CS 2110 - Computer Organization and Programming

Jerry Chen

Contents

1 Bits, Data Types, and Operations

1.1 Turing Machines and Turing-Completeness

1.1.1 Definitions and Properties

A system is Turing-complete if it can perform any computation that a Turing machine can, given enough time and memory. This is a capability statement: the system can express any algorithm that is computable in the Turing sense.

1.1.2 Turing Machine Components

A Turing machine is a mathematical model of computation consisting of:

- An infinite tape divided into cells.
  - Each cell contains a symbol: typically `0`, `1`, or blank.
- A head positioned at one cell at a time.
  - The head can read the current symbol, write a symbol, and move left or right by one cell.
- A finite set of states the machine can be in (e.g., start, halt/stop, accept, reject).
- A transition function that defines the machine's behavior:
  - It specifies what action to take based on the current state and the symbol currently under the head.
  - The action includes: what symbol to write, which direction to move, and what the next state is.

1.2 Machine Abstractions and the Translation Pipeline

1.2.1 From Problem to Hardware Execution

A computer system can be understood as a stack of abstraction layers that progressively remove ambiguity and translate human intent into hardware-executable operations. Each layer provides an interface to the layer above it while hiding lower-level implementation details.

1. Problem statement: A human description of what needs to be done, often expressed in natural language and therefore ambiguous.
2. Algorithm: A precise, unambiguous procedure that resolves the ambiguity in the problem statement (a step-by-step method that can be carried out mechanically).
3. High-level language / program: A human-friendly implementation of the algorithm in a programming language (e.g., C, C++, Python). This representation prioritizes readability and maintainability.
4. Assembly language: A low-level, human-readable representation of CPU operations using mnemonics and symbols; it is effectively a readable form of what will become 1s and 0s. An assembler translates assembly language into machine code.
5. Instruction Set Architecture (ISA): The hardware-facing "instruction API" defining the machine instructions, registers, memory model, and conventions for a family of processors (e.g., x86-64). Different CPU designs can implement the same ISA.
6. Microarchitecture: The internal design of a specific CPU that implements the ISA (pipelines, caches, execution units, etc.). Two CPUs can share an ISA but have different microarchitectures.
7. Digital logic, circuits, and physics: The physical realization of computation using logic gates, circuits, and ultimately electrical/physical behavior.

1.2.2 Assembly, Machine Code, and ISA Ownership

Assembly language is converted into a binary executable by an assembler. Machine code is the actual binary instruction encoding that the CPU executes.

Each chip manufacturer has its own ISA (or ISA family) or supports specific ISAs, and software must target an ISA to run directly on that hardware. The same ISA can be implemented differently by different CPUs via different microarchitectures.

1.3 Bits, Binary, Octal, Decimal, and Hexadecimal

1.3.1 Definition and Properties

A bit is the smallest unit of information and can be in one of two states: 0 or 1 (often described as off/on). In hardware, computers use tiny electronic switches that respond to voltage levels: - Presence of voltage is treated as 1 - Absence of voltage is treated as 0

This binary approach is practical because detecting "present vs absent" is simpler and more reliable than continuously measuring precise analog values.

1.3.2 Binary Notation

Binary is just a string of 1's and 0's. However, just like the decimal (base-10) system, binary can represent any integer, but just in base-2.

For example, 1011_2 is equal to $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$.

There are both signed and unsigned integers. For unsigned integers, all bits are used to represent the value. For signed integers, the leftmost bit is reserved to display the sign. See two's complement below for more details.

1.3.3 Octal Notation

Octal is base-8. One octal digit corresponds to 3 binary bits. In written notation, octal notation beings with a leading `0`, or `0o`.

It is easy to convert between octal and binary due to the 1:3 ratio/correspondance. For example, `010110` in binary converts to `26` in octal.

1.3.4 Hexadecimal Notation

Hexadecimal is base-16 and one hex digit corresponds to 4 binary digits. In written notation, hexadecimal begins either with a leading `x` or `0x`.

Similar to octal notation, we read from binary to hexadecimal by taking chunks of four binary digits. However, due to the limitations of the decimal system in representing numbers 10-15 (the entire hex range is 0-15), we use letters A-F to represent 10-15.

1.4 Two's Complement and Integer Operations

1.4.1 Definition and Properties

An unsigned integer uses all bits to represent a non-negative magnitude. A signed integer needs a way to represent negative values as well as positive values.

The standard signed representation on modern hardware is two's complement. The algorithm goes: 1. Take a target negative number (e.g., -5). Express it as its positive binary counterpart (imagine it is an unsigned integer, but with a guaranteed leading zero): `0101`. 2. Invert the bits: `1010`. 3. Add 1 to the inverted bits: `1011`.

Note: The negative value doesn't have much interpretability by itself. However, if you apply the same algorithm (invert bits, then +1), you get the original positive number again, which allows you to confirm the identity of negative numbers. - However, with this system, comparisons between negative numbers can be done properly using the same logic as with unsigned integers (i.e., -4 (`1100`) is greater than `1011` simply by comparing the bits starting from the left).

1.4.2 Advantages of Two's Complement

Core advantages over naively using the leftmost bit as sign indicator: 1. Only contains 1 zero; if we only used the leftmost bit to determine sign, both `0000` and `1000` would be 0. 2. Subtraction can use the same adding logic as addition (i.e., 7 + (-3) = 4 after doing math using two's complement for -3). 3. Easier overflow detection (explained below).

1.4.3 Range of Representation

For an n-bit two's complement integer: - The representable range is from -2^{n-1} to $2^{n-1}-1$. - The negative range has one extra value compared to the positive range because zero is included in the non-negative side. - Essentially, half of 2^n on negative side, that same value minus one on the positive side.

For an unsigned integer, the range is from 0 to 2^n-1. For a signed integer, but not 2's complement, the range goes from $-2^{n-1}+1$ to $2^{n-1}-1$.

1.4.4 Comparing Signed Integers

To compare two's complement signed integers using their bit patterns:

- If the most significant bit (MSB) is the same for both numbers (same sign):
  - Compare the remaining bits as usual; equivalently, you can "ignore" the first bit and compare magnitudes within that sign group.
  - Example: with 4 bits, `1111` (-1) is greater than `1110` (-2). Ignoring the first bit gives `111` vs `110`, and the ordering matches (-1 > -2).
- If the MSB differs (different signs):
  - The one with MSB = 0 is positive and is always greater than the one with MSB = 1 (negative).

1.4.5 Adding/Subtracting Signed Integers

Addition works exactly the same way as before, with any extra bits from carry being discarded if exceeding the number of bits allocated to storing the integer. Subtraction then becomes an addition operation with a negative integer represented using two's complement. However, with this new system, we need to catch potential overflow errors, which will not throw an error at runtime, but instead silently cause unintended behavior.

Overflow occurs when the mathematical result is outside the representable range for the fixed number of bits. - For example, when adding two positive numbers (leftmost bit is originally 0), a carry could cause the MSB (most significant bit) to become 1, making the number negative, which is incorrect. - Similarly, when adding two negative numbers (leftmost bit is both 1), exceeding the negative representation limit could cause the leftmost bit to become 0, making the number positive, which is also incorrect.

The good news is that when overflow does occur, as long as you are adding only two binary numbers together, the sign will always flip, which allows us to easily detect overflow. Additionally, it is impossible to overflow if one number is positive and the other is negative, because the resulting magnitude must be smaller than the maximum of the absolute values of either operand.

1.4.6 Widening Conversions and Multiplication with Signed Integers

Sign extension preserves the numeric value when increasing bit-width in two's complement (e.g., from 4 bits to 32 bits). For example, this is used during widening casts, or during arithmetic right shifts, or during signed integer multiplication. - If the number is positive (leading bit is 0), extend by adding 0s on the left. - If the number is negative (leading bit is 1), extend by adding 1s on the left.

When multiplying two signed two's complement integers in a fixed-width machine representation, the operation is performed as if the numbers were extended to a width large enough to hold the product, then truncated back to the chosen width.

1. Sign-extend both operands to a wider width appropriate for the result.
   - Example: multiplying two 4-bit signed values conceptually produces an 8-bit signed result (before truncation).
2. Perform binary multiplication using the same structure as base-10 multiplication: form partial products by shifting and adding.
3. Keep only the fixed-width result bits for the chosen width, discarding any higher-order bits beyond that width.

1.5 Bitwise Operations and Bitmasks

1.5.1 Core Bitwise Operators

Bitwise operators act on bitstrings (individual bits of an integer representation).

- Bitwise NOT (~, ~): flips each bit ($0 \to 1$, $1 \to 0$).
  - Note: `!` is logical NOT in C/C++ (it converts to a boolean-like result), not bitwise NOT.
- Bitwise AND (&, &): output bit is 1 only if both input bits are 1 at that position.
- Bitwise OR (|, |): output bit is 1 if at least one input bit is 1 at that position.
- Bitwise XOR (^): output bit is 1 if exactly one input bit is 1 at that position (equivalently, an odd number of 1s among the two inputs).

Derived operators: - NAND: apply AND, then invert. - NOR: apply OR, then invert. These appear frequently in digital circuits because NAND/NOR gates can be efficient building blocks at the transistor level.

1.5.2 Shifts

A shift moves bits left or right and fills in new bits on the vacated side.

- Logical left shift (<<)
  - Leftmost bits shifted out are discarded.
  - Rightmost vacated bits are filled with 0.
  - Often corresponds to multiplying by 2 per shift position for unsigned values (and for signed values when no overflow occurs).
  - Note: In C/C++, left-shifting into or past the sign bit of a signed type can be undefined behavior; use unsigned types when you intend pure bit manipulation.
- Logical right shift (>> for unsigned types)
  - Rightmost bits shifted out are discarded.
  - Leftmost vacated bits are filled with 0.
  - Often corresponds to dividing by 2 per shift position and discarding fractions (for unsigned values).
- Arithmetic right shift
  - The leftmost bit (the sign bit) is replicated to preserve the sign.
  - Rightmost bits shifted out are discarded.
  - Note: In many languages/architectures, right shift of signed integers behaves like an arithmetic shift, but in C/C++ the behavior for signed right shift is implementation-defined.

1.5.3 Bitmasks as Packed Boolean Flags

A bitmask uses individual bits of an integer to store independent on/off flags. For example, a 32-bit integer can store up to 32 boolean flags (one per bit position).

To create an individual flag mask, start from 1 and shift it to the desired bit position. This produces a mask with exactly one bit set.

Example:

```
int O_RDWR = 1 << 1; // 0x2
```

The left shift turns `...00000001` into `...00000010`, creating a single "on" bit that can represent one flag.

Multiple masks can be combined into a single `flags` value using bitwise OR, since OR preserves any bit that is set in either operand.

1.5.4 Logical Operations with Bitmasks

CLEAR at bit X: -> 0 - Use AND where bit X is 0 and all others are 1

SET at bit X: -> 1 - Use OR with a mask where bit X is 1 and all else is 0

TOGGLE at bit X: -> invert - Use XOR with a mask where bit X is 1 and all else is 0 - This is the ^ symbol

---

To check whether a specific flag is present inside a packed `flags` value, use bitwise AND with the corresponding mask:

```
if (flags & O_RDWR) {
    // do something
}
```

The AND result is nonzero exactly when the masked bit(s) are set in `flags`.

1.6 Radix Point, Fractional Values, and IEEE Floats

1.6.1 Radix Point Notation

Just like 10.5 is a decimal number representing numbers which are not whole numbers, there is an equivalent in binary.

For example, `101.011` is equal to $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 5.375$ - The `.` here is called a radix point.

1.6.2 IEEE Floating Point Representation

IEEE 754 floats (single precision) uses 32 bits: `SEM` - 1 bit sign (S) - 8 bits exponent field (E), stored with a bias of 127 - 23 bits fraction/mantissa field (M)

For normal floats: $V = -1^S \times 1.M \times 2^{E-127}$. - Just like how multiplying by 10^x shifts x positions in decimal, multiplying by 2^{E-127} shifts by $E - 127$ places.

The exponent is stored in biased form (rather than two's complement) so exponent magnitudes can be compared efficiently by comparing the exponent field as an unsigned integer after handling the sign.

IEEE 754 doubles use 64 bits: `SEM` - 1 bit sign (S) - 11 bits exponent field (E), stored with a bias of 1023 - 52 bits fraction/mantissa field (M)

For normal doubles: $V = -1^S \times 1.M \times 2^{E-1023}$

1.6.3 Converting IEEE Decimal Representations

To move from a decimal representation to an IEEE floating point representation, we need to determine the three different components first, being the sign, exponent, and mantissa.

1. The sign is zero if it's positive and one if the number is negative.
2. Next, we determine the mantissa, which is essentially a binary decimal point value, such as 1.1101. The way we determine this is by converting the integer portion into binary and then also converting the decimal portion into binary. For example $7.125 \rightarrow 111_2 + 0.001_2$, where 0.001_2 represents 1/8.
3. Finally, we convert 111.001_2 into $1.11001_2 \times 2^2$. It is raised to the second power because we shift the decimal point two places to the left from the initial binary decimal notation. In this way, we have found both M and E, where M represents the number to the right of the decimal point and $E - \text{bias} = \text{exponent}$. Note that M is zero-padded for further digits to the right, as it's 23-bit for floats.
4. From 2^2, we obtain that, assuming this is a floating-point number, we have $E - 127 = 2$, and so $E = 129$, and we must then convert that to binary.

1.6.4 Special Numbers (Exponent All 1s)

Special cases are determined by the exponent and mantissa fields:

- Zero: E = 0, M = 0 (sign bit distinguishes +0 and -0 in IEEE 754)
- Infinity: E = all 1s, M = 0, sign determines $+\infty$ vs $-\infty$
- NaN: E = all 1s, M $\neq$ 0 (many possible NaN payloads)

Note: The raw notes used "M = 255" for infinity/NaN, but 255 refers to the all-ones exponent value in 8-bit exponent fields (i.e., E = 255) for single precision.

