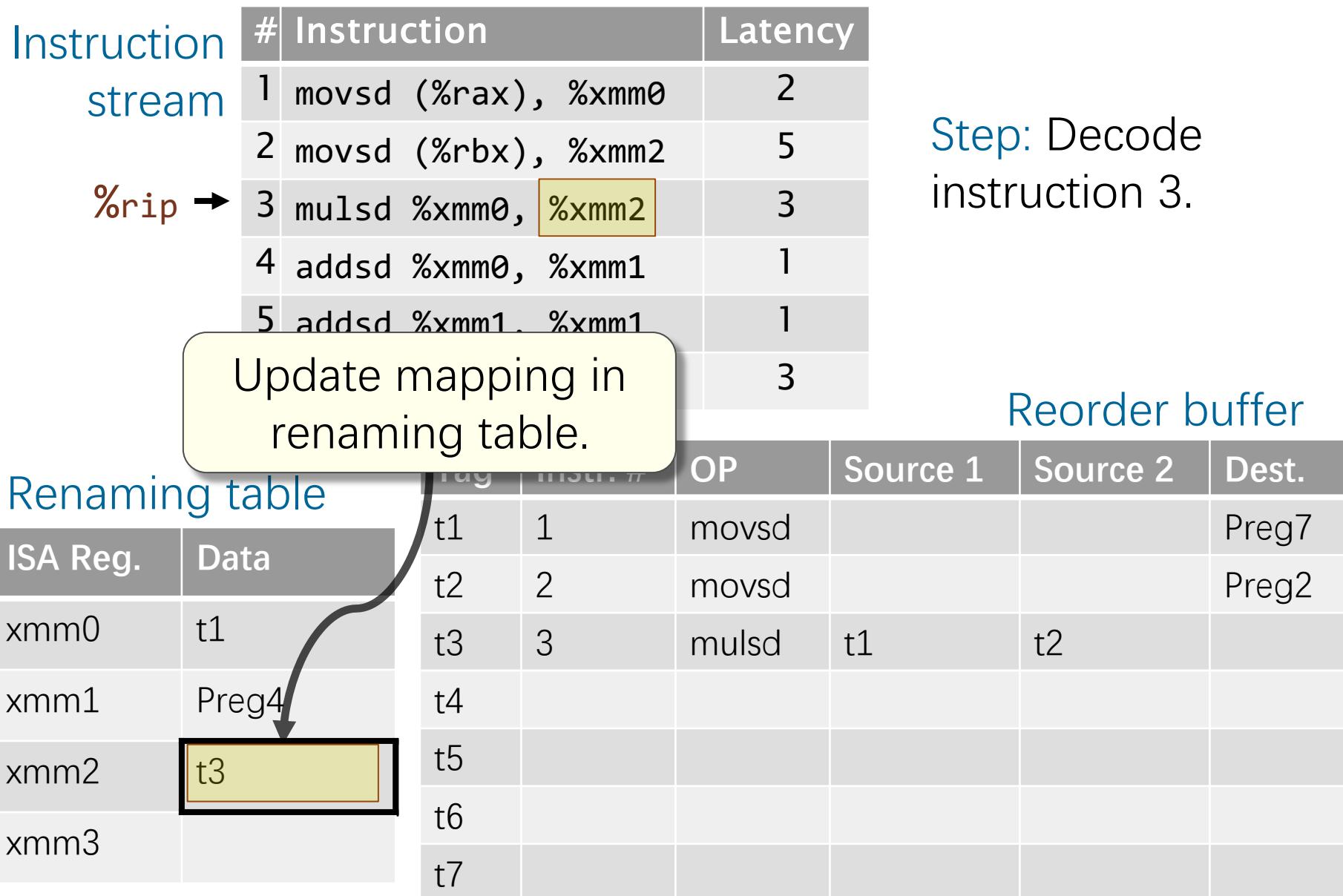
Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	Preg4
xmm2	t2
xmm3	

Reorder buffer

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

Dynamic Reordering and Renaming: Example



Dynamic Reordering and Renaming: Example

Instruction stream

`%rip` →

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

Update mapping in renaming table.

