COE 530 Midterm Exam Study Guide

Quantum Computers and Architecture — Term 252 (Lectures 1–4, 7–11)

Built from Lectures 1–4, 7–11. **Excludes** any slide titled with a trailing asterisk ∗ (marked out-of-scope by the instructor). Course focus (syllabus CLOs 1–4): classical vs. quantum architecture differences; quantum circuits; qubit control; implementing basic algorithms.

Explicitly excluded by asterisk (don't study these beyond the single-line summary given):

- L2: Carry-Lookahead Adder details (P/G logic, CLA block), 1's complement representation/addition, 2's complement representation/computation/addition (as *representations*), Binary Multiplier circuit.
- L7: "What does that mean?" deeper Gottesman-Knill consequences bullet; "Basic Concept Beyond MPS" (SVD theory); SVD image-compression example.
- L11: Zhegalkin-polynomial → circuit algorithm, step-by-step oracle synthesis algorithm, Genetic-algorithm algorithm-discovery paper demo.

Still in scope (even if a neighboring slide had a ∗): the *circuit* adder/subtractor for 2's complement (the XOR-flip + C_0=1 trick), the *concept* that 1's/2's complement exist, Gottesman-Knill *theorem statement*, MPS definition/example, Zhegalkin basis functions & the key idea that `U_f` is block-diagonal of I/X blocks.

Lecture 1 — Classical Computer System I (History & Foundations)

History of Computing

- **Abacus (~1300 AD / 12th c.) → Analytical Engine (1830s, Babbage/Lovelace) → ENIAC (1945) → PCs (1990s) → HPC (Shaheen III, 2025)**.
- Mechanical calculators were specific-purpose; Babbage proposed the first **general-purpose** mechanical design.

Babbage's Difference Engine (1823)

- **Idea 1:** any continuous function can be *approximated* by a polynomial.
- **Idea 2:** any polynomial is computable from **difference tables** (successive differences become constant).
- Implication: **any continuous function can be implemented with an adder.**
- Example table: `f(n) = 41,43,47,53,61`; d_1 `= 2,4,6,8`; d_2 `= 2,2,2` (constant).

Analytical Engine

- Inspired by **Jacquard looms** (punched cards = program; thread color = data).
- First general-purpose (arguably Turing-complete) design; includes an **ALU**, **control flow** (conditional branches + loops), and **integrated memory**.

Theory of Computing

- **Boolean algebra** (George Boole): AND, OR, NOT via truth tables.
- **λ-calculus** (Alonzo Church, 1930s) → functional languages (LISP, Prolog, Scheme).
- **Turing machine** (Alan Turing): infinite tape + read/write head + state-transition rules.
  - Tape ↔ memory; head ↔ CPU; rules ↔ program.
- **Equivalence:** Church's λ-calculus and Turing's machine compute the same set of functions.
- **Von Neumann architecture:** CPU (Control Unit + ALU) + Memory + I/O.

Analog → Digital

- Analog: resistor networks add/average voltages continuously.
- Digital trick: don't measure precisely; ask "is it *high* (1) or *low* (0)?" → binary.
- Valid-level bands (e.g., ≥0.9 V = 1, ≤0.2 V = 0) make signals cleanly interpretable, communicable, and duplicable.

Electronic Computers

- **ENIAC (1943–45, Eckert & Mauchly, UPenn):** first fully-electronic general-purpose calculator — 30 t, 72 m^2, 200 kW; addition **200 μs**, division **6 ms**; 18 000 vacuum tubes; program was **external** (patch cables ⇒ rewiring to change program).
- **EDVAC (1944, + von Neumann):** solves ENIAC's rewiring problem via the **stored-program concept** — *"program = a sequence of instructions stored in memory as data."* Uses 4 000 tubes, 2 000 words of mercury-delay-line storage.
- **Vacuum tube:** cathode (heated, emits e^-) / anode (collects) / grid (controls); a small grid voltage gates a large anode current → logic or amplification. Downsides: bulky, power-hungry, unreliable.
- **Semiconductor diode:** p–n junction.
- **MOSFET:** Metal-Oxide-Semiconductor FET (gate, source, drain, channel); basis of all modern CPUs.
- **Moore's Law:** transistor count per IC ~doubles every 2 years; nearing atomic-scale limits. Feynman: *"There is plenty of room at the bottom."*
- **Intel evolution:** 386 (1985, 0.3 M) → Pentium 4 (2000, 42 M) → Core i7 Skylake (2015, 1.9 B). Process nodes: 600 nm (Pentium Pro, 1995) → Intel 7 (Alder Lake, 2022).

Levels of Abstraction

- **C/Python programmer** (high-level code) ↕ **Assembly programmer** (CPU = PC, registers, condition codes; Memory = code/data/stack) ↕ **Computer designer** (gates, clocks, layout).
- The parallel **1950s-stack vs. today vs. quantum stack** is a recurring slide:
  - 1950s: Algorithms → Assembly + Circuit synthesis → Vacuum tubes.
  - Today: Algorithms → High-level languages → Compiler & OS → Architecture → Transistors.
  - Quantum: Algorithms → Quantum DSL, Compilation, Unitary synthesis, Pulse shaping, Noise mitigation, Error correction → Qubits.

Conceptual Models of Computation

Turing machines, λ-calculus, **Circuit model**. All three are equivalent; **computer architecture follows the circuit model.**

Key takeaway

Modern computer architecture trade-off = **cost (area, components, $)** vs. **performance (speed, efficiency, accuracy)**.

Lecture 2 — Classical Computer System II (ALU)

ALU basics

- CPU executes machine instructions. The **ALU** is a **combinational digital circuit** that performs a pre-identified set of arithmetic/logical ops (ADD, SUB, MUL, AND, OR, ...).
- Inputs = **two operands A, B + an OP CODE**; output = **result R** (+ status flags V, C).

Binary Addition

- Represent decimals in binary: `11` = 1011_2, `14` = 1110_2. Add column-by-column with carry (`11 + 14 = 25` = 11001_2).
- Addition hierarchy: **Half-Adder → Full-Adder → Ripple-Carry Adder → Carry-Look-Ahead Adder** (CLA is ∗ — out of scope; just know it "reduces 12-gate-delay critical path to 8" by computing carries in parallel from $P_i{=}A_i{\oplus}B_i$, $G_i{=}A_iB_i$).

Half-Adder (HA)

- 2 inputs (X, Y); 2 outputs: **Sum `S = X ⊕ Y`**, **Carry `C = X·Y`**. Built from 1 XOR + 1 AND.

Boolean Gates (review)

- NOT `Z = X⁻`; OR `Z = X+Y`; AND `Z = XY`; XOR `Z = X ⊕ Y`.
- Can be built from transistors (example: AND via two NPN BJTs in series). A **7400 IC** packages four NAND gates in 14 pins.

Full-Adder (FA)

- 3 inputs (X, Y, Z=carry-in); 2 outputs.
  - `S = X⁻ȲZ + X⁻YZ⁻ + XȲZ⁻ + XYZ = X ⊕ Y ⊕ Z`
  - `C = XY + XȲZ + X⁻YZ = XY + Z(X ⊕ Y)`
- Implemented from **two half-adders + one OR gate**.

Ripple-Carry Adder (RCA)

- N full-adders chained by carries. 4-bit RCA adds $B_3B_2B_1B_0 + A_3A_2A_1A_0 \rightarrow C_4S_3S_2S_1S_0$.
- Critical (worst-case) path = `3 + 3 + 2(n−1)` gate delays if XOR = 3 delays; for n=4 → **12 gate delays** (this counting slide *is* marked ∗ so remember only the concept that delay grows ~linearly in n — CLA brings it down to 8 for n=4).

Adder / Subtractor (signed, 2's complement) — IN SCOPE

- One circuit computes **A + B or A − B** depending on a single control S.
- **S = 0 (add):** XORs pass B unchanged; $C_0 = 0$.
- **S = 1 (sub):** XORs flip B (⇒ 1's complement of B) and $C_0 = 1$ supplies the "+1" ⇒ the adder sees `A + B⁻ + 1 = A + (−B)` in 2's complement. **No correction step needed.**
- (The *representations* of 1's and 2's complement are ∗ — out of scope. You only need to know the circuit trick above.)

ALU design — 3-bit function code (IN SCOPE)

$F_2F_1F_0$	Op	Result
000	ADD	A + B
001	ADD+1	A + B + 1
010	SUB−1	A − B − 1
011	SUB	A − B
100	AND	A & B
101	OR	A | B
110	NOR	~(A | B)
111	XOR	A ^ B

- Built from: **n-bit adder** (with XOR gates on B that feed F_1; $F_0 \rightarrow C_0$), **per-bit AND / OR / XOR arrays**, a **4-to-1 MUX** selected by F_1F_0 (with an inverter on the OR line to make it NOR), and a final **2-to-1 MUX** selected by F_2 choosing between "arithmetic result" and "logic result".
- Produces status flags **V (overflow = C_{n} ⊕ C_{n−1})** and **C (carry-out)**.

Full-fledge ALU (context)

Adds MUL, DIV, POW, SQRT, NAND, XNOR, SHL, SHR, etc.; OP code widens to 4 bits. Multiplication circuits (Binary Multiplier examples) are ∗ — excluded.

