Performance Engineering of Software Systems

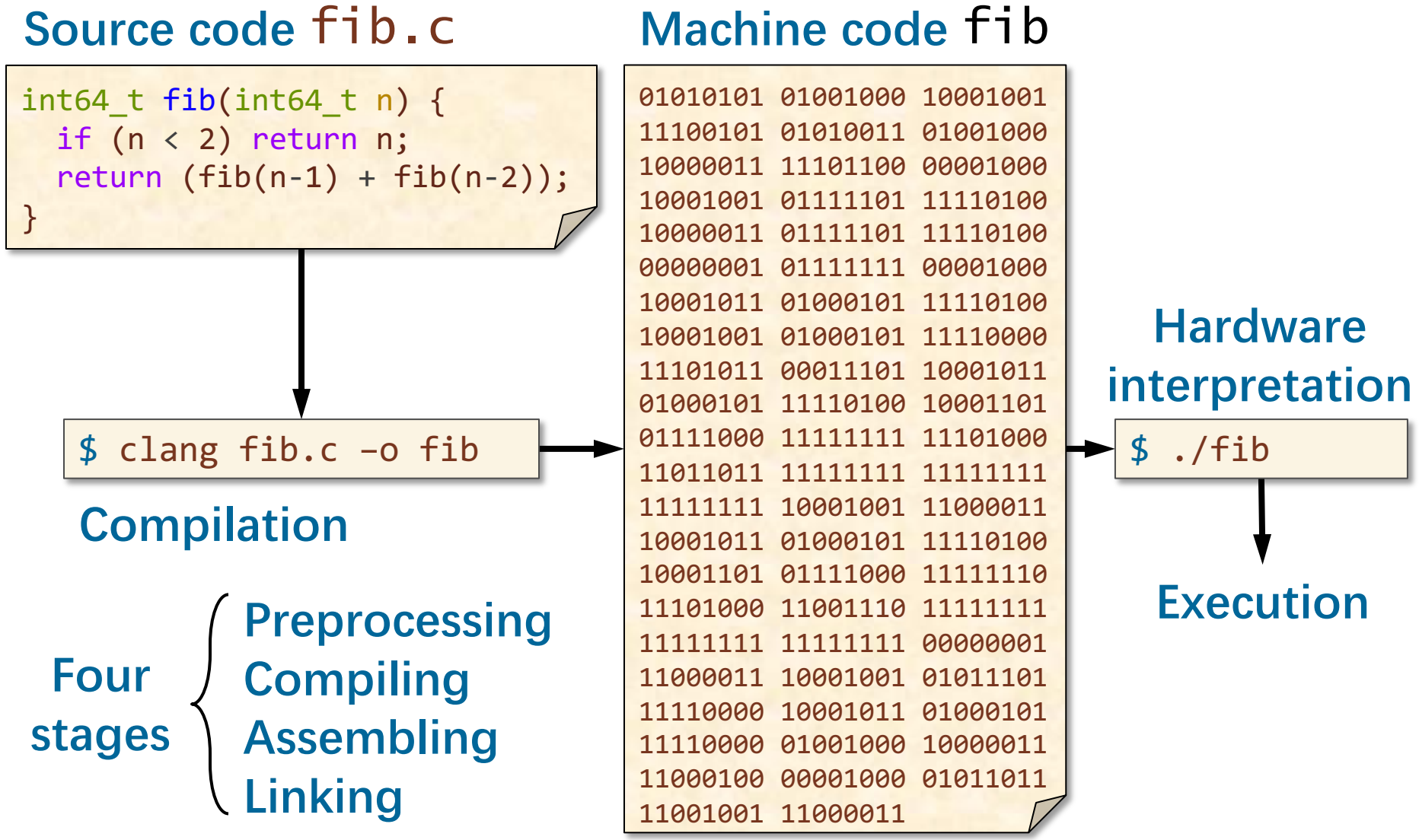
LECTURE 4

Assembly Language and Computer Architecture

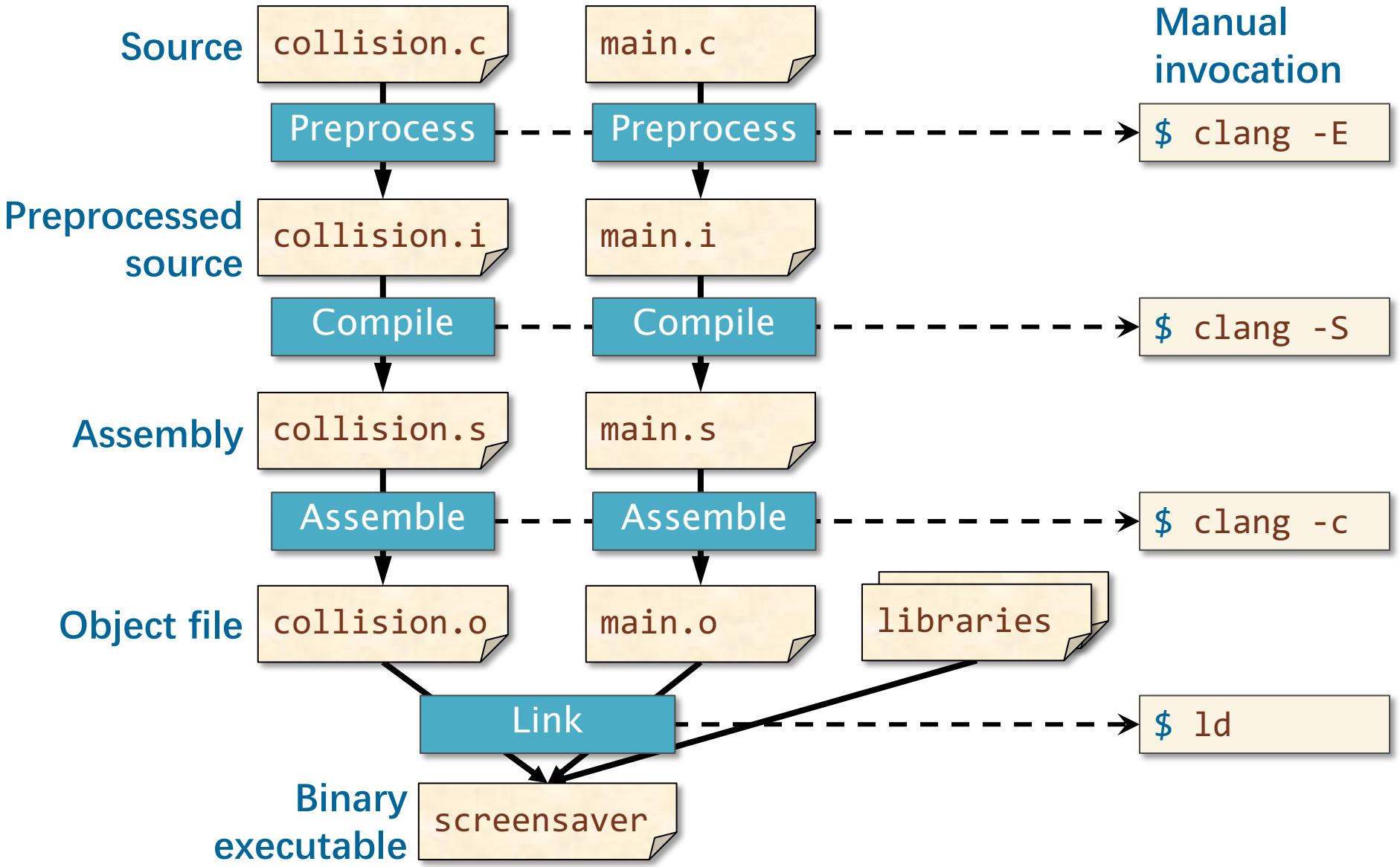
Srini Devadas

September 20, 2022

Source Code to 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:   ## @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`

```
            .globl      _fib
            .p2align    4, 0x90
_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
```

