UCLA COMPUTER SCIENCE FIELD NOTES

CS M51A

Logic Design of Digital Systems

Boolean algebra, combinational and sequential design, arithmetic, control, and error codes

How these notes are written. I wrote each chapter the way I wish the material had been explained the first time: the real idea, the point where I usually get stuck, the simplest mental model that still stays honest, and a worked decision instead of a polished answer appearing from nowhere. Every chapter closes with practice and a fully written solution. If a sentence sounds impressive but does not help me solve or explain something, it does not belong here.

Daniel Mastick

Curriculum map, working notes, practice, and solutions

Course map

Week	Core thread	What should be solid by the end
1	**Number systems and Boolean algebra**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
2	**Combinational specification and minimization**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
3	**Standard combinational modules**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
4	**Arithmetic circuits**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
5	**Latches, flip-flops, and timing**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
6	**Finite-state machines**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
7	**Datapaths and controllers**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
8	**Error detection and correction**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
9	**Integration workshop**	Connect at least three chapters in one end-to-end problem and explain where their contracts meet.
10	**Synthesis and project review**	Audit assumptions and explain a complete solution from interface to failure handling.

Scope anchor: https://catalog.registrar.ucla.edu/course/2024/COMSCIM51A. The sequence is aligned to the official catalog description and expanded into a practical ten-week study plan.

How I would use this packet

Treat each numbered section as one chapter. Read the formal explanation, pause at the student interpretation, and see whether the easy version still says the same thing. Rework the example without looking, then cover the solution in the chapter practice box. The cumulative set at the end is deliberately mixed: knowledge becomes durable when I have to choose the tool instead of being told which chapter I am in.

Contents

CHAPTER 1 Number systems and Boolean algebra

The idea

Binary, hexadecimal, signed representations, and Boolean identities connect information to circuits. Duality, De Morgan's laws, and canonical forms provide algebraic ways to transform logic.

Rebuild the chapter from first principles

The claim I need to own is this: Binary, hexadecimal, signed representations, and Boolean identities connect information to circuits. Duality, De Morgan's laws, and canonical forms provide algebraic ways to transform logic. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of number systems and boolean algebra. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach number systems and boolean algebra on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from the course foundations to combinational specification and minimization. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Binary, hexadecimal, signed representations, and Boolean identities connect information to circuits.
- **Mechanism.** Duality, De Morgan's laws, and canonical forms provide algebraic ways to transform logic.
- **Boundary.** The useful test is whether I can apply number systems and boolean algebra to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when number systems and boolean algebra is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I keep value representation separate from the bit pattern; the same bits can mean different things under signed and unsigned interpretation.

The easy version

Binary, hexadecimal, signed representations, and Boolean identities connect information to circuits.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Convert decimal 45 to binary and hex. The

conclusion is 45=32+8+4+1, so binary is 00101101 in eight bits and hexadecimal is 0x2D.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Convert decimal 45 to binary and hex.
Solution. 45=32+8+4+1, so binary is 00101101 in eight bits and hexadecimal is 0x2D.

Practice 2. Give a short oral explanation of number systems and boolean algebra that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct number systems and boolean algebra from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a number systems and boolean algebra problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 2 Combinational specification and minimization

The idea

Combinational outputs depend only on current inputs. Truth tables, minterms, maxterms, Karnaugh maps, and don't-care conditions move from specification to simplified equations.

Rebuild the chapter from first principles

The claim I need to own is this: Combinational outputs depend only on current inputs. Truth tables, minterms, maxterms, Karnaugh maps, and don't-care conditions move from specification to simplified

equations. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of combinational specification and minimization. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach combinational specification and minimization on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from number systems and boolean algebra to standard combinational modules. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Combinational outputs depend only on current inputs.
- **Mechanism.** Truth tables, minterms, maxterms, Karnaugh maps, and don't-care conditions move from specification to simplified equations.
- **Boundary.** The useful test is whether I can apply combinational specification and minimization to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when combinational specification and minimization is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I minimize only after the truth table is trustworthy; a beautiful circuit for the wrong specification is still wrong.

The easy version

Combinational outputs depend only on current inputs.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Simplify F=AB+AB'. The conclusion is Factor A(B+B')=A. In a K-map, the two adjacent minterms group by eliminating B.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Simplify F=AB+AB'.
Solution. Factor A(B+B')=A. In a K-map, the two adjacent minterms group by eliminating B.

Practice 2. Give a short oral explanation of combinational specification and minimization that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison

according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct combinational specification and minimization from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a combinational specification and minimization problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 3 Standard combinational modules

The idea

Multiplexers, decoders, encoders, comparators, and programmable arrays are reusable logic blocks. A multiplexer can implement any Boolean function by using variables as selects and cofactors as data inputs.

