

Machine Learning Compilers

Bridging Models and Hardware
for Maximum Performance

A first-principles course: computers, compilers, GPUs,
and the systems that make AI fast

by Laxmen Murali

Copyright © 2026 by Laxmen Murali

All rights reserved. No part of this publication may be reproduced, stored or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise without written permission from the publisher. It is illegal to copy this book, post it to a website, or distribute it by any other means without permission.

Laxmen Murali has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Websites referred to in this publication and does not guarantee that any content on such Websites is, or will remain, accurate or appropriate.

First edition

Contents

Chapter 1 — What Is a Computer? From Switches to Thinking Machines	1
1.1 Learning Objectives	1
1.2 A Story: The Room Full of Computers	1
1.3 What Is a Computer, Really?	3
1.4 The Humble Switch: Where It All Begins	4
1.5 From Switches to Logic Gates	5
1.6 Making Gates Do Math: The Adder	7
1.7 The Mathematics: Boolean Algebra	9
1.8 Remembering Things: A One-Paragraph Preview	11
1.9 The Big Idea: The Stored Program	11
1.10 The Heartbeat: Fetch, Decode, Execute	13
1.11 The Ladder of Abstraction: The Map of This Entire Book	16
1.12 Engineering Perspective: The Real Thing	17
1.13 Performance Analysis: Learning to Think in Nanosec-	18
1.14 Optimization: The Three Big Ideas (Already!)	19
1.15 Build It: Gates and Adders in Python	20
1.16 Mini Project: Build TOY-100 Itself	23
1.17 Debugging Guide: When the Machine Does the Wrong	25
1.18 Interview Questions	26
1.19 Common Mistakes and Misconceptions	27
1.20 Exercises	28
1.21 Chapter Summary	30
1.22 Key Takeaways	31
Chapter 2 — Binary: The Language of Machines	34
2.1 Learning Objectives	34

2.2 A Story: The Fifteen-Year-Old Who Invented a Binary	35
2.3 What Is a Bit?	35
2.4 Counting When You Only Have Two Digits	37
2.5 Everything Is Numbers; Numbers Are Interpretations	39
2.6 Binary Arithmetic and the Wrap-Around World	40
2.7 Negative Numbers: The Two's Complement Miracle	42
2.8 The Fraction Problem: Floating Point	44
2.9 The Formats That Run AI	47
2.10 Mathematical Foundations	50
2.11 Engineering Perspective	51
2.12 Performance Analysis: Bytes Are Time	53
2.13 Optimization Techniques	53
2.14 Build It: Binary in Python	54
2.15 Debugging Guide: Hunting NaNs and Other Numerical	57
2.16 Interview Questions	58
2.17 Common Mistakes and Misconceptions	59
2.18 Exercises	60
2.19 Chapter Summary	61
2.20 Key Takeaways	62
Chapter 3 — Memory: The Computer's Notebook	65
3.1 Learning Objectives	65
3.2 A Story: The Bits That Lived in Mercury	66
3.3 What Is Memory, Really?	67
3.4 What the Mailboxes Are Made Of	68
3.5 Inside the DRAM Warehouse: Why Order Beats Chaos	69
3.6 Flattening the World: How Matrices Live in 1-D	70
3.7 The Numbers: A Tour of the Latency Ladder	72
3.8 Bandwidth: The Other Axis	73
3.9 Locality: The Two Habits of Highly Effective Programs	74
3.10 Mathematical Foundations	75
3.11 Engineering Perspective	76
3.12 Performance Analysis: We Measure the Cliff	77

3.13 Optimization Techniques	78
3.14 Build It: Memory in Python, and TOY-100 Learns to Point	79
3.15 Debugging Guide: When Memory Misbehaves	81
3.16 Interview Questions	82
3.17 Common Mistakes and Misconceptions	83
3.18 Exercises	84
3.19 Chapter Summary	86
3.20 Key Takeaways	86
Chapter 4 — The CPU: The Engine of Computation	89
4.1 Learning Objectives	89
4.2 A Story: Ninety-Three Minutes	90
4.3 The Clock, and Why It Stopped Getting Faster	91
4.4 Registers: The Countertop, Formalized	91
4.5 The Pipeline: Fetch-Decode-Execute Meets Henry Ford	92
4.6 Branch Prediction: The Machine Bets on Your Future	94
4.7 Out-of-Order Execution: The Great Illusion	95
4.8 Superscalar and SIMD: More Per Beat, Twice Over	97
4.9 ISA versus Microarchitecture: The Contract and the Fac-	98
4.10 Mathematical Foundations	99
4.11 Engineering Perspective: Where the Silicon Actually	99
4.12 Build It: A Pipelined TOY-100, Cycle by Cycle	100
4.13 Debugging Guide: Reading a Core’s Mind	102
4.14 Interview Questions	102
4.15 Common Mistakes and Misconceptions	104
4.16 Exercises	104
4.17 Chapter Summary	106
4.18 Key Takeaways	106
Chapter 5 — Caches and the Memory Hierarchy	109
5.1 Learning Objectives	109
5.2 A Story: Two Pages in 1965	110
5.3 What a Cache Is (and Isn’t)	110
5.4 Anatomy: How a Cache Actually Finds Things	111

5.5 The Three C's: A Taxonomy of Misses	113
5.6 Prefetching: The Cache Sees the Future	114
5.7 Multicore: Coherence and the Fourth C	114
5.8 Mathematical Foundations	115
5.9 The Showcase: Tiling, Step by Step	116
5.10 Engineering Perspective	117
5.11 Build It: A Cache Simulator in 25 Lines	118
5.12 Debugging Guide: Seeing the Invisible	119
5.13 Interview Questions	120
5.14 Common Mistakes and Misconceptions	121
5.15 Exercises	122
5.16 Chapter Summary	123
5.17 Key Takeaways	124
Chapter 6 — Processes, Threads, and Virtual Memory	127
6.1 Learning Objectives	127
6.2 A Story: The Computer That Pretended	128
6.3 The Operating System: A Program That Runs Programs	128
6.4 Processes: The Unit of Pretending	129
6.5 Threads: Shared Everything, and the Price of It	130
6.6 Virtual Memory: The Grand Illusion	131
6.7 The TLB: Chapter 5 Moonlights as a Translator	133
6.8 Mathematical Foundations	134
6.9 Engineering Perspective	134
6.10 Build It: An Operating System for TOY-100	135
6.11 Debugging Guide: When the Government Is the Suspect	136
6.12 Optimization Techniques	137
6.13 Interview Questions	138
6.14 Common Mistakes and Misconceptions	139
6.15 Exercises	140
6.16 Chapter Summary	141
6.17 Key Takeaways	142
Chapter 7 — Performance: Thinking Like a Speed Engineer	145

7.1 Learning Objectives	145
7.2 A Story: The Law That Was Born Losing an Argument	146
7.3 The Three Questions	146
7.4 Amdahl and Gustafson: The Two Ceilings	147
7.5 Little's Law: The Formal Home	148
7.6 The Roofline Model: Every Kernel's Fate on One Napkin	149
7.7 Percentiles: The Tail Runs the Business	150
7.8 Benchmarking: The Seven Lies	151
7.9 Profiling: Finding Where Time Actually Goes	152
7.10 Mathematical Foundations	153
7.11 Engineering Perspective: Performance as a Practice	154
7.12 Build It: The Roofline Kit and the TOY-100 Profiler	155
7.13 Debugging Guide: The Master Checklist	156
7.14 Interview Questions	157
7.15 Common Mistakes and Misconceptions	158
7.16 Exercises	159
7.17 Chapter Summary	160
7.18 Key Takeaways	161
Chapter 8 — What Is Programming? Variables, Types, and Functions	164
8.1 Learning Objectives	164
8.2 A Story: The Woman Who Taught Machines to Meet Us	165
8.3 What a Program Is	166
8.4 Variables: The Two Great Models	166
8.5 Types: Interpretation Agreements, Promoted to Law	168
8.6 Control Flow: Costumes Over JZ and JUMP	169
8.7 Functions: The Treaty's Crown Jewel	170
8.8 The Two-Language Problem: Why ML Speaks Python and	172
8.9 Mathematical Foundations	173
8.10 Engineering Perspective: Reading the Treaty Table	174
8.11 Performance Analysis: Pricing the Treaty	174
8.12 Build It: Semantics on the Bench	175
8.13 Debugging Guide: Reading the Treaty's Error Messages	176

8.14 Interview Questions	177
8.15 Common Mistakes and Misconceptions	178
8.16 Exercises	179
8.17 Chapter Summary	180
8.18 Key Takeaways	181
Chapter 9 — Pointers, References, and the Shape of Memory	184
9.1 Learning Objectives	184
9.2 A Story: The Billion-Dollar Convenience	185
9.3 The Grammar: & and *	185
9.4 Pointer Arithmetic: The Stride Formula Becomes an Op-	186
9.5 Structs: Boxes With Floor Plans	187
9.6 The Catastrophes: A Field Guide	188
9.7 Where Boxes Live: Stack and Heap, Finally Explicit	190
9.8 Python Unmasked: Tags Are Pointers All the Way Down	190
9.9 Mathematical Foundations	191
9.10 Build It: malloc-lite	192
9.11 Debugging Guide: Pointer Forensics	194
9.12 Interview Questions	194
9.13 Common Mistakes and Misconceptions	196
9.14 Exercises	196
9.15 Chapter Summary	198
9.16 Key Takeaways	199
Chapter 10 — Recursion and the Call Stack	201
10.1 Learning Objectives	201
10.2 A Story: The Amsterdam Smuggling Operation	202
10.3 Recursion: The Two-Rule Discipline	203
10.4 The Call Stack: Frames Are Structs	204
10.5 TOY-100 Gets Its Stack	205
10.6 Recursion Shapes: Linear, Tree, Tail, and the Exponen-	206
10.7 The Explicit-Stack Recipe: Any Recursion → A Loop	207
10.8 Mathematical Foundations	208
10.9 Engineering Perspective: Recursion Is How Compilers	209

10.10 Debugging Guide: When the Stack Is the Story	210
10.11 Interview Questions	210
10.12 Common Mistakes and Misconceptions	212
10.13 Exercises	213
10.14 Chapter Summary	214
10.15 Key Takeaways	215
Chapter 11 — Data Structures: Organizing Information	217
11.1 Learning Objectives	217
11.2 A Story: The Man Who Taught Filing Cabinets Arith-	218
11.3 The Dynamic Array: The Default Answer	218
11.4 Linked Lists: The Honest Verdict	220
11.5 Hash Tables: Luhn’s Arithmetic, Built From Scratch	220
11.6 The LRU Cache: Paying Chapter 5’s Promise	222
11.7 Trees, Heaps, and the Sorted World	222
11.8 Mathematical Foundations	223
11.9 Performance Analysis: The Two-Ledger Table	224
11.10 Build It: The Chapter’s Artifacts	225
11.11 Debugging Guide: When the Structure Is the Bug	226
11.12 Interview Questions	227
11.13 Common Mistakes and Misconceptions	228
11.14 Exercises	229
11.15 Chapter Summary	231
11.16 Key Takeaways	231
Chapter 12 — Algorithms and Complexity	234
12.1 Learning Objectives	234
12.2 A Story: The Question That Built the Computer	235
12.3 Big-O: The Growth Law, Measured	235
12.4 Recurrences: Pricing Recursion, Formally	236
12.5 Sorting: The Grand Tour	237
12.6 Binary Search: Twenty Questions, and a Famous Bug	238
12.7 P, NP, and the Compiler’s Daily NP-Hardness	239
12.8 The Halting Problem: The Wall Itself	240

12.9 The Consequence: Compilers as Engines of Safe Approx-	241
12.10 Mathematical Foundations	242
12.11 Engineering Perspective: Living With the Walls	243
12.12 Debugging Guide: When Growth Is the Bug	243
12.13 Interview Questions	244
12.14 Common Mistakes and Misconceptions	245
12.15 Exercises	246
12.16 Chapter Summary	248
12.17 Key Takeaways	248
Chapter 13 — Python: How the Interpreter Really Works	251
13.1 Learning Objectives	251
13.2 A Story: The Christmas Hobby That Ate Scientific Com-	252
13.3 What Actually Happens When You Run a Python File	252
13.4 Bytecode: Reading the Machine's Second Language	253
13.5 The Eval Loop: TOY-100's Heartbeat, Industrialized	254
13.6 The Forty-Step Adventure of $a + b$	255
13.7 Reference Counting, Live, and Why the GIL Exists	256
13.8 Why Python Is Slow, The Complete Ledger	256
0.028 GFLOPS vs BLAS 201 (7,100×, Ch. 5); sum loop vs np.sum 14-56×;	257
13.9 The Escape Hatches: Vectorize, Specialize, Compile	257
13.10 Optimization: Idioms That Respect the Machine	258
13.11 Build It: mini-ceval, Run Real Bytecode	258
13.12 Debugging Guide: Interpreter-Aware Diagnosis	260
13.13 Interview Questions	260
13.14 Common Mistakes and Misconceptions	262
13.15 Exercises	262
13.16 Chapter Summary	264
13.17 Key Takeaways	265
Chapter 14 — C++ for Systems and Compiler Engineers	267
14.1 Learning Objectives	267
14.2 A Story: The Simulation That Wouldn't Fit	268
14.3 RAI: The Idea That Fixes Chapter 9	269

14.4 Ownership: The Vocabulary That Documents Itself	270
14.5 Undefined Behavior: The Contract, Witnessed	271
14.6 Templates: The Compiler's Compiler	272
14.7 Virtual Dispatch: The Vtable, Priced	272
14.8 The Capstone: TOY-100, Twice, and the 83×	273
14.9 Engineering Perspective: The Build Model and the	274
14.10 Mathematical Foundations	274
14.11 Debugging Guide: The Field Kit, Assembled	275
14.12 Interview Questions	276
14.13 Common Mistakes and Misconceptions	277
14.14 Exercises	278
14.15 Chapter Summary	279
14.16 Key Takeaways	280
Chapter 15 — What Is a Compiler? The Grand Tour	283
15.1 Learning Objectives	283
15.2 A Story: The Eighteen Staff-Years That Almost Didn't	284
15.3 What a Compiler Is (Now That You've Built the Pieces)	285
15.4 The Pipeline: Three Stages, One Diagram	285
15.5 The Grand Tour: One Program, Every Artifact	286
15.6 Build It: microc, the Book's Compiler	288
15.7 Correctness: The Commuting Diagram and the Differ-	289
15.8 Mathematical Foundations	290
15.9 Engineering Perspective: Real Pipelines, Real Budgets	291
15.10 Debugging Guide: When the Translator Is the Suspect	291
15.11 Interview Questions	292
15.12 Common Mistakes and Misconceptions	293
15.13 Exercises	294
15.14 Chapter Summary	295
15.15 Key Takeaways	296
Chapter 16 — Lexing: Turning Characters into Words	299
16.1 Learning Objectives	299
16.2 A Story: From Neurons to grep in Three Strange Steps	300

16.3 Tokens: The Words, and Where They Live	301
16.4 Maximal Munch: The One Rule of Tokenization	301
16.5 The Theory, Hands On: Regex → NFA (Thompson's Con-	302
16.6 NFA → DFA: The Subset Construction	303
16.7 Performance: The Measured Ladder, Honestly Read	304
0.87 Mtok/s char interpreter ceremony	304
16.8 The Boundary: What Regular Languages Cannot Do	304
16.9 Engineering: Error Recovery and the Craft of Being	305
16.10 Mathematical Foundations	306
16.11 Debugging Guide: When the Words Are Wrong	306
16.12 Interview Questions	307
16.13 Common Mistakes and Misconceptions	308
16.14 Exercises	309
16.15 Chapter Summary	310
16.16 Key Takeaways	311
Chapter 17 — Parsing: Turning Words into Structure	314
17.1 Learning Objectives	314
17.2 A Story: The Notation That Outlived the Language	315
17.3 Grammars: The Rules of Shape	316
17.4 Ambiguity: One String, Two Trees	316
17.5 Recursive Descent: The Grammar Becomes Call Struc-	317
17.6 Pratt Parsing: The Ten Lines Inside Every Modern Front	318
17.7 Error Recovery: The Parser as Diplomat	319
17.8 The Zoo: A Working Engineer's Map of Parsing Tech-	320
17.9 Mathematical Foundations	321
17.10 Engineering Perspective: ASTs as Industrial Data	321
17.11 Debugging Guide: When the Tree Is Wrong	322
17.12 Interview Questions	323
17.13 Common Mistakes and Misconceptions	324
17.14 Exercises	325
17.15 Chapter Summary	326
17.16 Key Takeaways	327