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

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

```
$ objdump -S fib
```

Source, machine, & assembly

```
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

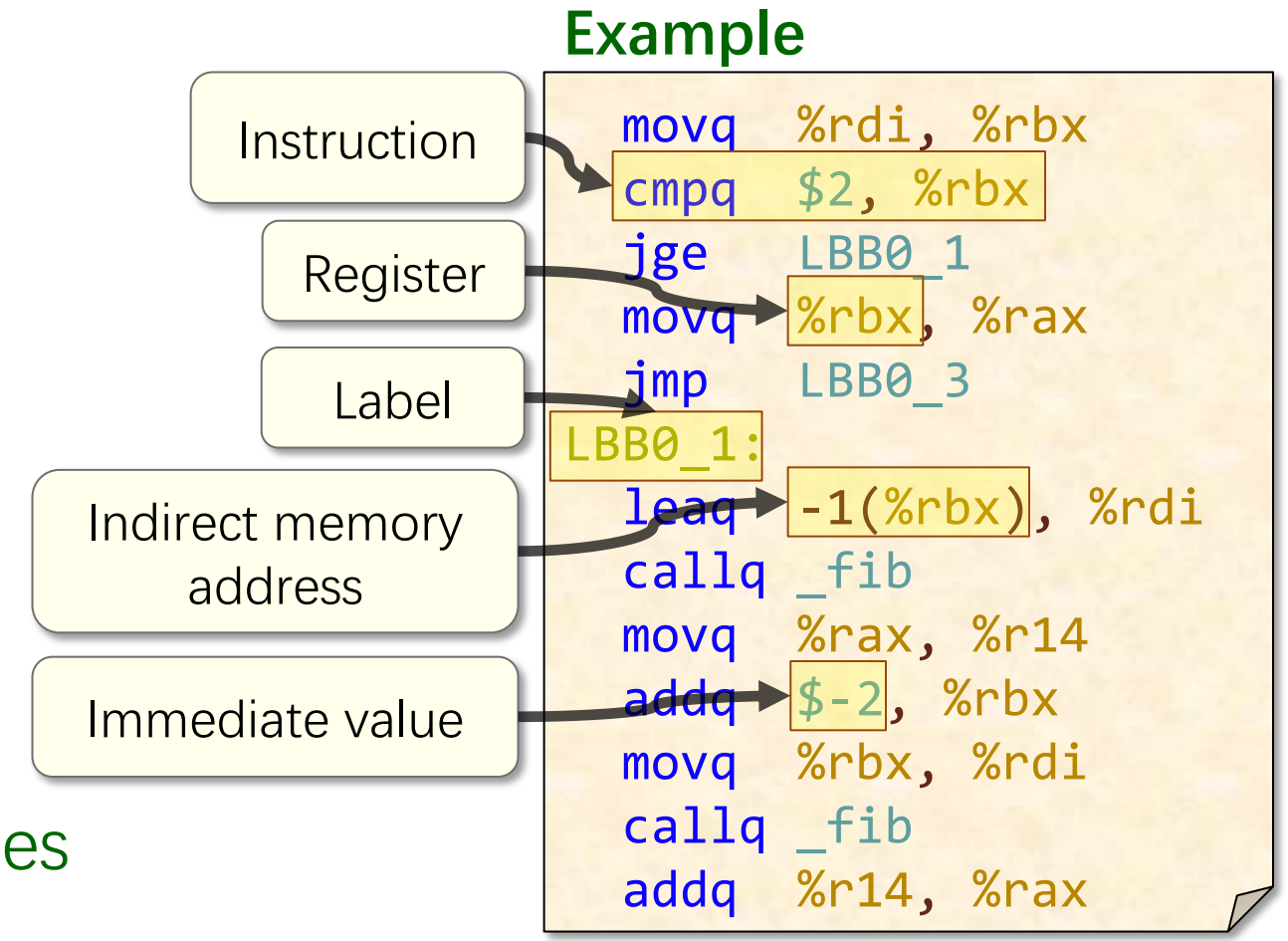
- x86-64 ISA Primer
  - Quick introduction
  - Opcodes
  - Operands
- Overview of Modern Computer Architecture
  - Quick Introduction
  - Dealing with Structural Hazards
  - Dealing with Control Hazards
  - Dealing with Data Hazards

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



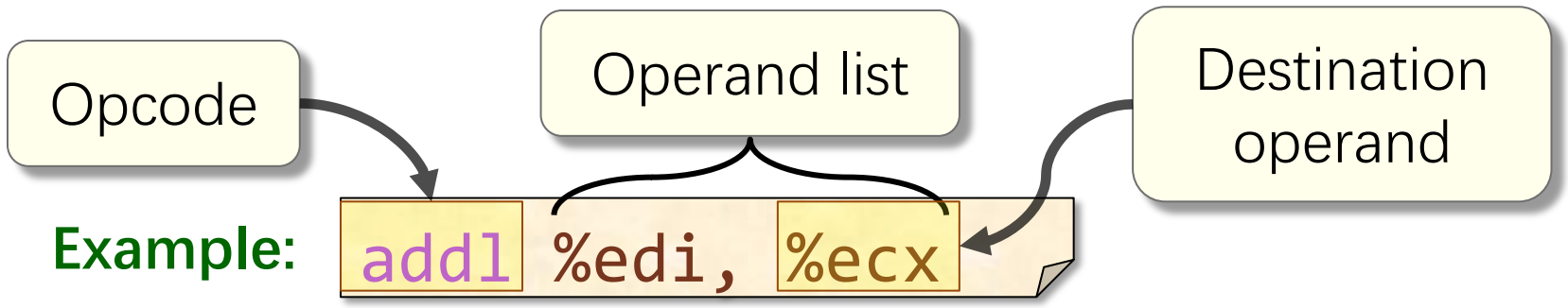
Common x86-64 Registers

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 <op> A → B	Intel Syntax A ← A <op> 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`, and 6.106 lectures

Used by Intel documentation

x86-64 ISA Primer: Opcodes

Common x86-64 Opcodes

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

- Note: the subtraction operation "`subq %rax, %rbx`" computes `%rbx = %rbx - %rax`

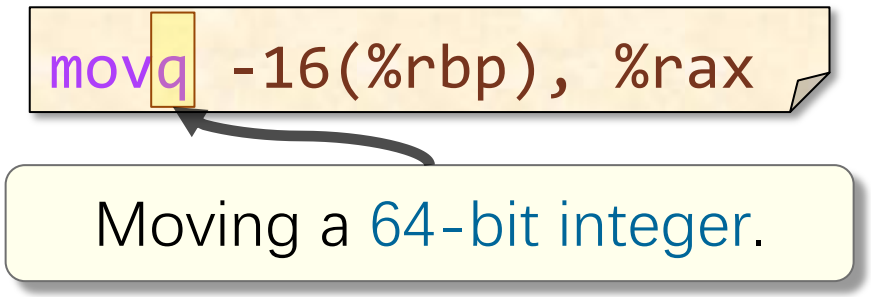
Opcode Suffixes

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



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		*Reserved*
2	PF	Parity
3		*Reserved*
4	AF	Adjust
5		*Reserved*
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12-63		*System flags or reserved*

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

- 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 $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

Examples

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

- **Register indexed**: The address is a constant offset of the value in the specified register

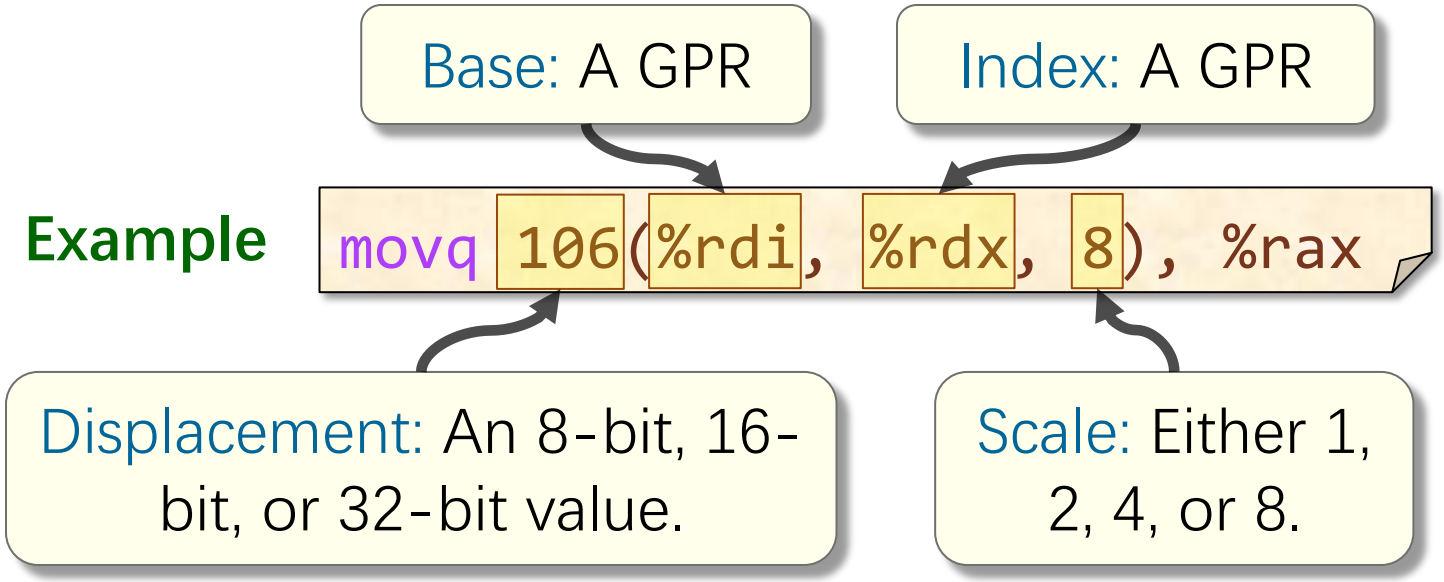
```
movq 106(%rax), %rdi
```

- **Instruction-pointer relative**: The address is indexed relative to `%rip`

```
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.