Multiplexers (MUX)

- Selects 1 of 2^n inputs onto a single output using n select bits.
- **2→1:** $Y = \bar{S} \cdot I_0 + S \cdot I_1$ — view as a 1-to-2 decoder + enabling circuits + OR.
- **4→1:** $Y = I_0\bar{S}_1\bar{S}_0 + I_1\bar{S}_1S_0 + I_2S_1\bar{S}_0 + I_3S_1S_0$; build from a 2-to-4 decoder + four 2-input ANDs + a 4-input OR.
- MUX selection logic can be duplicated **m** times in parallel to form m-bit-wide data paths.

Bridge to quantum

- A quantum ALU is constructible: find quantum equivalents for NOT, AND, OR. Whether this is *useful* is debated later in the course (classical ops are reversible in quantum form via Toffoli/Fredkin — see L11).

Lecture 3 — Classical Computer System III: Memory Units + QRAM

Combinational vs. Sequential Circuits

- Combinational: `output = f(input)` (e.g., FA).
- Sequential: `output = f(input, previous_output)` — needed for memory.

Clock

- Periodic signal (train of 1s and 0s). Vocabulary:
  - **Positive pulse** (level = 1), **negative pulse** (level = 0).
  - **Rising edge** (0→1), **falling edge** (1→0).
- **Clock cycle (period)** in s / ms / μs / ns / ps. **Frequency** `f = 1 / T`.
- Example: $T = 0.5\ \text{ns} \Rightarrow f = 1/(0.5\times10^{-9}) = 2\times10^{9}\ \text{Hz} = 2\ \text{GHz}$.

Memory Elements

Latches, flip-flops, registers — differ in input count and how inputs affect state.

SR Latch

- Cross-coupled **NOR** gates. Inputs S (Set), R (Reset); outputs Q, $\bar{Q}$.
- Behavior: | S | R | Result | |—|—|—| | 1 | 0 | **Set** (Q=1) | | 0 | 1 | **Reset** (Q=0) | | 0 | 0 | **Store** (hold previous Q) | | 1 | 1 | **Invalid** (both outputs = 0; if both then drop to 0 together ⇒ **race condition** / oscillation / unknown state) |

Clocked SR Latch

- AND each input with a clock enable C. When `C=0`, the latch ignores S and R (holds state). When `C=1`, normal SR behavior.

D Latch

- Single data input D with $S = D$, $R = \bar{D}$ (gated by clock C) — **eliminates the undefined state.**
  - `C=0`: hold.
  - `C=1, D=0`: reset (Q=0).
  - `C=1, D=1`: set (Q=1).

Registers

- An **n-bit register** = n D-latches sharing a common clock and reset; loads n bits in parallel.
- **Register-enable:** add a 2-to-1 MUX per bit at the D input:
  - $D_i = \text{Enable} \cdot I_i + \overline{\text{Enable}} \cdot Q_i$ — only loads when Enable = 1.

Random-Access Memory (RAM)

- Large array of storage cells, addressable randomly.
- **k address bits ⇒ 2^k words**; each word is **n bits** ⇒ capacity = $2^k \times n$ bits.
- Two controls: **Read** (`Data_out ← Memory[Address]`) and **Write** (`Memory[Address] ← Data_in`).
- Worked example: 10-bit address × 16-bit data = $2^{10} \times 16 = 16$ Kbit = 2 KB (16 384 bits across 1 024 words).

Bus & Memory Hierarchy

- **Bus** = parallel wires for address/data/control, shared by multiple devices. CPU reaches main memory through a **memory controller** over a **memory bus**.
- **Hierarchy (fastest/smallest → slowest/largest):** Registers (≈1 KB, 0.3 ns) → L1 (256 KB, 1.1 ns) → L2 (1 MB, 3.3 ns) → L3 (8 MB, 12.8 ns) → L4 (128 MB, 42.4 ns) → Main memory (32 GB, 62.9 ns, $5–10/GB) → SSD (~200 GB–1 TB, $0.67/GB, 0.1 ms) → Disk (2–5 TB, $50/TB, ~3 ms).

RAM Technologies

- **DRAM:** 1 transistor + 1 capacitor per bit; capacitor oriented vertically in a trench; must **refresh** periodically.
- **SRAM:** 6 transistors per bit; holds state indefinitely (faster and more expensive, less dense).

Quantum RAM (QRAM) — core for exam

- A memory whose input/output registers are **qubits**, while data can be addressed in **superposition**.
- **Classical RAM vs. QRAM comparison table:** | Attribute | Classical RAM | Quantum RAM | |—|—|—| | Info storage | bits (0/1) | qubits `\|ψ⟩ = α\|0⟩ + β\|1⟩` | | Access | transistors + capacitors | superposition encoding | | Read | read signal | quantum swap | | Write | write signal | qubits in input register | | Gate activations | $\Theta(2^n)$ | **$\Theta(n)$** | | Error correction | repetition codes | surface codes | | Scalability | #bits | #qubits |
- **QRAM operation:** with `N = 2^n` address lines,
  $\sum_{i=0}^{N-1} \alpha_i |i\rangle_r \xrightarrow{\text{QRAM}} \sum_{i=0}^{N-1} \alpha_i |i\rangle_r |X_i\rangle_o$
  — accesses *all* locations in superposition in one query.
- **Three components:** input (address) register, output (data) register, memory arrays.
- **Challenges:** (i) **no-cloning theorem** — can't copy qubits, so use entanglement instead; (ii) **decoherence time** — qubits need special encodings to survive.
- **Bucket-Brigade QRAM:** binary-tree routing. Quantum switches start in waiting state `|·⟩`; address qubits sent MSB → LSB flip switches along the active route to the target memory cell. Works in two modes:
  1. **Single address** (e.g., address `|01⟩` → data $|X_{01}\rangle$ returned).
  2. **Superposition** (e.g., `|+⟩^⊗2` activates all switches → returns $\frac{1}{2}\Sigma_i |X_i\rangle$).
- **Circuit implementation:** CNOTs + Toffoli gates on address, switch, and ancilla qubits route to the addressed data line; CNOTs highlighted "red" in the slide show the active route.
- **Other QRAM variants (context only):** Fanout, Flip-Flop, EQGAN, qudit-based, Approximate PQC-based.
- **Applications:** Grover auxiliary (database search), element distinctness, collision detection, quantum forking, storing classical data.

Lecture 4 — Classical Computer System V (ISA, x86-64 Assembly, OS, QOS)

Architecture vocabulary

- **Architecture (ISA):** the *parts of a processor you see when writing assembly/machine code* — instruction set, registers. Examples: **x86, IA32, Itanium, x86-64, ARM** (mobile), **RISC-V** (open-source).
- **Microarchitecture:** the *implementation* of the ISA (cache sizes, pipeline depth, core frequency).
- **Machine code** = byte-level executable. **Assembly** = text form of machine code.

C → binary toolchain

`p1.c` — `gcc -Og -S` → `p1.s` (asm) — `as` → `p1.o` (object) — `ld` (linker, pulls static libs `.a`) → `p` (executable).

x86-64 assembly essentials

- Grammar: `add %rbx, %rax` means `rax += rbx` (AT&T syntax: **source, destination**).
- **Registers (nouns):** | 64-bit | 32-bit alias | |—|—| | `%rax, %rbx, %rcx, %rdx` | `%eax, %ebx, %ecx, %edx` | | `%rsi, %rdi` | `%esi, %edi` | | `%rsp` (stack pointer) | `%esp` | | `%rbp` (base/frame pointer) | `%ebp` | | `%r8 … %r15` | `%r8d … %r15d` |
- **Memory** = everything not in a register; accessed via address.
- **Verbs (instruction categories):**
  1. **Data transfer** (load/store between memory and register).
  2. **Arithmetic** on register/memory data.
  3. **Transfer of control:** unconditional jumps, conditional branches, indirect branches.
- Example — compiling `int a=5; int b=7; int sum = a+b;`:

```
movl  $5, -4(%rbp)      ; a = 5
movl  $7, -8(%rbp)      ; b = 7
movl  -4(%rbp), %eax
addl  -8(%rbp), %eax    ; %eax = a + b
movl  %eax, -12(%rbp)   ; sum = eax
```

- Object-code bytes map ~1-to-1 to instructions (e.g., `03` = `add`). A full program prologue: `pushq %rbp; movq %rsp, %rbp; …; popq %rbp; ret`.

Programmer-visible CPU state

- **PC / RIP:** program counter — address of *next* instruction.
- **Register file:** frequently-used data (16 GPRs + special).
- **Condition codes:** status of the last arithmetic/logic op; drive conditional branches.
- **Memory:** byte-addressable array storing code, data, and stack.

Fetch–Decode–Execute cycle

1. **Fetch** — CPU retrieves the next instruction from RAM at the address in the **PC**, puts it in the **Instruction Register (IR)**, advances PC.
2. **Decode** — the **Control Unit** interprets the opcode: what operation, which operands, which control signals to fire.
3. **Execute** — ALU / memory unit performs the op; results land in registers or memory.