1.6.5 Subnormals (Exponent All 0s, Nonzero Mantissa)

When E = 0 and M $\neq$ 0, the number is subnormal (denormal). The leading significand bit is no longer assumed to be 1:

- Instead of `1.M`, the significand is `0.M`.
- For single precision, the effective exponent is `1 - 127 = -126` (rather than `0 - 127`).

Examples (single precision):

- Smallest normal: E = 1, M = 0 → `1.0 * 2^-126`
- Smallest subnormal: E = 0, M = 1 → `0.000...001_2 * 2^-126 = 2^-149`

1.7 Text Encoding

1.7.1 ASCII and Unicode Encodings

ASCII is an older character encoding that represents a limited set of characters. Modern text encoding is based on Unicode, with common encodings including UTF-16 and UTF-8.

- UTF-8 is backwards compatible with ASCII (ASCII bytes map directly to the same code points in UTF-8).
- UTF-8 represents characters using a variable number of bytes, typically 1 to 4 bytes depending on the character.

ascii -> 8 bits / 1 byte; delimited based on size

The conversion from all uppercase to lowercase letters in binary is 32 (+32 to become lowercase, -32 the other way around). This is equivalent to toggling the bit corresponding to 2^5.

2 Circuits and Transistors

2.1 Transistors in Logic Gates

2.1.1 Definition and Properties

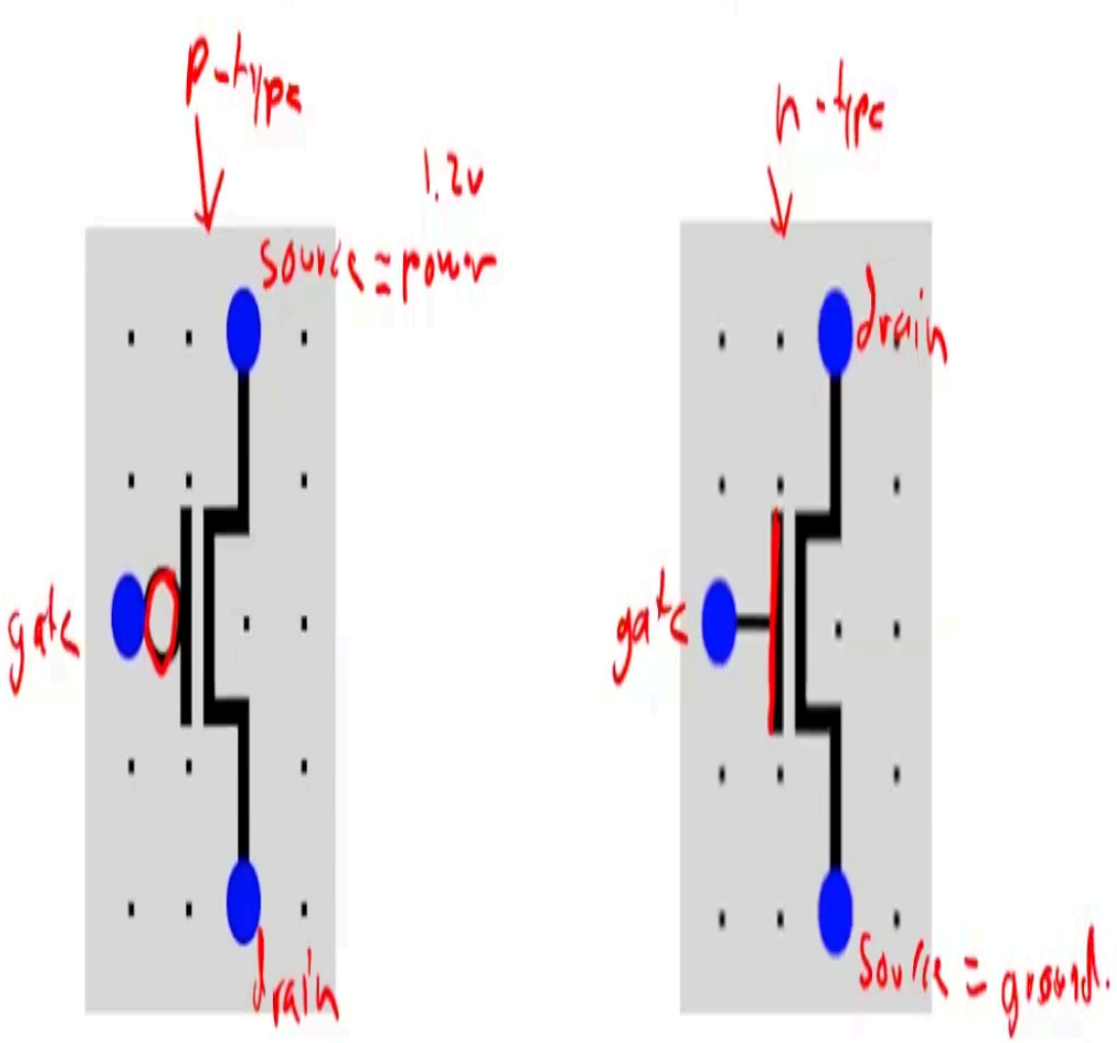
consists of switch (on/off) + amplifier (small signal in, triggers a bigger signal out)

MOSFET (metal-oxide-semiconductor field-effect transistor) has 3 connections (pins), depending on the type - Gate: control terminal - Source: where charge carriers come from - Drain: where charge goes to

2.1.2 n-type transistor and p-type transistors

For an *n-type (NMOS)* transistor, its source is GROUND (null). If its gate has a 0 signal, it acts as an open circuit. - In a diagram, it is denoted without a circle.

For a *p-type (PMOS)* transistor, its source is POWER. If its gate has a 0 signal, it acts as a closed circuit. - Denoted with a circle.



A MOS transistor acts as a voltage-controlled switch between source and drain. For a p-type transistor, the source is power while the drain is at the bottom, while for an n-type transistor the source is ground at the bottom while the drain is at the top.

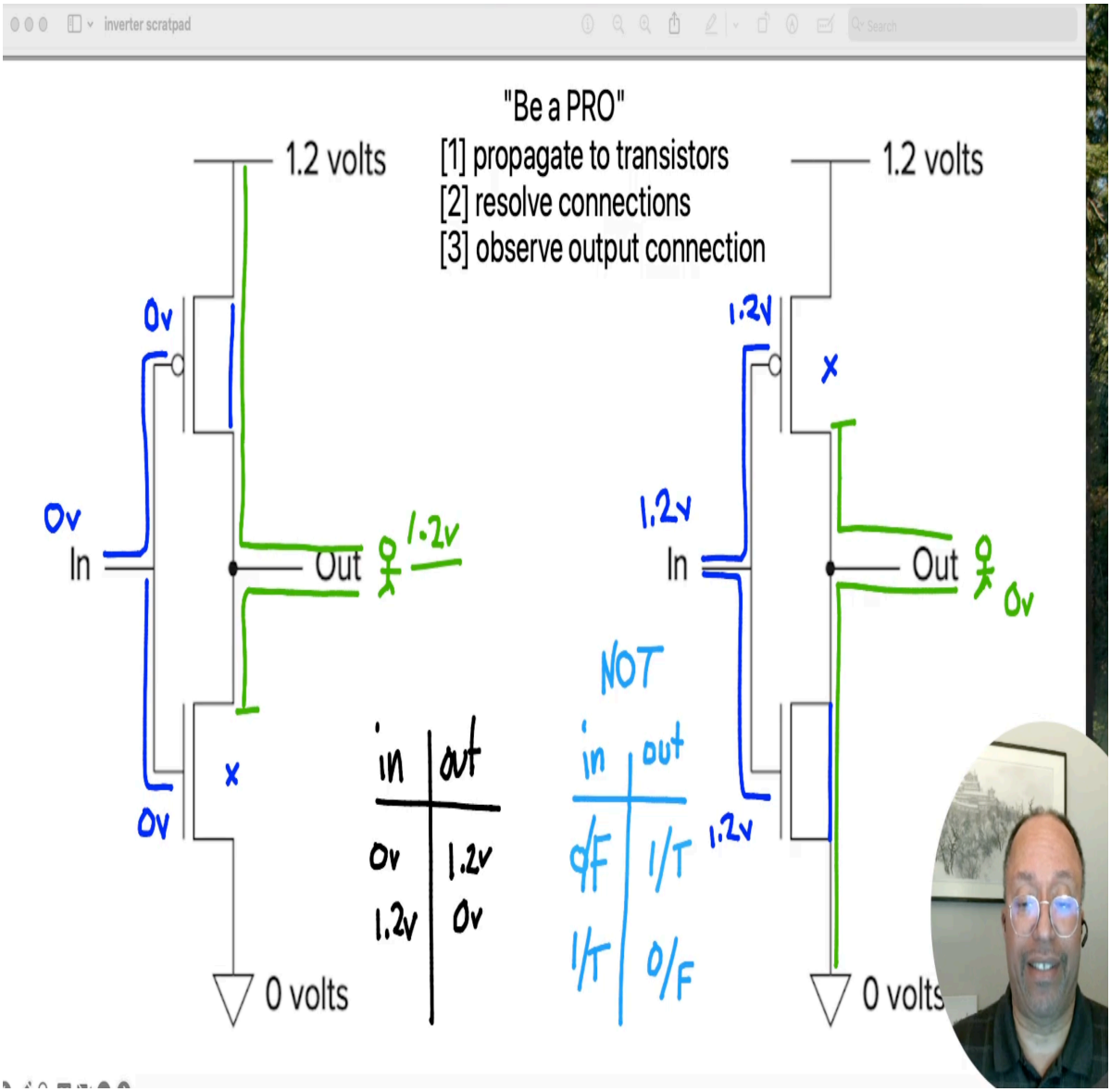
When p-type transistors are LOW, it acts as a CLOSED circuit so it does connect the source (power) to the drain. When it is HIGH, it acts as an OPEN circuit, providing no signal. Groups of p-type transistors form the "pull-up" network in more complex circuit components like NAND/NOR gates (in parallel for NAND, in series for NOR) outputting a signal of 1.

When n-type transistors are LOW, it acts as an OPEN circuit so it does not connect the source (ground) to the drain. When it is HIGH, it acts as a CLOSED circuit, connecting source/ground and "outputting" a signal of 0. This is why n-type transistors (in series for NAND gates, in parallel for NOR gates) are part of the "pull-down" network, pulling the output down to 0.

For any valid input combination, only one pull-up/pull-down network should be active at a time to influence the final output.

2.1.3 Tracing Circuit Diagrams with Transistors

1. Propagate from source to transistors
2. Resolve connections: based on the identity of transistors, determine how they will react ('x' for open circuit; draw line as connection for closed circuit)
3. Observe output connection: what is the output connected to? Is it able to reach the power line (top)? Is it able to reach the bottom (ground)?



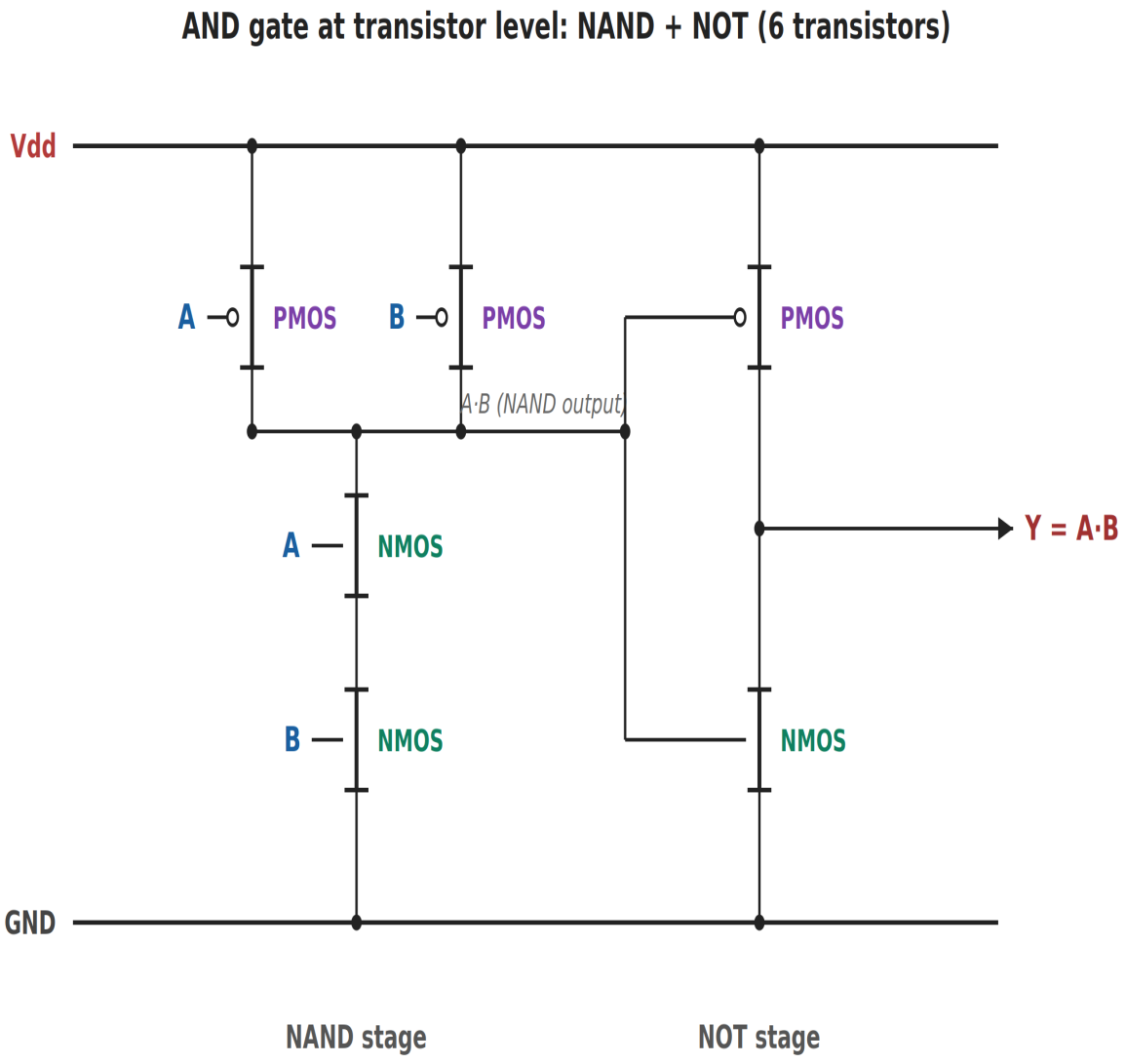
Working “NOT” logic gate (CMOS inverter). it is a circuit that implements a Boolean function, where the input is some voltage that is either 0 volts or a fixed value like 1.2 volts, and the output is also essentially an on/off state in voltage.