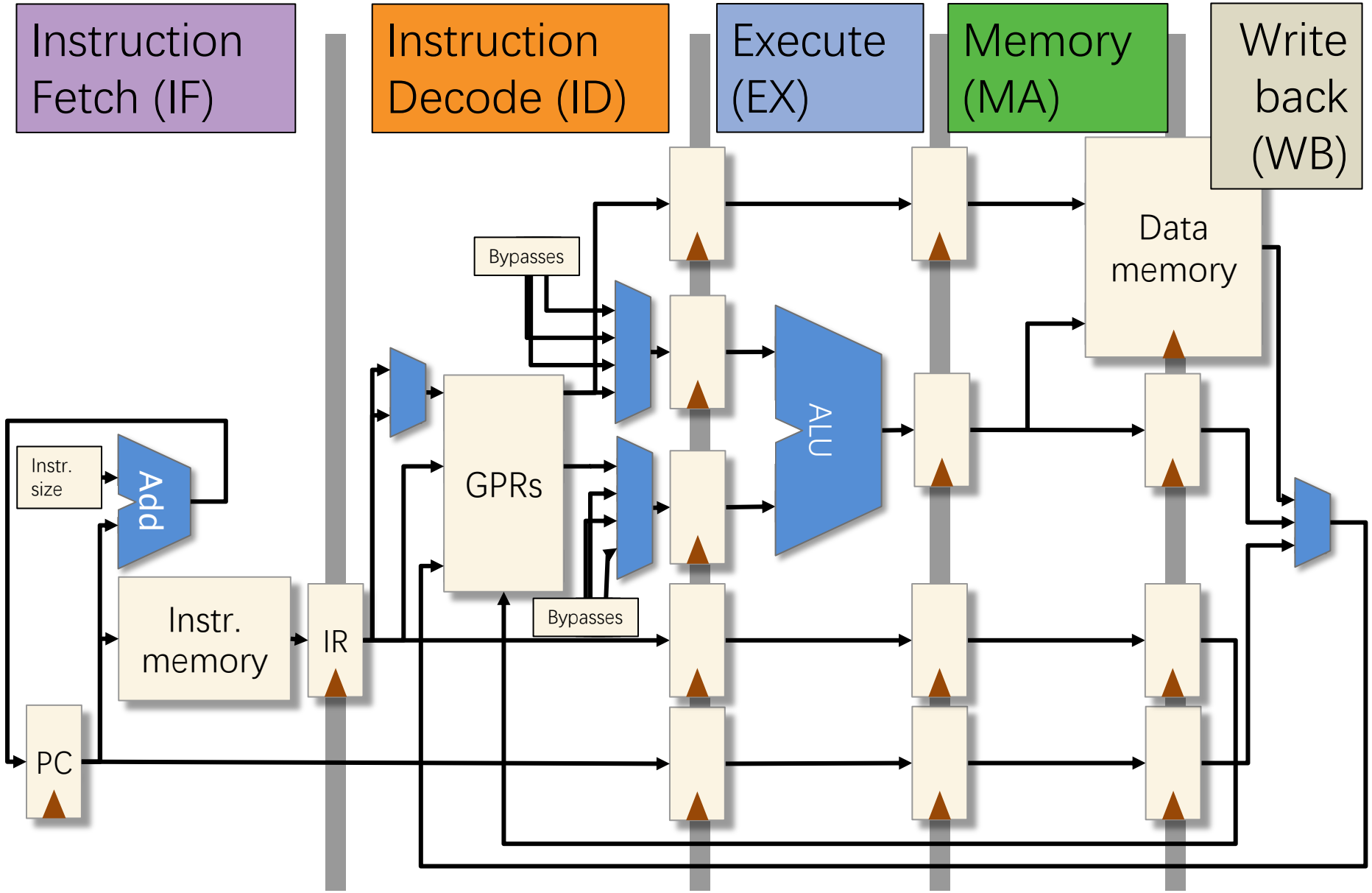


This mode refers to the address: Base + Index*Scale + Displacement

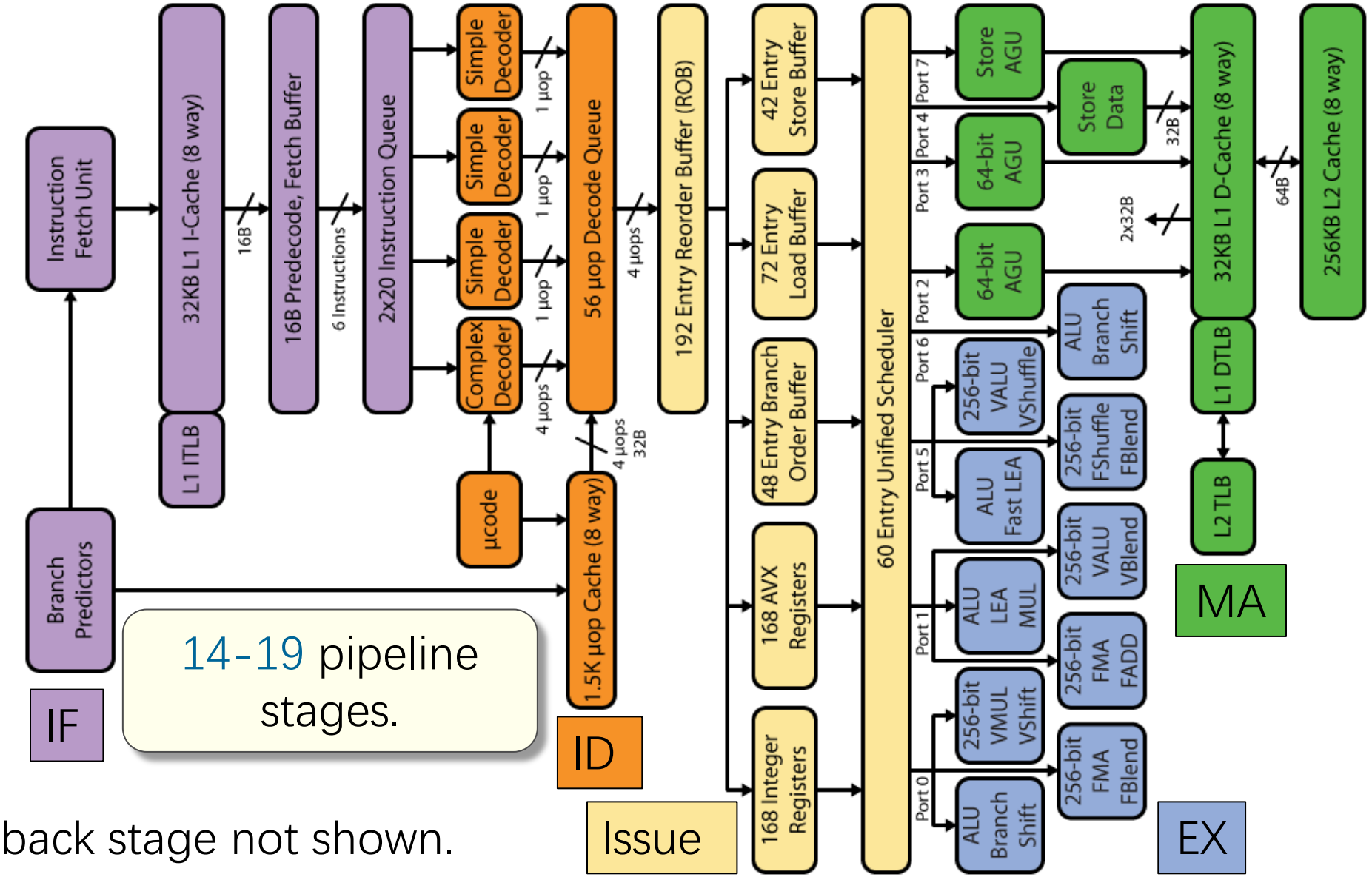
If unspecified, Index and Displacement default to 0, and Scale defaults to 1.

Overview of Computer Architecture

A Simple 5-Stage Processor



Intel Haswell Microarchitecture



Write-back stage not shown.

Courtesy of realworldtech.

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. #	Cycle 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.