Chapter 18 — ASTs and Semantic Analysis	330
18.1 Learning Objectives	330
18.2 A Story: The Barber, the Logician, and the Slogan	331
18.3 The Terrain: What the Grammar Cannot Know	331
18.4 Names and Scopes: The Symbol Table Employed	332
18.5 Type Checking: Structural Induction With a Badge	333
18.6 The Quiet Transformations: Desugaring and the First	334
18.7 Mathematical Foundations	335
18.8 Engineering Perspective: Sema at Industrial Scale	336
18.9 Debugging Guide: When Meaning Goes Wrong	337
18.10 Interview Questions	337
18.11 Common Mistakes and Misconceptions	339
18.12 Exercises	339
18.13 Chapter Summary	341
18.14 Key Takeaways	341
Chapter 19 — Intermediate Representations and SSA	344
19.1 Learning Objectives	344
19.2 A Story: The Form That Ate the Compiler World	345
19.3 Why an IR at All: The Goldilocks Argument	345
19.4 Three-Address Code: microc Gets an IR	346
19.5 The Assignment Problem, and the Rename That Solves	347
19.6 Why SSA Makes Optimization Easy: The Sparse Dividend	348
19.7 Constructing SSA: Dominance and the Frontier	349
19.8 Leaving SSA, and the Design Dimensions of Real IRs	350
19.9 Mathematical Foundations	351
19.10 Debugging Guide: Life Inside the IR	351
19.11 Interview Questions	352
19.12 Common Mistakes and Misconceptions	353
19.13 Exercises	354
19.14 Chapter Summary	355
19.15 Key Takeaways	356
Chapter 20 — Control Flow Graphs and Data Flow Analysis	359

20.1 Learning Objectives	359
20.2 A Story: The Teacher Who Stayed	360
20.3 The Control-Flow Graph: A Map of All Possible Runs	361
20.4 The Four Quadrants: What Analyses Ask	361
20.5 Liveness, Live: A Fixed Point You Can Watch	362
20.6 Reaching Definitions, and a Reunion Confirmed	363
20.7 The Framework: Lattices, Monotonicity, and the Termi-	364
20.8 Engineering: Worklists, Orders, and the Pass-Manager	365
20.9 Precision's Dials: MOP, Sensitivity, and the Aliasing	365
20.10 Debugging Guide: When the Facts Are Wrong	366
20.11 Interview Questions	367
20.12 Common Mistakes and Misconceptions	368
20.13 Exercises	369
20.14 Chapter Summary	370
20.15 Key Takeaways	371
Chapter 21 — Classical Optimizations	374
21.1 Learning Objectives	374
21.2 A Story: The Insult That Clarified the Field	375
21.3 The Catalogue, Organized by Scope	376
21.4 The Local Workhorses, and the Cascade	376
21.5 Loop Optimizations: Where the Time Is	377
21.6 Inlining: The Gateway	378
21.7 What Optimizers Cannot Do	379
21.8 Engineering: Anatomy of -O2	379
21.9 Mathematical Foundations	380
21.10 Debugging Guide: When Optimization Is the Suspect	380
21.11 Interview Questions	381
21.12 Common Mistakes and Misconceptions	383
21.13 Exercises	383
21.14 Chapter Summary	385
21.15 Key Takeaways	385

Chapter 22 — The Backend: Instruction Selection, Scheduling, Register Allocation, Codegen	388
22.1 Learning Objectives	388
22.2 A Story: The Same Building, Third Appearance	389
22.3 The Three Problems, and Their Knots	390
22.4 Register Allocation I: From Liveness to Interference	390
22.5 Register Allocation II: Run On Our Own Compiler	391
22.6 Mathematical Foundations: Chordality, or the Wall Tun-	392
22.8 Reading Production Assembly: The Backend Made Visi-	394
22.9 Engineering: Emission, Encodings, and the Last Mile	394
22.10 Debugging Guide: The Backend’s Bug Families	395
22.11 Interview Questions	395
22.12 Common Mistakes and Misconceptions	397
22.13 Exercises	398
22.14 Chapter Summary	399
22.15 Key Takeaways	400
Chapter 23 — LLVM: The Industry Workhorse	403
23.1 Learning Objectives	403
23.2 A Story: The Thesis That Ate the Toolchain	404
23.3 The IR, Read With Your Own Eyes	404
23.4 The Lab: Real LLVM, Live, From Python	405
23.5 The Pass Ecosystem: -O2 as a Library	406
23.6 The C++ Survival Kit	407
23.7 Backends and the GPU Substrate	408
23.8 Why ML Needed More: The “Sits Too Low” Argument	408
23.9 Mathematical Foundations (Brief, as Befits a Tour)	409
23.10 Debugging Guide: Driving the Big Rig	409
23.11 Interview Questions	409
23.12 Common Mistakes and Misconceptions	411
23.13 Exercises	412
23.14 Chapter Summary	413
23.15 Key Takeaways	414

Chapter 24 — MLIR: The Compiler Construction Kit	417
24.1 Learning Objectives	417
24.2 A Story: The Zoo	418
24.3 The Anatomy: Four Ideas	418
24.4 Reading Real MLIR: The Dialect Tour	420
24.5 Progressive Lowering: The Spine	421
24.6 Traits and Interfaces: Write the Pass Once	421
24.7 Mathematical Foundations	422
24.8 Engineering Perspective: The Ecosystem	423
24.9 Debugging Guide: Life on the Staircase	423
24.10 Interview Questions	424
24.11 Common Mistakes and Misconceptions	425
24.12 Exercises	426
24.13 Chapter Summary	428
24.14 Key Takeaways	428
Chapter 25 — Vectors and Matrices from First Principles	431
25.1 Learning Objectives	431
25.2 A Story: The Island Where Multiplication Was Rediscovered	432
25.3 Vectors: Lists With Commitments	433
25.4 Matrices Are Functions	433
25.5 Multiplication Is Composition (Everything Follows)	434
25.6 The Twenty-Line Network: Why Nonlinearity	435
25.7 Mathematical Foundations	436
25.8 The BLAS Ladder: Intensity Finally Meets Its Subject	437
25.9 Debugging Guide: Linear Algebra in the Field	438
25.10 Interview Questions	438
25.11 Common Mistakes and Misconceptions	440
25.12 Exercises	440
25.13 Chapter Summary	442
25.14 Key Takeaways	442
Chapter 26 — Tensors, Broadcasting, and Memory Layout	445
26.1 Learning Objectives	445

26.2 A Story: The Summation Sign That Vanished	446
26.3 The Tensor, Defined at Last	446
26.4 Broadcasting: The Zero-Stride Confession	447
26.7 Layout: The Compiler Decision Wearing a Memory Cos-	450
26.8 Mathematical Foundations	450
26.9 Debugging Guide: Shape Forensics	451
26.10 Interview Questions	451
26.11 Common Mistakes and Misconceptions	453
26.12 Exercises	454
26.13 Chapter Summary	455
26.14 Key Takeaways	456
Chapter 27 — Numerical Precision: FP32, FP16, BF16, FP8, INT8	459
27.1 Learning Objectives	459
27.2 A Story: Twenty-Eight Seconds	460
27.3 The Three Ways Half-Precision Kills a Run	460
27.4 Loss Scaling: Moving the Gradients Into Range	461
27.5 Accumulate Wide: The Master Policy, Measured	462
27.6 The NaN Hunt: Discipline, Formalized	463
27.7 Safe Softmax and the Stability Toolkit	464
27.8 Stochastic Rounding and the Determinism Treaty	465
27.9 Precision as a Compiler Pass	466
27.10 Mathematical Foundations	466
27.11 Debugging Guide: The Numerics Field Manual	467
27.12 Interview Questions	468
27.13 Common Mistakes and Misconceptions	470
27.14 Exercises	470
27.15 Chapter Summary	472
27.16 Key Takeaways	473
Chapter 28 — What Is Machine Learning?	476
28.1 Learning Objectives	476
28.2 A Story: The Program That Beat Its Author	477
28.3 The Three Ingredients	477

28.4 Gradient Descent: The Engine, Built and Broken	479
0.030 UNSTABLE — loss exploding 4.3×10^7	479
0.050 DIVERGED to infinity 9.4×10^{78}	479
28.5 The Calculus, Just Enough	480
28.6 The Vocabulary, By Example	481
28.7 Overfitting: The Field’s Central Danger, Measured	481
28.8 Mathematical Foundations	482
28.9 Engineering Perspective: The ML System Around the	483
28.10 Debugging Guide: When Learning Doesn’t	484
28.11 Interview Questions	485
28.12 Common Mistakes and Misconceptions	486
28.13 Exercises	487
28.14 Chapter Summary	488
28.15 Key Takeaways	489
Chapter 29 — Neural Networks and Backpropagation	492
29.1 Learning Objectives	492
29.2 A Story: The Idea That Was Discovered Four Times	493
29.3 The Network: XOR, Trained	493
29.4 Backpropagation, Built From Scratch	494
29.5 Forward-Mode vs Reverse-Mode: Why the Field Chose	496
29.6 Backpropagation Is a Compiler Pass	497
29.8 Mathematical Foundations	499
29.9 Engineering Perspective: From Micrograd to PyTorch	500
29.10 Debugging Guide: When Gradients Go Wrong	500
29.11 Interview Questions	501
29.12 Common Mistakes and Misconceptions	503
29.13 Exercises	504
29.14 Chapter Summary	505
29.15 Key Takeaways	506
Chapter 30 — Convolutional Neural Networks	509
30.1 Learning Objectives	509
30.2 A Story: The Cat’s Visual Cortex	510

30.3 Why Fully-Connected Fails on Images	511
30.4 What Convolution Actually Computes	512
30.6 The Full Architecture: Pooling, Depth, and Hierarchy	513
30.7 The Landmark Architectures: A Decade of Vision	514
30.8 Mathematical Foundations	515
30.9 Engineering Perspective: CNNs as Compute Workloads	516
30.10 Debugging Guide: When the CNN Won't See	517
30.11 Interview Questions	518
30.12 Common Mistakes and Misconceptions	519
30.13 Exercises	520
30.14 Chapter Summary	522
30.15 Key Takeaways	523
Chapter 31 — Transformers and Attention	526
31.1 Learning Objectives	526
31.2 A Story: The Architecture That Deleted Recurrence	527
31.3 Attention: A Soft Dictionary Lookup	528
31.4 The $(QK^T)V$ Chain and the $O(n^2)$ Wall	529
31.5 Multi-Head, Masking, and Position	530
31.6 The Transformer Block, Assembled	531
31.7 Mathematical Foundations	532
31.8 Engineering Perspective: The Transformer as the Work-	533
31.9 Debugging Guide: When Attention Misbehaves	534
31.10 Interview Questions	535
31.11 Common Mistakes and Misconceptions	536
31.12 Exercises	537
31.13 Chapter Summary	539
31.14 Key Takeaways	540
Chapter 32 — Training, Inference, and Large Language Models	543
32.1 Learning Objectives	543
32.2 A Story: The Bet That Scale Would Work	544
32.3 Optimizers: Beyond Plain Gradient Descent	544
32.4 Tokenization: Text Into Integers	546

32.5 The Compute Laws: Pricing Scale	547
32.6 Scaling Laws and Chinchilla’s Correction	547
32.7 The Training-Inference Asymmetry	548
32.8 Mathematical Foundations	549
32.9 Engineering Perspective: The LLM as the Book’s Desti-	550
32.10 Debugging Guide: Training and Serving at Scale	551
32.11 Interview Questions	552
32.12 Common Mistakes and Misconceptions	554
32.13 Exercises	555
32.14 Chapter Summary	556
32.15 Key Takeaways	557
Chapter 33 — Why GPUs? The Parallel Revolution	560
33.1 Learning Objectives	560
33.2 A Story: From Triangles to Transformers	561
33.3 The Architectural Bet: Throughput vs Latency	562
33.4 How the GPU Hides Latency: Oversubscription, Not Out-	562
33.5 SIMT: Thousands of Threads, One Instruction	563
33.6 The Roofline Verdict: What the GPU Wins and Loses	564
33.7 Mathematical Foundations	565
33.8 Engineering Perspective: The GPU as the AI Substrate	566
33.9 Debugging Guide: When the GPU Underperforms	566
33.10 Interview Questions	567
33.11 Common Mistakes and Misconceptions	569
33.12 Exercises	570
33.13 Chapter Summary	572
33.14 Key Takeaways	573
Chapter 34 — CUDA and the SIMT Programming Model	576
34.1 Learning Objectives	576
34.2 A Story: The Language That Freed the GPU	577
34.3 The Thread Hierarchy: Organizing Millions	578
34.4 Thread Indexing: “Which Data Do I Own?”	578
34.5 Warps and Divergence, Measured	579

34.6 Synchronization and the Parallel Reduction	580
34.7 The Compilation Ladder: CUDA → PTX → SASS	581
34.8 Mathematical Foundations	582
34.9 Engineering Perspective: From Hand-Written Kernels to	583
34.10 Debugging Guide: CUDA Kernel Pitfalls	584
34.11 Interview Questions	584
34.12 Common Mistakes and Misconceptions	586
34.13 Exercises	587
34.14 Chapter Summary	589
34.15 Key Takeaways	589
Chapter 35 — The GPU Memory Hierarchy	592
35.1 Learning Objectives	592
35.2 A Story: The Scratchpad You Load Yourself	593
35.3 The Hierarchy: Registers to HBM	593
35.4 Coalescing: The Access Pattern Is the Bandwidth	594
35.5 Bank Conflicts: The Shared-Memory Hazard	595
35.6 Tiled Matmul in Shared Memory: The Payoff	596
35.7 Mathematical Foundations	597
35.8 Engineering Perspective: Memory Management as the	598
35.9 Debugging Guide: GPU Memory Performance	599
35.10 Interview Questions	600
35.11 Common Mistakes and Misconceptions	602
35.12 Exercises	603
35.13 Chapter Summary	604
35.14 Key Takeaways	605
Chapter 36 — Tensor Cores, Occupancy, and Kernel Scheduling	608
36.1 Learning Objectives	608
36.2 A Story: The Chip That Does Only One Thing	609
36.3 Tensor Cores: The Matmul Organ	610
36.4 The 300 FLOP/Byte Mystery, Solved	611
36.5 Occupancy: The Resource-Balancing Act	611
36.6 Kernel Scheduling: Keeping the Machine Fed	612

36.7 Mathematical Foundations	613
36.8 Engineering Perspective: Feeding the Tensor Cores Is	614
36.9 Debugging Guide: GPU Compute Performance	615
36.10 Interview Questions	616
36.11 Common Mistakes and Misconceptions	618
36.12 Exercises	619
36.13 Chapter Summary	621
36.14 Key Takeaways	622
Chapter 37 — Anatomy of an ML Compiler: Graph IR and Tensor IR	625
37.1 Learning Objectives	625
37.2 A Story: The Compiler That Was Lucky	626
37.3 The Computation Graph: The Central Data Structure	627
37.4 Graph Transformations: The Optimizations Are Graph	628
37.5 The Two Levels: Graph IR and Tensor IR	629
37.6 Mathematical Foundations	630
37.7 Engineering Perspective: The ML Compiler Landscape	631
37.8 Debugging Guide: Working With Computation Graphs	632
37.9 Interview Questions	633
37.10 Common Mistakes and Misconceptions	635
37.11 Exercises	636
37.12 Chapter Summary	638
37.13 Key Takeaways	639
Chapter 38 — Shape Inference and Graph Optimization	642
38.1 Learning Objectives	642
38.2 A Story: The Thesis, Cashed	643
38.3 Shape Inference: Type Checking for Tensors	643
38.4 Symbolic Shapes: The Boundary to Runtime	644
38.5 Layout Assignment: Choosing the Memory Format	645
38.6 Graph Rewrites: Cleaning and Normalizing	646
38.7 Mathematical Foundations	647
38.8 Engineering Perspective: The Front-End Analysis	648
38.9 Debugging Guide: Shape and Layout Issues	649