2.1.4 AND and OR Gates (NAND/NOR)

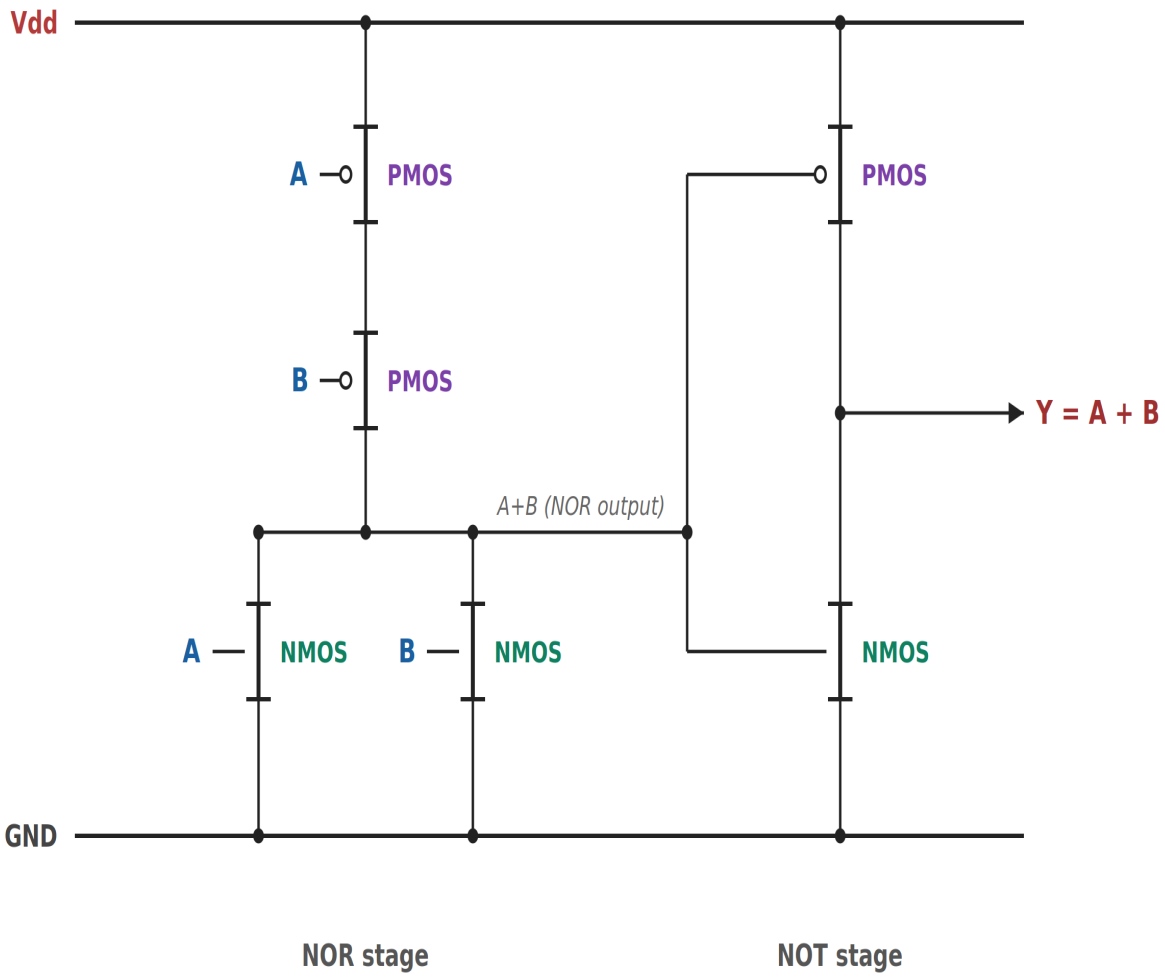
A dot means the wires are truly connected; if crossing but no dot at junction point, then they are not connected (i.e., imagine different heights).

Remembering Imagine that NAND is BOTH directions expansion (parallel on top stretching rightward, series on bottom stretching downward). Then NOR is ONE directional expansion (on the top series tries stretching downward, but is blocked by rightward expansion on the bottom being in parallel).

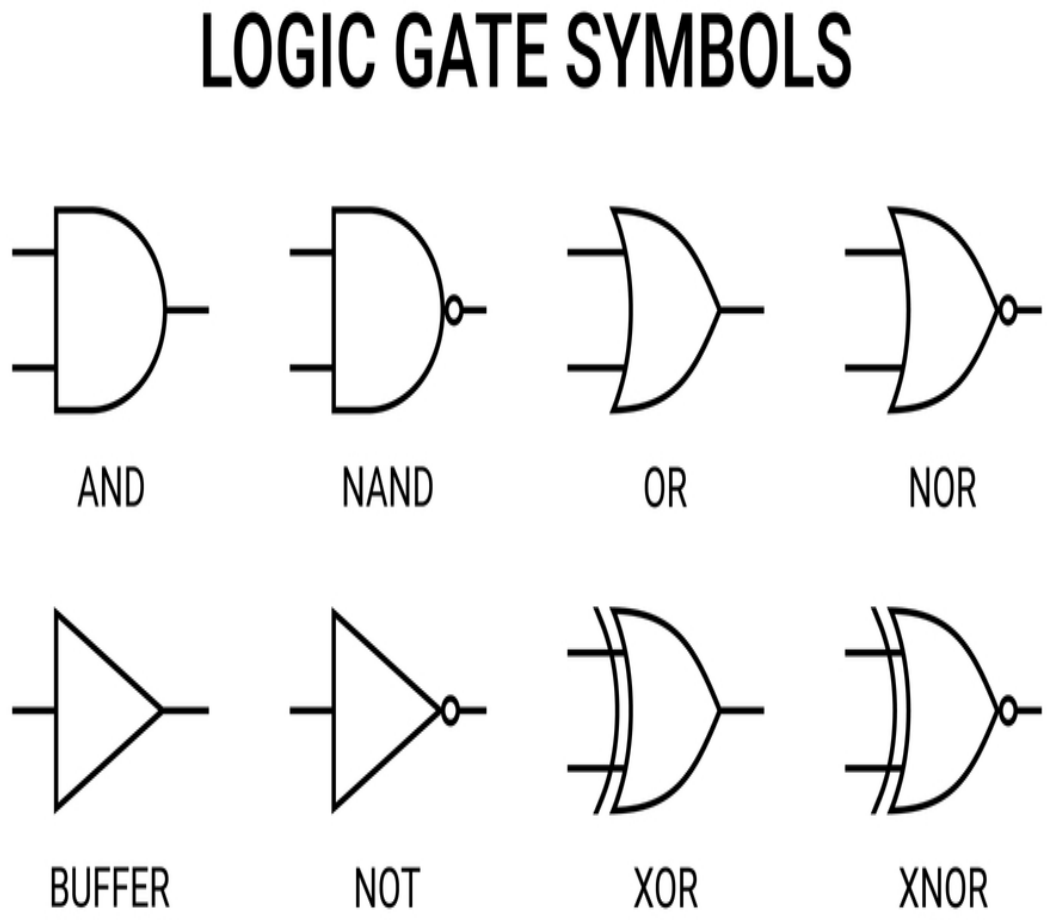
For both, transistors are stretched top-bottom (north-south). P-type is on top, with power, and n-type is on bottom, with ground. Additionally, sources contribute in to transistor gates in alternating form; i.e., each source feeds to 1 n-type and 1 p-type gate source.


AND gate at transistor level: NAND + NOT (6 transistors)
Vdd
A
PMOS
B
PMOS
PMOS
A·B (NAND output)
A
NMOS
B
NMOS
NMOS
Y = A·B
GND
NAND stage
NOT stage

OR gate at transistor level: NOR + NOT (6 transistors)



2.1.5 Logic Gate Symbols



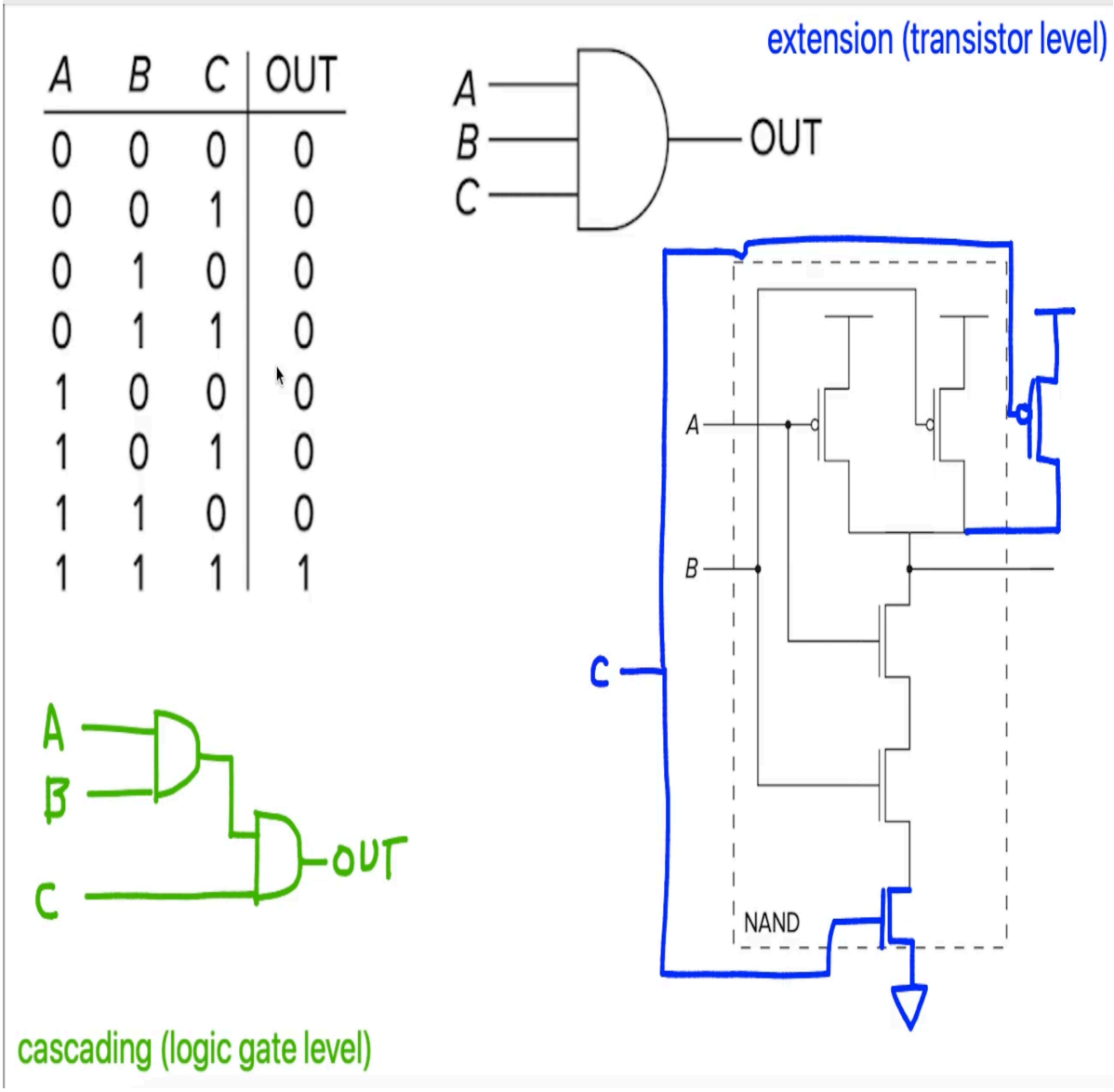
2.2 Building with Logic Gates

2.2.1 Gate Cascading and Generalizing to N Inputs

Sometimes we need to AND or OR more than just two inputs together. In these cases, we can either use cascading or extensions for each logic gate.

The naive solution is to chain multiple AND gates together. This is called cascading. For example, AND the first two inputs, and then AND that output with the third input. The two limitations with this is that, 1) it's not the most efficient way to use the least number of transistors to achieve the same logical purpose, and 2) there are certain operations that cannot be cascaded together. If the operation is not associative, i.e. we need inverters between stages, then we cannot cascade to create them, and this includes NAND and NOR. - An interesting concept is universality. When NAND or NOR is paired with NOT, it is sufficient to build any logic, including NAND for N inputs. It just cannot be purely cascaded.

The most efficient method is to extend existing logic gate structures and add another p-type and n-type transistor either in series or in parallel (depending on whether it's NAND/NOR). - The most efficient way to build AND/OR gates is through NAND/NOR + inverter.