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

Pipeline stalls

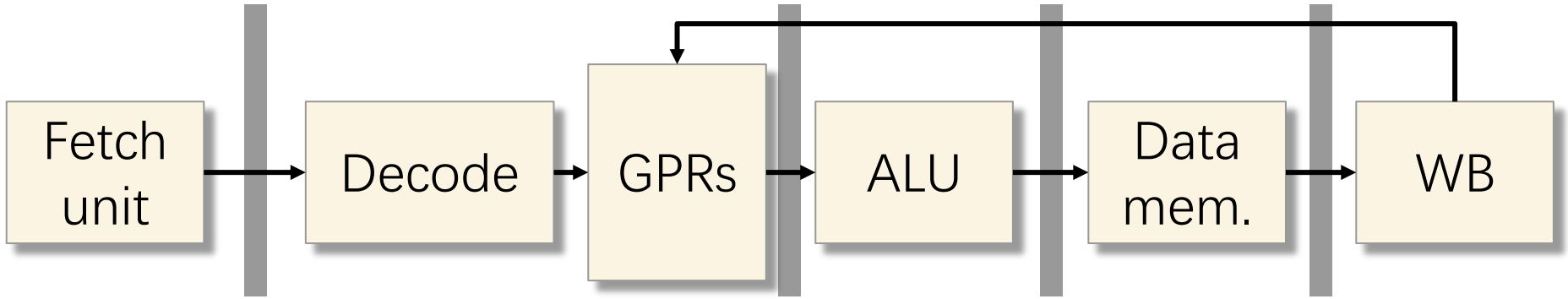
Sources of Pipeline Stalls

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

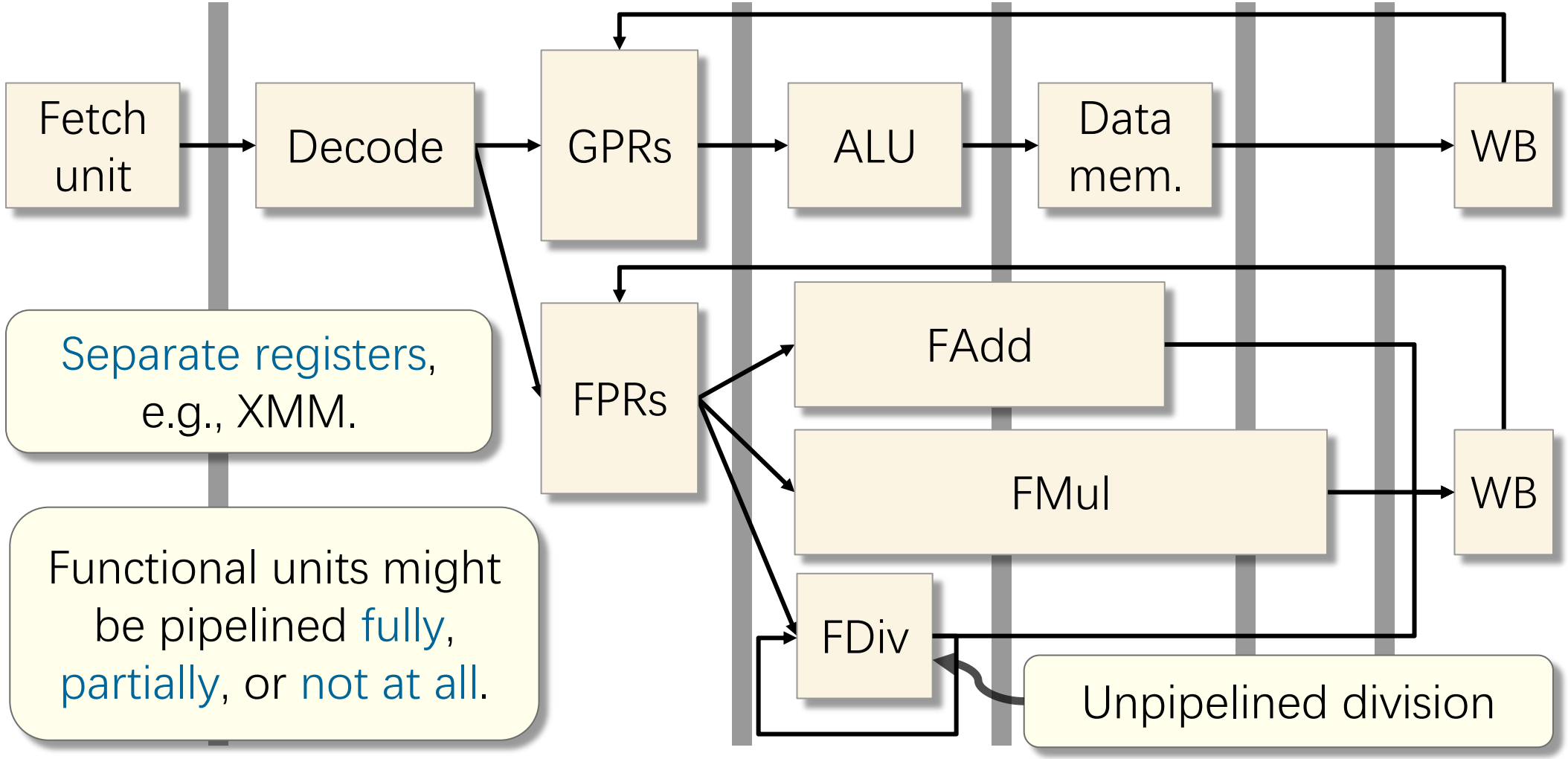
Some arithmetic operations are **complex** to implement in hardware and have **long latencies**.

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	`vfmass, vfmasd`	5

How can hardware accommodate these complex operations?