Renaming table

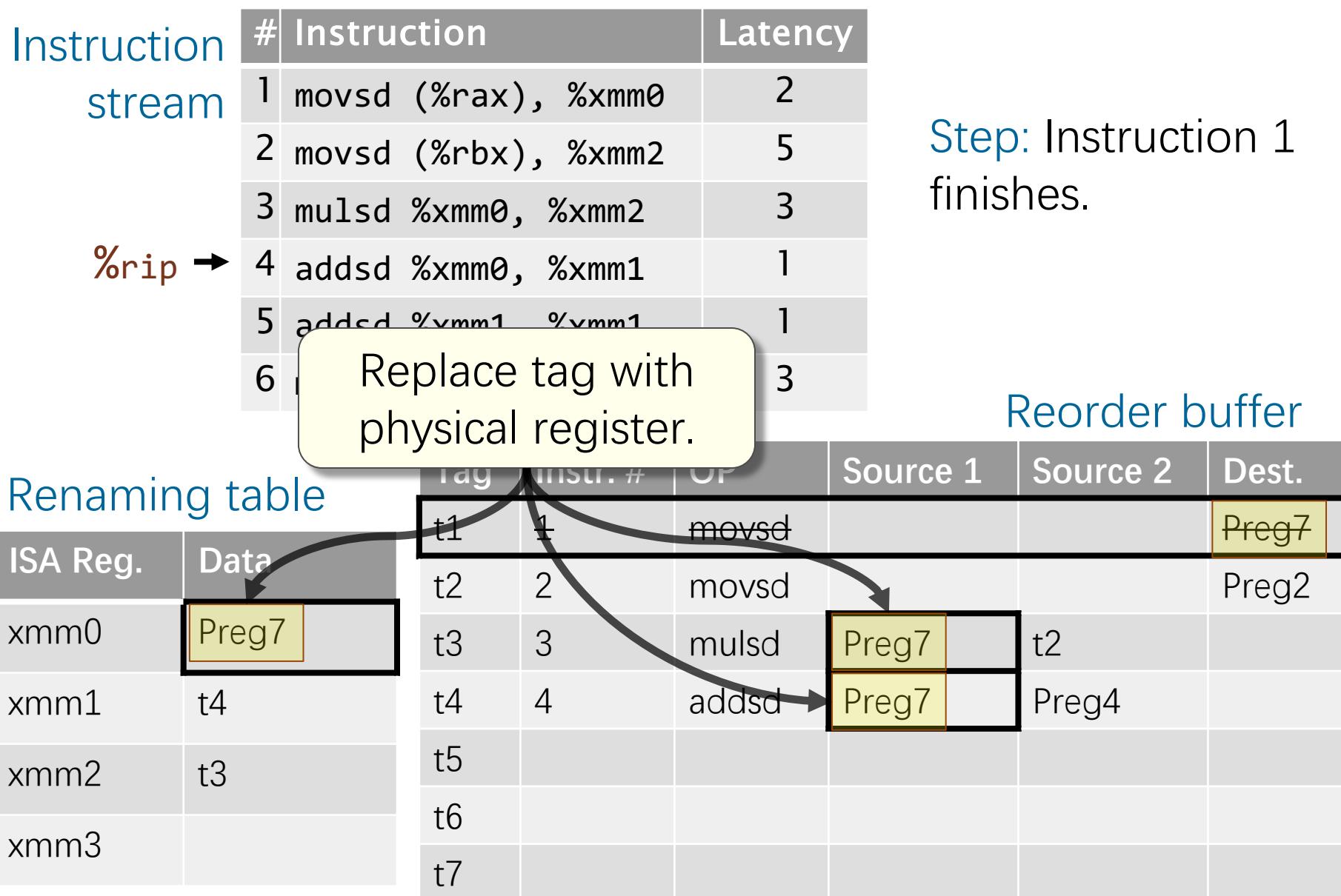
ISA Reg.	Data
xmm0	t1
xmm1	t4
xmm2	t3
xmm3	

Step: Decode instruction 4.