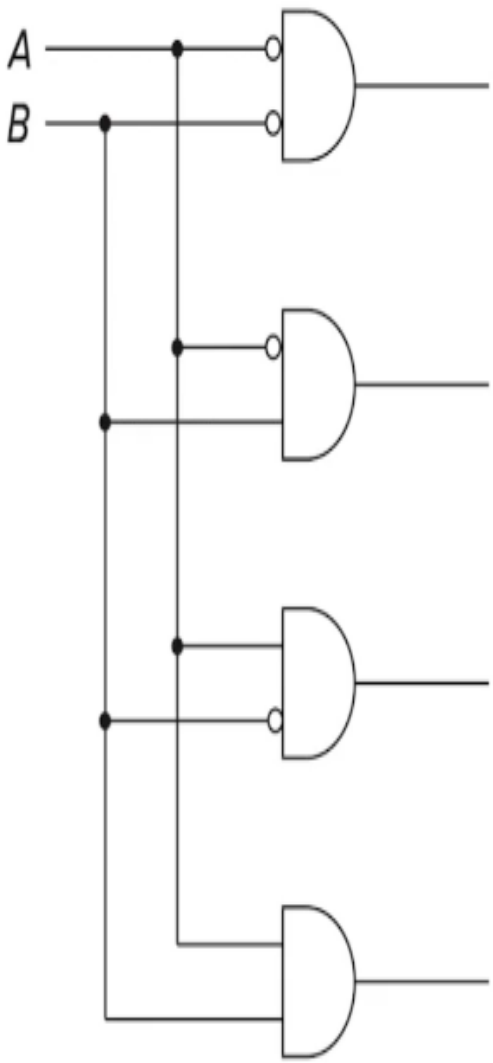


- NMOS in series at bottom
- PMOS in parallel at top
- Cascading results in 6 more transistors each time (another AND gate each time). However, extension only takes 2 more, which is significantly better.

2.2.2 Decoder

A decoder is a chain of AND gates which allows some smaller number of selector bits (input, but not formally considered input) to activate exactly 1 output signal. When taking in n selector bits, it asserts exactly *one* result from 2^n possible outputs.

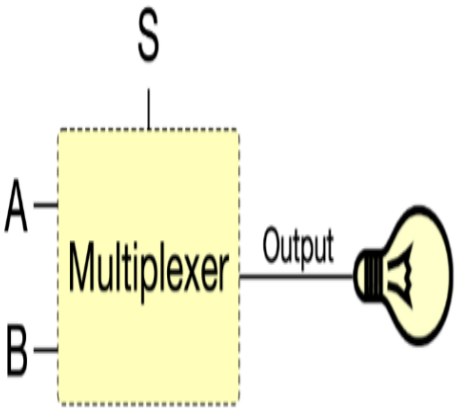
You can imagine "decoding" a compact bitstring in order to activate the single correct output.



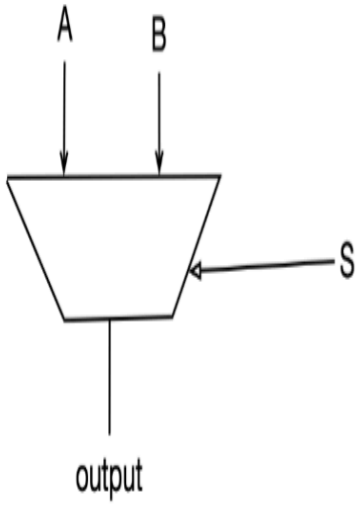
NOTE: On a test/exam, the decoder has 0 lines of inputs. Only has selector bits and output lines. - For example, address bits enter a decoder, whose output enables one register to be written to. - It points at exactly one target, given a compact selector bit input.

2.2.3 Selector/Multiplexer (Mux)

A selector, more commonly called a multiplexer/mux, has different *independent* data input channels. It can have up to 2^n *inputs*, with n selector bits for an output of either 0 or 1. - Many-to-one mapping.



S	A	B	Out
0	0	-	0
0	1	-	1
1	-	0	0
1	-	1	1



There are 2 ways to encode control lines. One way is binary-encoded: $k = \log(n)$ number of wires for n, meaning that combinations like 00, 01, 10, 11 (2 control lines) can encode 4 choices for input sources that we "look" at. The other way is to use one-hot encoding, where n sources are chosen based on 0/1 (1 line per source). Typically a MUX uses binary-encoded control lines.

For k control lines, there are 2^k logic gates, each taking in $k + 1$ inputs (k control lines and 1 input per gate).

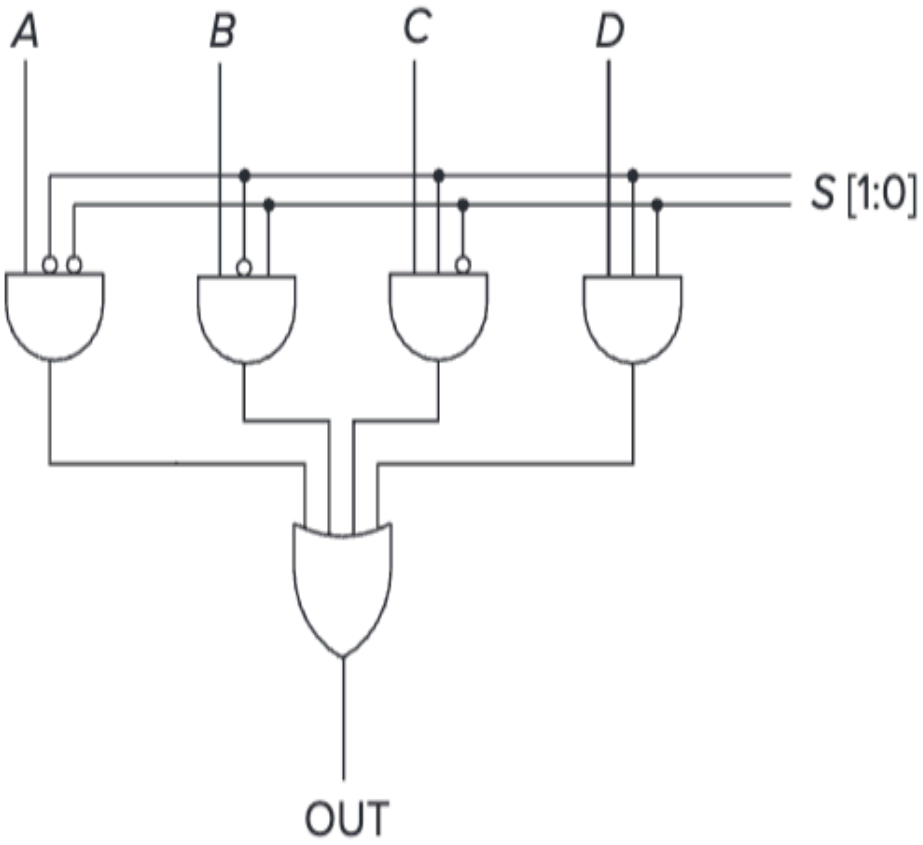


Figure 3.13 A four-input mux.

- In this example, `00` would correspond to A, `01` corresponds to B, `10` corresponds to C, and `11` corresponds to D.
- The series of AND gates make it so that only one input is "used" due to selector bits validating only a single AND gate. The OR gate acts as a "summer", combining the different inputs into a single output, relying on the identity that $X + 0 + 0... = X$.

2.2.4 A 1-Bit Adder with Logic Gates

A truth table can be used to express the logic for addition (in terms of how outputs are determined) from 3 distinct inputs: input A, input B, and the carry from the previous operation.

A_i	B_i	C_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3.14 The truth table for a one-bit adder.

Again, since there are 3 inputs, there are 3 inputs to each logic gate and 2^3 number of possible outcomes/outputs.

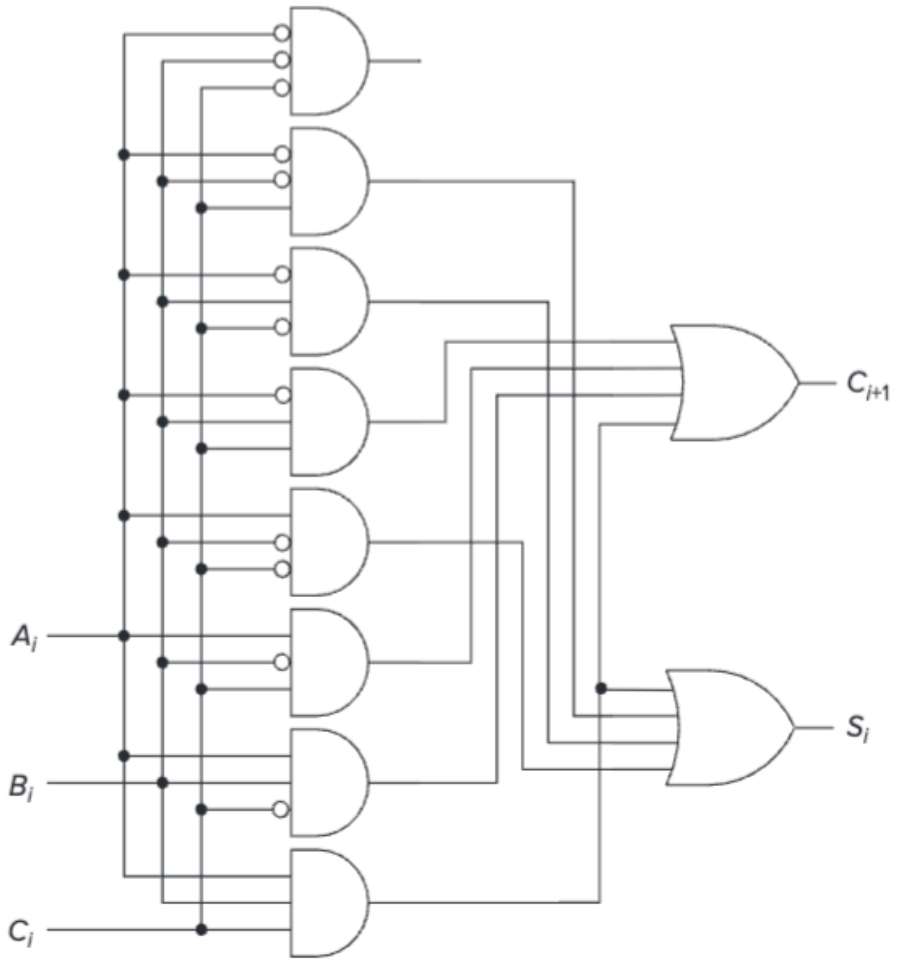
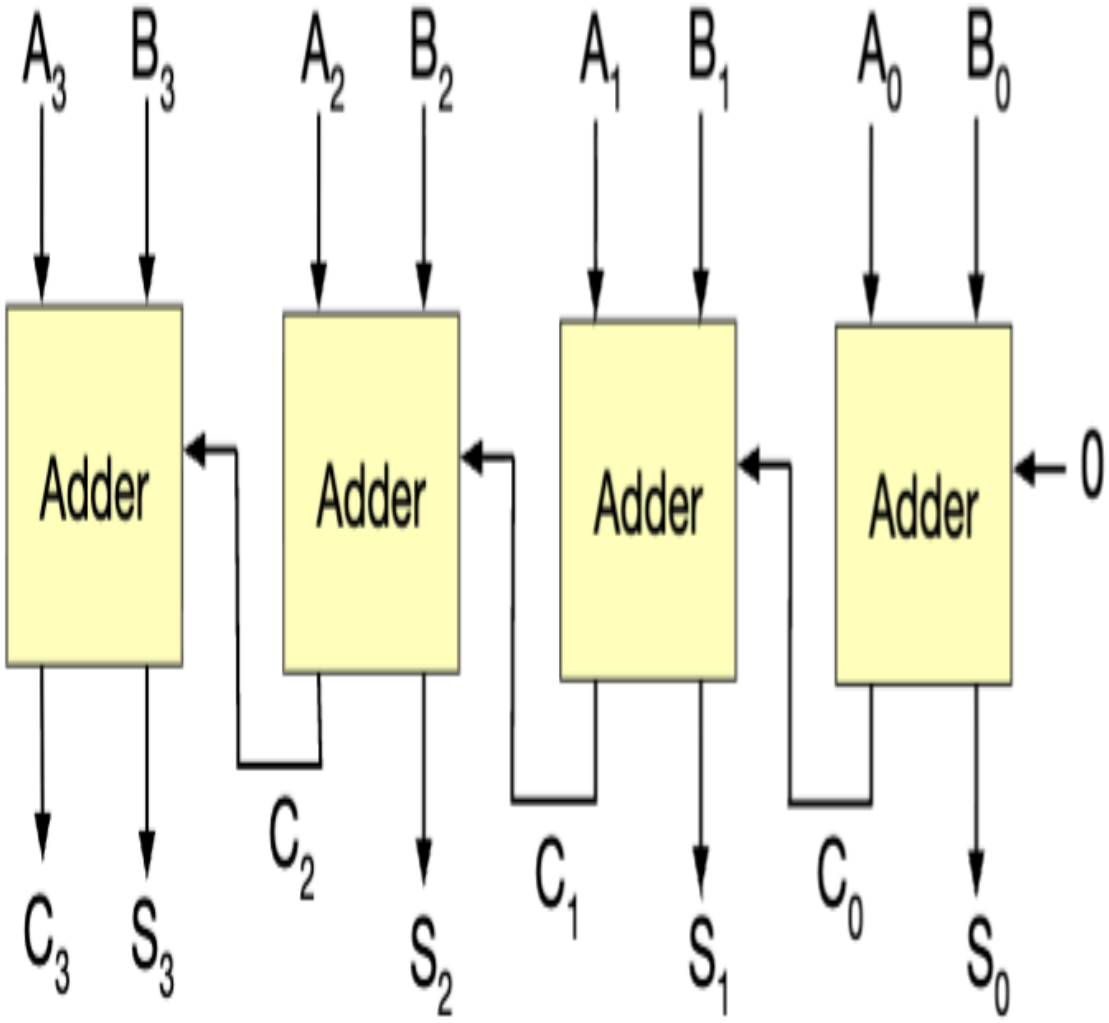


Figure 3.15 Gate-level description of a one-bit adder.

- Looking at the truth table, there is a sum of 1 when there is an odd number of 1's in the 3 inputs. Correspondingly, there is a sum of 0 when there's an even number of 1's in inputs.
- There is a carry when there are 2 or more 1's in the inputs.
- By sum of products, the end will be ORs right before the carry/sum. There will be ANDs for combining each of the inputs since we need a decoder to understand the input bits and trigger exactly 1 result per unique combination.