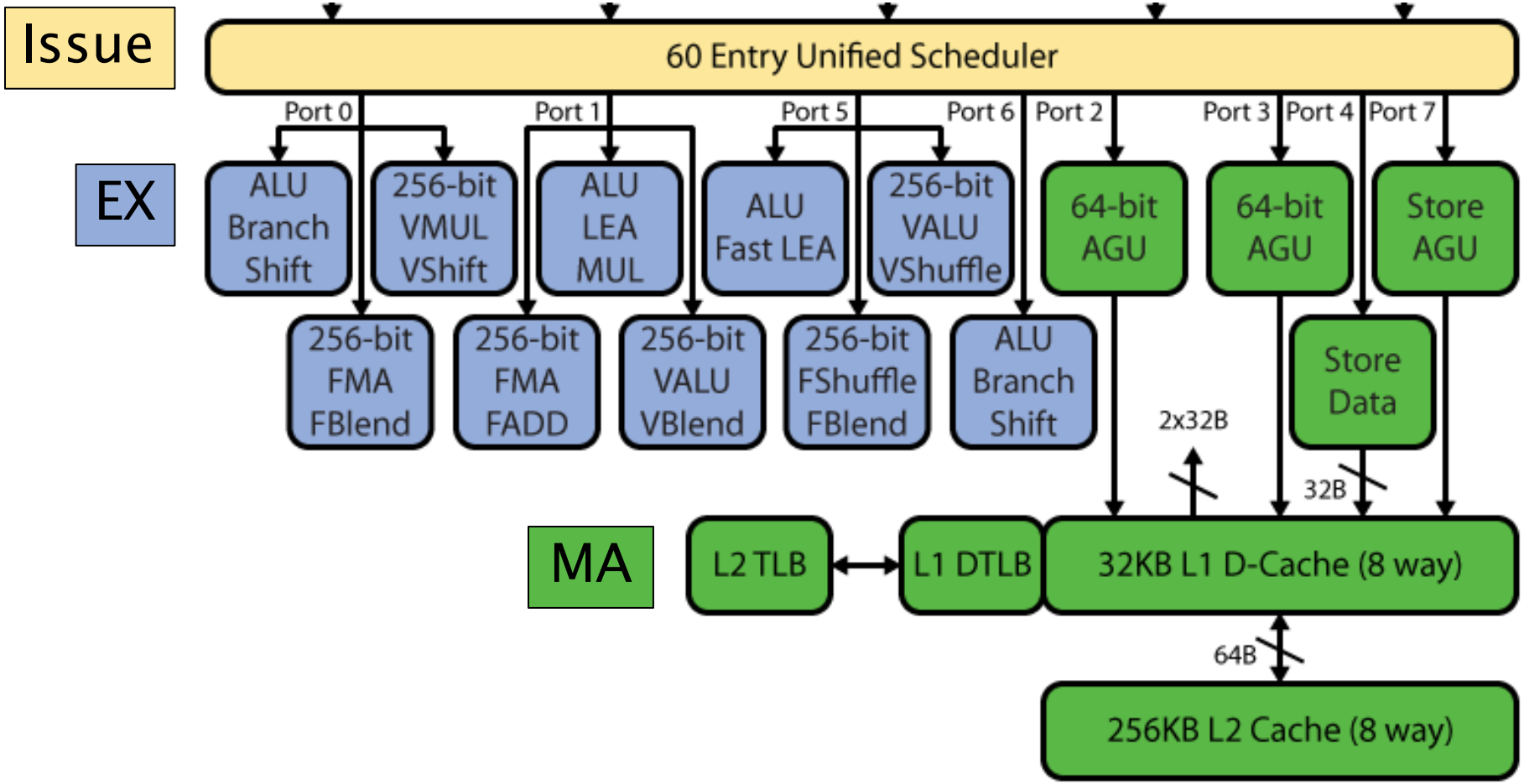
Solution: Complex Pipelining

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



Intel Haswell Functional Units

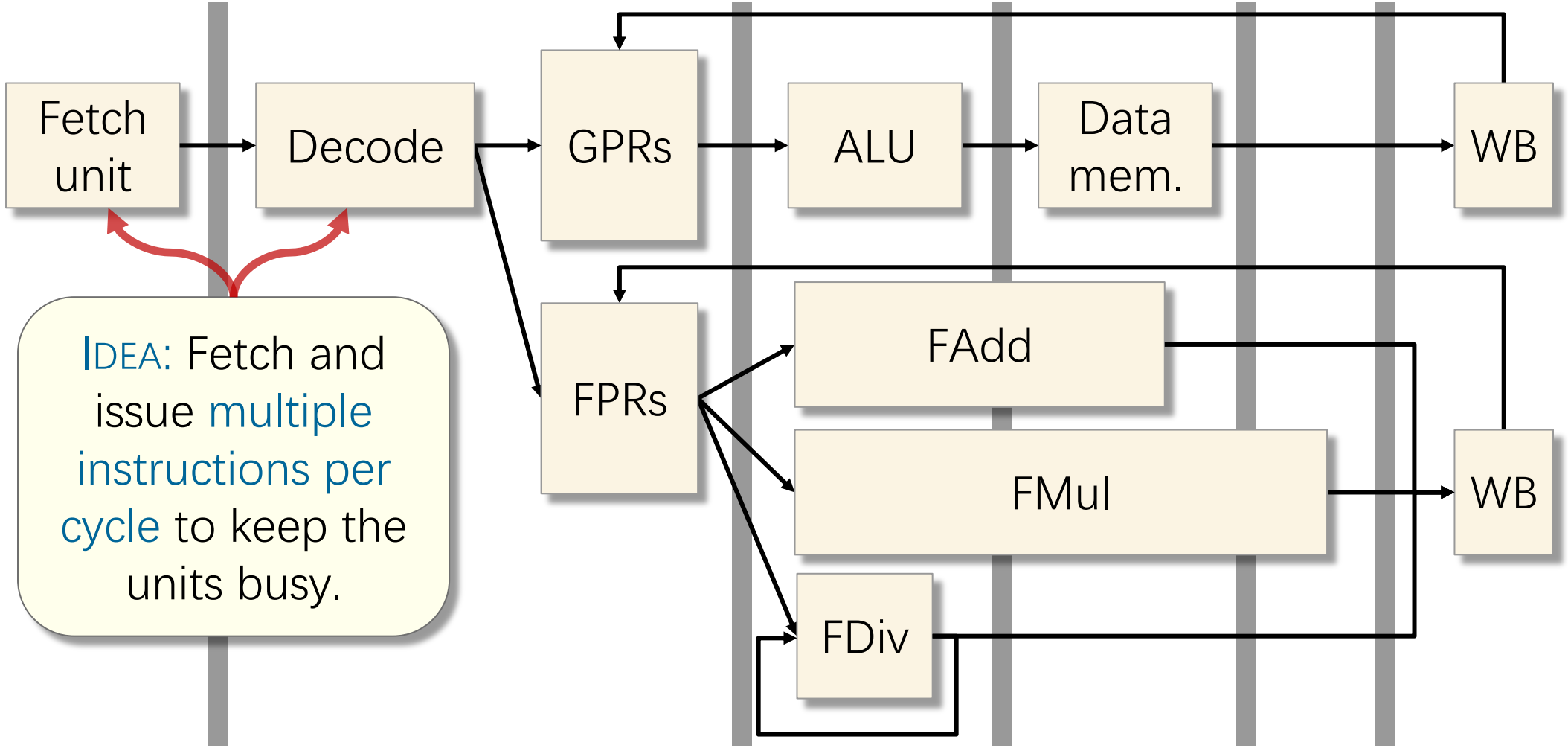
Haswell uses a suite of integer, vector, and floating-point functional units, distributed among 8 different ports.



Courtesy of realworldtech.

From Complex to Superscalar

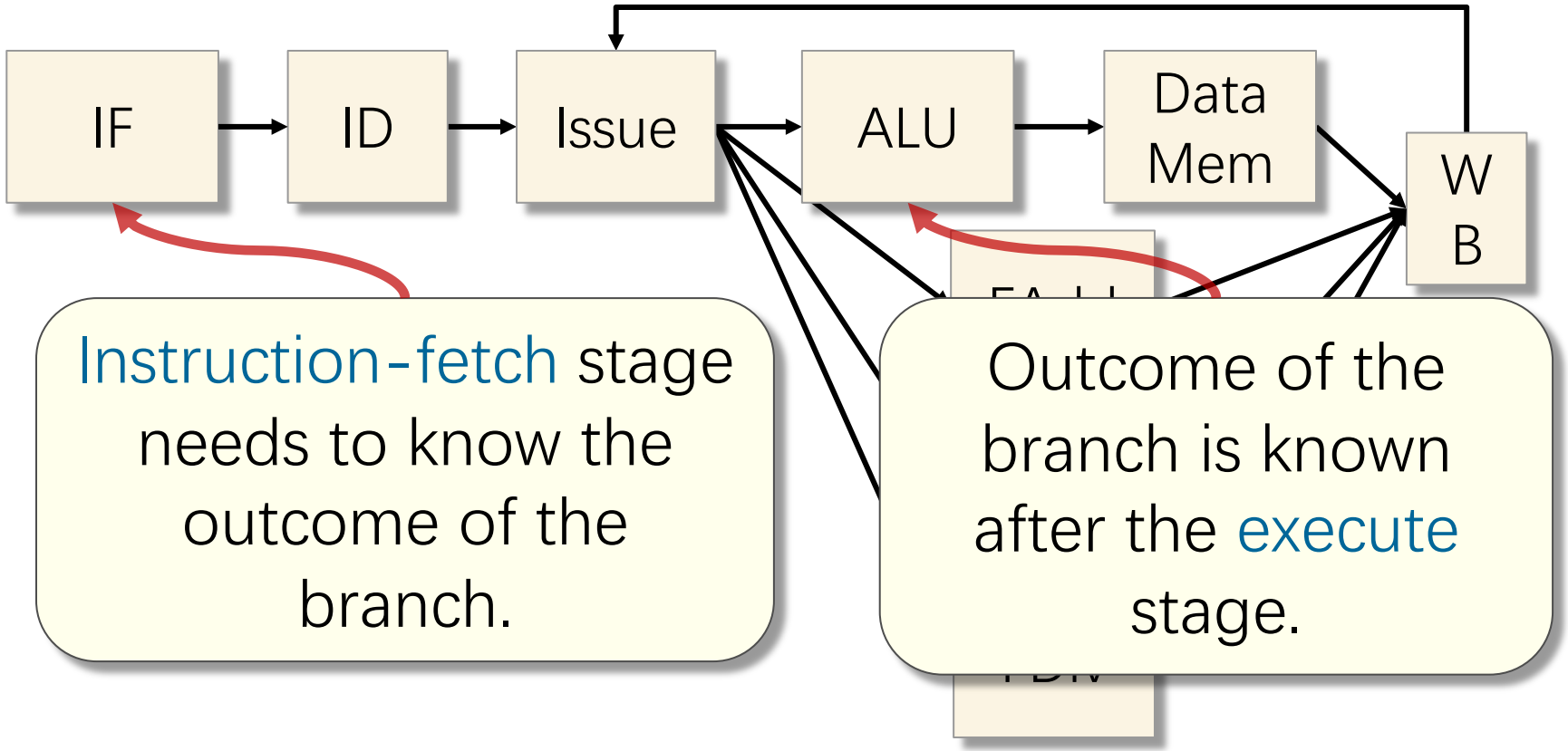
Given these additional functional units, how can the processor further exploit ILP?



Overview of Computer Architecture: Dealing with Control Hazards

Control Hazards

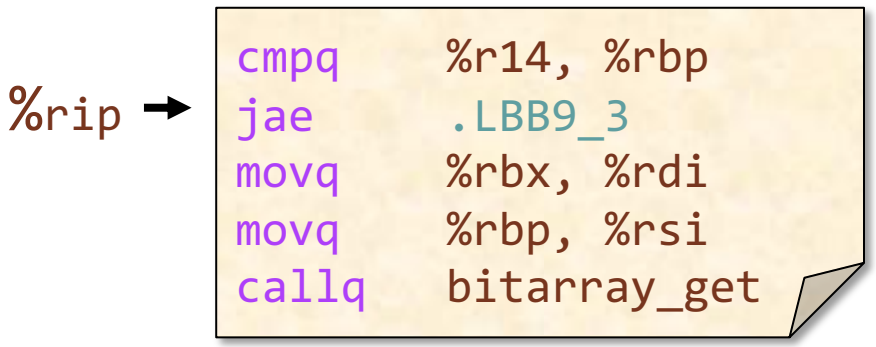
What happens if the processor encounters a conditional jump, a.k.a., a branch?