Rebuild the chapter from first principles

The claim I need to own is this: Multiplexers, decoders, encoders, comparators, and programmable arrays are reusable logic blocks. A multiplexer can implement any Boolean function by using variables as selects and cofactors as data inputs. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of standard combinational modules. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.

4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach standard combinational modules on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from combinational specification and minimization to arithmetic circuits. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Multiplexers, decoders, encoders, comparators, and programmable arrays are reusable logic blocks.
- **Mechanism.** A multiplexer can implement any Boolean function by using variables as selects and cofactors as data inputs.
- **Boundary.** The useful test is whether I can apply standard combinational modules to a new case and explain the assumption that makes the answer valid.

- **Decision test.** I should be able to say when standard combinational modules is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I look for a standard block before expanding gates; it keeps intent visible.

The easy version

Multiplexers, decoders, encoders, comparators, and programmable arrays are reusable logic blocks.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Implement a three-variable function with a 4-to-1 mux. The conclusion is Use two variables as select lines. For each select combination, evaluate the function as 0, 1, x, or x' and connect that cofactor to the corresponding data input.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Implement a three-variable function with a 4-to-1 mux.
Solution. Use two variables as select lines. For each select combination, evaluate the function as 0, 1, x, or x' and connect that cofactor to the corresponding data input.

Practice 2. Give a short oral explanation of standard combinational modules that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct standard combinational modules from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a standard combinational modules problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the

numerical or verbal result as unverified.

CHAPTER 4 Arithmetic circuits

The idea

Half and full adders build ripple adders; carry lookahead trades more logic for less delay. Subtraction uses two's complement addition. Overflow, carry, saturation, and comparison depend on representation.

Rebuild the chapter from first principles

The claim I need to own is this: Half and full adders build ripple adders; carry lookahead trades more logic for less delay. Subtraction uses two's complement addition. Overflow, carry, saturation, and comparison depend on representation. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of arithmetic circuits. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.

- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach arithmetic circuits on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from standard combinational modules to latches, flip-flops, and timing. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Half and full adders build ripple adders; carry lookahead trades more logic for less delay.
- **Mechanism.** Subtraction uses two's complement addition.
- **Boundary.** Overflow, carry, saturation, and comparison depend on representation.
- **Decision test.** I should be able to say when arithmetic circuits is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I do not use the carry-out bit as the signed overflow flag; they answer different questions.

The easy version

Half and full adders build ripple adders; carry lookahead trades more logic for less delay.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Detect signed addition overflow. The conclusion is Overflow occurs when operands share a sign and the result has the opposite sign, equivalently when carry into the sign bit differs from carry out.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Detect signed addition overflow.
Solution. Overflow occurs when operands share a sign and the result has the opposite sign, equivalently when carry into the sign bit differs from carry out.

Practice 2. Give a short oral explanation of arithmetic circuits that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct arithmetic circuits from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a arithmetic circuits problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 5 Latches, flip-flops, and timing

The idea

Sequential systems store state. Latches are level-sensitive; flip-flops are edge-triggered. Setup, hold, clock-to-Q, propagation delay, and clock skew determine whether a synchronous design is physically safe.

Rebuild the chapter from first principles

The claim I need to own is this: Sequential systems store state. Latches are level-sensitive; flip-flops are edge-triggered. Setup, hold, clock-to-Q, propagation delay, and clock skew determine whether a synchronous design is physically safe. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of latches, flip-flops, and timing. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach latches, flip-flops, and timing on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from arithmetic circuits to finite-state machines. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Sequential systems store state.
- **Mechanism.** Latches are level-sensitive; flip-flops are edge-triggered.
- **Boundary.** Setup, hold, clock-to-Q, propagation delay, and clock skew determine whether a synchronous design is physically safe.
- **Decision test.** I should be able to say when latches, flip-flops, and timing is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I draw the clock boundary and calculate the longest combinational path before choosing a frequency.

The easy version

Sequential systems store state.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: State the basic setup timing inequality. The conclusion is Clock period must cover clock-to-Q plus maximum combinational delay plus setup time, with skew and uncertainty included according to sign convention.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. State the basic setup timing inequality.
Solution. Clock period must cover clock-to-Q plus maximum combinational delay plus setup time, with skew and uncertainty included according to sign convention.

Practice 2. Give a short oral explanation of latches, flip-flops, and timing that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct latches, flip-flops, and timing from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and

choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a latches, flip-flops, and timing problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 6 Finite-state machines

The idea

An FSM combines a finite state register with next-state and output logic. Moore outputs depend on state; Mealy outputs also depend on input. State diagrams, tables, encoding, minimization, and unreachable-state recovery complete design.