38.10 Interview Questions	650
38.11 Common Mistakes and Misconceptions	652
38.12 Exercises	653
38.13 Chapter Summary	655
38.14 Key Takeaways	656
Chapter 39 — Operator and Kernel Fusion	659
39.1 Learning Objectives	659
39.2 A Story: The Wall, and the Door Through It	660
39.3 The Memory-Traffic Accounting	661
39.4 Why Fusion Is The Answer: Arithmetic Intensity	662
39.5 The Fusion Pass: Finding the Fusions	663
39.6 The Taxonomy and the Limits	664
39.7 Mathematical Foundations	664
39.8 Engineering Perspective: The Optimization That Justi-	666
39.9 Debugging Guide: Fusion Problems	667
39.10 Interview Questions	668
39.11 Common Mistakes and Misconceptions	670
39.12 Exercises	671
39.13 Chapter Summary	673
39.14 Key Takeaways	674
Chapter 40 — Scheduling: Separating What from How	677
40.1 Learning Objectives	677
40.2 A Story: The Idea That Untangled Performance	678
40.3 The Demonstration: One Algorithm, Many Schedules	679
40.4 The Schedule as a First-Class Object	679
40.5 The Affine and Polyhedral Model	680
40.6 Autotuning: Searching the Schedule Space	681
40.7 Mathematical Foundations	682
40.8 Engineering Perspective: The Idea That Made ML Com-	683
40.9 Debugging Guide: Scheduling Problems	684
40.10 Interview Questions	685
40.11 Common Mistakes and Misconceptions	687

40.12 Exercises	688
40.13 Chapter Summary	690
40.14 Key Takeaways	691
Chapter 41 — Memory Planning	694
41.1 Learning Objectives	694
41.2 A Story: The Promise from Chapter 20, Cashed	695
41.3 Tensor Liveness: Chapter 20 with Gigabyte Nouns	696
41.4 The Peak, the Floor, and the Allocation	696
41.5 In-Place Operations: The No-Alias Gift Cashed	697
41.6 The Full Picture: Fitting the Model	698
41.7 Mathematical Foundations	699
41.8 Engineering Perspective: What Lets the Model Fit	700
41.9 Debugging Guide: Memory Problems	701
41.10 Interview Questions	702
41.11 Common Mistakes and Misconceptions	704
41.12 Exercises	705
41.13 Chapter Summary	706
41.14 Key Takeaways	707
Chapter 42 — Cost Models and Auto-Tuning	710
42.1 Learning Objectives	710
42.2 A Story: The Oracle the Compiler Needs	711
42.3 The Analytical Cost Model: The Roofline Mechanized	712
42.4 Autotuning: Searching the Space	713
42.5 Learned Cost Models: ML Optimizing ML	714
42.6 The Cost Model Is the Compiler’s Crux	714
42.7 Mathematical Foundations	715
42.8 Engineering Perspective: The Compiler’s Decision-	716
42.9 Debugging Guide: Cost Model and Autotuning Problems	717
42.10 Interview Questions	718
42.11 Common Mistakes and Misconceptions	720
42.12 Exercises	721
42.13 Chapter Summary	723

42.14 Key Takeaways	723
Chapter 43 — Kernel Generation	726
43.1 Learning Objectives	726
43.2 A Story: The Plan Becomes Code	727
43.3 The Lowering Ladder: From Graph to Code	727
43.4 The Generation Targets: CUDA, Triton, LLVM	728
43.5 The Complete Part VII Pipeline	729
43.6 Mathematical Foundations	730
43.7 Engineering Perspective: The Last Mile	731
43.8 Debugging Guide: Kernel Generation Problems	732
43.9 Interview Questions	733
43.10 Common Mistakes and Misconceptions	735
43.11 Exercises	736
43.12 Chapter Summary	737
43.13 Key Takeaways	738
Chapter 44 — TVM: The Full-Stack Pioneer	741
44.1 Learning Objectives	741
44.2 A Story: The Heretical Question	742
44.3 TVM’s Architecture: Part VII in a Real System	742
44.4 The Compute/Schedule Separation: Halide, Cashed	743
44.6 TVM Unity: The Cross-Level Design	745
44.7 Engineering Perspective: The Pioneer’s Legacy	746
44.8 Debugging Guide: Working with TVM	746
44.9 Interview Questions	747
44.10 Common Mistakes and Misconceptions	749
44.11 Exercises	750
44.12 Chapter Summary	752
44.13 Key Takeaways	753
Chapter 45 — PyTorch Compilation: FX, Dynamo, and Inductor	756
45.1 Learning Objectives	756
45.2 A Story: The Compiler That Came to the Models	757
45.3 TorchDynamo: Capturing the Graph from Eager Python	758

45.4 Guards and Recompilation: Chapter 13’s JIT, Cashed	759
45.5 AOTAutograd and TorchInductor: The Back-End	760
45.6 Engineering Perspective: The Compiler You Actually Use	761
45.7 Debugging Guide: torch.compile Performance	762
45.8 Interview Questions	763
45.9 Common Mistakes and Misconceptions	765
45.10 Exercises	766
45.11 Chapter Summary	767
45.12 Key Takeaways	768
Chapter 46 — XLA and OpenXLA	771
46.1 Learning Objectives	771
46.2 A Story: The Opposite Bet	772
46.3 HLO and StableHLO: The Portability Layer	772
46.4 The Static-Shape Advantage	773
46.5 Fusion and the TPU	774
46.6 OpenXLA: The Open Ecosystem	775
46.7 Engineering Perspective: The Static-Graph Compiler	775
46.8 Debugging Guide: Working with XLA	776
46.9 Interview Questions	777
46.10 Common Mistakes and Misconceptions	779
46.11 Exercises	780
46.12 Chapter Summary	781
46.13 Key Takeaways	782
Chapter 47 — IREE and the MLIR-Based Stacks	785
47.1 Learning Objectives	785
47.2 A Story: The Cost of Building Your Own Stack	786
47.3 The Dialect Lowering Ladder	787
47.4 The linalg Dialect: Chapter 24’s Indexing Maps, Cashed	787
47.5 The HAL: One Program, Many Devices	788
47.6 Engineering Perspective: The MLIR-Based Movement	789
47.7 Debugging Guide: Working with IREE and MLIR	790
47.8 Interview Questions	791

47.9 Common Mistakes and Misconceptions	793
47.10 Exercises	794
47.11 Chapter Summary	795
47.12 Key Takeaways	796
Chapter 48 — Triton: Python-to-GPU Kernels	799
48.1 Learning Objectives	799
48.2 A Story: The Right Altitude	800
48.3 Block-Level Programming: The Tile as the Unit	800
48.4 What the Compiler Handles: The Thread Level	801
48.5 Compilation and the ML Compiler’s Target	802
48.6 Engineering Perspective: The Accessible Kernel Lan-	803
48.7 Debugging Guide: Triton Kernels	804
48.8 Interview Questions	805
48.9 Common Mistakes and Misconceptions	807
48.10 Exercises	808
48.11 Chapter Summary	809
48.12 Key Takeaways	810
Chapter 49 — TensorRT and ONNX Runtime	813
49.1 Learning Objectives	813
49.2 A Story: A Different God	814
49.3 TensorRT: The Inference Optimizer	815
49.4 INT8 Calibration: Low-Precision Inference	815
49.5 ONNX and ONNX Runtime: The Interchange and the	816
49.7 Debugging Guide: Inference Deployment	818
49.8 Interview Questions	819
49.9 Common Mistakes and Misconceptions	821
49.10 Exercises	822
49.11 Chapter Summary	823
49.12 Key Takeaways	824
Chapter 50 — GEMM: From Naive to Near-Peak	827
50.1 Learning Objectives	827
50.2 A Story: The One Kernel	828

50.3 The Optimization Ladder	828
50.4 The Roofline Climb: Why Each Step Helps	829
50.5 The Gap to Near-Peak: The Expert Craft	830
50.6 Engineering Perspective: The Kernel the Book Built To-	831
50.7 Debugging Guide: GEMM Performance	832
50.8 Interview Questions	832
50.9 Common Mistakes and Misconceptions	834
50.10 Exercises	835
50.11 Chapter Summary	837
50.12 Key Takeaways	837
Chapter 51 — Convolution Kernels	840
51.1 Learning Objectives	840
51.2 A Story: Four Roads to the Same Answer	841
51.4 Implicit GEMM: The Speed Without the Memory	842
51.5 Winograd: Fewer Multiplies	843
51.6 Choosing the Algorithm	844
51.7 Engineering Perspective: The Algorithm-Selection Ker-	845
51.8 Debugging Guide: Convolution Performance	846
51.9 Interview Questions	846
51.10 Common Mistakes and Misconceptions	848
51.11 Exercises	849
51.12 Chapter Summary	851
51.13 Key Takeaways	851
Chapter 52 — Attention Kernels and FlashAttention	854
52.1 Learning Objectives	854
52.2 A Story: The Wall, and Every Tool You Were Given	855
52.3 The Online Softmax: The Algorithmic Key	856
52.4 FlashAttention: Fusion, Tiling, and the Wall Removed	857
52.5 The Backward Pass: Recompute Rather Than Store	858
52.6 Engineering Perspective: Where Every Thread Con-	859
52.7 Debugging Guide: Attention Kernels	860
52.8 Interview Questions	861

52.9 Common Mistakes and Misconceptions	863
52.10 Exercises	864
52.11 Chapter Summary	865
52.12 Key Takeaways	866
Chapter 53 — Normalization, Activations, and Reductions	869
53.1 Learning Objectives	869
53.2 A Story: The Operations Nobody Watches	870
53.3 Welford’s Algorithm: Chapter 27’s Promise, Cashed	871
53.4 Reductions: The Tree, and Why It’s Also More Accurate	872
53.5 The Normalization Kernels: LayerNorm and RMSNorm	873
53.6 Why Fusion Pays Most Here	873
53.7 Engineering Perspective: The Memory-Bound Majority	874
53.8 Debugging Guide: Norms, Activations, and Reductions	875
53.9 Interview Questions	876
53.10 Common Mistakes and Misconceptions	878
53.11 Exercises	878
53.12 Chapter Summary	880
53.13 Key Takeaways	881
Chapter 54 — Advanced Loop Craft	884
54.1 Learning Objectives	884
54.2 A Story: The Optimization That Did Nothing	885
54.3 Where Manual Effort Still Pays	886
54.4 The Tiling Hierarchy	887
54.5 Software Pipelining Where It Actually Matters: The GPU	888
54.6 Engineering Perspective: The Craft, Organized	889
54.7 Debugging Guide: Loop Performance	889
54.8 Interview Questions	890
54.9 Common Mistakes and Misconceptions	892
54.10 Exercises	893
54.11 Chapter Summary	895
54.12 Key Takeaways	896

Chapter 55 — Inference Serving: KV Cache, Continuous Batching, and Speculative Decoding	899
55.1 Learning Objectives	899
55.2 A Story: A Million Jobs a Second	900
55.3 Prefill, Decode, and the KV Cache	901
55.4 Batching: The Core Economics	902
55.5 PagedAttention: Chapter 6’s Virtual Memory, Applied	903
55.6 Continuous Batching: Don’t Wait for the Slowest	904
55.7 Speculative Decoding: Buying Parallelism with Cheap	904
55.8 Engineering Perspective: The Serving Stack	905
55.9 Debugging Guide: Serving Performance	906
55.10 Interview Questions	907
55.11 Common Mistakes and Misconceptions	909
55.12 Exercises	910
55.13 Chapter Summary	911
55.14 Key Takeaways	912
Chapter 56 — Parallelism: Tensor, Pipeline, and Expert	915
56.1 Learning Objectives	915
56.2 A Story: Where to Cut	916
56.3 Data and Tensor Parallelism	917
56.4 Pipeline Parallelism and the Bubble	918
56.5 Expert Parallelism and Mixture-of-Experts	919
56.6 The Collectives Underneath	920
56.7 Composing Them: 3D Parallelism	920
56.8 Engineering Perspective: Choosing Where to Cut	921
56.9 Debugging Guide: Distributed Performance	922
56.10 Interview Questions	923
56.11 Common Mistakes and Misconceptions	925
56.12 Exercises	926
56.13 Chapter Summary	927
56.14 Key Takeaways	928
Chapter 57 — Quantization	931

57.1 Learning Objectives	931
57.2 A Story: Sixteen Bits, and How Many You Actually Need	932
57.3 What Quantization Is: The Affine Map	933
57.4 The Central Asymmetry: Weights Give In, Activations	934
57.6 Choosing the Numbers: Calibration, GPTQ, AWQ, and	936
57.7 Quantization as a Compiler Pass	938
57.8 Engineering Perspective: Picking a Quantization Recipe	939
57.9 Debugging Guide: When Quantization Goes Wrong	940
57.10 Interview Questions	941
57.11 Common Mistakes and Misconceptions	943
57.12 Exercises	943
57.13 Chapter Summary	945
57.14 Key Takeaways	946
Chapter 58 — Design and Architecture	949
58.1 Learning Objectives	949
58.2 A Story: The First Decision	950
58.3 What We Are Building: Scope, Non-Goals, and the Walk-	950
58.4 The Narrow Waist: The IR Is the Architecture	951
58.5 The Pass Pipeline: A Sequence of Rewrites	953
58.6 The Two Ends: Frontend, Backend, and the Interpreter	954
58.7 The Walking Skeleton, Running	955
58.8 Engineering Perspective: Architecting for Change	956
58.9 Debugging Guide: When the Architecture Fights You	957
58.10 Interview Questions	958
58.11 Common Mistakes and Misconceptions	959
58.12 Exercises	960
58.13 Chapter Summary	961
58.14 Key Takeaways	962
Chapter 59 — Building the IR and Optimization Passes	965
59.1 Learning Objectives	965
59.2 A Story: The Stubs Come Alive	966
59.3 Hardening the IR	966

59.4 Shape Inference: Type Checking Over Tensors	967
59.5 Simplification and CSE: Canonicalize, Then Deduplicate	968
59.6 Fusion: The Memory Wall, in Our Compiler	969
59.7 The Pipeline, Running	970
59.8 Engineering Perspective: Writing Passes That Compose	971
59.9 Debugging Guide: When a Pass Misbehaves	971
59.10 Interview Questions	972
59.11 Common Mistakes and Misconceptions	974
59.12 Exercises	974
59.13 Chapter Summary	976
59.14 Key Takeaways	977
Chapter 60 — The Backend and Code Generation	979
60.1 Learning Objectives	979
60.2 A Story: Truth Was Never Fast	980
60.3 Lowering: From Graph to Loops to Code	980
60.4 Memory Planning: The Buffers That Ping-Pong	981
60.5 Quantization in the Backend	982
60.6 The Backend Meets the Oracle	983
60.7 Engineering Perspective: Correctness Before Speed	984
60.8 Debugging Guide: When Generated Code Goes Wrong	984
60.9 Interview Questions	985
60.10 Common Mistakes and Misconceptions	987
60.11 Exercises	987
60.12 Chapter Summary	989
60.13 Key Takeaways	990
Chapter 61 — Testing, Benchmarking, and Debugging	992
61.1 Learning Objectives	992
61.2 A Story: How Compilers Actually Fail	993
61.3 The Differential Test Harness: Fuzzing Against the Ora-	993
61.4 Delta Debugging: Shrinking a Failure to Its Essence	995
61.5 Benchmarking Honestly: Percentiles, Throughput, and	996
61.6 Debugging Tools: Dumps, the Verifier, and Provenance	996

61.7 Engineering Perspective: What Makes a Compiler Trust-	997
61.8 Debugging Guide: Debugging the Compiler Itself	998
61.9 Interview Questions	999
61.10 Common Mistakes and Misconceptions	1000
61.11 Exercises	1001
61.12 Chapter Summary	1003
61.13 Key Takeaways	1003
Chapter 62 — How Real Teams Build ML Compilers	1006
62.1 Learning Objectives	1006
62.2 A Story: Nine Teams, One Compiler	1007
62.3 The Common Anatomy	1007
62.4 Google: The Co-Design Archetype	1008
62.5 NVIDIA: The Incumbent and the Software Moat	1009
62.6 The Challengers and the MLIR Convergence	1009
62.7 The Framework Owners and the Labs	1010
62.8 Engineering Perspective: What Is Universal, What Is	1011
62.9 Debugging Guide: Orienting Yourself in an Unfamiliar	1012
62.10 Interview Questions	1013
62.11 Common Mistakes and Misconceptions	1015
62.12 Exercises	1015
62.13 Chapter Summary	1016
62.14 Key Takeaways	1017
Chapter 63 — The Road Ahead: Careers, Interviews, and the Future	1020
63.1 Learning Objectives	1020
63.2 A Story: The View from the Summit	1021
63.3 The Interview, Synthesized	1021
63.4 The Four Employers and What They Value	1022
63.5 Breaking In: Your Portfolio and the Open Field	1023
63.6 The Future: The Durable and the Ephemeral	1024
63.7 Career Perspective: Building on a Field Still Being In-	1025
63.8 Debugging Guide: When the Search or the Interview	1026
63.9 Interview Questions	1027

63.10 Common Mistakes and Misconceptions	1028
63.11 Exercises	1028
63.12 Chapter Summary	1030
63.13 Key Takeaways	1030

Chapter 1 — What Is a Computer? From Switches to Thinking Machines

“What I cannot create, I do not understand.”