Bridge to quantum — reflection questions on the slides (likely exam prompts)

- What is the quantum architecture?
- What would be the quantum instruction set?
- What is "quantum" object code?
- At what levels are errors handled (quantum vs. classical)?
- Can multiple quantum circuits share one QPU simultaneously?
- Where are qubits stored for long computations?
- Where does the compilation of a quantum circuit take place?

Operating System (OS)

- A program that sits between a user and the hardware.
- **Goals:** run user programs, convenient use, efficient hardware utilization.
- **Functionalities:** process management (threads, concurrency), memory management (virtualization), CPU scheduling, file system, security.

Quantum OS (QOS — TUM, USENIX OSDI 2025)

- Pipeline (big circuit → post-processed results):
  QOS API → Error Mitigator (pre/post-processor) → Estimator → Multi-programmer → Scheduler → QPUs + Classical nodes → (bundled results) → Post-processed results.
- **Four main components:** error mitigator, estimator, multi-programmer, scheduler.

Lecture 7 — Classical Simulation for Quantum Computing

Why simulate?

Software & hardware verification, algorithm debugging, teaching. Five main methods: **State Vector, Unitary, Density Matrix, Stabilizer, Matrix Product States (MPS).**

1. State Vector Simulation

- n qubits ↔ state vector $|\psi\rangle = \Sigma_i\ \alpha_i\ |i\rangle$ in a **2^n-dimensional complex (Hilbert) space**.
- **Steps:**
  1. **Initialize** $|0\rangle^{\otimes n}$ (vector of size 2^n).
  2. **Gates = matrix-vector multiplication:** $|\psi_{new}\rangle = U\ |\psi_{old}\rangle$, with U a $2^n \times 2^n$ unitary.
  3. **Measurement probabilities:** $P(i) = |\alpha_i|^2$.
- **Pros:** exact, simple, directly interpretable.
- **Cons:** memory $O(2^n)$, time per gate $O(2^n)$ — infeasible past ~40–50 qubits. Not good for highly entangled states.

2. Unitary Simulation

- Track the *entire circuit unitary*: $U_{circuit} = U_N\ U_{N-1}\ \dots\ U_1$, then $|\psi_{final}\rangle = U_{circuit}\ |\psi_{initial}\rangle$.
- Asks: *"What does the circuit do to any input?"* (vs. state vector: "what happens to this specific input?").
- Example (Bell circuit $H \otimes I$ then CNOT): $U_{circuit} = CNOT \cdot (H \otimes I) = (1/\sqrt{2})[[1,0,1,0],[0,1,0,1],[0,1,0,-1],[1,0,-1,0$ — 4×4.
- **Pros:** analyze *all inputs* at once; great for small-circuit verification / operator properties.
- **Cons:** inherits state-vector limits; memory $O(4^n)$.

3. Density Matrix Simulation

- Track $\rho = \Sigma_i\ p_i\ |\psi_i\rangle\langle\psi_i|$ (mixed state) with $p_i \geq 0, \Sigma\ p_i = 1$, ρ Hermitian PSD, $Tr(\rho) = 1$.
- **Evolution (unitary):** $\rho_{new} = U\ \rho_{old}\ U\dagger$.
- **With errors (Kraus):** $\rho_{new} = \Sigma_k\ E_k\ \rho_{old}\ E_k\dagger$.
- Used for: **mixed states, noise, decoherence, partial systems, open quantum dynamics.**
- **Example — EPR state $|\psi\rangle = (|00\rangle+|11\rangle)/\sqrt{2}$:**
  - Pure: $\rho = \frac{1}{2}\cdot[[1,0,0,1],[0,0,0,0],[0,0,0,0],[1,0,0,1]]$.
  - Phase-damped (λ=1): off-diagonals vanish → $\rho = \frac{1}{2}\cdot diag(1,0,0,1)$ = 50/50 mixture of $|00\rangle\langle 00|$ and $|11\rangle\langle 11|$.
- **Cons:** 4^n parameters (computationally intensive).

4. Stabilizer Simulation

- Efficient for circuits of **Clifford gates**: gates U with $U\ P\ U\dagger \to P'$, mapping Pauli operators to Pauli operators.
- **Clifford examples:** Hadamard H, CNOT, Phase S.
- **Stabilizer:** a Pauli S with $S|\psi\rangle = |\psi\rangle$. Store stabilizers instead of amplitudes.
- **Conjugation rules (memorize):**
  - $H\ X\ H = Z, H\ Z\ H = X, H\ Y\ H = -Y$.
  - CNOT: $X_1 \to X_1X_2, Z_1 \to Z_1, X_2 \to X_2, Z_2 \to Z_1Z_2$.
- **Worked example — Bell state via $H_1 \cdot$ CNOT on $|00\rangle$:**
  - Start: stabilizers Z_1Z_2 (actually Z_1 and Z_2 separately).
  - After H on qubit 1: X_1, Z_2 (since $Z_1 \to X_1$).
  - After CNOT (control=1, target=2): X_1X_2, Z_1Z_2 ⇒ state is $(|00\rangle+|11\rangle)/\sqrt{2}$.
- **Gottesman–Knill Theorem (IN SCOPE):** any circuit built from (i) stabilizer-state inputs, (ii) **Clifford gates {H, S, CNOT}**, (iii) measurements in the **Z basis** can be **simulated efficiently** (polynomial time) on a classical computer.
  - *(The deeper "what it means" bullet — about Clifford-only ≠ quantum speedup and its role in quantum error correction — is marked ∗; not required to memorize, but very useful intuition.)*
- **Pros:** polynomial time for stabilizer circuits.
- **Cons:** limited to Clifford; any **non-Clifford gate (e.g., T)** forces exponential simulation; needs circuit decomposition.

5. Matrix Product States (MPS)

- Represent |ψ⟩ = Σ_{σ₁…σ_N} c_{σ₁…σ_N} |σ₁…σ_N⟩ compactly as a product of per-site matrices: $|\psi\rangle = \sum_{\sigma} \mathrm{Tr}\left(U^{[1]}_{\sigma_1} U^{[2]}_{\sigma_2} \cdots U^{[N]}_{\sigma_N}\right) |\sigma_1, \ldots, \sigma_N\rangle$
- **Worked example — GHZ (|000⟩+|111⟩)/√2:** choose
  - U₀^[1] = [1/√2, 0], U₁^[1] = [0, 1/√2];
  - U₀^[2] = [[1,0],[0,0]], U₁^[2] = [[0,0],[0,1]];
  - U₀^[3] = [1,0]ᵀ, U₁^[3] = [0,1]ᵀ.
  - Then |000⟩ amplitude = [1/√2, 0]·[[1,0],[0,0]]·[1,0]ᵀ = 1/√2; |111⟩ similarly 1/√2; all other strings give 0. ✓
- **Works well when: limited entanglement**, circuit is mostly **1D (nearest-neighbor)**, depth is small.
- **Memory:** O(n·χ²) with n qubits and **bond dimension** χ (entanglement measure).
- **Scale example:** 50 qubits require **18 PB** as a state vector but only **~6.6 MB** as an MPS.
- **Pros:** compact for low-entanglement states; used in DMRG for ground-state physics; captures local entanglement well.
- **Cons:** poor for highly-entangled / 2D systems.
- *(Underlying SVD theory and the image-compression analogy are marked ∗ — excluded.)*

Comparison at a glance

	State vec / Unitary / Density	Stabilizer	MPS
Applicability	Any circuit	Clifford + stab. states	Low-entanglement, 1D
Efficiency	Exponential	Polynomial	Linear (1D, low-ent.)
Memory	O(2ⁿ) (4ⁿ for ρ/unitary)	Polynomial	O(n·χ²)
Exact	Yes	Yes (Clifford)	Yes (low-ent.)
Scope	Universal	Clifford only	1D low-ent.

Lecture 8 — Quantum Computing Systems & Quantum Assembly (OpenQASM)

Quantum Computing Stack (layered)

- **User:** Languages, SDKs/Frameworks, Algorithms, Applications.
- **Platform:** Simulation, Compilation, Error Correction.
- **Hardware:** Control Layer, Quantum (QPU).
- Three abstraction layers — **Application**, **Systems software** (DSL, compilation, unitary synthesis, pulse control, noise mitigation, error correction), **Hardware** (controller + qubits).
- The classical 1950s stack (Algorithms → ASM → Vacuum Tubes) structurally resembles today's QC stack (Algorithms → Q-DSL/Compilation/Pulse/… → Qubits).

Compiler Pipeline

QDSL (Qiskit, myQLM, Cirq) → Compiler → QASM (e.g., OpenQASM) → **QIR / QISA** (hardware-specific IR) → **Pulses** (physical control).

QASM vs. QIR

	QASM (e.g., OpenQASM)	**QIR** (Intermediate Rep.)
Level	Human-readable, gate-level	Lower-level, machine-oriented
Portability	Hardware-agnostic	Hardware-specific (device topology, native gate set)
Use	Circuit exchange between tools, simulators, cloud APIs	Backend compilation, gate fusion, routing, qubit mapping
Classical analog	Classical assembly	**LLVM IR**

	QASM (e.g., OpenQASM)	**QIR** (Intermediate Rep.)

Generic Quantum Compilation Flow

Quantum algorithm → high-level program (quantum + classical syntax) → automated compiler passes (resolve syntax, function cloning, loop unrolling, **gate decomposition**, **arithmetic synthesis**, scheduling, memory mgmt, error correction, dynamic compilation) constrained by **architectural constraints** (qubit connectivity, coherence, gate error, available parallelism) → classical controller → quantum + classical processors.