Rebuild the chapter from first principles

The claim I need to own is this: An FSM combines a finite state register with next-state and output logic. Moore outputs depend on state; Mealy outputs also depend on input. State diagrams, tables, encoding, minimization, and unreachable-state recovery complete design. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of finite-state machines. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.

- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach finite-state machines on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from latches, flip-flops, and timing to datapaths and controllers. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** An FSM combines a finite state register with next-state and output logic.
- **Mechanism.** Moore outputs depend on state; Mealy outputs also depend on input.
- **Boundary.** State diagrams, tables, encoding, minimization, and unreachable-state recovery complete design.
- **Decision test.** I should be able to say when finite-state machines is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I name states by what the machine remembers, not by arbitrary numbers.

The easy version

An FSM combines a finite state register with next-state and output logic.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Design a detector for input suffix 101. The conclusion is Use states representing longest matched prefix: none, 1, 10, and detected as output or transition. Preserve overlaps by returning to the state matching the longest suffix.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Design a detector for input suffix 101.
Solution. Use states representing longest matched prefix: none, 1, 10, and detected as output or transition. Preserve overlaps by returning to the state matching the longest suffix.

Practice 2. Give a short oral explanation of finite-state machines that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct finite-state machines from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a finite-state machines problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 7 Datapaths and controllers

The idea

Algorithmic systems separate a datapath of registers, operators, muxes, and buses from a controller that sequences control signals. Register-transfer notation describes work per cycle and exposes resource sharing.

Rebuild the chapter from first principles

The claim I need to own is this: Algorithmic systems separate a datapath of registers, operators, muxes, and buses from a controller that sequences control signals. Register-transfer notation describes work per cycle and exposes resource sharing. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of datapaths and controllers. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach datapaths and controllers on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from finite-state machines to error detection and correction. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Algorithmic systems separate a datapath of registers, operators, muxes, and buses from a controller that sequences control signals.
- **Mechanism.** Register-transfer notation describes work per cycle and exposes resource sharing.
- **Boundary.** The useful test is whether I can apply datapaths and controllers to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when datapaths and controllers is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I mark which value is stored at each clock edge; that prevents mixing combinational calculation with state update.

The easy version

Algorithmic systems separate a datapath of registers, operators, muxes, and buses from a controller that sequences control signals.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Sketch repeated-add multiplication. The conclusion is Datapath holds multiplicand, multiplier/count, and accumulator. Controller clears,

tests, conditionally adds, shifts or counts, and asserts done when iterations finish.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Sketch repeated-add multiplication.
Solution. Datapath holds multiplicand, multiplier/count, and accumulator. Controller clears, tests, conditionally adds, shifts or counts, and asserts done when iterations finish.

Practice 2. Give a short oral explanation of datapaths and controllers that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct datapaths and controllers from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a datapaths and controllers problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 8 Error detection and correction

The idea

Parity detects any odd number of bit flips but cannot locate them. Hamming distance determines detection and correction capability; linear block codes use generator and parity-check matrices, and syndrome identifies an error pattern.

Rebuild the chapter from first principles

The claim I need to own is this: Parity detects any odd number of bit flips but cannot locate them. Hamming distance determines detection and correction capability; linear block codes use generator and parity-check matrices, and syndrome identifies an error pattern. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of error detection and correction. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach error detection and correction on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from datapaths and controllers to the cumulative course model. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Parity detects any odd number of bit flips but cannot locate them.
- **Mechanism.** Hamming distance determines detection and correction capability; linear block codes use generator and parity-check matrices, and syndrome identifies an error pattern.
- **Boundary.** The useful test is whether I can apply error detection and correction to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when error detection and correction is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I translate the coding claim into distance: detect d-1 errors, correct floor((d-1)/2).

The easy version

Parity detects any odd number of bit flips but cannot locate them.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: What minimum distance corrects one error and detects two? The conclusion is Correction of one needs distance at least 3; simultaneous guarantee of detecting two also needs at least 3. Extended Hamming distance 4 adds stronger double-error detection.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. What minimum distance corrects one error and detects two?
Solution. Correction of one needs distance at least 3; simultaneous guarantee of detecting two also needs at least 3. Extended Hamming distance 4 adds stronger double-error detection.

Practice 2. Give a short oral explanation of error detection and correction that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that

licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct error detection and correction from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a error detection and correction problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

Final study check

I should be able to close this packet and reconstruct the core models, not merely recognize their names. My final check is to explain one complete problem aloud: what the inputs mean, which invariant or contract controls the work, why the method is legal, what it costs, how it can fail, and what evidence would convince another engineer.