Richard Feynman, written on his blackboard at the time of his death

1.1 Learning Objectives

By the end of this chapter, you will be able to:

- Explain what a computer fundamentally is, and what it is not, in plain language.
- Describe how a simple on/off switch (a transistor) becomes a logic gate, how logic gates become an adder, and how adders become a machine that computes.
- Explain the **stored program concept**: the single most important idea in all of computing.
- Draw and explain the **von Neumann architecture** and the **fetch-decode-execute cycle**.
- Trace, by hand, the execution of a real program on a simple machine, one step at a time.
- Explain the **ladder of abstraction**, the tower of layers from physics up to AI models, and point to exactly where ML compilers live in that tower.
- Build a complete, working computer simulator (**TOY-100**) in about 40 lines of Python, a machine we will keep extending throughout this book.
- Answer the classic entry-level systems interview questions about how computers work.

This chapter is the foundation stone. Every single topic in this book, GPUs, LLVM, tensor IRs, FlashAttention, all of it, is built from the ideas on these pages. Take your time here.

1.2 A Story: The Room Full of Computers

We’ll begin in 1944, in a secret laboratory on a mesa in New Mexico called Los Alamos. The people there were designing the atomic bomb, and they had a problem that had

nothing to do with physics and everything to do with arithmetic.

To predict what would happen inside an imploding bomb core, they needed to solve enormous systems of equations. Not solve them cleverly, solve them *numerically*: millions of small multiplications and additions, each one feeding the next. No human being could do this alone in a lifetime.

So they filled a room with people and mechanical calculators, clattering desktop machines called Marchants that could multiply two numbers if you cranked them. And here is the beautiful part, the part that matters for this entire book: the person who figured out how to make that room *fast* was a young physicist named Richard Feynman.

Feynman's insight was that the problem wasn't the speed of any one person. It was the *organization* of the work. He and his colleagues broke each big calculation into tiny steps and arranged the people like an assembly line: this person only multiplies, that person only adds, this one only squares a number and passes the card to the next desk. A single problem flowed through the room like a car through a factory.

Then they got greedy. Why should the room work on only one problem at a time? They ran several calculations *simultaneously*, using different colored cards for each problem, flowing through the same room, through the same people, interleaved. When someone discovered an error in one calculation, they didn't restart everything, they figured out exactly which cards were contaminated, recomputed just those, and spliced the corrections back into the stream.

Stop and look at what this room invented, decades before the vocabulary existed:

- Breaking a computation into tiny, dumb, repeatable steps → **instructions**
- One specialist per step, work flowing between them → **pipelining**
- Multiple problems in flight at once, sharing the same workers → **parallelism and multi-tenancy**
- Recomputing only what an error touched → **incremental computation and dependence analysis**
- Deciding who does what, when, to keep everyone busy → **scheduling**

Here's the punchline. And I want you to hold onto it for the next thousand pages: **that room was a computer**. The people were the processing units. The cards were the memory. Feynman's arrangement of the work was the *program*, and the act of arranging it well was *compilation*.

Everything in this book, every GPU kernel, every compiler pass, every scheduling algorithm, is a descendant of that room. The machines got faster by a factor of a trillion. The ideas barely changed.

One more historical note, because it reframes everything: the word "computer" originally meant *a person*. From the 1600s until roughly the 1950s, a "computer" was a job title, like "baker" or "carpenter", someone employed to do calculations. The women who calculated rocket trajectories at NASA (you may know Katherine Johnson's story from the film *Hidden Figures*) were called computers. The machine was named after the human, not the other way around. A computer is not a magical thinking box. It is a worker that follows instructions. It just happens to be made of silicon instead of flesh, and to work about a trillion times faster.

1.3 What Is a Computer, Really?

Explain It Like I'm Five

A computer is a machine that follows a recipe.

That's it. That's the whole thing.

You give it a list of very simple steps, "take this number, add it to that number, write down the answer, if the answer is zero skip to step 7", and it follows them. One at a time. In order. Without getting bored, without getting distracted, without asking why.

The steps it can follow are *incredibly* dumb. A computer cannot "recognize a cat." It cannot "understand a sentence." It can only do things like: copy a number from here to there, add two numbers, compare two numbers, and jump to a different step in the recipe. That's roughly the whole menu.

So how does a machine that can only add and copy numbers end up recognizing cats and writing poetry? Because it follows *billions* of those dumb steps *per second*, and because very clever people have figured out how to write recipes in which billions of dumb steps add up to something that looks like intelligence.

Hold onto this: **a computer is not smart. It is fast.** All the smartness lives in the recipes. And this book is about the programs that write the recipes for the fastest machines on Earth.

The Core Intuition

Every computer, from the chip in a musical greeting card to the GPU cluster training a frontier AI model, is the same three things:

1. **A place to keep numbers** (memory)
2. **A thing that changes numbers** (a processor)
3. **A recipe saying which changes to make in which order** (a program)

Memorize that triad. When we meet a GPU in Part VI, it will be: a *huge* place to keep numbers, *thousands* of things that change numbers, and recipes called kernels. When we build an ML compiler in Part XI, its job will be: given a recipe written for mathematicians, rewrite it into the best possible recipe for the machine. Different costumes, same skeleton.

The Analogy We'll Reuse All Book: The Kitchen

Picture a professional kitchen.

- The **pantry** is memory: shelves of ingredients (numbers), each shelf with a label (an address).
- The **chef** is the processor: they can chop, mix, and heat (add, multiply, compare), but only what's on the counter in front of them.
- The **counter** is the registers: a tiny workspace holding just the few ingredients being worked on *right now*.
- The **recipe** is the program: numbered steps, followed in order.
- The **finished dish** is the output.

Notice something important that will dominate this book from Chapter 5 onward: the chef is fast, but *walking to the pantry is slow*. A great kitchen isn't one with a faster chef, it's one organized so the chef almost never has to walk to the pantry. Remember this sentence. In modern computing, **arithmetic is nearly free; moving data is expensive**. A huge fraction of ML compiler engineering is pantry logistics.

1.4 The Humble Switch: Where It All Begins

Every concept in this book, we'll build from the ground up. So let's go all the way to the ground. What is a computer *made of*?

ELI5

A computer is made of tiny light switches. Billions of them. Each switch is either ON or OFF. That's all a computer physically is: an unimaginably large box of switches flipping each other on and off very, very fast.

The Real-World Object: The Transistor

The switch is called a **transistor**, invented at Bell Labs in 1947 by John Bardeen, Walter Brattain, and William Shockley (they got the Nobel Prize for it). Before the transistor, computers used vacuum tubes, glowing glass bulbs the size of your thumb that burned out constantly. ENIAC, completed in 1945, used about 17,468 of them, filled a large room, and could do around 5,000 additions per second.

A transistor does the same job as a tube, acts as a switch, but it's a solid speck of silicon with no moving parts and nothing to burn out. And crucially, it can be made *small*. Absurdly small. Modern transistors are a few nanometers across; tens of thousands of them would fit across the width of a human hair. A modern GPU packs on the order of a hundred billion of them onto a chip the size of your fingernail-to-palm.

What makes a transistor magical isn't that it's a switch. Your wall has switches. It's *who flips it*:

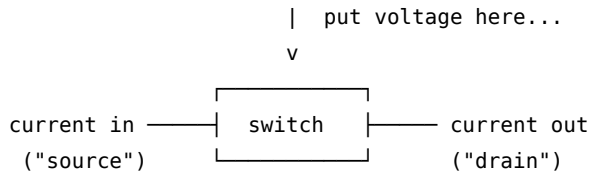
A transistor is a switch that is flipped by electricity, which means switches can flip other switches.

Read that again. A wall switch needs a human finger. A transistor needs only a voltage on its control wire. So the output of one switch can be the control of another switch. Switches controlling switches controlling switches, that's a chain reaction of logic, and it needs no human in the loop. This one property is why computers can exist.

Visually

A transistor has three terminals. Think of it as a gate in a garden fence for electric current:

control wire ("gate")
|



Voltage ON the control → current flows (switch CLOSED, "1")
 Voltage OFF the control → no current flows (switch OPEN, "0")

We give the two states names: a wire carrying voltage is a **1**, a wire without voltage is a **0**. That's the entire origin of binary, which gets its own full treatment in Chapter 2. For now: ON = 1, OFF = 0.

What Problem Does It Solve?

The transistor solves the problem of **automation of logic**. Before it (and its ancestor the relay/tube), any decision, "if this, then that", required a human or a slow mechanical contraption. The transistor made decisions: electrically fast (a modern transistor can switch billions of times per second), microscopically small, essentially free to operate, and, critically, *composable*: outputs can drive inputs.

Composability is the deep idea. One switch is a toy. Switches that control switches are a *language*.

1.5 From Switches to Logic Gates

ELI5

If you wire a few switches together in clever patterns, you get little machines that make tiny decisions. These are called **logic gates**. A gate looks at one or two input wires (each carrying a 1 or a 0) and produces one output wire (a 1 or a 0), according to a fixed rule.

There are three rules to start with, and they have friendly names:

- **AND**: "I output 1 only if *both* my inputs are 1."
- **OR**: "I output 1 if *at least one* of my inputs is 1."
- **NOT**: "I output the opposite of my input."

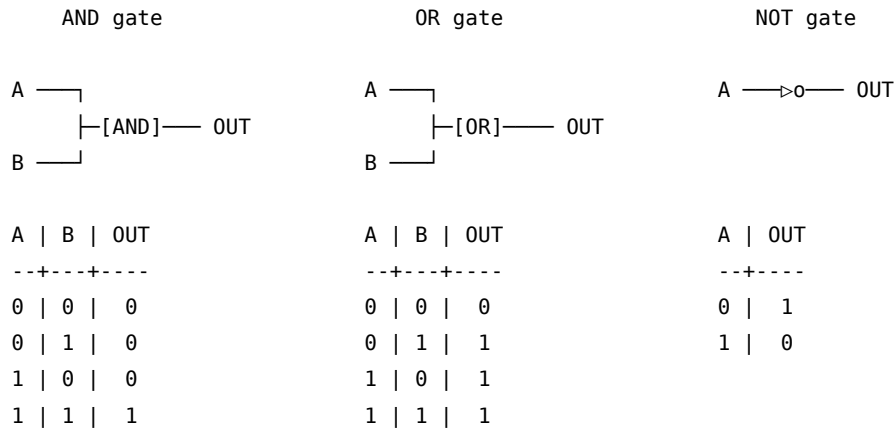
Analogy

Think of a nightclub with two bouncers:

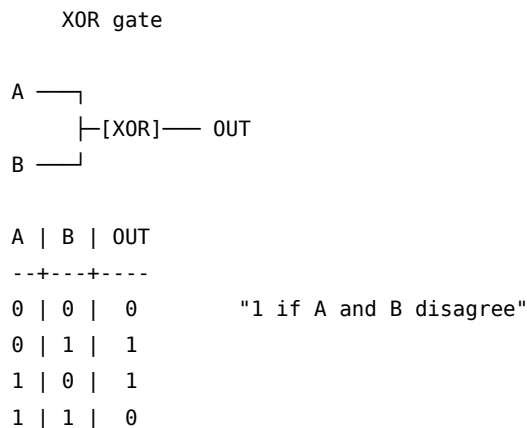
- An **AND** gate is a door with two bouncers where *both* must nod to let you in. Strict.
- An **OR** gate is a door where *either* bouncer nodding gets you in. Lenient.
- A **NOT** gate is a contrarian who says the opposite of whatever they're told. Told "yes," says "no."

Visually

We draw gates like this, and we describe their complete behavior with a **truth table**, a table listing every possible input combination and the resulting output. Truth tables matter because they are *complete*: two circuits with the same truth table are interchangeable, no matter how differently they're built. (File that thought away, "same behavior, different implementation" is the legal foundation of every compiler optimization you will ever write.)

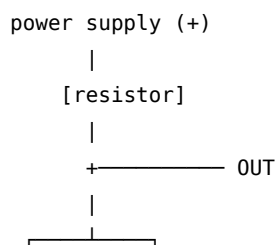


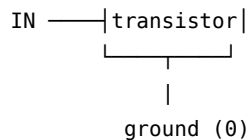
One more gate earns a seat at the table, because arithmetic will need it. **XOR** ("exclusive or") outputs 1 if its inputs are *different*:



How Are Gates Built From Transistors?

Here is a NOT gate, the simplest, so you can see there's no magic. In real chips we use a technology called CMOS, which pairs two complementary kinds of transistor, but the essence is one transistor and a resistor:





IN = 1 → transistor conducts → OUT is drained to ground → OUT = 0
 IN = 0 → transistor blocks → OUT stays at supply → OUT = 1

An AND gate takes about 6 transistors in CMOS; XOR takes about 8-12 depending on the design. The details are electrical engineering and we won't need them again. What we need is the punchline:

Transistors → gates. Gates are the atoms of computation. From here up, we never think about electrons again. We think in 1s, 0s, and gates.

You've just climbed the first rung of the most important ladder in computing, the ladder of abstraction. We'll name it properly in Section 1.10.

A Delicious Mathematical Fact

How many possible two-input gates could exist? A two-input gate is just a truth table with 4 rows, and each row's output can independently be 0 or 1. So there are $2^4 = \mathbf{16}$ **possible two-input gates**. That's it. That is the entire universe of two-input logic. AND, OR, XOR are three of the sixteen; the others include NAND (NOT-AND), NOR, "always 0," "ignore B and output A," and so on.

And here's a fact that delights everyone who meets it: **one single gate type, NAND, can build all the others**, and therefore can build an entire computer. Every CPU and GPU on Earth could, in principle, be built from one gate design repeated billions of times. (You'll prove a piece of this yourself in Exercise 4.)