Use-case platforms

- **Qaptiva (Eviden / Atos QLM):** Python circuit language, universal/gate-based (LinAlg emulator, noise, MPS, QPEG), annealing (Simulated Quantum Annealing = SQA), analog (Hamiltonian schedules), optimization (topological + pattern-circuit optimizers, noise models), resources (batch, tomography, scheduler). On-premise or as a service.
- **OpenQASM 3.0 compilation/execution:** application → OpenQASM circuits + extern functions → Logical OpenQASM → compilation passes (basis-gate translation, physical qubit mapping, circuit optimization, timing resolution, pulse optimization, control-flow optimization) → Physical OpenQASM → Machine Target Code → Global Controller → Local Controllers → Qubits.

Quantum Architecture Constraints (memorize the 4)

1. **Instruction set** — universal gate set (e.g., Clifford + T).
2. **Quantum communication** — multi-qubit ops need qubits to be brought near via **SWAP chains**.
3. **Hardware noise** — memory errors (decoherence) + gate errors (imprecise control).
4. **Available parallel control** — limited by control hardware and error-mitigation protocols.

DiVincenzo Criteria for a Viable Qubit (2000)

1. A **scalable** physical system with **well-characterized** qubits.
2. Ability to **initialize** to a simple fiducial state (e.g., $|000\ldots0\rangle$).
3. Long decoherence times $T_1, T_2 \gg t_{gate}$.
4. A **universal set of quantum gates** (single-qubit + CNOT).
5. **Qubit-specific measurement** (readout).

Plus two for **quantum communication**: (6) **interconvert** stationary ↔ flying qubits; (7) **faithfully transmit** flying qubits between locations.

OpenQASM 2.0

- Imperative language for quantum circuits; IR between high-level compilers and hardware; **machine-independent**; based on the **quantum circuit model**; pronounced *"kazm"*.
- **Execution model (4 phases):** (1) **Compilation** (offline, problem-independent) → (2) **Circuit generation** (online, with problem parameters) → (3) **Execution** (real-time on controllers) → (4) **Post-processing** (classical analysis of measurements).

Header & registers

- `OPENQASM 2.0;` — version declaration (must be first line).
- `qreg q[5];` — 5-qubit register (auto-initialized to $|0\rangle$).
- `creg c[5];` — 5 classical bits (initialized to 0).

Built-in universal basis

- `U(θ, φ, λ) q[0];` — arbitrary single-qubit: `U = Rz(φ)·Ry(θ)·Rz(λ)`.
- `CX q[0], q[1];` — controlled-NOT.
- Example: `U(pi/2, 0, pi) q[0];` = Hadamard.

Other statements: `measure q -> c;` (Z-basis), `reset q[0];` (→ |0⟩), `barrier q;` (prevent optimizer from reordering across this line).

Custom gates (hierarchical; declared before use; no recursion):

```
gate h a { U(pi/2,0,pi) a; }
gate cu1(lambda) a,b {
  U(0,0,lambda/2) a; CX a,b;
  U(0,0,-lambda/2) b; CX a,b;
  U(0,0,lambda/2) b;
}
```

Header `qelib1.inc` provides x, y, z, h, s, t, cx, ccx, cu1, ….

Classical control (`if`-statement): `if(c==1) z q[2];` — apply Z only when classical register `c` equals integer 1 (bit[0] = low-order).

Statement table (v2): `OPENQASM`, `qreg`, `creg`, `include`, `gate`, `opaque`, `// comment`, `U`, `CX`, `measure`, `reset`, `<gatename>(…)`, `if(creg==int) qop`, `barrier`.

Examples (QASM 2)

- **Quantum Ripple-Carry Adder (Cuccaro et al. quant-ph/0410184):** defines `gate majority a,b,c { cx c,b; cx c,a; ccx a,b,c; }` and `gate unmaj a,b,c { ccx a,b,c; cx c,a; cx a,b; }`; then chains majority → CX(a[3],cout[0]) → unmaj → measure to add `a = 0001` to `b = 1111`.
- **Quantum Fourier Transform on `|1010⟩`:** `x q[0]; x q[2]; h q[0]; cu1(π/2) q[1],q[0]; h q[1]; cu1(π/4) q[2],q[0]; cu1(π/2) q[2],q[1]; h q[2]; cu1(π/8) q[3],q[0]; cu1(π/4) q[3],q[1]; cu1(π/2) q[3],q[2]; h q[3]; measure q -> c;` Output read in **reverse** order: `c[3], c[2], c[1], c[0]`.

Why OpenQASM 3?

- **v2 limits:** qubits/gates only, **no classical control flow**, no timing/pulse descriptions, can't express *repeat-until-success* algorithms.
- **v3 adds:** classical branching, calibration & error-mitigation via timing, **dynamical decoupling** (precise pulse scheduling), **multi-level IR**.

OpenQASM 3 features

- **Program structure:** `OPENQASM 3; include "stdgates.inc";`. Declarations: `qubit[n] q; bit[n] c;` + `int, uint, float, bool` (widths like `int[8]`, `float[64]`, `bit[n]`).
- **Gates:** built-in `U(θ,φ,λ)`, `gphase(γ)`; custom `gate h a { U(π/2,0,π) a; }`; **modifiers** `ctrl @`, `inv @`, `pow(r) @`.
- **Control flow:** `if/else`, `for`, `while`; subroutines via `def`; `extern` for external classical functions.
- **Timing:** `duration` (ns / μs / `dt` units), `stretch` (compiler-resolvable flexible duration), `delay` (idle time on specific qubits), `box` (scope timing constraints).
- **Pulse:** `defcal` attaches pulse-level (OpenPulse) implementations to gates.

Two Timescales of Q-C Interaction

	Real-time	Near-time
Timing	Must finish **within coherence times**	Looser
Coupling	Tightly coupled to QPU	Loosely coupled
Examples	Mid-circuit measurement, feed-forward, conditional ops	VQE outer loop, Shor co-prime search
OpenQASM 3 scope	**Yes — this is what v3 targets**	Normal classical code

Gate Modifiers

- **`ctrl @ gate`** — add a control qubit (`ctrl @ x = CNOT`; `ctrl @ ctrl @ x = Toffoli`).

- **`inv @ gate`** — adjoint (reverse the gate sequence and invert each gate).
- **`pow(r) @ gate`** — r-th power (`pow(0.5) @ z = S` = $\sqrt{Z}$).
- **Composable:** `ctrl @ inv @ pow(2) @ s`.

OpenQASM 3 — multi-level IR

1. **High-level (Logical):** virtual qubits, gate modifiers, subroutines.
2. **Mid-level (Circuit):** basis gates, qubit mapping, timing constraints.
3. **Low-level (Physical):** `defcal` pulse definitions, physical qubits (`$0, $1`).

OpenQASM vs. AQASM

	OpenQASM	**AQASM**
Origin	Open-source, IBM / Qiskit ecosystem	Part of Atos/Eviden **myQLM**
Purpose	Describe & exchange circuits across tools	Internal circuit representation inside QLM
Characteristics	Human-readable, hardware-oriented, gate-level; v3 adds classical flow, timing, hybrid features	Not a universal interchange format; Python/myQLM-integrated; works with QLM simulators & compilers
Versions	2.0, 3.0	myQLM-specific

Use Cases (real motivation for v3)

- **Dynamic circuits:** mid-circuit measurement + conditional ops, **repeat-until-success** (e.g., Rz synthesis from Nielsen & Chuang), **Iterative Phase Estimation** with just one extra qubit.
- **Dynamical decoupling:** timing-sensitive pulse sequences written with `stretch`; compiler resolves to exact durations at compile time. **Spin-echo** sequences cancel errors from slowly drifting hardware parameters.
- **Key insight:** OpenQASM 3 enables the *same language* from algorithm development through calibration and error mitigation — no separate tools needed.

Lecture 9 — Quantum Program Debugging, Verification & State Tomography

Two Notions of Verification

- **Software verification:** the *program* is bug-free AND the compiler's transformations preserve logical equivalence.
- **Hardware verification:** the *hardware* performs the intended quantum logic.
- Tools: **classical simulators** (Lecture 7 methods), **assertion-based** methods, **formal methods**.

Strong vs. Weak Classical Simulation

	Strong	Weak
Goal	Compute $p(x) = \|\langle x\|C\|0^{\otimes n}\rangle\|^2$ for **every** $x \in \{0,1\}^n$	Sample from distribution D with $\|D - D_C\| \leq \varepsilon$
Output	Full probability distribution over 2^n outcomes	ε-close samples
Requires amplitudes?	Yes	No
Cost on general circuits	**Exponential**	Can be **polynomial** for restricted circuits (e.g., Clifford via Gottesman–Knill)

Hardware Verification → Quantum Tomography

- Quantum objects can't be observed directly; **tomography** reconstructs them from measurement data.

Quantum State Tomography (QST)

Reconstruct an unknown density matrix ρ from measurement statistics on many identical copies. Analogous to CT scans.