Speculative Execution

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

Example



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

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

%rip →

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

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

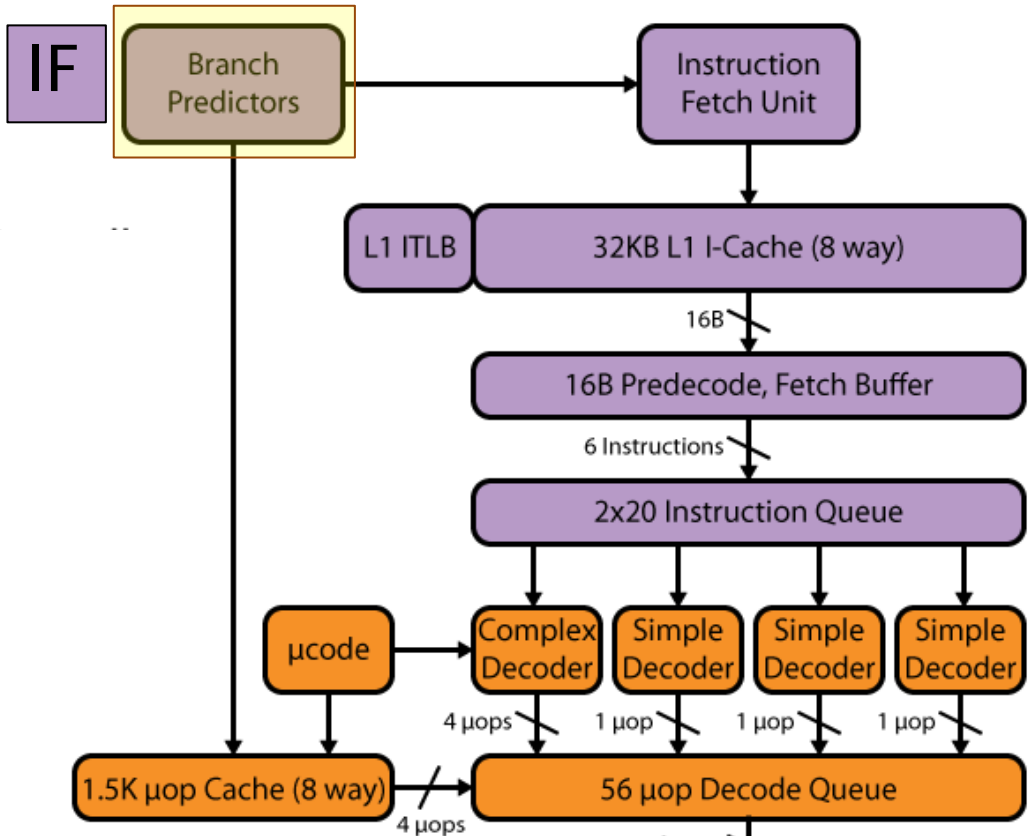
Problem: The effect on throughput of undoing computation is just like stalling.

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.

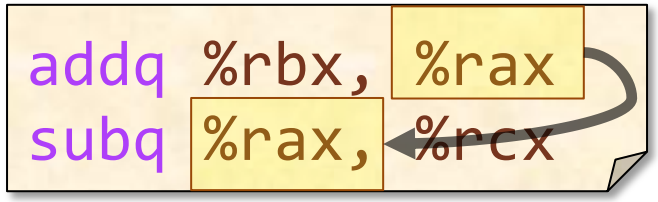
Overview of Computer Architecture: Dealing with Data Hazards

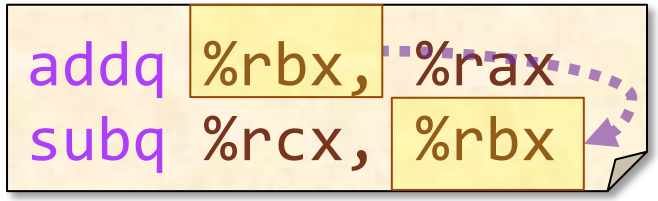
Sources of Data Hazards

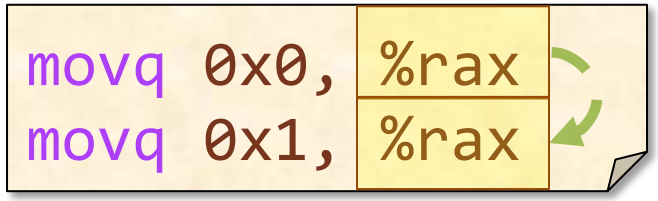
An instruction i can create a data hazard with a later instruction j due to a dependence between i and j.

- 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



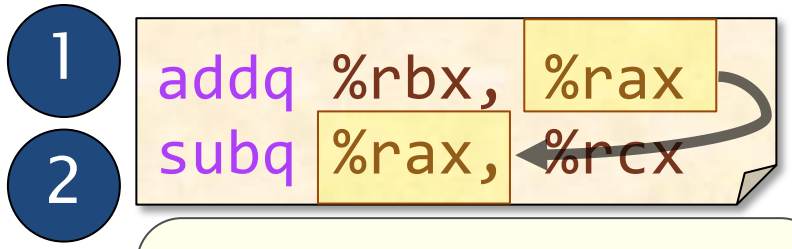




Bypassing

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

Example



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

Without bypassing

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

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			

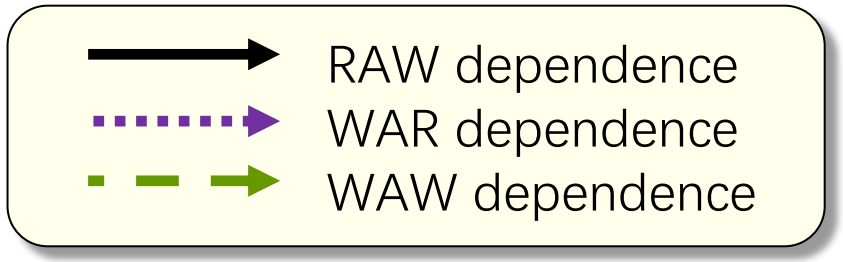
Stall eliminated.

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

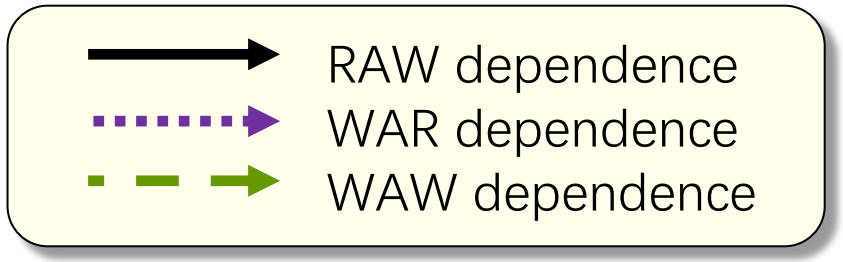
- The issue stage issues 1 operation per cycle.
- The processor has plenty of functional units for all operations.
- We'll ignore the non-execute stages of the pipeline.
- Latencies are chosen to simplify the example.

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

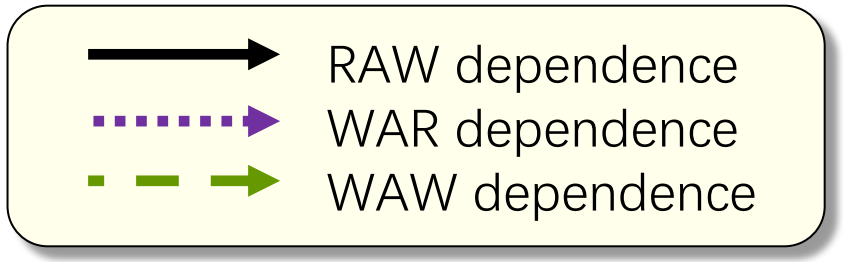
- The issue stage issues 1 operation per cycle.
- The processor has plenty of functional units for all operations.
- We'll ignore the non-execute stages of the pipeline.
- Latencies are chosen to simplify the example.

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

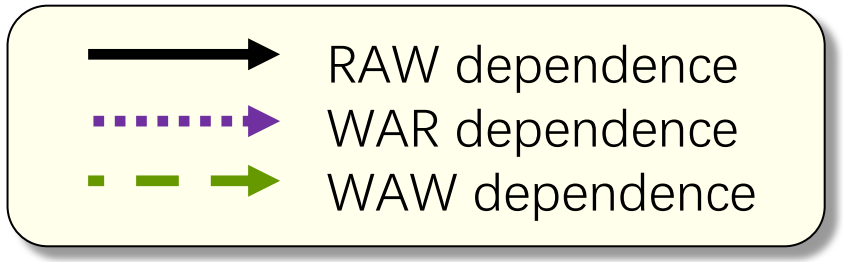
- The issue stage issues 1 operation per cycle.
- The processor has plenty of functional units for all operations.
- We'll ignore the non-execute stages of the pipeline.
- Latencies are chosen to simplify the example.