1.6 Making Gates Do Math: The Adder

Now the first real payoff. We'll wire gates together into a circuit that performs *addition*, the workhorse operation of all computing, and (as you'll see in Parts V-IX) essentially the only thing AI hardware does all day, at a rate of quadrillions per second.

ELI5

We want a circuit where you put in two numbers and out comes their sum. We'll start laughably small: adding two *one-digit binary numbers*. Each input is a single bit, 0 or 1.

There are only four possible problems this circuit will ever face:

0 + 0 = 0
 0 + 1 = 1
 1 + 0 = 1
 1 + 1 = 2 ...uh oh.

“2” doesn’t fit in a single binary digit, binary digits only go 0, 1. Just like 7+5 in grade school, where you write 2 and *carry the 1*, binary 1+1 is written as 10: sum digit 0, carry 1. (Chapter 2 makes binary counting second nature; for today, trust the table.)

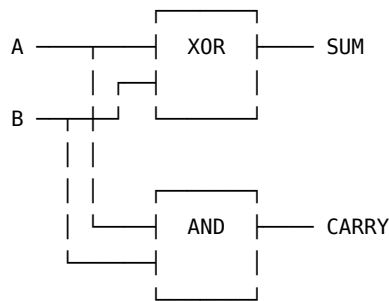
So our circuit needs **two outputs**: a SUM bit and a CARRY bit.

The Discovery

Write out what we need as a truth table, then stare at it:

A	B	CARRY	SUM
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Look at the CARRY column: it’s 1 only when both inputs are 1. *That’s exactly the AND gate*. Now the SUM column: it’s 1 when the inputs differ. *That’s exactly XOR*. We don’t need to invent anything. Addition was hiding inside the gates all along:

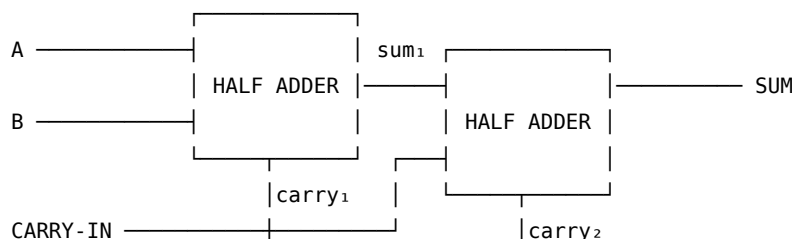


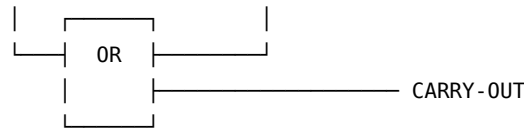
THE HALF ADDER: two gates that do arithmetic.

This circuit is called a **half adder**. Take a moment. Two bouncer-logic gates, wired side by side, and *arithmetic emerges*. This is the first miracle of computer science: **math is logic in disguise**. (George Boole suspected this in 1854; Claude Shannon nailed it to circuits in his 1937 master’s thesis, arguably the most consequential master’s thesis ever written.)

From Half to Full: Step-by-Step

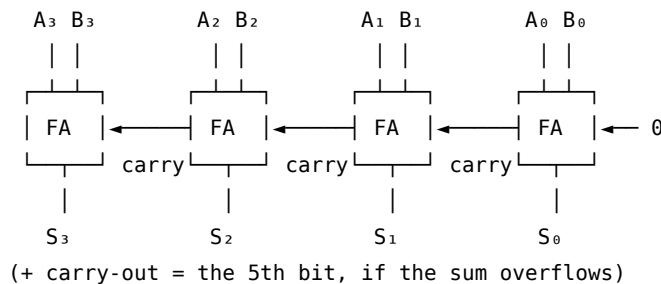
Why “half”? Because when we add multi-digit numbers, middle digits must accept a *carry from the previous column*, grade-school addition again. A circuit that adds three bits (A, B, and carry-in) is a **full adder**, and it’s just two half adders plus an OR:





In words: add A and B (first half adder), then add the incoming carry to that result (second half adder). A carry-out happens if *either* stage produced a carry, hence the OR. (Both can't carry at once, Exercise 5 asks you to prove it.)

Now the grand construction. Chain full adders, each column's carry-out feeding the next column's carry-in, exactly like grade school, and you can add numbers of any size. Here is a 4-bit **ripple-carry adder** (the carry "ripples" left, like a wave):



A 64-bit adder is this picture, 64 columns wide, around 1,000 transistors. Your laptop's CPU contains many dozens of adders. An H100 GPU contains, in effect, *hundreds of thousands* of adder-like circuits running simultaneously. When you hear "the GPU does a quadrillion operations per second," picture fields of these gate-chains, all rippling at once.

The Performance Seed (Plant It Now, Harvest It All Book)

Notice something subtle about the ripple-carry adder: column 3 *cannot finish* until column 2's carry arrives, which waits on column 1, which waits on column 0. The gates are all fast, but they're stuck in a *dependency chain*, a bucket brigade where each bucket waits for the previous one. A 64-bit ripple adder is ~64 gate-delays long even though it's only using 2-3 gate-delays of "real work" per column.

Hardware designers fixed this with cleverer circuits (carry-lookahead adders) that compute carries in parallel. But savor the shape of the problem, because you will meet it at every single layer of this book: **the enemy of speed is rarely the amount of work, it's the chain of waiting.** Dependency chains will reappear as pipeline stalls (Ch. 4), memory latency (Ch. 5), sequential bottlenecks in parallel algorithms (Ch. 33), and the reason attention kernels are hard to make fast (Ch. 52). Feynman's room knew it too: the assembly line is only as fast as its slowest, most-waited-on desk.

1.7 The Mathematics: Boolean Algebra

Time to put on slightly more formal glasses. Everything above has an elegant algebra underneath, invented by George Boole in 1854, almost a century before electronic computers, as an attempt to write the "laws of thought" as equations.

The Setup

Boolean algebra is arithmetic in a universe with only two numbers: 0 and 1. We define three operations, matching our gates. Notation you'll see in the literature:

- AND: written $A \cdot B$ or $A \wedge B$ (behaves like multiplication: $1 \cdot 1 = 1$, anything $\cdot 0 = 0$)
- OR: written $A + B$ or $A \vee B$ (like addition, except $1 + 1 = 1$, the output can't exceed 1)
- NOT: written $\bar{A}$ or $\neg A$

A **Boolean function** takes n bits in, produces bits out: mathematically, a function $f: \{0,1\}^n \rightarrow \{0,1\}^m$. Our half adder is a function from $\{0,1\}^2 \rightarrow \{0,1\}^2$. Truth tables are simply the complete listing of such a function, possible only because the input space is finite (2^n rows).

The Laws

These are the rewrite rules of the two-valued universe. Each is easy to verify by checking all input combinations:

Identity:	$A \cdot 1 = A$	$A + 0 = A$
Annihilation:	$A \cdot 0 = 0$	$A + 1 = 1$
Idempotence:	$A \cdot A = A$	$A + A = A$
Complement:	$A \cdot \bar{A} = 0$	$A + \bar{A} = 1$
Commutativity:	$A \cdot B = B \cdot A$	$A + B = B + A$
Associativity:	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C = A + (B + C)$
Distributivity:	$A \cdot (B + C) = A \cdot B + A \cdot C$	
De Morgan's:	$\neg(A \cdot B) = \bar{A} + \bar{B}$	$\neg(A + B) = \bar{A} \cdot \bar{B}$

De Morgan's laws are the celebrities: "NOT (A AND B)" is the same as "(NOT A) OR (NOT B)." In bouncer terms: failing a two-bouncer strict door means at least one bouncer rejected you. Obvious when said aloud; priceless when rewiring circuits.

Why a Compiler Book Cares About 170-Year-Old Algebra

Because these laws are **equalities**, and an equality is a license to *substitute*: anywhere the left side appears, you may install the right side instead, and *no observer can tell the difference*. If the right side is cheaper, fewer gates, fewer instructions, fewer nanoseconds, you just optimized for free.

That is a compiler's whole soul: **find provably-equal-but-cheaper rewrites, apply them, repeat**. Circuit designers use these laws to shrink transistor counts. LLVM uses richer algebraic rules to shrink instruction counts ($x * 2 \rightarrow x << 1$; $x + 0 \rightarrow x$, Chapter 21). ML compilers use tensor-level rules to shrink whole GPU workloads (fusing `add` into `matmul`, folding transposes, Chapter 39). Same move, three altitudes. Boolean algebra is where you learn the move with training wheels on.

Two little counting results worth keeping in your pocket:

1. **n wires can represent 2^n distinct states.** Ten switches: 1,024 states. 64 wires: 18.4 quintillion. This exponential is why fixed-size chunks of wires (Chapter 2's "words") can name astronomically many things.

2. **There are $2^{(2^n)}$ distinct Boolean functions of n inputs.** For $n=2$: 16 gates, as we counted. For $n=6$, it's $2^{64} \approx 1.8 \times 10^{19}$ possible functions. The design space explodes instantly, a first hint of why “search for the best circuit/kernel” (auto-tuning, Chapter 42) is genuinely hard.
-

1.8 Remembering Things: A One-Paragraph Preview

Gates as described are *forgetful*: change the inputs and the output changes instantly; nothing is stored. A computer also needs to *remember*. The trick, one of the loveliest in engineering, is **feedback**: wire two gates so each one's output feeds the other's input, and the pair locks into a stable state, holding a 1 or a 0 until deliberately changed. Such a loop is called a **latch** or **flip-flop**, and it is one bit of memory built from the same gates as everything else. Chain millions of them (or use denser cousins like DRAM cells, a capacitor holding charge like a leaky bucket) and you have the pantry. Chapter 3 is devoted entirely to memory; here we only needed to establish that it exists and is made of the same stuff. From now on, assume we have a big array of numbered shelves that hold numbers.

1.9 The Big Idea: The Stored Program

Everything so far, gates, adders, memory, existed in some form in the mechanical and electromechanical machines of the early 1940s. Now comes the idea that created *software*, and with it the entire modern world. It is the single most important idea in this book.

The Problem It Solved

ENIAC, the great room-sized calculator of 1945, had a dirty secret: to change what it computed, technicians **rewired it**. Programming ENIAC meant days of re-plugging cables and setting switches by hand. The machine could add 5,000 numbers a second, but changing its *mind* took a week. The recipe was built into the machine's body, like a music box that plays one tune, forever, because the tune *is* the drum's pins.

The Idea

In 1945, John von Neumann circulated a report (building on work by Eckert, Mauchly, and, conceptually, Alan Turing a decade earlier) proposing something that sounds almost too simple:

Store the instructions in the same memory as the data. Instructions are just numbers.

Sit with this. A recipe step like “add shelf 11's contents to the counter” gets encoded as a number, say, 211, and placed on a shelf *in the pantry itself*, right alongside the ingredients. The machine reads shelf 0, interprets the number found there as a command, obeys it, reads shelf 1, obeys, and so on.

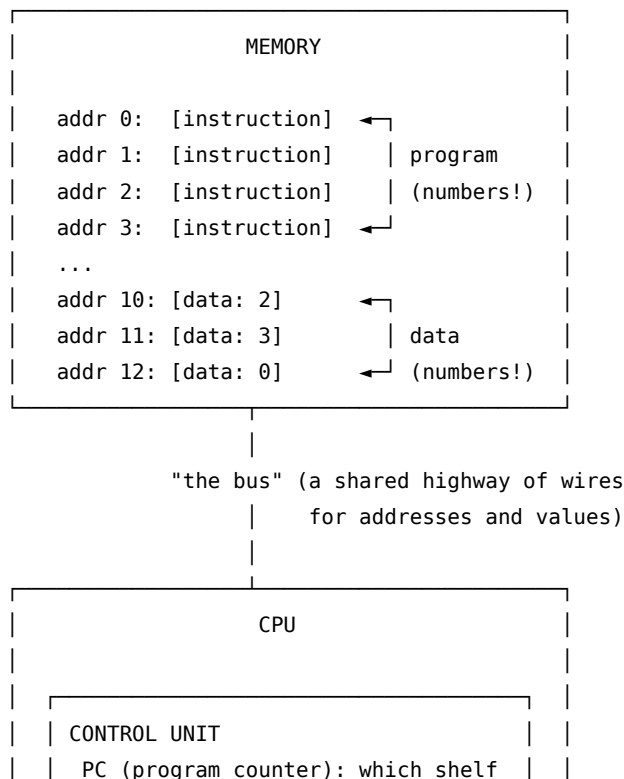
The consequences cascade fast, and each one is enormous:

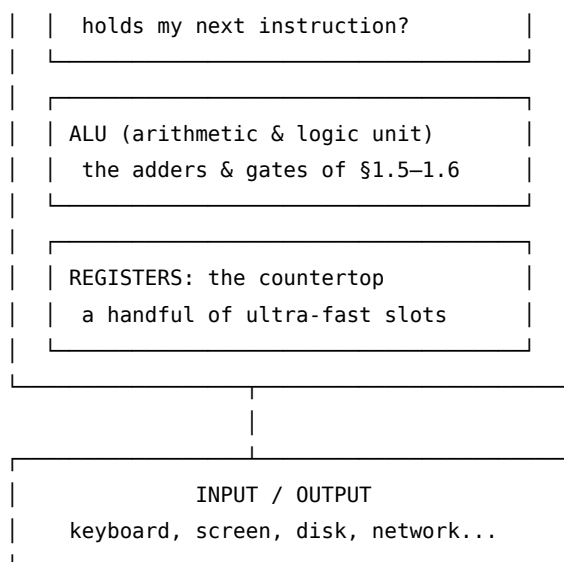
1. **Changing the program = changing memory contents.** No rewiring. Loading a new program takes milliseconds, not days. Software is born as something *soft*.
2. **Programs can be shipped, copied, and sold**, they're just numbers. Every app store, every `pip install`, every model checkpoint you'll ever download descends from this.
3. **Programs can read and write programs.** A program whose input is a program and whose output is a *better* program is called... a **compiler**. The tool this book teaches you to build is only possible because instructions are data.
4. **Programs can even modify themselves** while running, mostly a party trick today, but the principle powers JIT (just-in-time) compilers, which generate fresh machine code mid-execution. PyTorch 2's `torch.compile` (Chapter 45) does exactly this: it manufactures new instructions and places them in memory, and the machine gleefully runs numbers that didn't exist a second ago.

I want to be dramatic about this because it deserves drama: **the stored program concept is the reason “computer science” exists as a field.** Machines that calculate are engineering. Machines whose *behavior is data*, data that other machines can generate, analyze, transform, and optimize, that's a new kind of thing in the universe. When you write your first compiler pass in Part III, you'll be exercising the fourth consequence directly. You'll be the program that rewrites programs.

The Architecture, Drawn

The scheme von Neumann described is still, three-quarters of a century later, the blueprint of essentially every processor on Earth, your laptop's CPU, your phone's chip, and (with a twist we'll meet in Part VI: multiply everything by thousands) every GPU:





Map it back to the kitchen: memory is the pantry (one big one, holding both recipe cards and ingredients, that’s the von Neumann twist), the ALU is the chef’s hands, the registers are the countertop, and the control unit is the chef’s eyes reading the recipe, with the **program counter (PC)** as the finger keeping their place on the card.