Five steps:

1. **Prepare** many identical copies of ρ (measurement destroys each copy, so you need N independent copies).
2. **Choose measurement bases.** One basis is insufficient (no-cloning). For a **single qubit** you need three: **Z** {|0⟩, |1⟩}, **X** {|+⟩, |−⟩} (apply H before a Z-measurement), **Y** {|i⟩, |−i⟩} (apply S†H before a Z-measurement). For **N qubits** the required measurements grow **exponentially** (tensor products of X/Y/Z per qubit).
3. **Measure** each copy in a chosen basis; tally outcome frequencies.
4. **Estimate expectation values.** For one qubit ρ = ½(I + x σ_x + y σ_y + z σ_z) with x = ⟨σ_x⟩, y = ⟨σ_y⟩, z = ⟨σ_z⟩, and ⟨σ_B⟩ = p_{+B} − p_{−B} for basis B.
5. **Reconstruct ρ:**
   $\rho = \frac{1}{2}\begin{pmatrix} 1+z & x-iy \\ x+iy & 1-z \end{pmatrix}$
   For **N qubits** this extends to a $2^N \times 2^N$ matrix.

Worked QST Example (MUST be able to reproduce)

Unknown state |ψ⟩ = (√3/2)|0⟩ + (½)|1⟩. Prepare 10 000 copies per basis.

- **Z-basis:** P(|0⟩) = |√3/2|² = 3/4 = 0.75, P(|1⟩) = |½|² = 1/4 = 0.25. ⟨σ_z⟩ = 0.75 − 0.25 = 0.50.
- **X-basis** (apply H first): P(|+⟩) = |(√3+1)/(2√2)|² = (2+√3)/4 ≈ 0.933, P(|−⟩) ≈ 0.067. ⟨σ_x⟩ ≈ 0.866 ≈ √3/2.
- **Y-basis** (apply S†H first): amplitudes are real ⇒ P(|i⟩) = P(|−i⟩) = 0.50. ⟨σ_y⟩ = 0.
- **Bloch vector** r = (0.866, 0, 0.50), |r| = √(0.75 + 0 + 0.25) = 1 ⇒ **pure state** (on the Bloch-sphere surface).
- **Reconstruct:**
  $\rho = \frac{1}{2}\begin{bmatrix} 1.5 & 0.866 \\ 0.866 & 0.5 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 3/4 & \sqrt{3}/4 \\ \sqrt{3}/4 & 1/4 \end{bmatrix}$ (after factoring ½)
- **Verification:** Tr(ρ) = ½(1.5 + 0.5)/(… well with the factor) = 1 ✓, ρ† = ρ ✓, Tr(ρ²) = 1 (pure) ✓; matches |ψ⟩⟨ψ|. ✓ **QST successful.**

Four Types of Quantum Tomography

1. **Quantum State Tomography (QST)** — reconstruct an unknown state ρ.
2. **Quantum Process Tomography (QPT)** — characterize an unknown **operation** (gate or noise process) by applying it to known inputs and measuring outputs.
3. **Quantum Detector Tomography (QDT)** — characterize measurement devices; determine the **POVM** elements describing the detector.
4. **Gate Set Tomography (GST)** — simultaneously reconstruct states + processes + measurements; best for diagnosing **systematic** hardware errors.

Lecture 10 — Quantum Circuit Synthesis: Unitary Decomposition

Algorithm → Executable Pipeline

Quantum algorithm → **Unitary matrix** (target transformation, SU(2ⁿ)) → **Logic circuit design** (synthesis + optimization targeting gate count, depth, CNOT count, T count) → **Quantum compiling** (qubit mapping, routing, SWAP insertion, timing) → executable program on a physical topology.

Circuit Representations

- **Gate Model:** sequence of gates on horizontal qubit wires, left-to-right.
- **DAG (Directed Acyclic Graph):** nodes = gates, edges = data dependencies on shared qubits → enables scheduling and parallelism analysis.
- **Tensor Networks:** MPS, TTN, PEPS, MERA — trade-offs in entanglement structure handled.

- **ZX-Diagrams:** tensor-network-like; two spider types (green Z, red X); can represent **any linear map** between qubits. (Reference: *Quantum in Pictures* by Coecke & Gogioso.)
- **QMDD:** Quantum Multiple-valued Decision Diagram — layered representation of $2^n \times 2^n$ ops.
- **Unitary matrix:** the mathematical target; exponential in size ($SU(2^n)$).

Universal Gate Sets

- A gate set **S** is **computationally universal** iff gates from S can realize an arbitrary quantum circuit to any accuracy.
- **"Gold Standard" (Clifford + T):** H (superposition), S ($\pi/2$ Z-rotation), CNOT (entangling), T ($\pi/4$ Z-rotation) — common in fault-tolerant research.
- **Discrete:** {H, T, CNOT} — best for error-corrected systems.
- **Continuous:** {$R_x(\theta)$, $R_y(\theta)$, CNOT} — common on NISQ devices where θ can be tuned precisely.
- **OpenQASM universal `U(θ,φ,λ) + CX`:**
  $U(\theta,\varphi,\lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\varphi}\sin(\theta/2) & e^{i(\varphi+\lambda)}\cos(\theta/2) \end{bmatrix}$
  θ = Y-rotation, φ = Z after θ, λ = Z before θ. Standard mappings: | Standard | OpenQASM | |—|—| | I | `U(0,0,0)` | | X | `U(π,0,π)` | | H | `U(π/2,0,π)` | | Z | `U(0,0,π)` | | S | `U(0,0,π/2)` |
- **IBM Heron / Nighthawk native set `{RZ, SX, X, CZ}`** (current standard):
  - **`RZ(λ)` (Virtual-Z):** "zero-cost" phase shift implemented by changing the phase of the microwave drive in the classical controller. **Duration = 0 ns, Error ≈ 0%.**
  - **`SX` = `√X`:** $\pi/2$ X-rotation — the physical building block used to create Hadamards and other rotations.
  - **`X`:** full π — often implemented as two SX pulses back-to-back.
  - **`CZ` (Controlled-Z):** implemented via **tunable couplers** that bring two qubits into resonance momentarily; faster and higher-fidelity than the older CX (cross-resonance). Convert: $CNOT_{0,1} = (I \otimes H) \cdot CZ_{0,1} \cdot (I \otimes H)$.

Unitary Decomposition — the goal

- Rewrite a $2^n \times 2^n$ U as a product of 1- and 2-qubit gates: $U \approx G_m\, G_{m-1} \ldots G_2\, G_1$.
- **Why?** Hardware executes only short sequences of 1- and 2-qubit native gates, and n-qubit U grows exponentially.

Three Exact Decomposition Methods

1. Two-Level / Givens Rotation

- Zero out matrix entries one at a time using **2D-subspace rotations** (Givens rotations), then translate each Givens into controlled rotations + CNOTs.
- **Strategy:** Element-wise elimination (bottom-up).
- 2-qubit 4×4 example: apply G_{12}, G_{13}, G_{14}, G_{23}, G_{24}, G_{34} in order to reduce U to I. Then $U = G_{12}^\dagger\, G_{13}^\dagger\, G_{14}^\dagger\, G_{23}^\dagger\, G_{24}^\dagger\, G_{34}^\dagger$.
- **Numerical snippet (single Givens):** for `U = [[0.6, −0.8],[0.8, 0.6]]` (rotation by ≈ 53.13°):
  - $r = \sqrt{0.6^2 + 0.8^2} = 1$, $\cos\varphi = 0.6$, $\sin\varphi = 0.8$.
  - `G = [[0.6, 0.8],[−0.8, 0.6]]` (rotation by $-\varphi$); $G \cdot U = I$, so $U = G^\dagger$.
- For `N × N`, this requires ~`N(N−1)/2` Givens rotations.
- **Best for:** proving universality, small matrices.
- **CNOT count: `O(n³·4ⁿ)`** — very high.

2. Cosine–Sine / Quantum Shannon Decomposition (CSD / QSD)

- $U = \mathrm{diag}(U_1, U_2) \cdot [[C, -S],[S, C]] \cdot \mathrm{diag}(V_1, V_2)$ with $C = \mathrm{diag}(\cos\theta_i)$, $S = \mathrm{diag}(\sin\theta_i)$, $C^2 + S^2 = I$.
- Middle block = **multiplexed (uniformly-controlled) rotation**; block-diagonals recurse on `(n−1)` qubits until 1-qubit.
- **Strategy:** Recursive subspace splitting (top-down).
- Sketch for $U = H \otimes I = (1/\sqrt{2}) \cdot [[1,0,1,0],[0,1,0,1],[1,0,-1,0],[0,1,0,-1]]$:
  - Partition into 2×2 blocks: $A = B = (1/\sqrt{2})I$, $C = -D = (1/\sqrt{2})I$.
  - CSD yields a middle $R_x(\theta)$ (or R_y) controlled on qubit 0 — here a single $R_y(\pi/4)$.
  - The multiplexed rotation implements as **CNOT + Ry(θ) + CNOT**.
- **Best for:** arbitrary n-qubit unitaries. **CNOT count: ≈ `½·4ⁿ`** (near-optimal asymptotically).