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

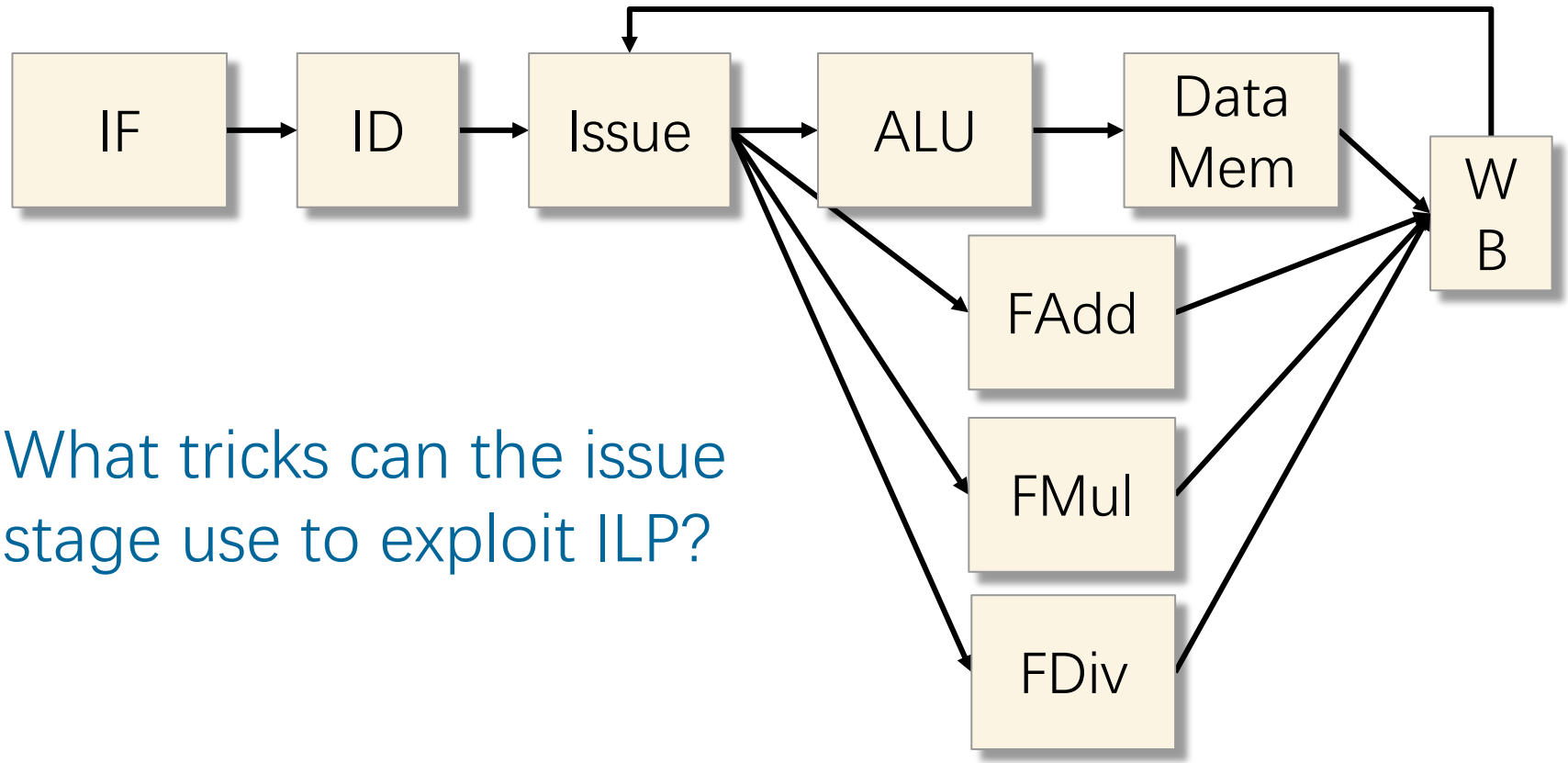


Simplifying assumptions

- 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?

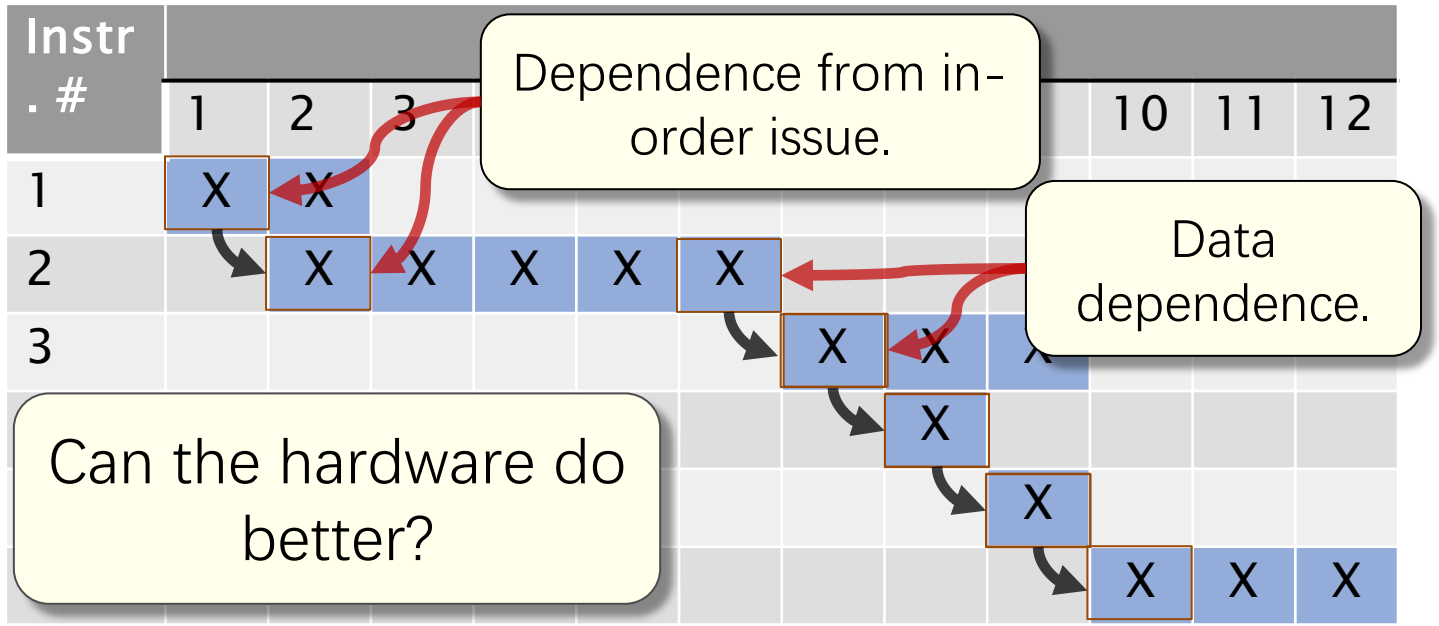
Example: In-Order Issue

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

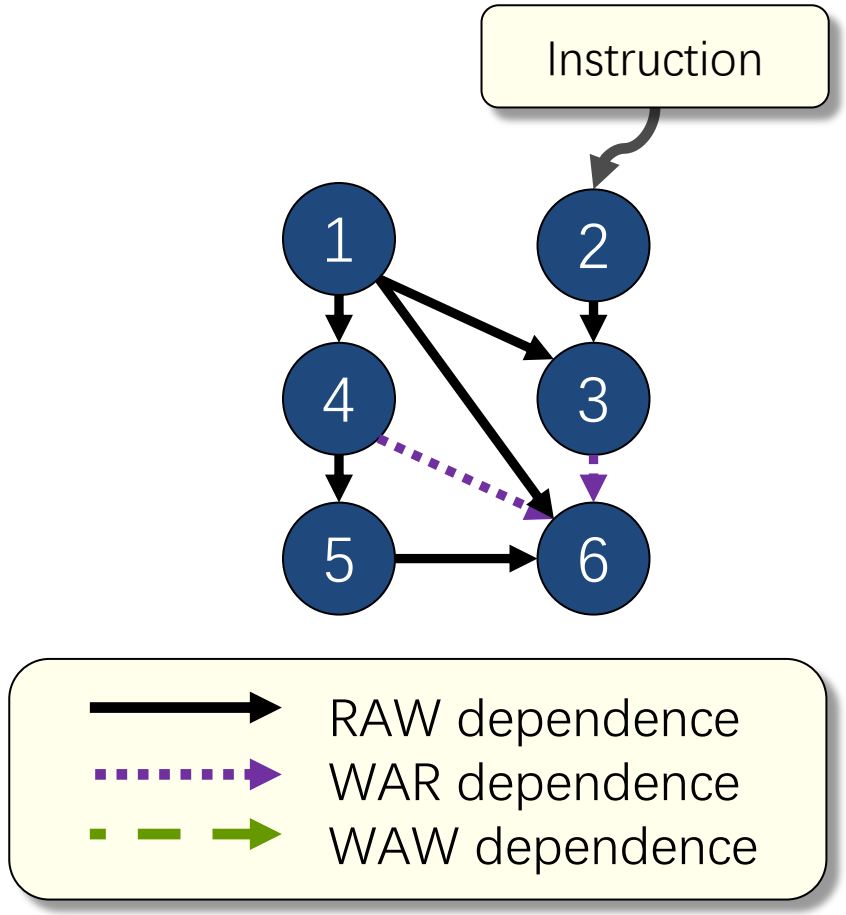


Data-Flow Graphs

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

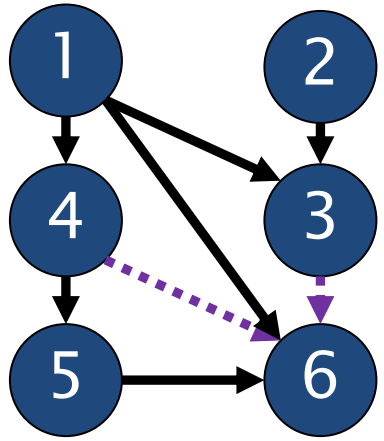


In-Order Issue, Revisited

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

Data-flow graph



Instruction timing for use of functional units

Instr. #	Cycle 1	2	3	4	5	6	7	8	9	10	11	12
1	X	X										
2		X	X	X	X	X						
3							X	X	X			
4								X				
									X			
										X	X	X