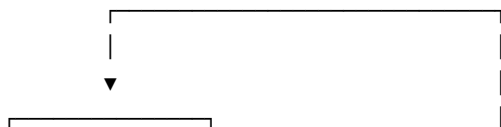
The Famous Flaw: The von Neumann Bottleneck

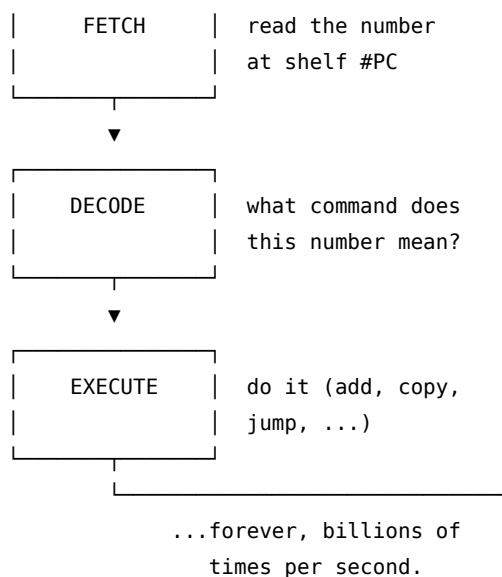
One highway (the bus) connects CPU and memory, and *everything*, instructions and data, travels on it. The CPU can compute far faster than the highway can deliver, so the processor spends much of its life waiting, like a chef drumming their fingers while an intern jogs to the pantry. John Backus (inventor of Fortran) named this the **von Neumann bottleneck** in his 1977 Turing Award lecture, and grumbled that we’d built our whole industry around “pushing vast numbers of words back and forth” through a straw.

He was right, and it got worse: since then, compute speed has improved orders of magnitude faster than memory speed. The entire memory hierarchy (Chapter 5), GPU shared memory (Chapter 35), operator fusion (Chapter 39), and FlashAttention (Chapter 52) are all, at heart, schemes for smuggling work around this one bottleneck. It is not an exaggeration to say **a large fraction of ML compiler engineering is a 75-year argument with this diagram.**

1.10 The Heartbeat: Fetch, Decode, Execute

So memory holds numbers, some of which are instructions. How does the machine actually *run*? With one loop. The simplest, most important loop in all of computing, the machine’s heartbeat:





Fetch: read memory at the address in the PC. **Decode:** split the fetched number into “which operation?” and “operating on what?”. **Execute:** fire the right circuit, the adder, the memory port, or (for jumps) overwrite the PC itself, which is how loops and decisions happen. Then advance the PC and go again. A 3 GHz processor sings this song roughly three billion times per second. It is the only song it knows.

Let’s Build a Machine: TOY-100

Abstract explanations breed fuzzy understanding, and this book doesn’t deal in fuzz. So we now define a complete, real (if tiny) computer, and by the end of the chapter you’ll have built and programmed it yourself. We’ll call it **TOY-100**, and we will keep it as a pet for the whole book, feeding it new abilities in Parts II and III until, in Part XI, we build a compiler that targets it.

The design (savor how little a “computer” requires):

- **Memory:** 100 shelves, addresses 0–99, each holding one number.
- **One register:** the accumulator, **ACC**, a countertop with room for exactly one number.
- **The PC:** which shelf to read the next instruction from. Starts at 0.
- **Instruction format:** each instruction is a 3-digit number. First digit = opcode (which command), last two digits = operand (which address). So 211 decodes as opcode 2, operand 11.

The eight commands (the **instruction set architecture**, or ISA, the machine’s complete vocabulary):

opcode	name	meaning
0	HALT	stop the machine
1	LOAD a	ACC ← memory[a] (pantry → countertop)
2	ADD a	ACC ← ACC + memory[a]
3	SUB a	ACC ← ACC – memory[a]
4	STORE a	memory[a] ← ACC (countertop → pantry)

```

5    JUMP a  PC ← a          (jump to step a)
6    JZ   a  if ACC == 0: PC ← a  (the machine's only "if"!)
7    PRINT a  print memory[a]

```

Eight commands. That’s the whole vocabulary, and it’s genuinely enough to compute anything computable, given time and memory. (This shocking claim is called Turing completeness; we’ll touch it in Chapter 12.) Real ISAs like x86 have thousands of instructions, but they’re conveniences, not necessities.

A Program, and a Hand Trace

Here’s a complete TOY-100 program that adds $2 + 3$ and prints the result. The left column is the shelf address; the middle is the number actually stored there; the right is what it means:

addr	contents	meaning
0	110	LOAD 10 (ACC ← mem[10])
1	211	ADD 11 (ACC ← ACC + mem[11])
2	412	STORE 12 (mem[12] ← ACC)
3	712	PRINT 12
4	000	HALT
...		
10	2	← data: first number
11	3	← data: second number
12	0	← data: room for the answer

Notice: shelf 0 holds 110 and shelf 10 holds 2, and *there is nothing marking one as “instruction” and the other as “data.”* They’re both just numbers on shelves. Only the PC’s path through memory makes some numbers instructions. That’s the stored-program idea made concrete.

Now trace it, slowly, by hand, the way you’d never let yourself do again but must do once:

step	PC	fetches	decoded	effect	ACC	mem[12]
1	0	110	LOAD 10	ACC ← mem[10] = 2	2	0
2	1	211	ADD 11	ACC ← 2 + mem[11] = 5	5	0
3	2	412	STORE 12	mem[12] ← 5	5	5
4	3	712	PRINT 12	prints "5"	5	5
5	4	000	HALT	machine stops	5	5

Congratulations, you just executed a program by hand, doing exactly what the control unit’s gates do. Every program you have ever run, your browser, a game, GPT generating text, is this table, with more rows. A large language model producing one word of output is this table with a few hundred billion rows, mostly MULTIPLY and ADD. There is no other magic. **This is the whole trick, done quickly.**

Where’s the “If”? Where’s the Loop?

Two of the eight instructions deserve special ceremony. JUMP and JZ write into the *PC itself*, they steer the machine’s attention. JZ (“jump if zero”) is the machine’s entire capacity for

decision: every if statement, every while loop, every branching choice in every program ultimately compiles down to some flavor of “compare, then conditionally jump.” Here’s a countdown loop, 5-4-3-2-1, to show a program that revisits its own steps:

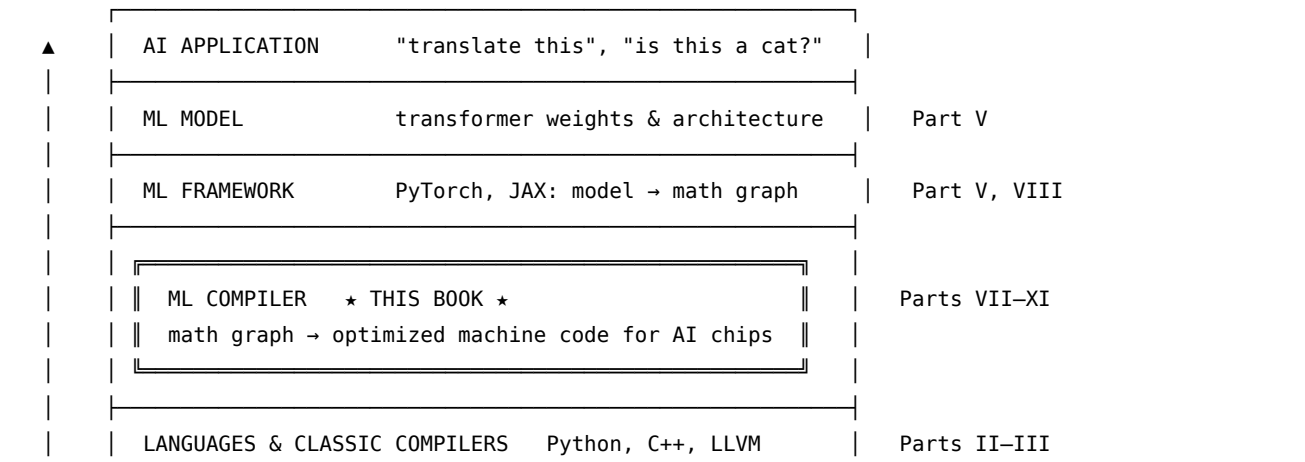
addr	contents	meaning
0	110	LOAD 10 ACC ← counter
1	608	JZ 8 if counter hit 0 → addr 8 (done)
2	410	STORE 10 save counter
3	710	PRINT 10 print it
4	110	LOAD 10
5	311	SUB 11 counter minus 1
6	410	STORE 10 save it back
7	500	JUMP 0 ...and around again
8	000	HALT
10	5	← the counter
11	1	← the constant 1

Trace three iterations of this yourself with a pencil (Exercise 2 makes it official). Feel how JUMP 0 at address 7 throws the PC backward, and how JZ 8 is the escape hatch that keeps it from looping forever. When you can predict this program’s every step, you understand program execution better than most working programmers, not because they couldn’t, but because they’ve never sat with it.

And a small aside that will pay off in Part III: look how much ceremony the loop needs, load, store, load, store, juggling one lonely accumulator. High-level languages exist to hide this ceremony; *compilers* exist to translate the pleasant version back into this, ideally with less ceremony than a human would write. The gap between “what you meant” and “what the machine needs” is the space every compiler lives in.

1.11 The Ladder of Abstraction: The Map of This Entire Book

We’ve climbed from electrons to running programs in ten sections. We’ll pause on the landing, turn around, and look at the staircase, because the staircase itself is the most important mental model in computing, and it is literally the map of this book.



ISA / ASSEMBLY	x86, ARM, PTX: the machine's vocab	Ch. 4, 34
MICROARCHITECTURE	pipelines, caches, schedulers	Ch. 4–5, 33–36
LOGIC GATES	AND, OR, NOT, adders	← you are here
TRANSISTORS	switches that flip switches	← and here
PHYSICS	electrons in silicon	

Three rules govern life on this ladder. They deserve frames on the wall:

Rule 1, Each layer only needs the layer below. The gate designer thinks in transistors, never electrons. The CPU designer thinks in gates, never transistors. You just programmed TOY-100 thinking in instructions, never gates, notice that you *could*, and that it felt natural. This is the only known way humans manage systems with 10^{11} parts: nobody understands the whole thing; everybody understands one floor and trusts the floor below. Abstraction is civilization’s best trick, and computing is its purest expression.

Rule 2, Each layer is a *language*, and the borders between layers are *translators*. A PyTorch model is a sentence in the “tensor math” language. The machine speaks only the “instructions” language. Something must translate, and every translator on the ladder is called a compiler (or its sibling, an interpreter, Chapter 13 draws that line). ML compilers are the translators for the strangest border: from “mathematics on billion-element arrays” down to “keep 100,000 tiny adders fed without tripping over the von Neumann bottleneck.”

Rule 3, Layers leak, and performance work is the study of the leaks. Correctness respects the floors; *speed does not*. Why is one PyTorch program $10\times$ faster than another that computes the same thing? The answer is never on the PyTorch floor, it’s three floors down, in cache behavior or warp scheduling, whispering up through the ceiling. A framework user can ignore the leaks. A performance engineer cannot. **An ML compiler engineer is a person who is comfortable on every floor of this building**, who can ride the elevator from “attention layer” to “memory coalescing” and back without getting dizzy. That’s why this book insists on starting from transistors when its title says ML Compilers: we are not taking a detour; we are surveying the building you’ll be working in.

Keep this diagram bookmarked. Every subsequent chapter is me pointing at one floor and saying: this one, now, in detail.

1.12 Engineering Perspective: The Real Thing

TOY-100 is honest but tiny. We’ll calibrate against the machines this book ultimately targets, so you know how much the picture scales without changing shape.

The chips. A high-end CPU (say, an AMD EPYC or Apple M-series) packs tens of billions

of transistors; NVIDIA's H100 GPU, the workhorse of the current AI boom, has ~80 billion; Apple's M2 Ultra, ~134 billion. At ~5 nanometer feature sizes, transistors are so small that ~20,000 of them fit across a human hair's width. They are manufactured, mostly by one company, TSMC in Taiwan, using lithography machines built by one company, ASML in the Netherlands, each machine costing more than a Boeing 787, by projecting patterns of light onto silicon wafers, layer by layer, like silk-screening a t-shirt 60 times with atomic precision. A modern fab costs \$20+ billion. This is, by most reckonings, the most precise manufacturing humanity does.

The clock. All those gates march to a metronome, the **clock**, ticking ~3-5 billion times per second (3-5 GHz) on CPUs, ~1-2 GHz on GPUs. Each tick, every circuit settles into its next state: fetch, decode, execute at industrial scale. Chapter 4 reveals the tricks (pipelining! Feynman's assembly line, in silicon) that let one tick retire *multiple* instructions.

The scale of the vocabulary. TOY-100 has 8 instructions; x86 has ~1,500+ variants; ARM, hundreds; NVIDIA GPUs speak PTX and SASS (Chapter 34). But every one of them is still a number in memory, fetched-decoded-executed. Trace table's the same; it just has more columns.

And the same trinity everywhere. Memory, processor, program. An H100 is: 80 GB of memory (pantry), ~16,000 arithmetic lanes organized into 132 "streaming multiprocessors" (a vast brigade of chefs), running kernels (recipes), with a von Neumann bottleneck so severe that feeding the chefs, not cooking, is the central engineering problem. You already have the concepts; Part VI just adds zeros.

1.13 Performance Analysis: Learning to Think in Nanoseconds

This is a performance book wearing a compiler book's clothing, so let's start training the reflex now: **always know the numbers.**

How Fast Is Fast?

A 3 GHz clock means one tick every **0.33 nanoseconds**. A nanosecond is a billionth of a second, for humans, an unimaginable sliver, so use this yardstick instead: **in one nanosecond, light travels about 30 centimeters**. One foot. A ruler on your desk.

Now the fact that makes chip design a branch of geography: in one 0.33 ns clock tick, light travels ~10 cm, and electrical signals in silicon are slower, managing only a few centimeters. **A signal cannot cross a large chip in one tick.** Distance on the chip is time. This is why chips can't just be made bigger, why "make the clock faster" hit a wall in the mid-2000s (plus heat, power grows roughly with the cube of clock speed), and why the industry's answer to "how do we go faster?" pivoted from *faster* to *more*: more cores, more parallel lanes, which handed the difficulty to software, which is why your career (this book) exists. When someone asks why we can't just build a 100 GHz CPU instead of wrestling with parallelism: physics said no, around 2005, permanently.

The Trillion-Fold Ascent

computer	additions/second	vs. previous
Human with pencil	~0.1	—
Los Alamos room (1944)	~10s, pipelined	~100×
ENIAC (1945)	~5,000	~500×
Cray-1 supercomputer (1976)	~160 million	~30,000×
Laptop CPU (2020s)	~100 billion	~1,000×
NVIDIA H100 GPU (2020s)	~10 ¹⁵ (a quadrillion)	~10,000×

From Feynman’s room to an H100 is a factor of about **10¹⁴**, a hundred trillion. If walking speed had improved that much, you could stroll from Earth to the Sun in about a minute. This is why “dumb steps, but fast” wins: no amount of cleverness in a slow medium competes with adequate cleverness executed 10¹⁴ times faster.

The Two Numbers That Rule Everything: Latency and Throughput

Feynman’s room teaches the distinction. Ask: *how long does one calculation take, start to finish?* That’s **latency**, maybe hours, even in the optimized room, because hundreds of small steps happen in sequence. Now ask: *how many calculations finish per day?* That’s **throughput**, and the assembly line made it enormous even though each individual problem still crawled.

- **Latency** = time for one thing. (One customer’s meal: 20 minutes.)
- **Throughput** = things per unit time. (Kitchen output: 200 meals/hour.)

They are *not* reciprocals of each other, they respond to different medicine (parallelism boosts throughput but does nothing for a single item’s latency), and confusing them is the most common performance-reasoning error in industry. The distinction will chase us everywhere: a GPU is a low-latency-be-damned throughput monster; interactive AI chat wants low latency per token; batch inference wants throughput; serving systems (Chapter 55) walk the tightrope between them daily.

Amdahl’s Whisper

One more seed. In the room, suppose card-sorting between desks takes 10% of total time, and you parallelize the *arithmetic* infinitely, infinite clerks, free multiplication. Your maximum speedup is 10×, forever, because the sorting remains. The unimprovable part sets the ceiling: that’s **Amdahl’s Law** (formally in Chapter 7: $\text{speedup} \leq 1/(\text{serial fraction})$). It’s why profiling (measuring where time actually goes) precedes optimizing in every competent performance effort, and why “we made the matmuls 2× faster but the model isn’t faster” is a sentence you will someday say, understand, and fix.