3. KAK / Cartan Decomposition (2-qubit)

- $U = (A_1 \otimes A_2) \cdot \exp[i(\alpha\, X\otimes X + \beta\, Y\otimes Y + \gamma\, Z\otimes Z)] \cdot (B_1 \otimes B_2)$.
- Local gates A, B are single-qubit; the middle "entangling core" uses $\leq$ **3 CNOTs**.
- **Cartan coordinates (α, β, γ)** measure the entangling power of the gate — and determine exactly how many CNOTs are needed.
- **CNOT example:** KAK for CNOT has $\alpha = \pi/4$, $\beta = \gamma = 0$, with local gates $A_1 = Ry(\pi/2)$, $A_2 = I$, $B_1 = Rz(-\pi/2)$, $B_2 = Rz(-\pi/2)$. Because $\beta = \gamma = 0$ the entangling core needs only **1 CNOT** (the minimum).
- **Strategy:** Geometric Lie-algebra mapping.
- **Best for:** optimal 2-qubit synthesis. **CNOT count: 3 CNOTs** = absolute minimum for any general 2-qubit unitary.

Exact Decomposition — Summary Table

Feature	Givens	CSD/QSD	KAK/Cartan
Strategy	Element-wise elimination (bottom-up)	Recursive subspace splitting (top-down)	Geometric Lie-algebra mapping
Best for	Proving universality, 2×2	Arbitrary n-qubit	**Optimal 2-qubit synthesis**
CNOT count	$O(n^3 \cdot 4^n)$ (High)	$\approx \frac{1}{2} \cdot 4^n$ (Near-optimal)	**3** (Minimum for general 2-qubit)
Complexity	Simple, iterative	Highly recursive, algebraic	Geometric / abstract
Exact	✓	✓	✓

Solovay–Kitaev Theorem (Approximate Synthesis)

- **The problem:** a finite gate set (e.g., $\{H, T\}$) produces only **countably many** unitaries, but $SU(2)$ is continuous (uncountable) — so most gates can only be **approximated**.
- **Naive brute force:** enumerate gate sequences of length $\ell \to$ needs $\ell = O(1/\varepsilon)$ to reach accuracy ε. Impractical.
- **Solovay–Kitaev theorem:** any finite gate set G that generates a *dense* subgroup of $SU(2)$ can approximate any $U \in SU(2)$ to accuracy ε using only
  $O(\log^c(1/\varepsilon))$ gates, $\quad c \approx 3.97$.
  Exponentially fewer than brute force.
- **Algorithm (recursive):** to improve from ε_n to $\varepsilon_{n+1} \approx \varepsilon_n^{3/2}$:
  1. **Base case:** look up a "rough" approximation U_0 (accuracy ε_0) in a precomputed table.
  2. Compute the error $U_{err} = U \cdot U_n\dagger$.
  3. Factor the error as a **group commutator** $U_{err} \approx V\, W\, V\dagger\, W\dagger$.
  4. Recursively approximate V and W.
  5. $U_{n+1} = \tilde{V}\, \tilde{W}\, \tilde{V}\dagger\, \tilde{W}\dagger \cdot U_n$.
- **Key insight:** a commutator $V\, W\, V\dagger\, W\dagger$ has error $\approx \varepsilon^2$, so each recursion level **roughly squares the accuracy**.

Lecture 11 — Quantum Circuit Synthesis (Examples, Oracle Synthesis, AI/ML, QAS)

Synthesizing Controlled-U Gates

- **Theorem.** For any unitary U, there exist single-qubit gates $A_1, \ldots, A_m$ and $B_1, \ldots, B_m$ and a phase α such that
  $C\text{-}U = e^{i\alpha} A_m B_m A_{m-1} B_{m-1} \cdots A_1 B_1$
  satisfying $A_m \ldots A_1 = I$ and with each B_i simpler than C–U (commonly $B_i = X$).
- **"ABC" form.** An arbitrary U decomposes as $U = e^{i\alpha} \cdot C\, X\, B\, X\, A$ with $ABC = I$. Procedure:
  1. Decompose U into A, B, C satisfying ABC = I.
  2. Apply C to the target qubit.
  3. Apply CNOT (control toggles target).
  4. Apply B, then another CNOT, then A to the target.
  5. Apply the global phase $e^{i\alpha}$ as a controlled-phase gate on the control qubit.

Synthesizing Quantum Oracles — the 2-step recipe

Quantum algorithms (**Deutsch–Jozsa, Simon, Bernstein–Vazirani, Grover**) treat oracles as black boxes. Implementing an oracle is a two-step problem:

1. Find an **efficient reversible-logic implementation** of the Boolean function using **Toffoli, Fredkin, or Peres** gates. **Ancilla qubits** (initialized `|0⟩`) may be needed for reversibility and must be **uncomputed** after use.
2. Implement each reversible logic gate using quantum gates (e.g., Toffoli ⇒ 6 CNOTs + single-qubit H, T, T†). Goal: **minimize gate count and depth** for NISQ hardware.

Toffoli (CCNOT)

- 3-qubit gate that flips the target iff both controls are `|1⟩`.
- **Universal for classical reversible computation.** (And **Toffoli + Hadamard** is universal for quantum computation.)
- **Must be decomposed** because hardware supports only native 1- and 2-qubit gates.
- **Standard decomposition: 6 CNOTs + several H, T, T†** single-qubit gates.
  - Sketch: `H₃; CX 2,3; T†₃; CX 1,3; T₃; CX 2,3; T†₃; CX 1,3; T₂; T₃; H₃; CX 1,2; T₁; T†₂; CX 1,2` (or equivalent).
- **Why it matters:**
  - Hardware supports only native 1- and 2-qubit gates.
  - Toffoli is expensive (~6 CNOTs each) → **key optimization target** in circuit compilation.

Oracle Synthesis for Specific Algorithms

- **Deutsch oracle** `U_f : |x, y⟩ → |x, y ⊕ f(x)⟩`:
  - **Constant f:** oracle is either Identity (`f = 0`) or X on the output qubit (`f = 1`).
  - **Balanced f:** oracle is a **CNOT** with input as control and output as target.

Grover's Algorithm & the Oracle Problem

- Achieves **`O(√N)` search** vs. classical `O(N)`.
- **Three steps of the algorithm:**
  1. **Superposition** — apply Hadamard to all qubits → equal probability for all `N` items.
  2. **Oracle** — flips the **phase** of the target item (`U_f |s⟩ = −|s⟩` if `s = ω` (target), else `+|s⟩`). Marks the target without revealing it.
  3. **Diffusion** — amplifies the marked state's amplitude.
- **Core challenge:** the oracle must be **custom-built for each search problem**. Three solution families: **Classical synthesis, Metaheuristics, Neural networks.**

Classical Synthesis Methods for Oracles (in scope — *concept* only; the algorithm is ∗)

- **Approach:** convert a Boolean function `f` into a reversible quantum circuit via algebraic decomposition. Key tool: **Zhegalkin polynomials** (= Algebraic Normal Form / ANF) represent any Boolean function as an XOR of AND terms, e.g., `f(x₁,x₂) = a₀ ⊕ a₁x₁ ⊕ a₂x₂ ⊕ a₁₂x₁x₂`.
- **Key idea:** every Boolean `f(x)` can be represented as a unitary. `U_f` is a `2^(n+1) × 2^(n+1)` **block-diagonal matrix of `I` and `X` blocks**.
- **1-bit function table (must memorize):** | f | Formula | `U_f` | Quantum circuit | |—|—|—|—| | `f₀ = 0` | constant 0 | `diag(I, I)` | **Identity** | | `f₁ = x` | identity | `diag(I, X)` | **CNOT** | | `f₂ = x¯` | NOT | `diag(X, I)` | **CNOT + NOT** | | `f₃ = 1` | constant 1 | `diag(X, X)` | **NOT gate** |
- **Each term** in the Zhegalkin polynomial maps to a gate: constant 1 → **X** on output; single variable `xⱼ` → **CNOT** (qubit j controls output); product `xⱼ₁·xⱼ₂` → **Toffoli/CCNOT**; product of m vars → **C^m-NOT**. So `f(x₁,x₂) = 1 ⊕ x₁ ⊕ x₂ ⊕ x₁x₂` → `X + CNOT(x₁→y) + CNOT(x₂→y) + CCNOT(x₁,x₂→y)`.
- **Qubits required:** `n + 1 + a` (n input + 1 output + a ancilla).
- **Limitations:** needs explicit Boolean-function construction; ancilla overhead; limited use of entanglement.

Metaheuristic Methodologies

- Use **optimization algorithms** (**Ant Colony**, **Genetic Programming**) to search for optimal circuits; circuits evolved iteratively with fitness metrics.
- **Mathematical foundation:**
  - **Fidelity:** $F(|y\rangle, |x\rangle) = |\langle y|x\rangle|$.
  - **Mean Square Fidelity:** $MSF(A, X, Y) = (\Sigma_i F(|y_i\rangle, A|x_i\rangle)^2) / |X|$, with A = realized unitary, X = all inputs, Y = all expected outputs.
- **Key methods:**
  - **QAP (Quantum Ant Programming):** graph-based pheromone trails reinforce good sub-circuits.
  - **Genetic Programming:** tree-structured circuits with **mutation** (insert/delete/replace subtrees) and crossover.
- **Limitations:** poor fidelity for large circuits; expensive runtime; doesn't scale linearly.