For multi-bit operations, the carry leftover from one operation is used as input for the next.



2.3 Logic Minimization

2.3.1 Definition and Properties

Logic minimization is the process of rewriting a boolean function into an equivalent form that uses fewer gates, fewer inputs per gate, or fewer logic levels (less delay) while preserving behavior of inputs/outputs. - In other words, we try to find a better representation in order to perform less work for the same outcomes.

Transistor Counts in CMOS 1-input NOT: 2 transistors 2-input NAND: 4 transistors 2-input AND: 6 transistors 2-input NOR: 4 transistors 2-input OR: 6 transistors

Example Consider an output where we want AB + AC + DB + DC. The “products” indicate AND while the “adding” indicates OR. - The naive way to represent this is using 4 2-input AND gates (each being 6 transistors) and then 3 2-input OR gates (each being 6 transistors), totalling to 42 transistors.

We can rewrite this in equivalent form as A(B + C) + D(B + C). - Then we have 1 2-input OR gates (only compute B + C once and feed it as an input twice), 2 2-input AND gates, and then 1 2-input OR gate at the end. That sums to only 24 transistors.

After simplifying further to (A + D)(B + C), we have even fewer transistors. - 2 2-input OR gates and 1 2-input AND gate for a total of 18 transistors.

Notice that by minimizing the total number of add/multiply operations (since AND/OR are the same number of transistors, we can just minimize the total number of operations), we can typically improve the number of transistors and reduce levels.

General Logic Minimization Concepts This remains an unsolved problem, since the search space is 2^n. In general, it helps to minimize add/multiply operations through boolean algebra. However, there is no "standard algorithm" that can guarantee even a somewhat optimal solution.

It also helps if you're able to apply double negation + De Morgan's law without increasing the number of operations, because that allows for usage of NAND/NOR gates.

A K-map (discussed later) provides a somewhat structured form of a search for a small number of inputs (but becomes uninterpretable for large number of inputs).

2.3.2 Sum of Products from Truth Tables

Let the Boolean function be F(A, B, C) with the truth table below.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

By looking only at the rows where the output is 1, we obtain a sum of products (ANDs ORed together). - (A,B,C) = (0,0,1) → A'B'C - (A,B,C) = (0,1,1) → A'BC - (A,B,C) = (1,1,0) → ABC'

The naive sum-of-products logic for this function is then: F(A,B,C) = A'B'C + A'BC + ABC'. - This can often be further optimized using logic minimization.

2.3.3 De Morgan's Law to Maximize NAND/NOR

There are two reasons why we want to minimize AND/OR gate usage. The first is that it adds a second level of depth for every operation (since they are fundamentally built of NAND + NOT or NOR + NOT, it takes longer for a signal to get through). The second is that this increases the number of transistors we need.

Example Imagine making a circuit for $(A + B)(C + D)$. At this point, there is no further "simplification" that can be done algebraically to reduce the number of operations.

Invert it twice, so you have an equivalent form. Then, apply De Morgan's Law: $((A + B)') + (C + D)')'$. - Essentially, negate twice so that one NOT is applied at the very end, and then we can use NOR internally.

Disregard the outer negation for a moment. We are then looking at NOT ((A OR B) AND (C OR D)) -> (NOT (A OR B)) OR (NOT (C OR D)). In the resulting form, we use 4 + 6 + 4 transistors, +2 at the end for the final negation, which is 16.

Compare to the initial form, which required 6 + 6 + 6 = 18 transistors. This is a slight improvement.

2.3.4 Gray Code

Gray code is a way to list binary numbers so that each step only changes a single bit.

For example, normal binary order would look like: 00, 01, 10, 11 - From the 2nd to 3rd element, both bits changed.

Gray code order would look like: 00, 01, 11, 10

Deterministic Gray Code Sequence The most widely-used ordering is binary-reflected gray code. For each integer 0 to $2^n - 1$, gray code is computed as

$$g(k) = k \oplus (k \gg 1)$$

- For example, if k is 011 (3), then the gray code for it would be $011 \oplus 001 = 010$.

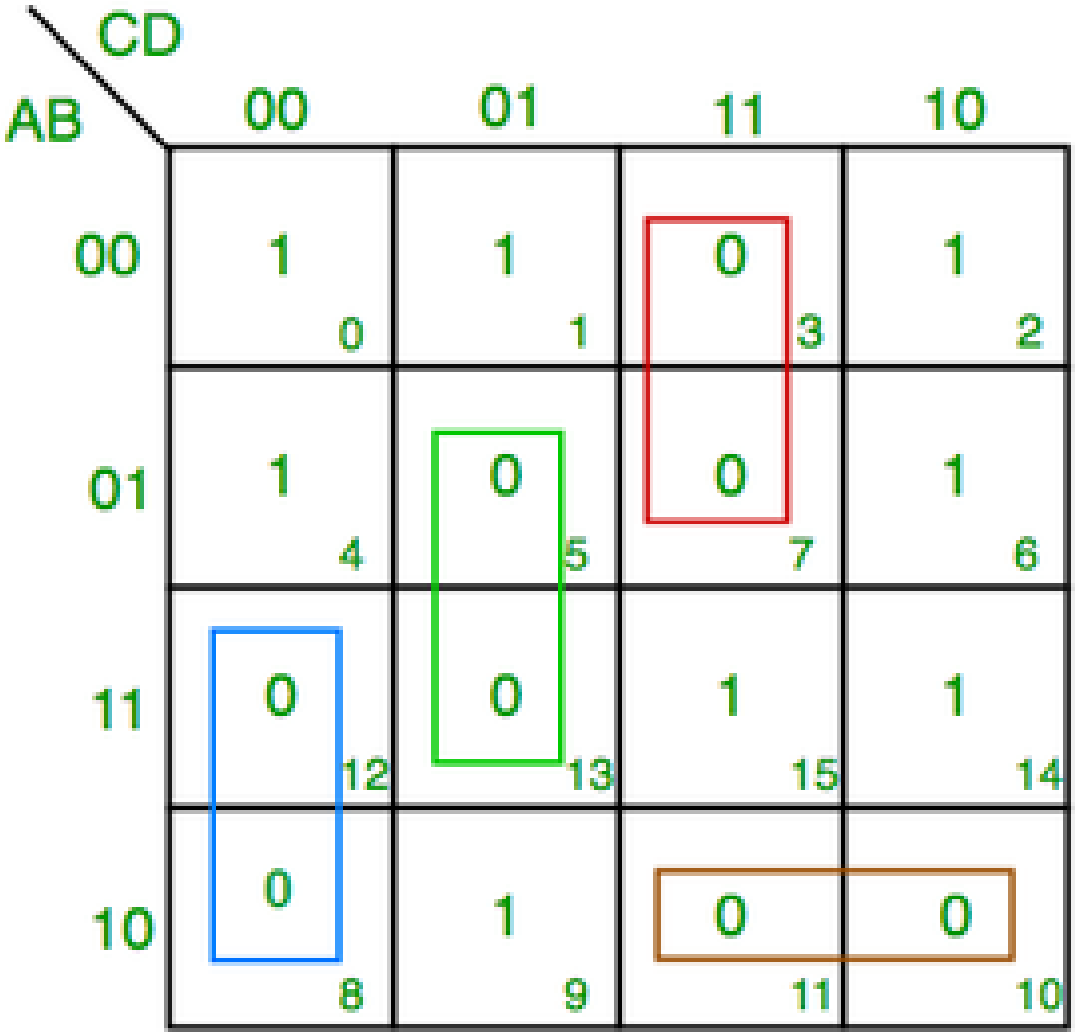
2.3.5 Mintern Numbers (M-Numbers)

In a truth table, each row has a specific input combination (e.g., 0 0 1 0). A mintern is a name for the row matching a specific pattern that *produces a 1*. For example, 0 0 1 0 -> 1 would correspond to m2.

So if a function is 1 on that row, you would say “the function includes m2”.

2.3.6 Karnaugh Maps (K-map)

A K-map is a graphical method to simplify boolean functions by exploiting adjacency patterns in a truth table.



- A mintern number of m1 would correspond to the 0001 combination, which is in the first row second column.

It is essentiallly a more compact form of a truth table where rows/cols can represent combinations of inputs (helping to reduce dimensionality and keep it in 2D or 3D to help with visualization).

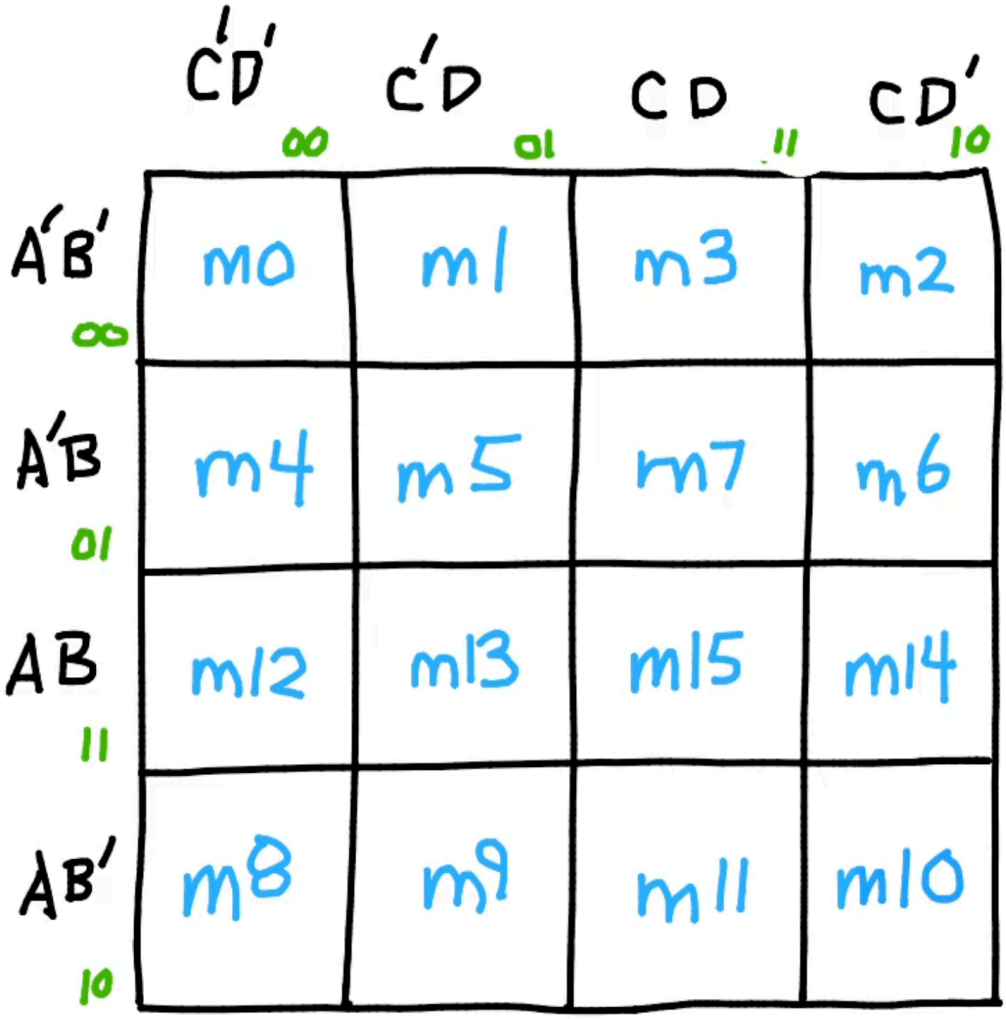
Its significance is a deterministic/representational form that can help achieve logic minimization by choosing blocks of 1's or 0's - which are the size of powers of 2 (e.g., 2, 4, 8) - and simplifying them. - When choosing 1's, we are simplifying a sum of products (i.e., these combinations result in 1's; if any of the combinations are the input we output a 1). - When choosing 0's, we are simplifying a product of sums (i.e., any of the input combinations being a specific value results in a 0; need this to apply across multiple combinations to ensure no 1 output). - Either choice could result in the global optimum for most logic minimized. Often 1's are grouped/chosen, but image example happened to be 0's.

For example, consider the blue rectangle. There, we see that if we have inputs 1 1 0 0 and 1 0 0 0, we have 0 either way regardless of whether B=1 or B=0. This means that instead of writing $AB\bar{C}\bar{D} \vee A\bar{B}\bar{C}\bar{D}$ for these two (sum of products; OR the rows of the truth table together), we can write $A\bar{C}\bar{D}$.

NOTE: Wrap-around logic can be used (i.e., think of the table as a sphere where the left/right and top/bottom edges are connected) due to how simplification works.

Application of Gray Code A K-map REQUIRES

when using gray-code ordering along each axis so that two horizontally/vertically adjacent cells differ in exactly 1 variable, there are there rectangular groups which correspond to the capability to eliminate variables



no don't-cares -> always optimal when there are don't cares -> not always optimal

2.4 Sequential Logic and Memory

2.4.1 Fixed, Fast, and (not) Flexible

A combinational logic circuit is fixed once it is physically wired in. It always computes the same boolean function.