1.14 Optimization: The Three Big Ideas (Already!)

It may seem premature to discuss optimization in Chapter 1. It isn’t, because at this altitude you can see something that’s invisible from inside the weeds: **there are only**

about three big ideas in all of performance engineering, and Feynman’s room already used every one. The rest of this book is these three ideas wearing forty different costumes.

Idea 1, Parallelism: more workers. The room ran multiple problems as interleaved card colors through many human calculators at once. Costumes it will wear later: multicore CPUs (Ch. 6), SIMD vector units (Ch. 22), the GPU’s tens of thousands of lanes (Ch. 33–34), tensor/pipeline/expert parallelism across whole datacenters (Ch. 56). Limit: Amdahl’s ceiling, plus coordination costs, even in 1944, someone had to sort the cards.

Idea 2, Pipelining: never let a worker idle. The multiplication desk shouldn’t wait for a problem to *finish*; the moment card #1 moves on, card #2 arrives. Latency per problem: unchanged. Throughput: multiplied by the number of stages. Costumes: CPU instruction pipelines (Ch. 4), GPU memory-latency hiding via warp swapping (Ch. 34, 36), software pipelining in kernels (Ch. 54), pipeline-parallel training (Ch. 56).

Idea 3, Specialization: shape the worker to the work. The person who *only multiplies* gets scarily good at multiplying, and a circuit that only multiplies 16×16 matrices beats a general processor at that job by 10–100 $\times$ in speed and energy. Costumes: the GPU itself (specialized for throughput), tensor cores (Ch. 36, matrix-multiply organs), TPUs, and every “AI accelerator” startup. The price of specialization is rigidity, and **that price is paid in software, by compilers.** The more specialized the hardware, the harder the translation from flexible human intent, and the more valuable the translator. Specialization is, quite directly, why ML compiler engineers are scarce and expensive.

And braided through all three, the meta-idea we planted in the kitchen: **feed the workers, data movement, not arithmetic, is usually the real cost.** Fusion, tiling, memory planning, FlashAttention: pantry logistics, all of it.

When you meet any performance technique from now on, in this book or in the wild, run the checklist: *which of the three is this? And is it really about compute, or about feeding?* You’ll be startled how completely those two questions dissect the field.

1.15 Build It: Gates and Adders in Python

Feynman’s blackboard rule is this book’s rule: what we cannot create, we do not understand. So we now create everything from this chapter, in Python. (If you’ve never written Python, follow along anyway, every line is explained in English, and Chapter 8 formally teaches programming. You can return and re-run this later; it will feel like meeting an old friend.)

Simulating Gates

```
# gates.py – the atoms of computation, simulated

def NOT(a):
    """The contrarian: output the opposite of the input."""
    if a == 1:
        return 0
```

```

    else:
        return 1

def AND(a, b):
    """The strict door: 1 only if BOTH inputs are 1."""
    if a == 1 and b == 1:
        return 1
    else:
        return 0

def OR(a, b):
    """The lenient door: 1 if AT LEAST ONE input is 1."""
    if a == 1 or b == 1:
        return 1
    else:
        return 0

def XOR(a, b):
    """The difference detector: 1 if the inputs DISAGREE.
    Built from the other gates to prove we can:
    XOR(a,b) = (a OR b) AND NOT(a AND b) - 'at least one, but not both'."""
    return AND(OR(a, b), NOT(AND(a, b)))

```

Line-by-line, for complete beginners:

- `def NOT(a):`, `def` *defines* a reusable recipe (a **function**) named `NOT` that takes one input, which we'll call `a` inside the recipe. The colon and indentation mean "the following indented lines are the recipe's body."
- `if a == 1:`, a decision. `==` asks "are these equal?" (one = would *assign*, a classic beginner trap). Note the pleasing echo: Python's `if` will ultimately execute as some machine's conditional jump, a `JZ`, just like TOY-100's.
- `return 0`, "the answer is 0; recipe over." `return` hands a value back to whoever called the function.
- In `XOR`, we don't use `if` at all, we *compose* gates, feeding outputs into inputs, exactly as wires would: "(a OR b) AND NOT (a AND b)" is a direct transliteration of a real XOR circuit. Switches flipping switches, in software.

A dirty secret worth smiling at: Python's `and`, `or`, `not` mean we're using a computer's gates to pretend to be gates. Every simulation in this book has this Escher quality, we always build the thing out of the thing. That's not cheating; that's Rule 1 of the ladder (each layer builds on one below), plus the stored-program idea (behavior is data, so any machine can imitate any other). Sit with the vertigo until it becomes cozy; it's the field's defining sensation.

Verify Exhaustively

For 2-input gates, the input space is 4 rows, so we don't spot-check, we check *everything*. Remember this luxury and mourn it later: exhaustive verification dies of exponential explosion the moment inputs grow (64-bit adder: 2^{128} cases), which is why testing strategy (Chapter 61) is a real discipline.

```

print(" A  B | AND OR XOR")
for a in (0, 1):           # try both values of a...
    for b in (0, 1):       # ...and for each, both values of b
        print(f" {a}  {b} | {AND(a,b)}  {OR(a,b)}  {XOR(a,b)}")

```

The nested `for` loops enumerate all four combinations, (0,0), (0,1), (1,0), (1,1). The `f"..."` is an f-string: the curly-brace holes get filled with each value. Output:

```

A  B | AND OR XOR
0  0 |  0  0  0
0  1 |  0  1  1
1  0 |  0  1  1
1  1 |  1  1  0

```

Compare with §1.5's truth tables: identical. Our simulated universe agrees with the paper one.

The Adders

Now transliterate §1.6's circuits into code, notice the code *is* the circuit diagram, read as prose:

```

def half_adder(a, b):
    """Add two bits. Returns (sum_bit, carry_bit)."""
    s = XOR(a, b)           # the SUM wire   (1 if inputs differ)
    c = AND(a, b)           # the CARRY wire (1 only for 1+1)
    return s, c

def full_adder(a, b, carry_in):
    """Add three bits (two addends + incoming carry).
    Exactly the two-half-adders-plus-OR circuit of §1.6."""
    s1, c1 = half_adder(a, b)           # first half adder: a+b
    s2, c2 = half_adder(s1, carry_in)   # second: (a+b) + carry_in
    return s2, OR(c1, c2)               # carry out if EITHER stage carried

def add_bits(a_bits, b_bits):
    """Ripple-carry adder for equal-length bit lists (leftmost = biggest digit).
    Returns (sum_bits, overflow_bit)."""
    result = []
    carry = 0
    # Grade school: start from the RIGHTMOST column, hence reversed()
    for a, b in zip(reversed(a_bits), reversed(b_bits)):
        s, carry = full_adder(a, b, carry)  # this column's sum; carry ripples on
        result.append(s)
    result.reverse()                       # we built it right-to-left; flip it
    return result, carry

```

New Python in `add_bits`, explained: `[]` is an empty **list** (an ordered collection); `.append(x)` adds to its end. `reversed(...)` walks a list backwards. `zip(xs, ys)` pairs lists up element-by-element, so the loop sees one *column* of the addition at a time, rightmost column first, exactly like pencil-and-paper. The variable `carry` is the wave rippling leftward: each

column's carry-out becomes, on the next loop spin, the next column's carry-in.

Test, $3 + 5$ (binary 0011 and 0101, again, Chapter 2 drills this; today, trust):

```
s, overflow = add_bits([0,0,1,1], [0,1,0,1])
print(s, overflow)          # → [1, 0, 0, 0] 0
```

[1,0,0,0] is binary 1000 = eight. **$3 + 5 = 8$, computed by six simulated gates in a bucket brigade.** You have personally built arithmetic out of decisions. On a real chip this happens in a fraction of a nanosecond, sixty-four columns wide, millions of instances at once, but it is *this*, exactly this.

1.16 Mini Project: Build TOY-100 Itself

Now the chapter's summit: implement the whole computer, memory, PC, ACC, and the fetch-decode-execute heartbeat, in about 40 lines. This program is a **virtual machine** (VM): a machine made of software. (That phrase should now provoke a knowing nod instead of confusion: programs are data, so a program can *be* a machine. CPython, the JVM, and CUDA's PTX layer are this same idea at industrial strength, Chapters 13 and 34.)

```
# toy100.py – a complete computer in ~40 lines

# The vocabulary (ISA): opcode numbers from §1.10, given names for readability.
HALT, LOAD, ADD, SUB, STORE, JUMP, JZ, PRINT = 0, 1, 2, 3, 4, 5, 6, 7

def decode(instruction):
    """Split a 3-digit number into (opcode, operand): 211 → (2, 11)."""
    opcode = instruction // 100      # integer division: 211 // 100 = 2
    operand = instruction % 100      # remainder:          211 % 100 = 11
    return opcode, operand

def run(memory, trace=False):
    """The fetch-decode-execute heartbeat. `memory` is a list of 100 ints."""
    pc = 0                          # program counter: start at shelf 0
    acc = 0                         # the accumulator (countertop): starts empty

    while True:                    # the heartbeat loops until HALT breaks it
        instruction = memory[pc]    # — FETCH
        opcode, operand = decode(instruction)  # — DECODE
        if trace:
            print(f"pc={pc:2d}  instr={instruction:03d}  acc={acc}")
        pc += 1                    # advance the bookmark BEFORE executing
                                   # (so jumps can overwrite it cleanly)

        # — EXECUTE: exactly one of these fires
        if opcode == HALT: break
        elif opcode == LOAD: acc = memory[operand]
        elif opcode == ADD: acc = acc + memory[operand]
        elif opcode == SUB: acc = acc - memory[operand]
```

```

elif opcode == STORE: memory[operand] = acc
elif opcode == JUMP: pc = operand
elif opcode == JZ:
    if acc == 0:
        pc = operand
elif opcode == PRINT: print(memory[operand])
else:
    raise RuntimeError(f"Unknown opcode {opcode} at address {pc-1}")

return memory # hand back memory so tests can inspect it

```

The lines that repay staring:

- HALT, LOAD, ... = 0, 1, ..., names for the opcode numbers, nothing more. The machine only ever sees numbers; the names are for *us*. (Foreshadowing: an assembler, Chapter 15's warm-up, is a program that converts names to numbers so humans never write 211 again. You'll build one in Project B.)
- decode, pure arithmetic surgery: // chops off the last two digits, % keeps them. Real CPUs decode with gates that slice *binary* fields rather than decimal digits, but it's the same knife.
- pc += 1 **before** execute, the little design decision with real consequences: since the PC already points *past* the current instruction, a JUMP simply overwrites it and nothing fights. Real ISAs do the same (when x86 code reads its own instruction pointer mid-instruction, it sees the *next* address). You've hit your first piece of *semantics pedantry*, precise rules about tiny orderings, which is the daily bread of compiler correctness (Chapter 18 onward).
- while True: + break, the heartbeat has no natural end; only HALT stops it. Feed it a program without a reachable HALT and it runs forever, you'll engineer this disaster deliberately in Exercise 7, because a bug you've built on purpose is a bug you'll recognize forever after.
- The if/elif chain, this is the control unit: look at the opcode, fire one circuit. A real control unit is this chain *as hardware*, all comparisons in parallel.

Run the Programs

```

# --- 2 + 3, from §1.10 ---
memory = [0] * 100 # 100 shelves, all holding 0
memory[0] = 110 # LOAD 10
memory[1] = 211 # ADD 11
memory[2] = 412 # STORE 12
memory[3] = 712 # PRINT 12
memory[4] = 0 # HALT
memory[10] = 2 # data
memory[11] = 3 # data
run(memory) # prints: 5

# --- countdown 5..1, from §1.10 ---
memory = [0] * 100
program = [110, 608, 410, 710, 110, 311, 410, 500, 0]

```

```
memory[:len(program)] = program      # copy program to shelves 0..8
memory[10] = 5                       # the counter
memory[11] = 1                       # the constant 1
run(memory)                          # prints: 5 4 3 2 1 (one per line)
```

(`[0] * 100` builds a 100-zero list; `memory[:len(program)] = program` pastes the program over shelves 0–8. Loading a program *is* writing numbers into memory, the stored-program concept as a single line of code.)

Run the first one with `trace=True` and watch §1.10’s hand-trace table print itself, live, machine-generated. There is something quietly moving about that: the pencil trace and the running program agree, because they are the same computation at two altitudes.

Why This 40-Line Toy Is Load-Bearing for a 1,000-Page Book

Because it is the **target**. When Part III teaches compilation, TOY-100 is what we’ll compile *to*, you’ll write a compiler that turns $x = 2 + 3$ into `[110, 211, 412]` automatically. When Part XI builds an ML compiler, a grown-up TOY-100 with tensor instructions will be its backend. And when you meet real IRs, LLVM (Ch. 23), PTX (Ch. 34), they’ll feel familiar, because you’ll have owned a machine since Chapter 1. We didn’t build a toy; we built the *anchor*.

1.17 Debugging Guide: When the Machine Does the Wrong Thing

Debugging is the half of engineering that tutorials skip and professionals live in. We start the habit in Chapter 1 and never drop it. Three lessons, sized to what we’ve built:

Lesson 1, The machine is never wrong. TOY-100 (real hardware, too, to a first approximation) does *exactly* what the numbers in memory say. “The computer is misbehaving” means, with probability indistinguishable from 1, “my numbers don’t say what I believe they say.” This mindset shift, from *why is it doing that?!* to *what did I actually tell it?*, is 60% of debugging skill, acquirable today, on a 40-line machine.

Lesson 2, Make the invisible visible, then read what it says. Execution is invisible; that’s why bugs survive. Our `trace=True` is the antidote, and it is not a toy feature. It is this book’s first **observability tool**, ancestor of a noble line: `gdb` single-stepping (Ch. 14), LLVM’s `-print-after-all` showing IR between passes (Ch. 23), `TORCH_LOGS` dumping Inductor’s decisions (Ch. 45), Nsight tracing GPU kernels (Ch. 36). Same instinct every time: *don’t theorize about what the machine did, make it tell you*. A worked example: suppose you reorganize the countdown program and it starts printing `5 4 3 2 1 0`, one extra line. You could re-read the code and squint. Instead, run the trace: it will show you the exact step where `PRINT` fired with `mem[10] = 0`, and the PC value tells you which instruction did it, you’ll see immediately that your rearrangement put the `PRINT` *before* the `JZ` check instead of after it. The trace converts “somewhere in my program” into “address 3, step 31.” Shrinking the *where* is the whole game of debugging.

Lesson 3, Binary-search the timeline. When a long run goes wrong, don’t read 10,000

trace lines. Check the middle: state healthy? Bug's in the second half. Sick? First half. Repeat. Feynman's room did precisely this to salvage contaminated calculations, find the first bad card, recompute forward from there. $\log_2(10,000) \approx 14$ checks beats 10,000. This is the same divide-and-conquer that will find miscompiled functions via bisection in Chapter 61 (`llvm-reduce`, `pass bisection`), a professional technique, fully learnable on a toy.

And one professional reflex to install immediately: **when you find a bug, first write the smallest program that triggers it** (ours: three instructions?), *then* fix it, then keep the small program as a permanent test. Chapter 61 will name this regression testing; you can practice it in tonight's exercises.

1.18 Interview Questions

These are real questions asked in systems/new-grad interviews (and, in sharper forms, in ML systems interviews). Attempt each before reading the answer, the struggle is the workout; the answer is just the stretch after.

Q1. “Explain what a computer does, to someone non-technical, in under a minute.” (*Tests: whether your understanding survives without jargon to hide behind.*)

Strong answer: A computer is a machine that follows a recipe of extremely simple steps, copy a number, add two numbers, and if some number is zero, skip to a different step. What makes it useful isn't cleverness, it's speed: billions of steps per second, without fatigue or boredom. Everything impressive, video, games, AI, is billions of dumb steps arranged cleverly by humans. The intelligence is in the recipes.