Neural-Network Methodologies

- Use classical NNs as encoders/decoders; learn to map unitaries (or I/O pairs) to gate sequences.
- **Foundations:**
  - **Matrix decomposition (GRU network):** learn $U \approx U_1 \cdot U_2 \cdot \ldots \cdot U_N$.
  - **Lie-algebra coefficients (ReLU network):** learn coefficients of $U = \Pi \exp(i \tau_k \theta_k)$ that translate directly to gates.
- **Key architectures:**
  - **GRU + ReLU (Swaddle):** matrix → simpler matrices → Lie-algebra coefficients → gates.
  - **Seq2Seq (Cruz-Benito):** encoder–decoder on OpenQASM text; generates and optimizes circuits.
- **Limitations:** data-hungry; scalability (tested ≤ 3 qubits); non-deterministic outputs.

AI/ML Approaches — Why

Traditional deterministic decomposition produces high gate counts / depths. **Four motivations:**

1. **Reduce depth** to minimize decoherence effects on NISQ.
2. **Reduce gate counts** to reduce error accumulation in noisy hardware.
3. **Discover novel topologies** classical heuristics may miss.
4. **Automate hardware-aware compilation** for specific quantum-processor architectures.

ML Techniques for Quantum Circuits

- **Supervised:** NNs trained on known optimal decompositions predict efficient gate sequences; **GNNs** learn circuit-structure representations and propose transformations.
- **Generative:**
  - **Variational Autoencoders (VAEs):** learn latent representations of circuit families and generate candidates.
  - **Generative Adversarial Networks (GANs):** produce circuits approximating target unitaries.
- **Evolutionary:** genetic programming with crossover + mutation of gate sequences; multi-objective optimization over depth, gate count, fidelity simultaneously.

Reinforcement Learning for Gate Optimization

Formulate circuit synthesis as a **sequential decision-making problem**:

1. **State:** current partial circuit (gate sequence so far).
2. **Action:** select the next quantum gate + target qubits from the allowed gate set.
3. **Reward:** fidelity of the synthesized circuit to the target unitary, penalized by circuit depth.

Common RL approaches:

- **Deep Q-Networks (DQN):** value functions over discrete gate choices.
- **Policy Gradient methods:** directly optimize parameterized policies for continuous gate parameters (rotation angles).
- **Monte Carlo Tree Search (MCTS):** explore gate-sequence trees with learned value estimates.

Quantum Architecture Search (QAS)

- QC analog of **Neural Architecture Search (NAS)**; **automatically discovers optimal VQC (variational quantum circuit) ansatz structures** for a given task.
- **Why QAS matters:**
  - Manual ansatz design requires deep expertise and often fails the **expressivity-vs-trainability** balance.
  - **Barren plateaus** + quantum noise make generic ansatz patterns unreliable on NISQ.
  - QAS can tailor circuits to both the **computational problem** and the **physical device**.
- **Four dimensions of any QAS method (must memorize):**
  1. **Search Space** — what gates/layers are available (gate-wise, layer-wise, block-wise).
  2. **Encoding** — how a circuit is represented (list, DAG, tensor, graph).
  3. **Search Strategy** — how the space is explored (RL, EA, differentiable, etc.).
  4. **Performance Evaluation** — how candidates are scored (full training, predictors, zero-cost proxies).

Variational Quantum Circuits (VQC) — the target of QAS

- Parameterized circuit with tunable rotations θ; a classical optimizer updates θ to minimize a cost function $C(\theta)$ (hybrid quantum-classical loop).
- **Typical ansatz:** layer of single-qubit $R_y(\theta)$ rotations + entangling layer of CNOTs, repeated **L** times.
- **Applications (preview of COE 531): VQE** (variational quantum eigensolver), **QAOA** (quantum approximate optimization algorithm), **QML** (quantum machine learning).
- **Optimization loop:** Measure → Evaluate $C(\theta)$ → Update θ → Re-run circuit → converge to $\min C(\theta*)$.

QAS — Methods & Frameworks

- **Search-space definition:** gate pool (Rx / Ry / Rz, CNOT / CZ, parametric gates), **topology constraints** (hardware connectivity graph), **depth budget**.
- **Search strategies at a glance:**
  - **Differentiable QAS (DQAS):** discrete choices relaxed to continuous parameters; optimized by gradient descent.
  - **RL-based QAS:** an agent places gates sequentially; reward = task performance.
  - **Evolutionary QAS:** populations evolve via selection + crossover + mutation.
  - **Bayesian Optimization:** surrogate models (Gaussian Processes, Tree Parzen Estimators) guide efficient exploration.

QAS — Seven Search Strategies (detail — memorize representatives)

1. **Reinforcement Learning (RL)** — Markov Decision Process; algorithms **PPO, DDQN, A3C**; **CRLQAS** uses curriculum learning to gradually increase task difficulty. **Weakness:** sample-inefficient (millions of rollouts).
2. **Evolutionary Algorithms (EA)** — population of circuits evolved via mutation + crossover. Notable: **MoG-VQE** (multi-objective), **EVQE** (identity-initialized), **QCEAT** (multi-species with robustness). Some EA variants **avoid crossover** because combining two circuits doesn't guarantee inherited performance.
3. **Differentiable QAS — DQAS** (Monte Carlo gradients), **QuantumDARTS** (Gumbel-softmax). Jointly optimizes architecture probabilities and gate parameters in one pass.
4. **Adaptive / Greedy Algorithms** — iteratively add/remove gates via gradients or heuristic rules. **ADAPT-VQE** greedily appends the fermionic operator with the largest gradient (**highly popular in chemistry**). **VAns** stochastically inserts then prunes via rule-based classical logic.
5. **Monte Carlo Tree Search (MCTS)** — encode the search space as a tree; nodes = gates, paths = circuits. Balances exploration vs. exploitation via **Upper Confidence Bounds**. **MCGS** extends MCTS to graphs.
6. **Bayesian Optimization (BO)** — Gaussian Processes / Tree Parzen Estimators guide efficient sampling. **Best for small search spaces with expensive evaluations.**
7. **Generative Models — GPT-QE** (GPT-2-style transformer) generates gate sequences as "language". Supports pre-training + fine-tuning for transfer learning. GANs/VAEs also applicable.

Open Challenges

- **Scalability** — most methods tested on ≤ 10 qubits; scaling to larger circuits is an open problem.
- **Hardware adaptability** — recalibrated QPUs can require expensive QAS retraining.
- **Benchmarks** — no standardized benchmark suite exists, making method comparisons difficult.

- **Noise modeling** — better noise-aware evaluation metrics are needed beyond simple expectation-value perturbations.
- **Transfer learning** — pre-trained QAS models rarely transfer cleanly between problem types.

Key takeaway

QAS lives at the intersection of **quantum computing + AutoML + hardware-aware optimization**. Today the most mature techniques are **RL-based** and **Adaptive** (especially **ADAPT-VQE**); **Differentiable** methods and **Generative** models are catching up as scalability bottlenecks are addressed.

Master Cross-Lecture Reference

Universal gate sets (memorize!)

Set	Typical use
{H, T, CNOT}	Fault-tolerant / error-corrected
{$R_x(\theta)$, $R_y(\theta)$, CNOT}	NISQ (tunable angles)
{U(θ,φ,λ), CX}	OpenQASM built-in
{RZ, SX, X, CZ}	**IBM Heron/Nighthawk native**
{Toffoli, H}	Alt. universal set

Simulation costs at a glance

Simulation	Memory	Best for
State vector	$O(2^n)$	General, small n
Unitary	$O(4^n)$	Small circuit analysis
Density matrix	4^n params	Noise / mixed states
Stabilizer	polynomial	Clifford-only circuits
MPS	$O(n \cdot \chi^2)$	1D, low-entanglement

Circuit-synthesis complexity landmarks (must memorize)

- **RCA delay:** 3 + 3 + 2(n−1) gate delays → 12 for n=4 (CLA reduces to 8).
- **Toffoli decomposition: 6 CNOTs + H, T, T†** single-qubit gates.
- **General 2-qubit gate:** minimum **3 CNOTs** (KAK).
- **Arbitrary n-qubit unitary: ~½·4^n CNOTs** (CSD/QSD).
- **Approximating U ∈ SU(2) to accuracy ε from {H, T}:** O(log^3.97(1/ε)) gates (**Solovay–Kitaev**).
- **QRAM gate activations: Θ(n)** vs. classical RAM's **Θ(2^n)**.