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

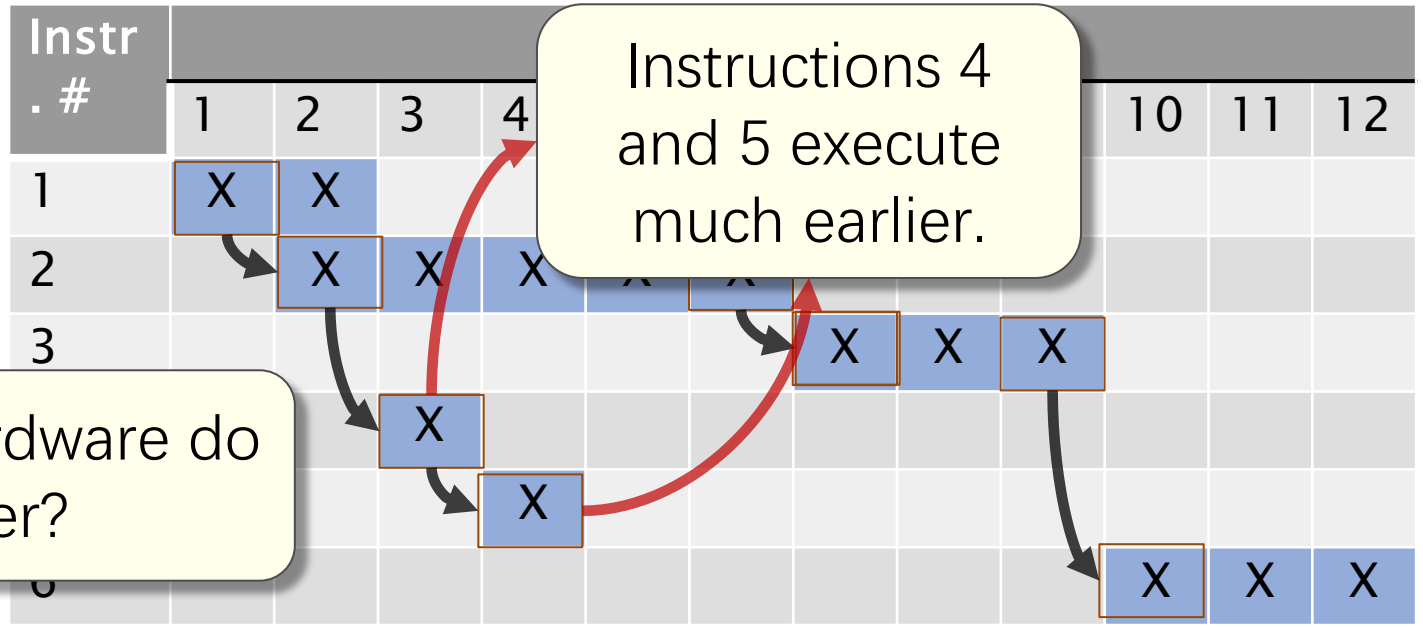
Example: Out-of-Order Execution

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

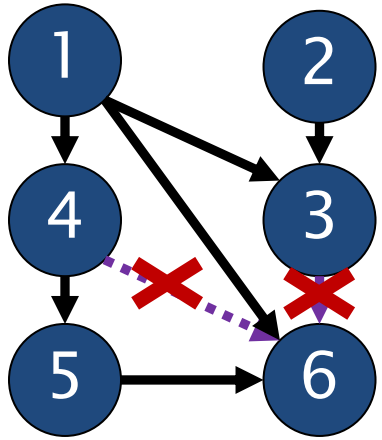


Eliminating Name Dependencies

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

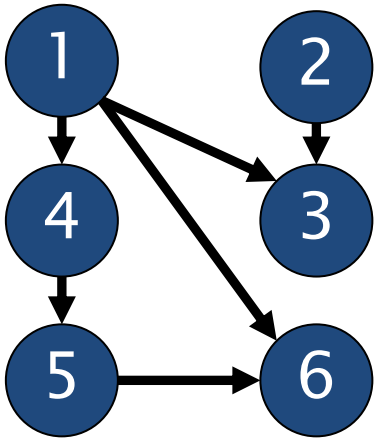
Data-flow graph



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!

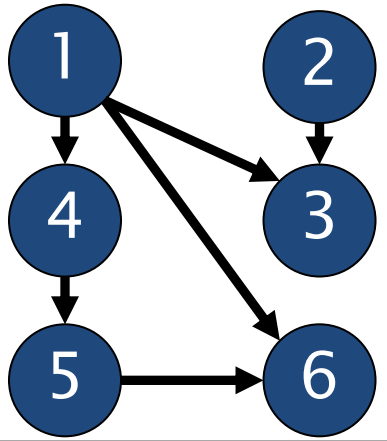
New data-flow graph



Effect of Eliminating Name Deps.

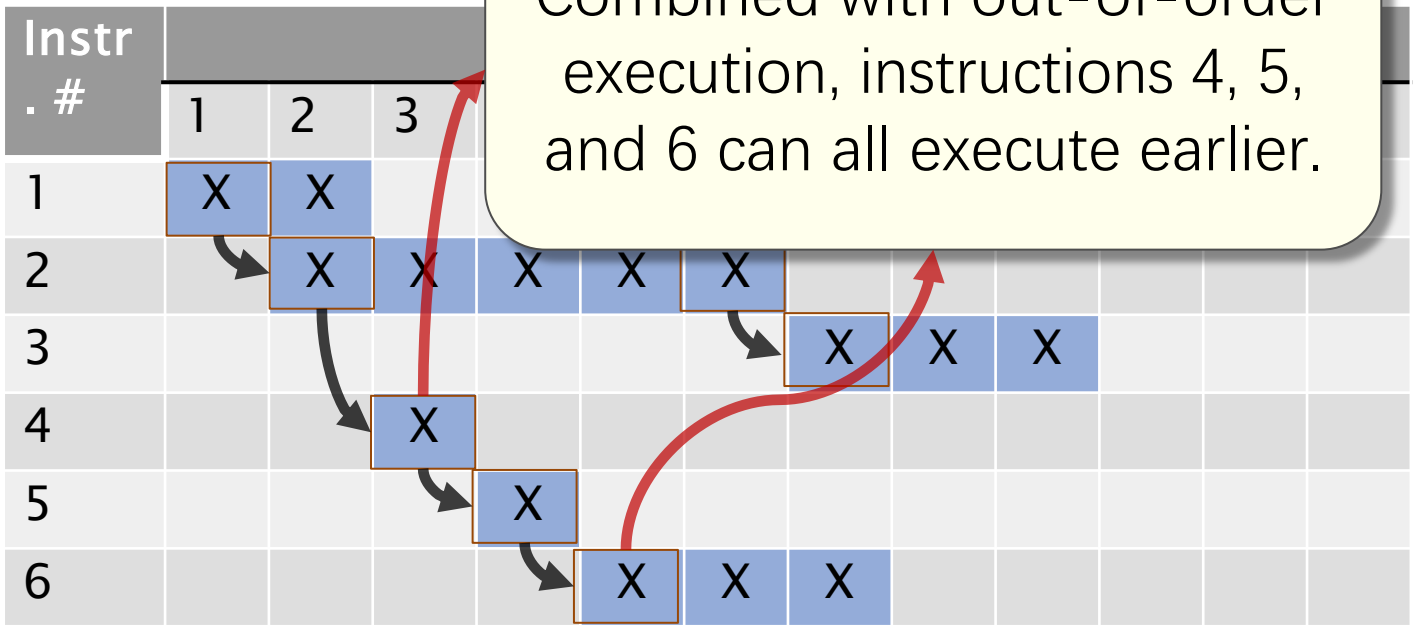
Idea: Let the hardware change destination register names on the fly

New data-flow graph



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

Instruction timing for use of functional units



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

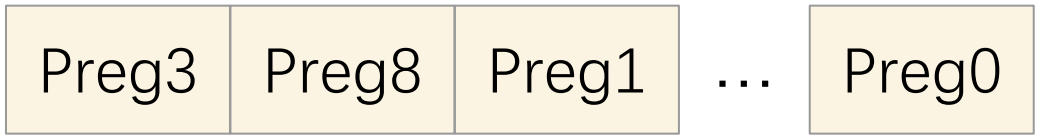
On-the-Fly Register Renaming

- 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



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

#	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` →

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

#	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` →

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

#	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` →

Step: Decode instruction 3.

Update mapping in renaming table.

Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	Preg4
xmm2	t3
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

#	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` →

Step: Decode instruction 4.

Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	Preg4
xmm2	t3
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	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` →

Step: Decode instruction 4.

Update mapping in renaming table.

Renaming table

ISA Reg.	Data
xmm0	t1
xmm1	t4
xmm2	t3
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	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
6	[illegible]	3

`%rip` →

Step: Instruction 1 finishes.

Replace tag with physical register.

Renaming table

ISA Reg.	Data
xmm0	Preg7
xmm1	t4
xmm2	t3
xmm3	

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	
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

`%rip` →

Step: Instruction 1 finishes.

Get the next available physical register from the free list.

Instruction 4 is ready to execute out of order.

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					
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`	3

`%rip` →

Step: Decode instruction 5.

Renaming table

ISA Reg.	Data
xmm0	Preg7
xmm1	t4
xmm2	t3
xmm3	

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

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` →

Step: Decode instruction 5.

Update mapping in renaming table.

Renamin

ISA Reg.	Data
xmm0	Preg7
xmm1	t5
xmm2	t3
xmm3	

Reorder buffer

		OP	Source 1	Source 2	Dest.
		~~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
5	`addsd %xmm1, %xmm1`	1
6	`mulsd %xmm1, %xmm0`	

`%rip` →

Step: Decode instruction 6.

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

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					

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