Q2. “What is the stored-program concept and why did it matter?” Programs are stored in the same memory as data, an instruction is just a number on a shelf. Consequences: reprogramming = rewriting memory (seconds, not days of rewiring); programs become shippable, copyable data; and programs can process other programs, which is what makes compilers, interpreters, operating systems, and JITs possible at all. Strong finisher: it also created computing's biggest security headache (data smuggled into being treated as code, buffer overflows), and its biggest performance headache, the von Neumann bottleneck, since instructions and data share one memory highway.

Q3. “Walk me through fetch-decode-execute.” Loop forever: **Fetch** the number at the address in the program counter. **Decode** it, split into operation and operands. **Execute**, fire the matching circuit; ordinary instructions then advance the PC by one, while jumps write a new value into the PC, which is how ifs and loops exist. Bonus points: mention that modern CPUs run many iterations of this loop *overlapped* (pipelining) and even out of order, while carefully preserving the illusion of one-at-a-time execution, the “illusion” framing signals real understanding (and sets up Chapter 4).

Q4. “Why can't we just keep increasing clock speed instead of going parallel?” Two walls. **Heat**: power dissipation grows superlinearly with frequency (roughly $\propto V^2 f$, and higher f demands higher V), so past ~ 4 – 5 GHz air-cooled silicon literally can't shed the watts. **Distance**: at 3 GHz a signal travels only a few centimeters per tick in silicon, near chip scale, so faster clocks would mean signals can't cross the chip in a cycle. Both walls arrived ~ 2005 ; the industry's pivot was “more workers, not faster workers,” which moved the burden onto parallel software, and is the direct economic reason GPU

programming and ML compilers are careers.

Q5. “What’s the difference between latency and throughput? Give an example where improving one doesn’t improve the other.” Latency: time for one item, start to finish. Throughput: items per second. A pipeline (assembly line) multiplies throughput while leaving each item’s latency untouched, or slightly worse. Adding lanes to a highway raises throughput; it doesn’t get *one* car there faster. Sharp example for ML-flavored interviews: batching inference requests raises tokens/second served (throughput) while *increasing* each user’s time-to-response (latency), the central tension of Chapter 55.

Q6. “What is the von Neumann bottleneck?” CPU and memory share one channel; instructions and data compete for it; the processor computes far faster than the channel delivers, so it starves. Modern mitigations: caches (Ch. 5), prefetching, wide memory buses, and, at the ML-compiler level, *restructuring computation itself* to touch memory less: fusion, tiling, recomputation (Ch. 39, 52). If you can name FlashAttention as “a scheme to defeat the bottleneck for attention,” you’ve turned a definition question into a hire signal.

Q7. “One gate type can build any computer. Which, and roughly why?” NAND (NOR also works). Why: NOT is NAND with both inputs tied together; AND is NAND then NOT; OR falls out via De Morgan’s law from NOTted inputs. With AND/OR/NOT you can realize any truth table (write it as a sum-of-products), and with gates + feedback you get memory too, so NAND suffices for logic *and* storage. (You’ll verify the first steps in Exercise 4.)

Q8. (Systems-design flavored) “You have 1,000 clerks with calculators and a month of arithmetic. Organize them.” They’re asking you to reinvent Feynman’s room; do it knowingly. Decompose into small steps; specialize clerks (multipliers, adders, checkers); pipeline the flow so nobody idles; run independent problems in parallel (colored cards); add error-checking rows and recompute only contaminated work; watch for the bottleneck desk (profile!) and re-balance staff toward it; batch similar operations to amortize setup. Every clause maps onto a chapter of this book, which is rather the point of this book.

1.19 Common Mistakes and Misconceptions

“The computer understands English / Python / code.” It doesn’t, any more than a music box understands music. Understanding-shaped behavior is produced by layers of translation (the ladder!) ending in gate-flips. Practical consequence: when the machine “misunderstands” you, some translator on the ladder turned your intent into instructions you didn’t expect, and *that’s* where to look. Blame flows down the ladder, never sideways into mysticism.

“Computers are smart.” Inverted: they are stupendously dumb and stupendously fast, and the trade is worth it because $10^{14}\times$ speed beats cleverness. Engineers who forget this start expecting the machine to “figure out what I meant”, a category error that costs real debugging hours. (Yes, LLMs complicate the *feeling* of this statement. They don’t complicate the fact: an LLM is dumb steps, arranged by training into smart-shaped behavior, Part V.)

“Hardware and software are fundamentally different things.” They’re interchangeable costumes for computation: any circuit can be simulated in software (you did it today, TOY-100), any program can be baked into a circuit. What differs is the *trade-off*: hardware is fast and frozen; software is slow and fluid. Where to draw that line is a live billion-dollar question, it is literally the “should we build a custom chip for transformers?” debate (Part XII), and engineers who see the line as movable think better than those who see it as natural law.

“Memory and storage are the same thing.” In this book, **memory** (RAM) = the fast, volatile pantry the CPU works from; **storage** (disk/SSD) = the slow, permanent warehouse. Muddling them muddles every performance conversation you’ll ever have; Chapter 3 formalizes the difference (spoiler: $\sim 100,000\times$ latency gap).

“More GHz = faster computer” / “GPU beats CPU.” Neither, unconditionally. Clock speed is one factor among many (work per tick, core count, memory speed, Ch. 4–7). GPUs devastate CPUs on throughput-parallel work and *lose* on serial, branchy, latency-bound work (Ch. 33). “Faster at *what*?” is the only defensible phrasing. Interviewers deduct for the unconditioned claim.

“The trace/hand-simulation stuff is beneath me; I get the concept.” The concept isn’t the skill. Predicting *exactly* what a machine does, step by step, is the skill, the same muscle that later reads IR dumps, disassembly, and kernel traces without flinching. Every strong systems engineer I can point to does low-level traces *fluently*, and fluency comes only from reps. Do the pencil traces. Tonight. Nobody’s watching.

“ $1 + 1 = 2$, so my half adder should output 2.” A single wire can’t say “2”, it’s ON or OFF. Two-bit answers need two wires (carry & sum: 10). Sounds trivial; it is the seed of the deepest recurring theme in numerical computing: **representations have finite capacity, and arithmetic that overflows its container corrupts silently**. In Chapter 27 this exact issue, sums outgrowing FP16’s container, will be why loss-scaling exists and why your training run NaN’d at 3 a.m.

1.20 Exercises

Pencil-and-paper first, resist the urge to skip to code. (Estimated effort in parentheses.)

Exercise 1, Truth-table drills. (10 min) Write full truth tables for: (a) NAND (NOT of AND); (b) NOR; (c) the three-input majority function $\text{MAJ}(A,B,C) = 1$ when two or more inputs are 1. Then express MAJ using AND/OR, and check your expression against all 8 rows. (Majority gates are the carry logic of full adders, compare your table’s pattern to §1.6.)

Exercise 2, The pencil trace. (15 min) Hand-trace the countdown program of §1.10 in a table with columns: step, PC, fetched, decoded, ACC, mem[10], output. Run it to completion (≈ 40 steps). Check yourself against `run(memory, trace=True)`. If any row disagrees, find the *first* disagreeing row, Lesson 3’s timeline-bisection, in miniature.

Exercise 3, Reverse engineering. (15 min) Without running it, determine what this program does, then verify on your VM: shelves 0–6 = [110, 611, 211...], wait, no hints from formatting; here it is plainly: `mem[0..6] = [110, 605, 312, 410, 500, 710, 0]`, `mem[10] = 12`,

`mem[11] = 0, mem[12] = 3.` (Sneaky detail: what’s at shelf 11, and does JZ ever fire? What happens on shelf 10 over time, and does this program *terminate*?)

Exercise 4, NAND builds the world. (20 min) Using *only* NAND: build NOT (hint: feed the same signal to both inputs), then AND, then OR (hint: De Morgan, $\text{OR}(a,b) = \text{NAND}(\text{NOT } a, \text{NOT } b)$). Verify each with truth tables. You have now shown every circuit in this chapter, hence every computer, could be one gate design, repeated.

Exercise 5, A tiny proof. (10 min) In the full adder, prove the two half adders can never *both* produce $\text{carry} = 1$ (hence the OR never receives 1,1, some texts use XOR there instead; explain why both work). Hint: the first half adder’s carry is 1 only when $A=B=1$; what is its *sum* output in that case, and what does the second half adder then see?

Exercise 6, De Morgan on the street. (10 min) “The alarm fires unless the door is closed and the badge is valid.” Write the alarm’s Boolean expression two ways, as a NOT-of-AND and, via De Morgan, as an OR, and argue in plain English that they describe the same policy. (This tiny rewrite is precisely the kind a compiler makes when one form is cheaper on the target machine.)

Exercise 7, Build the disaster. (10 min) Write the shortest TOY-100 program that never halts. Then write one that never halts *for a subtler reason*, a JZ that can never fire because of arithmetic upstream. Run both, feel the hang, Ctrl-C, and reflect: from the *outside*, can you tell “still working” from “looping forever”? (You’ve just met the halting problem’s shadow; Chapter 12 introduces it properly, and it will constrain what optimizing compilers can ever promise.)

Coding Exercises

Coding 1, Multiply on a machine with no multiply. (30 min) TOY-100 has no MUL. Write a TOY-100 program (numbers in memory, not Python!) that computes `mem[10] × mem[11]` into `mem[12]` by repeated addition with a loop counter. Test with $6 \times 7 = 42$, and the edge cases 0×5 and 5×0 . (Every ISA omits something; synthesizing missing operations from present ones is a compiler’s daily chore, Chapter 22 calls it *instruction selection* and *legalization*.)

Coding 2, Extend the ISA. (20 min) Add opcode 8, `INC a` (`mem[a] ← mem[a] + 1`) to `toy100.py`: pick the encoding, extend `decode`’s reach if needed, add the `elif`, write a test. Then the interesting question, in one written paragraph: rewrite Coding 1’s multiply using `INC`, how many *fewer* instructions execute for 6×7 ? You’ve just done a hardware/software co-design trade study, the game Chapter 36’s tensor cores play at billion-dollar stakes.

Coding 3, A 16-bit adder and its limits. (20 min) Using `add_bits`, compute 40,000 + 30,000 with 16-bit inputs. The true answer (70,000) exceeds 16 bits’ capacity (65,535), inspect the overflow bit. Now *ignore* the overflow bit and decode the 16 sum bits you got: what wrong-but-tidy number appears? (Preview of Chapter 2’s modular arithmetic, the machine doesn’t crash on overflow; it *wraps*, silently, which is worse.)

Coding 4, Self-modifying code (advanced, optional). (30 min) Write a TOY-100 program that STOREs a computed value *into an address the program itself will later execute*, e.g., manufacture the instruction `710 + k` in ACC and store it ahead of the PC. Predict the behavior, then trace it. One paragraph: why do modern systems mark memory pages

“no-execute” to forbid exactly this in most software (hint: Q2’s security note), and why do JIT compilers (Ch. 13, 45) get a special pass?

Mini Projects

Project A, TOY-100 Deluxe. (2-4 hrs) Grow your VM: (1) a `step()` method + interactive mode, press Enter to advance one instruction, printing state (you’re building a debugger; Chapter 14’s gdb will feel like family); (2) breakpoints: `run(memory, break_at={3})` pauses when PC hits address 3; (3) a cycle counter reporting instructions executed, your first *profiler*, plus a per-opcode histogram (which instruction dominates the countdown program? multiply?); (4) guardrails: catch reads of nonsense instructions and out-of-range addresses with *helpful* messages naming the address and PC. (Error-message craftsmanship is compiler UX, Chapter 16 onward.)

Project B, The Assembler: your first translator. (3-5 hrs) Write `assemble(text) -> memory` turning this pleasant text into TOY-100 numbers:

```
        LOAD  counter
top:    JZ    done
        STORE counter
        PRINT counter
        LOAD  counter
        SUB   one
        STORE counter
        JUMP  top
done:   HALT
counter: 5
one:    1
```

Requirements: two passes (first pass records what address each label like `top:` lands on; second pass emits numbers, filling label references in, try writing it single-pass and discover *why* forward references force two passes); comments with `#`; errors that name the offending line. **Understand what you will have built: a translator from human-friendly notation to machine numbers.** A small compiler. The 1,000-page journey from here to “ML compiler” is this project, deepened, better source languages (Parts II-III), better targets (Part VI), and above all *optimization* between the two (everything else). You’ve started.

1.21 Chapter Summary

We began in a room in 1944 where “computer” still meant a person, and Feynman was inventing pipelining with colored index cards. We ended with you building a working computer in Python. The route: a transistor is a switch that other switches can flip, which makes logic *composable*; a handful of switch-patterns, AND, OR, NOT, XOR, are the atoms of decision; arithmetic was hiding inside logic all along (half adder = one XOR + one AND), and Boolean algebra’s rewrite laws are the original optimization license: replace anything with anything provably equal but cheaper. Feedback loops give gates memory; one shared memory holding *both* data and instructions-as-numbers, the

stored-program concept, turns behavior into data and makes compilers possible; and a three-beat heartbeat, fetch-decode-execute, animates it all, with conditional jumps as the machine's entire capacity for choice. Above this rises the ladder of abstraction, with ML compilers occupying one specific, strange floor: translating mathematics-on-tensors down to instruction streams for the most parallel machines ever built. And already visible from Chapter 1: the three big ideas of performance (parallelism, pipelining, specialization), the tyranny of data movement over arithmetic, latency versus throughput, Amdahl's ceiling, and the debugging creed, the machine is never wrong; make it show you what you actually said.

1.22 Key Takeaways

- **A computer is not smart; it is fast.** All intelligence lives in the recipes, and in the people (and compilers) that write them.
- **Composability is the miracle.** Switches flip switches → gates → adders → machines. Each layer speaks only to the next: the ladder of abstraction.
- **Math is logic in disguise.** SUM = XOR, CARRY = AND. Shannon's observation started everything.
- **Equality is a license to optimize.** Boolean laws, and every compiler rewrite descended from them, swap equals for cheaper equals. Correct-but-cheaper is the entire compiler business model.
- **Instructions are numbers.** Therefore programs are data; therefore programs can write, analyze, and improve programs; therefore compilers. The stored-program concept is the field's founding act.
- **The heartbeat is fetch-decode-execute**, and jumps that overwrite the PC are all the "choice" a machine has.
- **The bottleneck is the bus.** Arithmetic is cheap; moving data is dear. Most of ML-compiler engineering is pantry logistics against the von Neumann bottleneck.
- **Three ideas rule performance:** parallelism, pipelining, specialization, all invented by a room full of humans in 1944, all wearing silicon costumes now.
- **Latency ≠ throughput**, and Amdahl's Law caps every parallel dream at $1/(\text{serial fraction})$.
- **Debug by making the machine confess:** trace it, bisect the timeline, shrink the reproducer. The machine is never wrong.
- **You own a computer now.** TOY-100, 8 instructions, 100 shelves, 40 lines, is real, it's yours, and this book will grow it into an ML compiler's target.

What's Next

TOY-100's shelves held friendly decimal numbers, a small lie of convenience. Real machines store *everything*, numbers, text, instructions, images, and the 70 billion weights of a language model, as patterns of 1s and 0s, and the *choice of pattern* turns out to be a genuine engineering decision with billion-dollar consequences (ask anyone who's debugged an FP16 overflow). Chapter 2, **Binary: The Language of Machines**, teaches you to read, write, and *feel* binary: counting, negative numbers via a beautiful trick called two's complement, why $0.1 + 0.2 \neq 0.3$ on every computer on Earth, and the first honest look at the floating-point formats, FP32, FP16, BF16, whose trade-offs quietly govern all of modern AI.

End of Chapter 1. Next: Chapter 2, "Binary: The Language of Machines."



FROM SWITCHES TO SYSTEMS. FROM IDEAS TO IMPLEMENTATION. NOW YOU CAN BUILD A WORKING COMPUTER IN PYTHON.