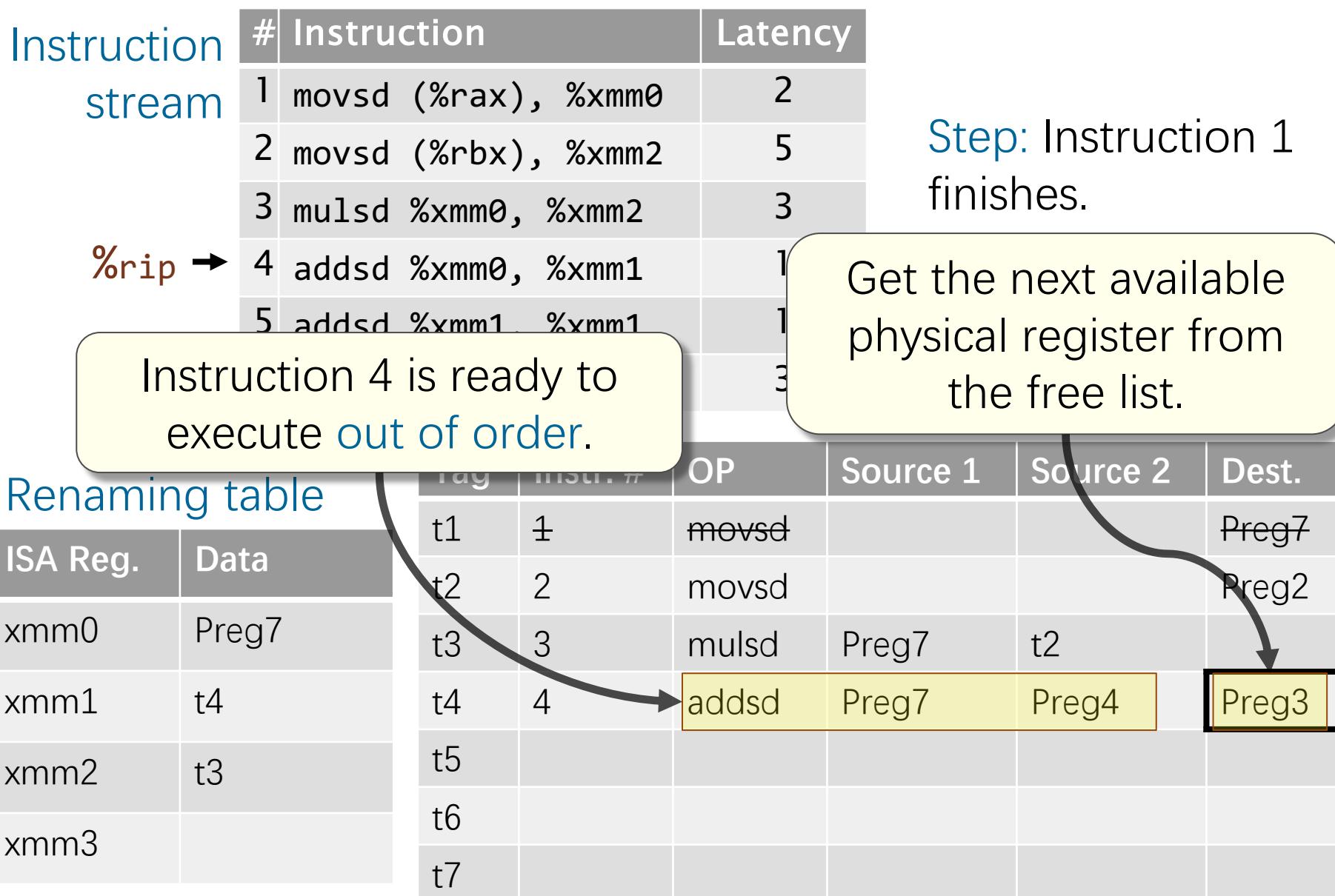
Reorder buffer

Tag	Inst. #	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



Dynamic Reordering and Renaming: Example



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	3

%rip →

Renaming table

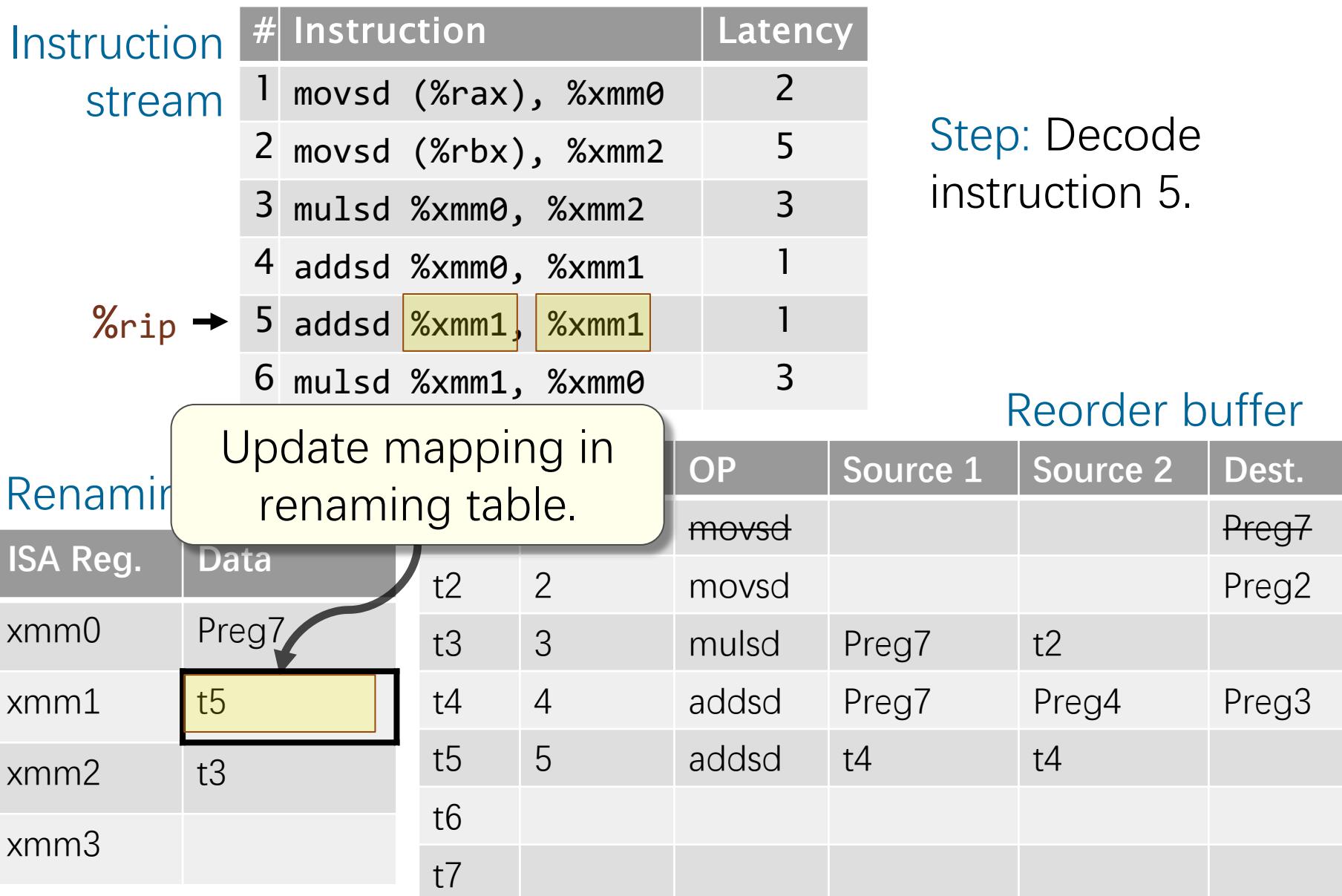
ISA Reg.	Data
xmm0	Preg7
xmm1	t4
xmm2	t3
xmm3	

Step: Decode instruction 5.

Reorder buffer

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



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	mov			
t2	2	mov			
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.

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