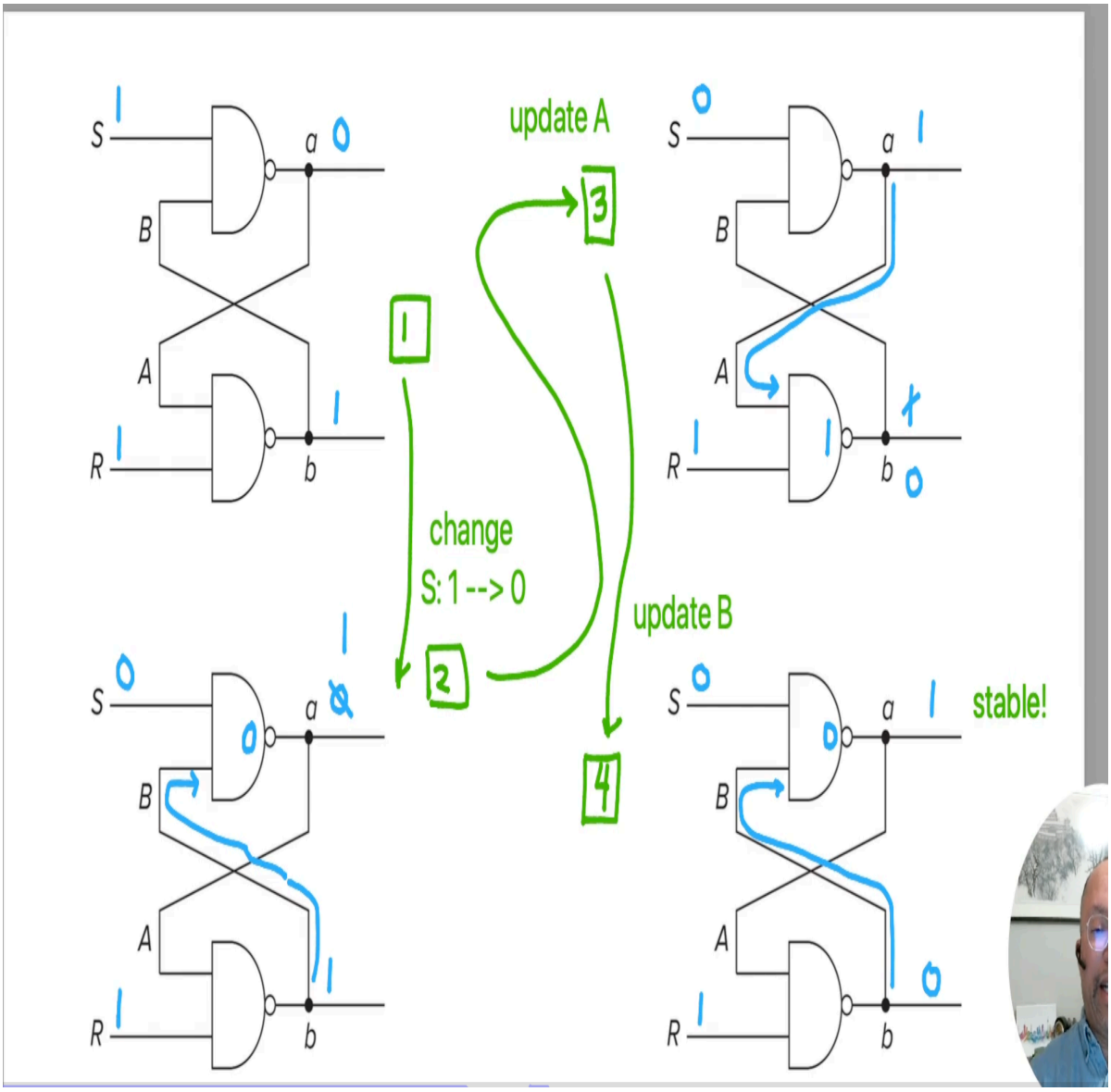
It is also extremely fast because computation occurs through electrical signals propagating through gates, near the speed of light.

However, hardware logic is not flexible in that we need variables and programmability.

2.4.2 R-S Latch

The R-S latch is incredibly significant because it introduces the concept of *memory* into digital circuits. It can store a single bit of information.

Previously, all combinational logic reacted only to the immediate inputs they received. But the RS Latch depends not only on the current inputs, but also on the previous state.



3 Possible States In all previous combinational logic, essentially any set of inputs would map to a specific output state, and the output state would update immediately. The RS latch changes this by basically having two NAND gates and then propagating the outputs of each into one another, while also having a set and reset input. This RS latch construction with two NAND gates results in the particular set and reset commands to be active when they are zero respectively.

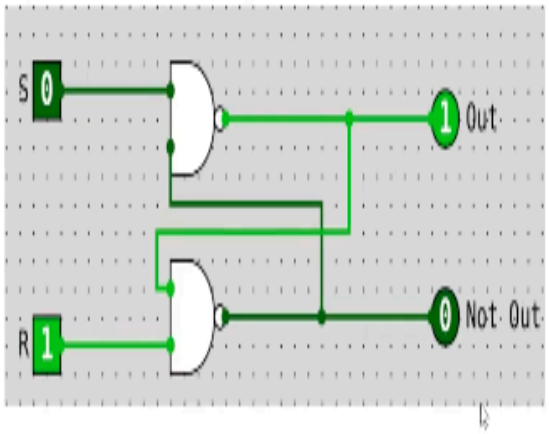
So first, we look at if S equals 1 and R equals 1, then we maintain a quiescent state. Essentially here, we preserve whatever output was originally in the RS latch. If we flip both to zero, we move out of the stable state into an unstable or invalid state.

Next, we look at the two "action states" where S and R need to be complementary to each other. In the case that S = 0, R = 1, then Q = 1. And so basically Q is the output of the NAND gate that has S as an input, and Q is always going to be opposite of S when it's not in the quiescent state. Similarly, if S = 1, R = 0, then Q = 0. And again, it's opposite of whatever S is.

The significance of this compared to the combinational logic that we had before is that in addition to the write operations for zero and one, there's a third combination of inputs where we can tell the circuit to perform a preserve option, or essentially do nothing.

RS Latch

- R = Reset
- S = Set
- Stores 1 bit in Out
  - Note that Out matches the value of R
  - Not Out is not used

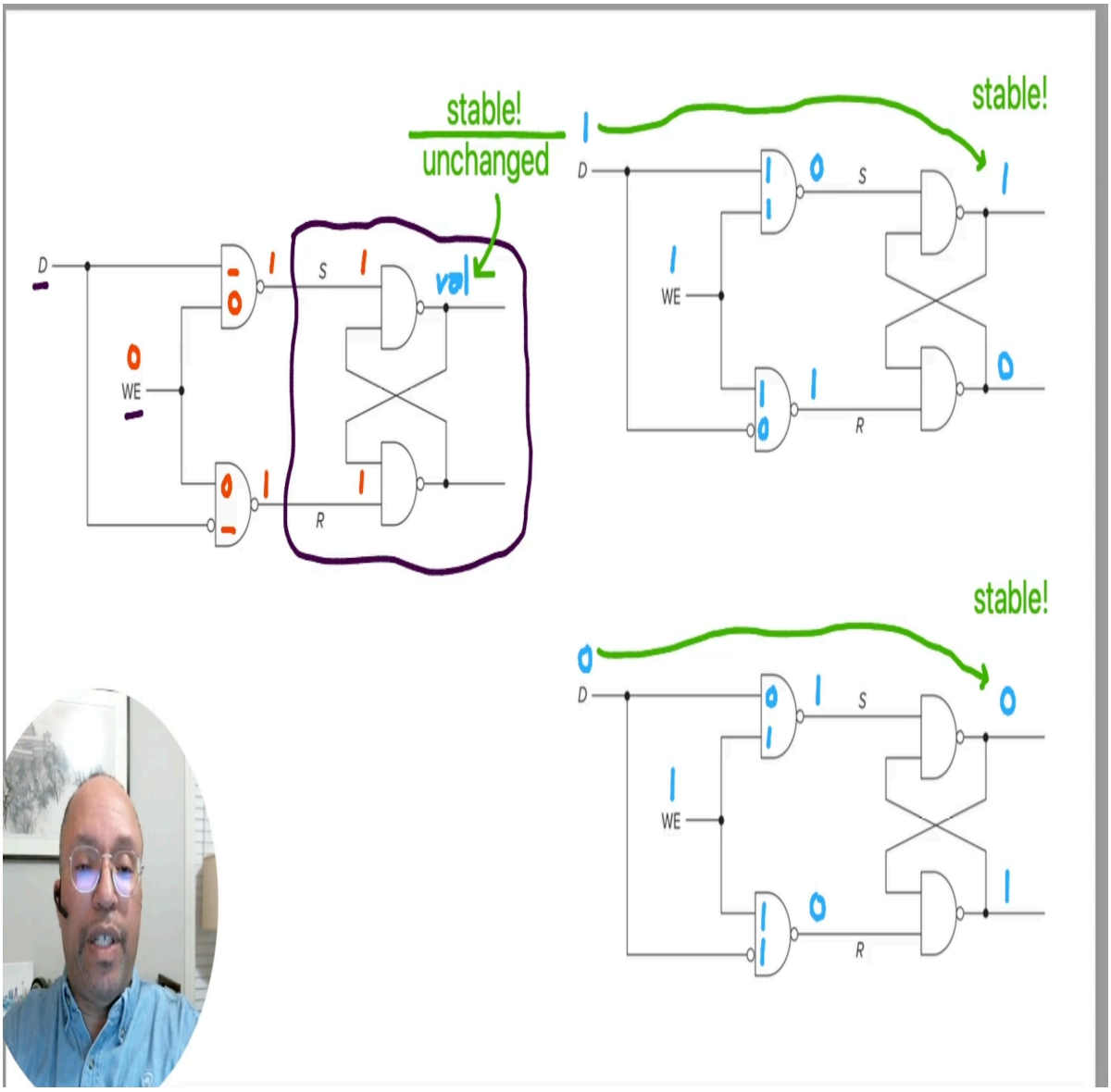


S	R	Out
1	1	Stays same
0	1	Set to 1
1	0	Reset to 0
0	0	Invalid

2.4.3 Gated D-Latch

Now that we have a basic concept of memory in that we have a combination of inputs that preserves the previous state from a write, we need to fix a few problems. The first major problem is that we have no control over when new writes happen. The RS latch is still combinational in the sense that even though it is sequential logic, the RS latch still responds to S and R whenever they change. So if the inputs to the RS latch come from some combinational logic that's still settling, then in between that period, S and R could take in all types of values in a random order, and the latch sees each of those values and responds to them. The second issue is that there is an unstable state where if both S and R are zero (for an RS latch made of two NAND gates), then since S and R are coming from upstream combinational logic, you have no way to prevent that unstable state from happening.

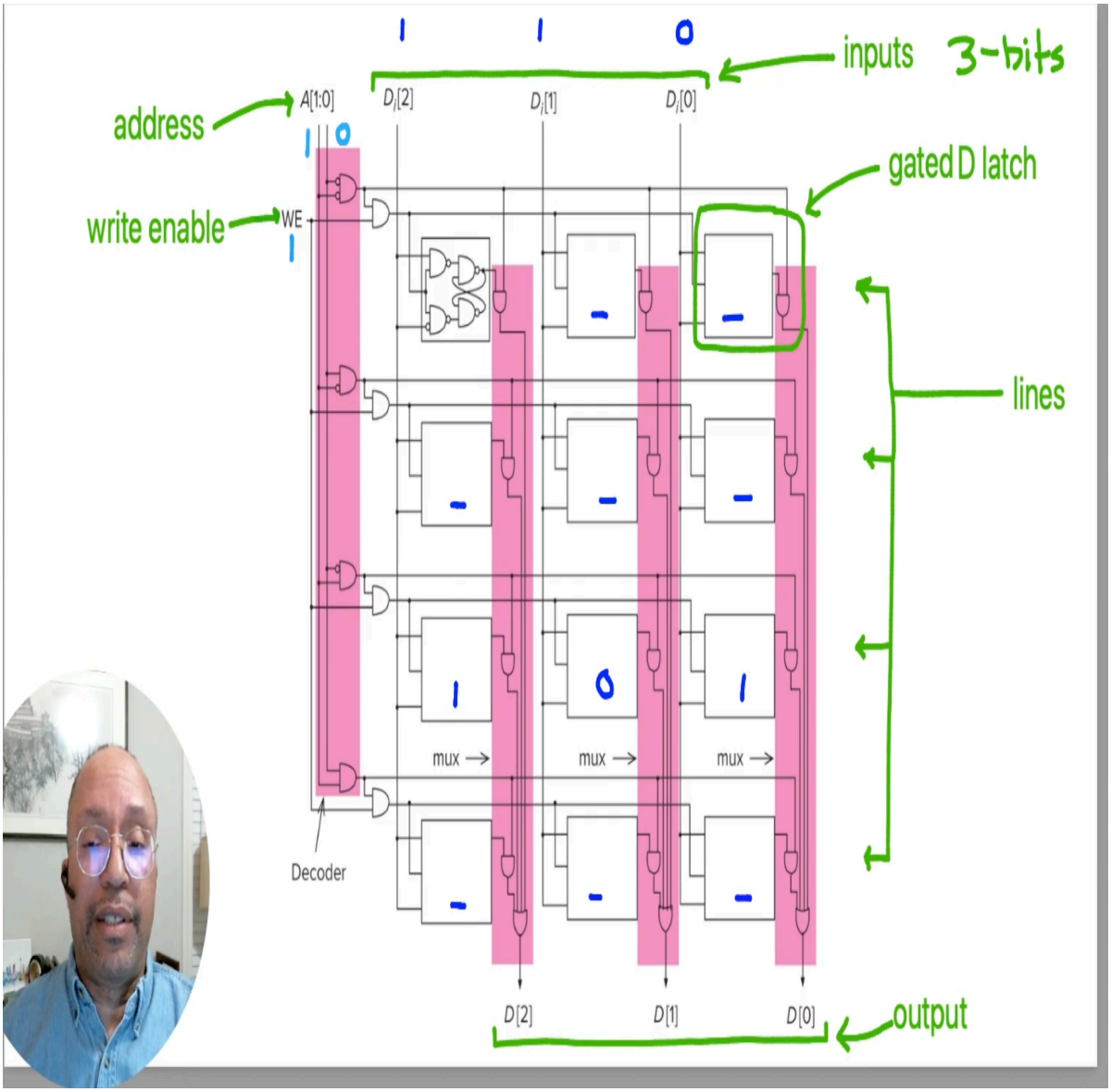
In order to fix this problem, we adjust the construction of the RS latch. We preserve the RS latch in its entirety, and then we add two more NAND gates. The inputs to these 2 NAND gates are both data and write enable. The output of the NAND gate corresponding to D is S, and the output of the NAND gate with $\bar{D}$ is R.