Classical ↔ quantum stack parallels

Classical 1950s	Today	Quantum
Algorithms	Algorithms	Algorithms
Assembly + Circuit synthesis	High-level language, Compiler, OS, Architecture	Q-DSL, Compilation, Unitary synthesis, Pulse shaping, Noise mitigation, Error correction
Vacuum tubes	Transistors	Qubits

OpenQASM 2 vs. 3 quick diff

	v2	v3
Version line	`OPENQASM 2.0;`	`OPENQASM 3;`
Classical types	only `creg bit[n]`	`int, uint, float, bool, bit[n]`
Control flow	only `if(c==int) qop;`	`if/else, for, while, def, extern`
Timing	none	`duration, stretch, delay, box`
Pulses	none	`defcal` (OpenPulse)
Standard header	`qelib1.inc`	`stdgates.inc`

Exam-Day Essentials — "If you can do these, you can ace it"

1. **Binary / adder circuitry.** Build HA, FA from gate equations; draw a 4-bit RCA; explain the adder/subtractor for 2's complement (XOR-flip + C_0=1 trick). Build a simple ALU from an n-bit adder + AND/OR/XOR arrays + MUXes selected by $F_0F_1F_2$. State the 8-op function table. *(Don't memorize 1's/2's representations or the multiplier circuit — they're asterisked out.)*
2. **Sequential / memory.** Draw SR-latch (cross-coupled NORs); give the 4-row truth table (Set/Reset/Store/Invalid). Convert SR → D-latch via `S=D, R=D⁻`. Assemble an n-bit register + register-enable (MUX in front of each D input). Compute RAM capacity from k,n.
3. **QRAM.** State the classical vs. quantum RAM comparison (especially **$\Theta(2^n)$ vs. $\Theta(n)$** gate activations). Describe bucket-brigade routing in both single-address and superposition modes.
4. **ISA / x86-64 / Fetch-Decode-Execute.** Walk through the 3-phase cycle. Read a short x86-64 snippet: recognize `movl, addl, pushq, popq, ret` and `-4(%rbp)` addressing. Distinguish ISA vs. microarchitecture. Name 3 ISAs.
5. **QOS pipeline.** Name the **four QOS components** (error mitigator, estimator, multi-programmer, scheduler) and the pipeline order: API → error-mitigator → estimator → multi-programmer → scheduler → QPUs.
6. **Simulation methods.** Explain each of the 5 methods; match a use-case to the right method (noise → density matrix; Clifford circuits → stabilizer; low-entangled 1D → MPS). State **Gottesman–Knill** precisely (stabilizer inputs + Clifford gates + Z-basis measurement ⇒ polynomial-time classical simulation).
7. **QST end-to-end (single qubit).** Given 10 000 samples in Z, X, Y bases: compute probabilities → `⟨σ_x⟩, ⟨σ_y⟩, ⟨σ_z⟩` → Bloch vector → `ρ = ½(I + x σ_x + y σ_y + z σ_z)`. Verify `Tr(ρ) = 1, ρ† = ρ, Tr(ρ²) = 1 for pure`.
8. **Strong vs. weak simulation.** Give definitions and costs; know Clifford → weak poly-time.
9. **OpenQASM 2 vs. 3.** Write a 3-line OpenQASM 2 snippet (e.g., Bell pair preparation). State what v3 adds (classical flow, `duration/stretch/delay/box`, `defcal`, multi-level IR). Recall gate modifiers (`ctrl @`, `inv @`, `pow(r) @`).
10. **DiVincenzo criteria.** List the five (plus two communication extensions).
11. **Unitary decomposition.** Decompose a 2-qubit unitary via **KAK**; count CNOTs via Cartan coordinates `(α,β,γ)`. Describe CSD for n-qubit (recursive). Know Givens for 2×2. Fill out the **3-method summary table**.
12. **Solovay–Kitaev.** State the result: `O(log^3.97(1/ε))` gates. Recall the 4-step recursive algorithm (look-up → error $U \cdot U_n\dagger$ → commutator `VWV†W†` → recurse). Key insight: accuracy squares each recursion.
13. **Toffoli / Oracle synthesis.** Toffoli = 6 CNOTs + H/T/T†. Deutsch oracle: constant → I or X; balanced → CNOT. Grover's three steps: **Superposition → Oracle → Diffusion**. Toffoli + H = universal quantum.
14. **C-U "ABC" form.** Know the 5-step procedure: decompose U into A,B,C with `ABC = I`, then `C → CNOT → B → CNOT → A` on target + controlled-phase `e^{iα}`.
15. **QAS — 4 dimensions + 7 search strategies.** Match: ADAPT-VQE = adaptive/greedy, DQAS = differentiable, GPT-QE = generative, PPO/DDQN/A3C = RL, MoG-VQE = evolutionary, MCGS = MCTS on graph, TPE/GP = Bayesian.

Common Pitfalls & Gotchas

- **Don't confuse "stabilizer simulation" with "stabilizer states".** Stabilizer simulation stores the stabilizer *generators* (Pauli operators), not amplitudes.
- **Gottesman-Knill requires all three:** Clifford gates AND stabilizer inputs AND Z-basis measurement. Miss any one and the guarantee collapses.
- **Solovay-Kitaev's exponent is `c ≈ 3.97`, not 3.** The slides say so explicitly.

- **OpenQASM 2 register semantics:** `qreg` auto-initializes to `|0⟩`; `creg` to 0. Don't write `reset` unnecessarily.
- **OpenQASM 3 `defcal` ≠ `def`.** `def` is a subroutine (classical+quantum); `defcal` attaches a pulse-level calibration to a gate.
- **`RZ` is "Virtual-Z" on IBM Heron:** 0 ns, 0% error — this is often **free** in circuit-depth budgets. Pack phase corrections into `RZ` wherever possible.
- **Toffoli is universal for *classical reversible*, not quantum alone.** You need **Toffoli + Hadamard** for quantum universality.
- **KAK counts CNOTs from Cartan coordinates:** $\alpha=\pi/4$, $\beta=\gamma=0 \Rightarrow$ **1 CNOT** (like CNOT itself). Only $\alpha \neq 0$ gives 1 CNOT; two non-zero $\Rightarrow$ 2 CNOTs; all three $\Rightarrow$ 3 CNOTs.
- **QRAM beats classical RAM *in gate activations* ($\Theta(n)$ vs $\Theta(2^n)$)**, not in raw bit count. That's the whole point of the comparison.
- **Bloch-vector purity check:** `|r| = 1` $\Rightarrow$ pure state (on sphere); `|r| < 1` $\Rightarrow$ mixed (inside sphere). Equivalent to `Tr(ρ²) = 1` vs. `< 1`.
- **`U(θ,φ,λ)` — get the order right.** λ acts *before* θ (Z-rotation), then the θ Y-rotation, then φ Z-rotation. So `H = U(π/2, 0, π)` with $\lambda=\pi$, $\varphi=0$.
- **"Real-time" classical (OpenQASM 3 target)** means within T_1, T_2; **"near-time"** (VQE outer loop, Shor co-prime) is looser.

One-Page Formula & Number Cheat Sheet

Binary / ALU

- `S_HA = X⊕Y, C_HA = XY · S_FA = X⊕Y⊕Z, C_FA = XY + Z(X⊕Y)`
- RCA delay = `3 + 3 + 2(n−1)` gate delays; CLA for n=4 → 8 gates; ALU 3-bit function code → 8 ops.

Memory / clock

- `f = 1/T. T = 0.5 ns ⇒ f = 2 GHz`.
- RAM capacity = $2^k \times n$ bits.
- Hierarchy times: reg 0.3 ns, L1 1.1 ns, L2 3.3 ns, L3 12.8 ns, RAM 62.9 ns, SSD 0.1 ms, disk 3 ms.

Quantum states / measurement

- State vector: $|\psi\rangle = \Sigma\, \alpha_i |i\rangle \in \mathbb{C}^{2^n}$; $P(i) = |\alpha_i|^2$.
- Density matrix: $\rho = \Sigma\, p_i |\psi_i\rangle\langle\psi_i|$, Hermitian PSD, $\mathrm{Tr}(\rho) = 1$, $\mathrm{Tr}(\rho^2) = 1 \Leftrightarrow$ pure.
- Single-qubit Bloch reconstruction: $\rho = \frac{1}{2}(I + x\,\sigma_x + y\,\sigma_y + z\,\sigma_z)$, with $\langle\sigma_B\rangle = p_{+B} - p_{-B}$.

Synthesis

- `U(θ,φ,λ) = [[cos(θ/2), −e^{iλ}sin(θ/2)], [e^{iφ}sin(θ/2), e^{i(φ+λ)}cos(θ/2)]]`.
- KAK: $U = (A_1 \otimes A_2)\cdot\exp[i(\alpha XX + \beta YY + \gamma ZZ)]\cdot(B_1 \otimes B_2)$; **3 CNOTs** generic, **1** for CNOT itself.
- CSD CNOT cost: **$\sim\frac{1}{2}\cdot 4^n$**. Givens cost: $O(n^3 \cdot 4^n)$.
- Solovay-Kitaev: **$O(\log^{3.97}(1/\varepsilon))$** gates.
- QRAM superposition query: $\Sigma_i\ \alpha_i |i\rangle_r \to \Sigma_i\ \alpha_i |i\rangle_r\ |X_i\rangle_o$.

Oracles

- Deutsch constant → `I` or `X`; balanced → `CNOT`.
- Toffoli = **6 CNOTs + H, T, T†**.
- Grover: **H → U_f (phase flip) → Diffusion**; $O(\sqrt{N})$ search.
- C-U = $e^{i\alpha} \cdot C \cdot X \cdot B \cdot X \cdot A$ with `ABC = I`.

Constraints to remember

- DiVincenzo: (1) scalable qubits, (2) init, (3) long T_1/T_2, (4) universal gate set, (5) readout. +2 for communication: interconvert & faithfully transmit flying qubits.
- Architecture constraints: **instruction set, quantum communication (SWAP chains), hardware noise, available parallel control**.