Note the downstream impact of the two changes in its construction: - If WE = 0, then the outputs of the two initial NAND gates are guaranteed to be 1. That means that both R and S are going to be 1, and so a write enable of 0 guarantees preservation of whatever data was outputted by the Gated D-Latch originally. - If WE = 1, then the input data D is guaranteed to be propagated as the output of the RS latch. Note that there is no longer a reversal: $D = \overline{S} = Q$.

Memory in the LC-3 is built using Gated D-Latches, decoders, and multiplexers. Here, the address space is $2^2 = 4$ unique locations, with each address storing 3 bits of information.

1. A decoder takes in n address bits mapping to a 2^n size address space. It "actives" a corresponding word line.
2. This output of the decoder is ANDed with Write-Enable, which determines whether the access is a read/write operations.
3. A "word line" is the row select wire for memory. In this case, memory has 4 rows. Let `M0`, `M1`, `M2`, and `M3` represent the 3-bit words stored in each row. Let `D[0]`, `D[1]`, and `D[2]` represent each of the columns. Then `WLx` is ANDed with `Mx[2]` for `D[2]`. This way, the word line picks the row for a specific column data input.
4. For each data input line, say `D[2]`, the candidate inputs to the mux are the 4 stored bits in that column. The selector bit is the set of word-line signals from the decoder



- The multiplexor's selector bits are the wordlines of the address decoder, which pick a single row.
- The multiplexor's inputs is the bit state stored in the Gated D-Latch in each row of a specific column (e.g., Q(row0, colj), Q(row1, colj)).
- Data line at the top indicates 3 bit input (if write enable is 1, then these inputs are written to memory).

2.4.4 Address Space and Addressability

Memory in a computer consists of a large number of locations, which have a unique identifier and the ability to store a value. Here are a few vocabulary terms for components of memory: - Addressability is a property of the memory system. It is the number of bits of information that live at a particular memory address/location. - Word size is a property of the processor. It is the unit of data which ALU and registers operate on. In LC-3, word size is 16 bits. - Instruction size is a property of the ISA encoding. It is how many bits it takes to encode one instruction. - Address space is a property of the ISA encoding. It is 2^n where n is the number of bits within any particular memory address. The address space is the number of different memory locations that can be represented.

The total memory/storage within a computer is equal to addressability (# of bits per address) x address space (total number of addresses).

2.4.5 Finite State Machines

Finite state machines allow us to model algorithms/solutions for a large number of scenarios. It consists of the following core components: 1. Finite set of states 2. Finite set of inputs 3. Finite set of outputs 4. Specification of all state transitions (valid movement between states) 5. Specification of what determines output value

Finite state machines track all valid states in a system, and all possible transitions between those states. - It is called *finite* because the memory required to store the states is bounded by the definition of the system/scenario. - An infinite case would be when the system needs to remember an unbounded amount of information, like if we wanted to recognize strings of the form $a^n b^n$ (some number of a's followed by same number of b's). - No FSM an do this because the count of letters seen previously (remember number of a's to check b's) is unbounded.

Now let us understand the usefulness of a FSM. It is helpful because it forces us to consider all possible valid states and transitions between those valid states in a system. It is used to rigorously model certain processes, like network protocols, game logic, or compilers/parsers. - It is not necessarily used often for practical purposes, but it is the simplest model of computation and is the theoretical foundation.

2.4.6 The Clock (Synchronus Finite State Machine)

A synchronus finite state machine transitions from its current state to the next after a fixed interval of time. A clock circuit produces a signal which alternates between 0 and some specified voltage. We represent that as 0 and 1.

One clock cycle is defined as the time between one 0 -> 1 transition and the next 0 -> 1 transition. Of course, this means that it must go back to 0 sometime in between the two clock cycles, but that part isn't too important.

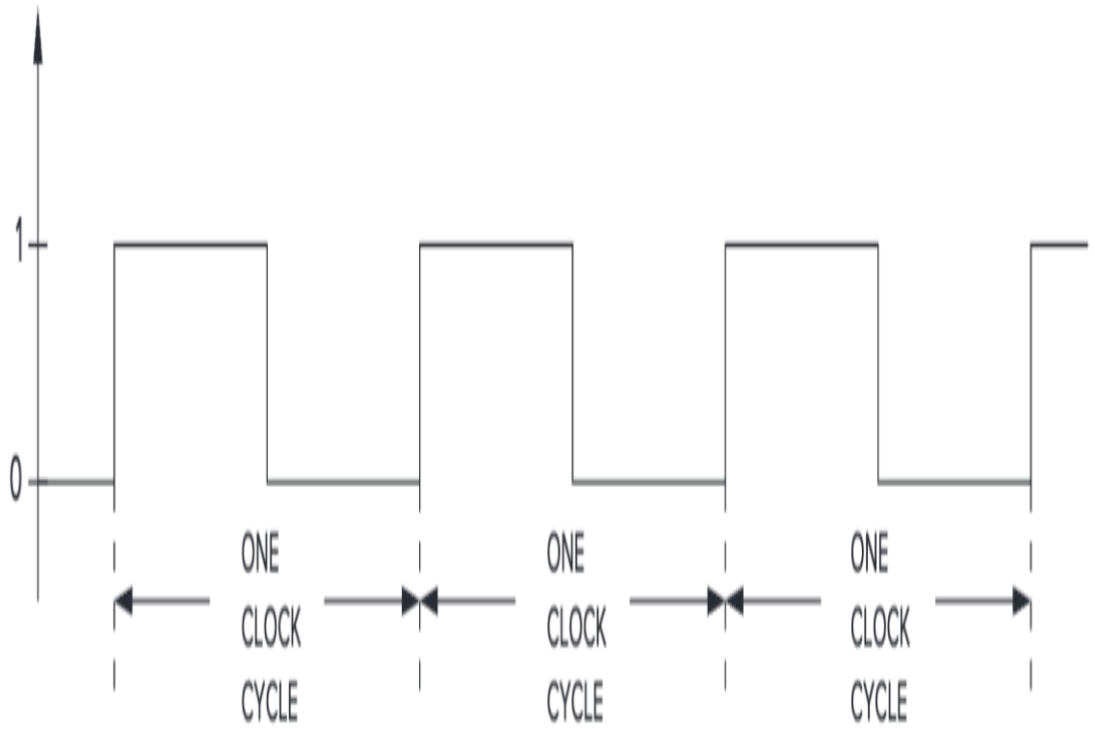


Figure 3.28 A clock signal.

Circuits may be either level-triggered or edge-triggered. - If a circuit's output can change whenever the clock signal is at a particular level (high or low), it is level-triggered. - Think the RS-Latch or Gated D-Latch. Whenever a write-enable is on, updates to the output can propagate through. - If a circuit's output can only change on clock transitions (either rising edge or falling edge), then it is edge-triggered. - Think the Flip-Flop. The master-slave combination only allows for output updates on a clock transition.

2.4.7 The D Flip-Flop

Flip-flops provide a digital system with memory across time. On a rising clock edge, it samples an input bit *D* and holds that value on *Q* until the next clock edge.

It makes up for a deficiency that the Gated D-Latch has. The issue with a Gated D-Latch is that while WE=1, the latch is *transparent*, so any new inputs are automatically propagated to the output. This results in an uncontrollable set of transitions all within a single clock cycle.

Rather, we need a storage mechanism that can consider the input *D* at the start of each clock cycle, and then ignore any changes to *D* afterward.

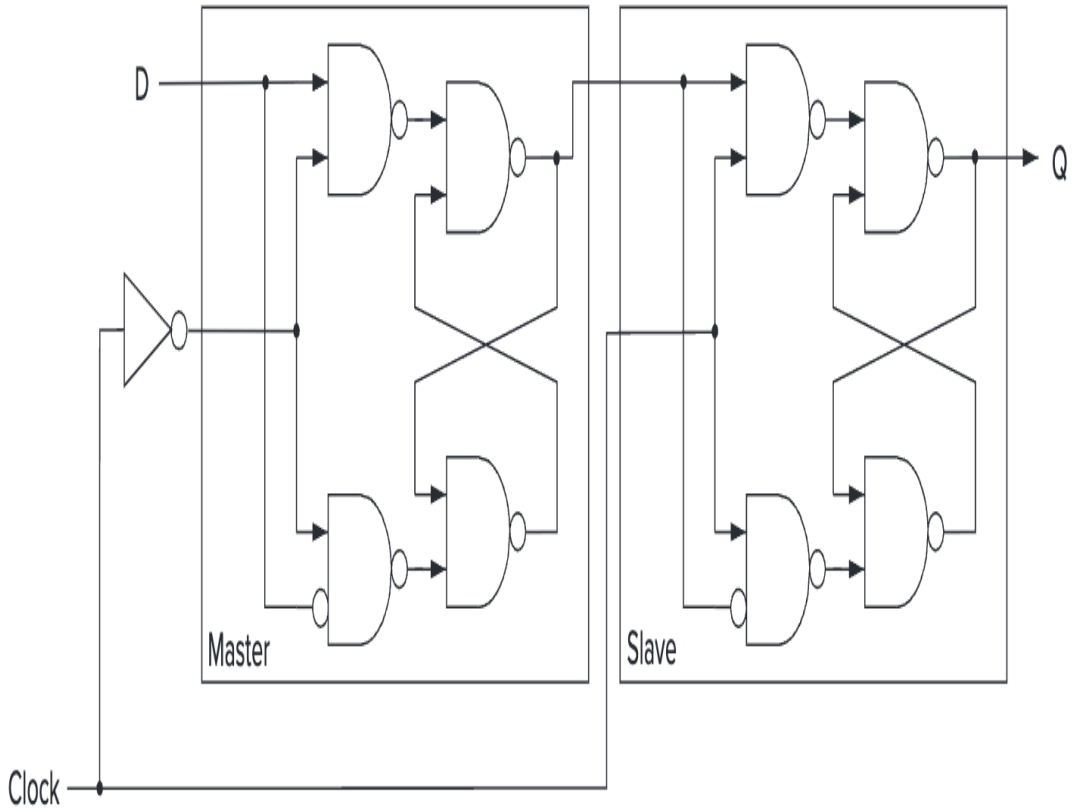


Figure 3.33 A master/slave flip-flop.

- Two Gated D-Latches placed in series, with the first taking in clock inverted as WE and input D
- The second takes in clock as WE and output of first as input D

Let's walk through what occurs at a single clock cycle: 1. CLK=0, -CLK=1: master is transparent (D propagates to master's output), but slave is opaque (ultimately Q stays the same value that is was). 2. Rising edge, CLK becomes 1, -CLK becomes 0: master becomes opaque and locks whatever value input D had as its output; slave becomes transparent and propagates master's output to Q - This is where Q updates; at the beginning of each clock cycle. 3. CLK=1, -CLK=0: master remains opaque, so any changes to D will not influence the output Q. - During this time, a combinational next-state computation logic can occur to udpate D using new Q but the master is opaque so there is no race condition. 4. Falling edge, CLK becomes 0, -CLK becomes 1: master becomes transparent again, so the new D propagates to become the master's output. But the slave becomes opaque and locked, so Q doesn't change yet until the next rising edge.

Essentially, the two Gated D-Latches are always in complementary states. One is opaque while the other is transparent. This maps to clock cycles well and controls updates at predictable rates.

2.4.8 Registers: Parallel Flip-Flops

A register is an array of some number of flip-flops. Each flip-flop stores one bit of output, and they all share the same clock wiring so they all update on the same clock cycles.

They are parallel in that data inputs are independent of each other and their outputs are also computed/updated independent of one another.

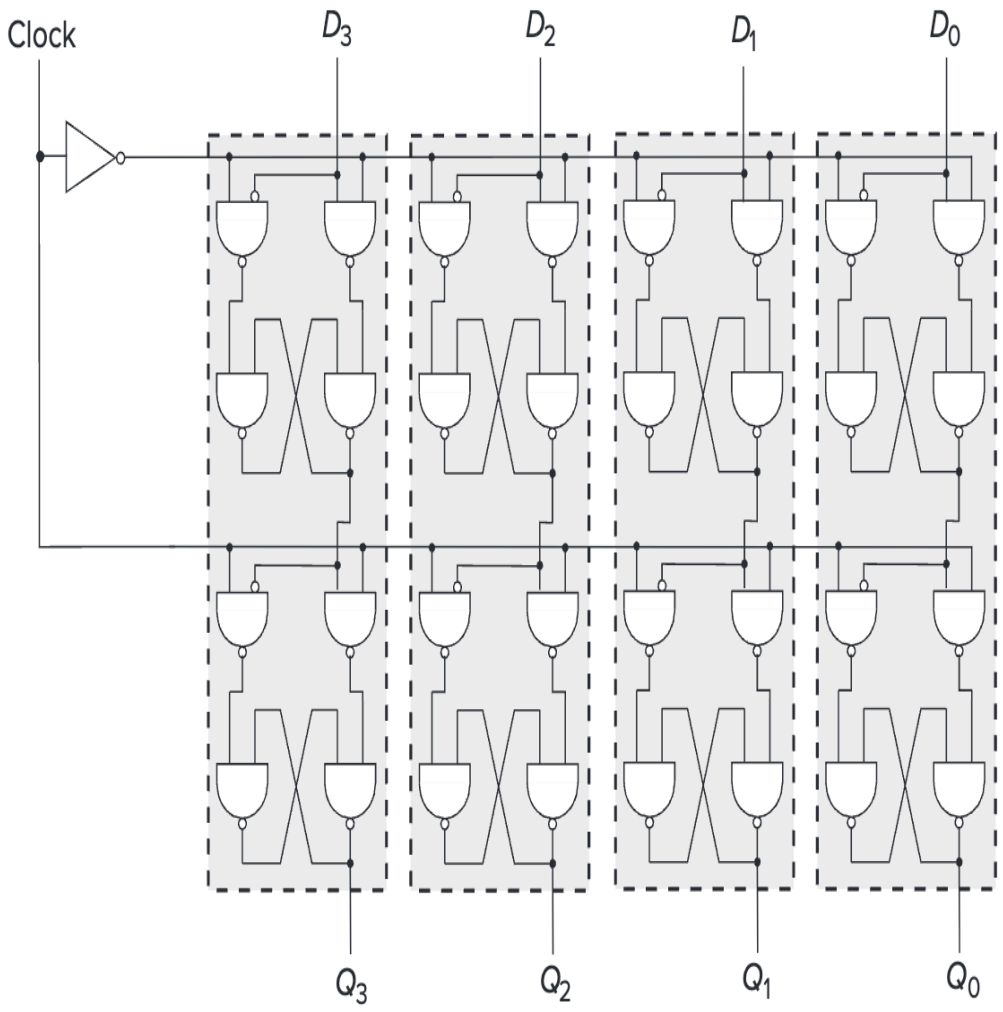
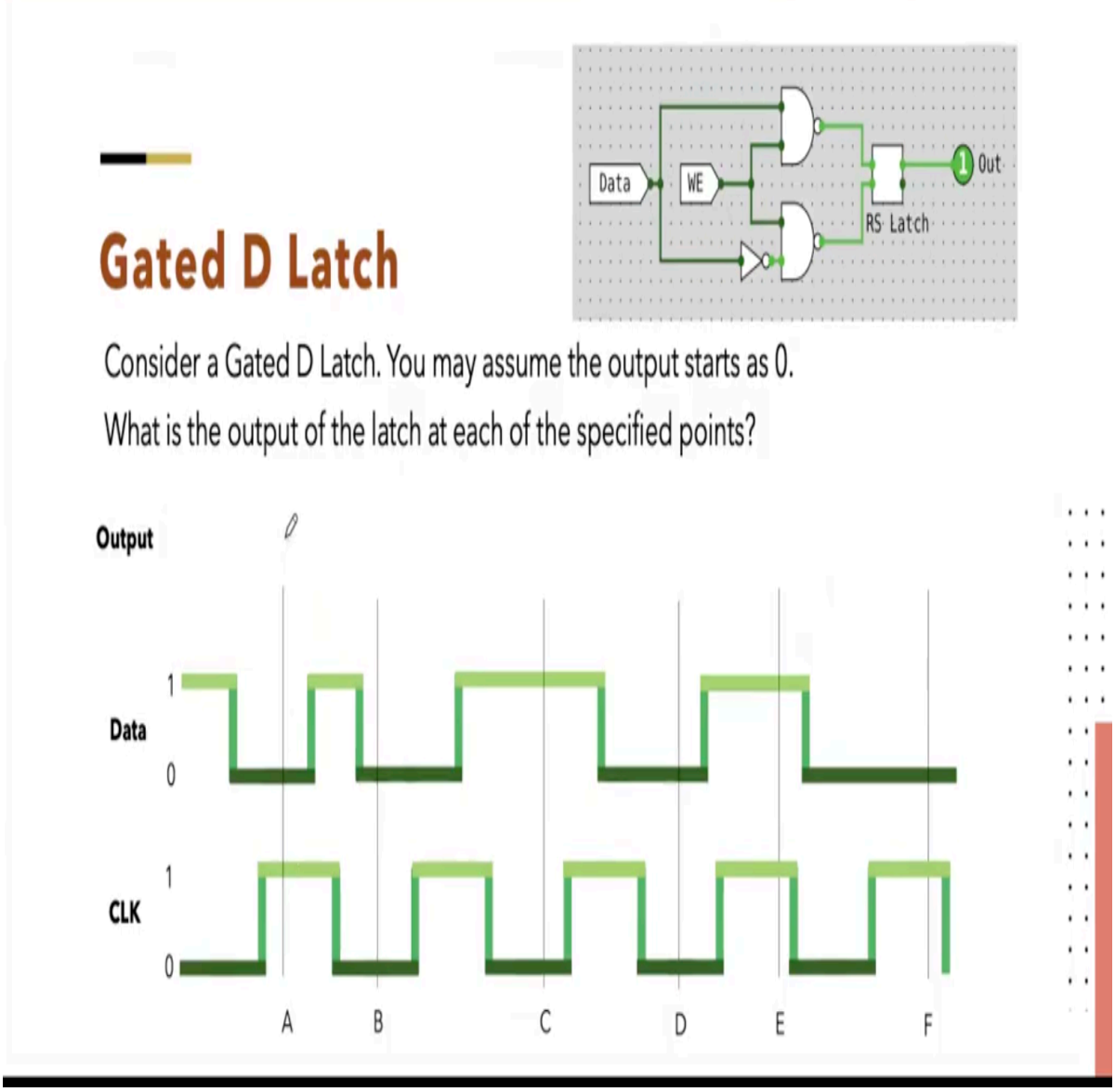


Figure 3.36 A four-bit register.

2.5 Misc

2.5.1 Level vs. Edge Triggered

For a Gated D-Latch, it is level-triggered. Since the CLK is essentially WE, the data propagates to become the output whenever the CLK is 1.



- A -> 0, because CLK=1 and D=0 at that point.
- B -> 1, because when CLK became 0, output was D=1. Even though D became 0 afteward, that occured while CLK=0, so the output never updated.
- C -> 1, because D=1 while CLK=1, and after CLK->0, output remains 1.
- D -> 0, becase D->0 while CLK=1 and then CLK->0 locked that output=0.
- E -> 1, because D=1 and CLK=1.
- F -> 0, because D=0 and CLK=1, so D=0 propagates to output.

The Gated D-Latch and memory in general are level-triggered, and so whenever WE = 1, or it's turned on, then any data written will be propagated and the state will be updated. For D flip-flops and registers, which are constructed of parallel D flip-flops, the type of logic will be edge triggered logic. State is only updated on rising/falling edges.

2.5.2 FSM to Truth Table

In a finite state machine, there are 3 main concerns: - The states - The transitions - And the output

In a truth table, if there are k states, then there must be $\lceil \log_2 k \rceil$ columns in the truth table allocated to represent those states. These columns can be labeled S_0, S_1 ....

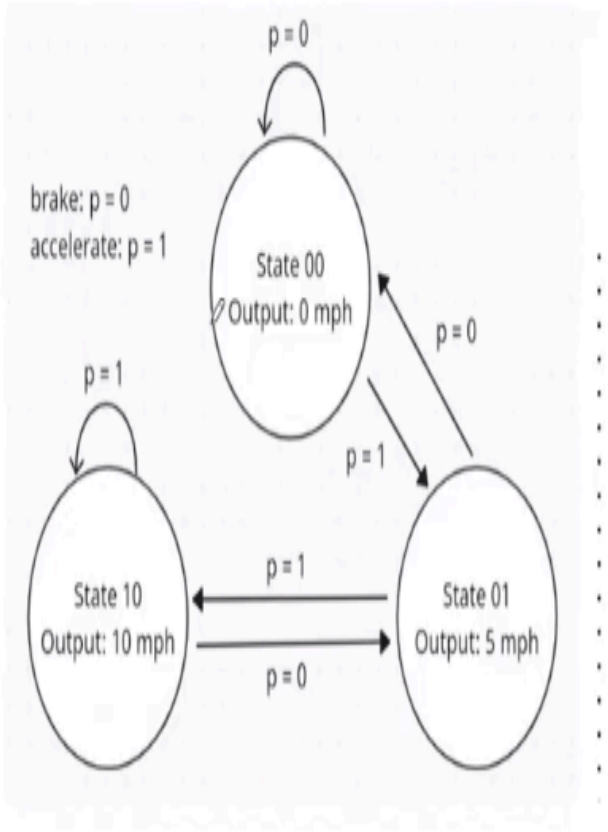
Similarly, the action must also be encoded as columns of the truth table.

The columns that must be filled in are the next states, encoded as $N_0, N_1, \ldots$ as well as the output (which is really just the tracked value of concern).

State Machines

Fill out the truth table based on the provided state machine diagram.

S1	S0	P	N1	N0	OUT
0	0	0	0	0	0 mph
0	0	1	0	1	0 mph
0	1	0	0	0	5 mph
0	1	1	1	0	5 mph
1	0	0	0	1	10 mph
1	0	1	1	0	10 mph
1	1	0	X	X	X
1	1	1	X	X	X



- The OUT depends on the CURRENT state ($S_0, S_1, \ldots$).

3 Von Neumann Machine

3.1 Components of a Computer

3.1.1 Achieving Tasks using a Computer

To perform an action on a computer, there are two core components. The first is a computer program that specifies what the machine must do, and then the computer which will carry out the specified algorithm.

The Von Neumann model is significant because it establishes the model that nearly all computers follow. There are 5 core components: 1. Processing Unit: ALU performs all logic and arithmetic operations 2. Control Unit: Orchestrator to read the next instruction from memory, decodes the opcode to do the required task, and then generates signals for other components 3. Memory: Memory Address Register (MAR) and Memory Data Register (MDR) 4. Input 5. Output

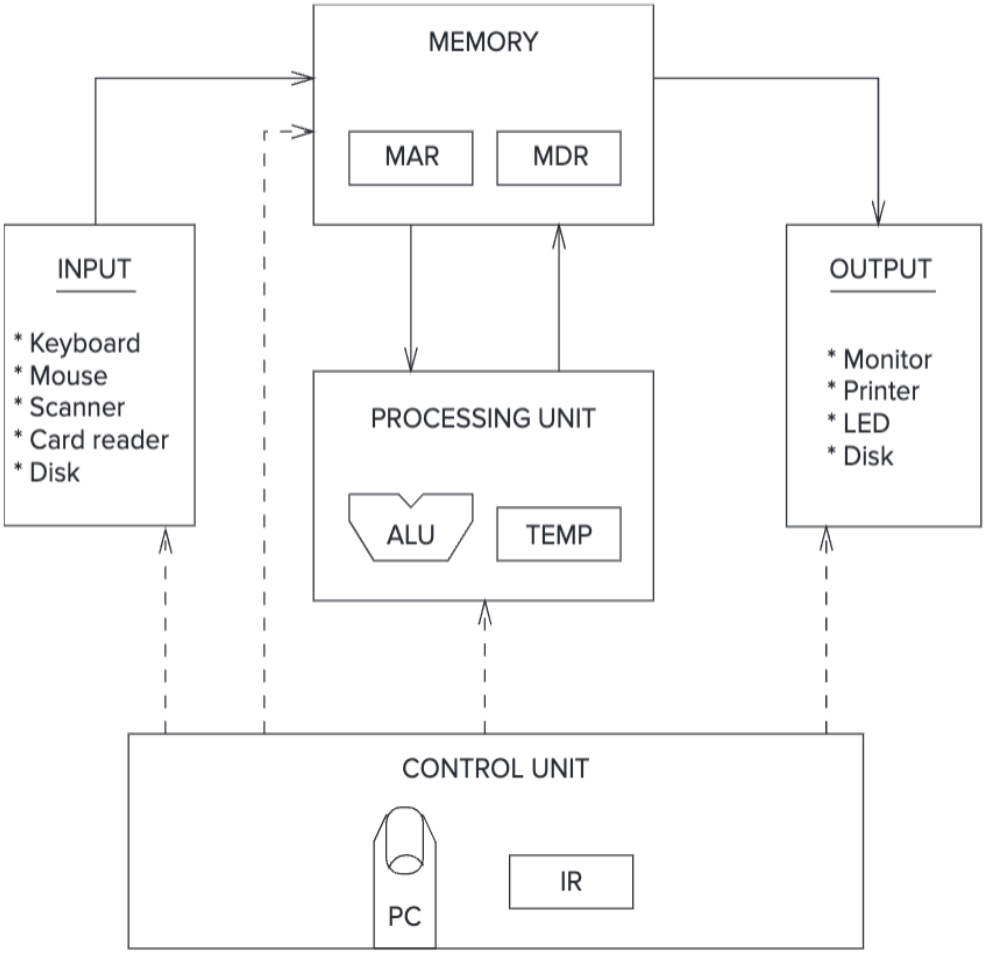


Figure 4.1 The von Neumann model, overall block diagram.

3.1.2 Memory in a Computer (RAM)

Memory stores the general program outside of the CPU (e.g., store variables, stack, heap).

Memory Address Register (MAR) and Memory Data Register (MDR) are used to bridge RAM and CPU.

For example, fetching instructions (this happens at the beginning of each instruction): 1. `MAR <– PC`: PC currently holds some address (address of instruction which is to be executed). The control unit asserts `LD.MAR` (load to MAR), so the PC's value gets placed onto the bus and loaded into the MAR. 2. Memory read: The memory system sees the address in the MAR and looks up what's stored there and places those bits into the MDR (which will hold actual bits of instructions). 3. `PC <– PC + 1`: While this is happening, the PC gets updated. The PC "increments", so that the address of next instruction is what gets stored in the register. 4. `IR <– MDR`: The raw instruction bits are loaded from the MDR to IR. - Here, the IR still holds the raw instruction bits. However, the reason it can't just stay on the MDR is because the MDR is the only gateway between memory and the rest of the CPU. - After fetching the instruction, we might need the MDR to store something else during that same instruction's execution. 5. The FSM/Control Unit reads the Instruction Registry (IR) and decodes it based on opcode and remaining operand bits.

The sizes of MAR and MDR are telling. If an MAR is 16 bits, then there are 2^{16} total addresses available (approximately 64 KB). If an MDR contains 16 bits, then each memory location contains 16 bits (i.e., word size is 16 bits).

3.1.3 Processing Unit (Different from Control Unit)

The actual processing of information in a computer is done by its processing unit. In the von Neumann model, this processing unit is the Arithmetic and Logical Unit (ALU).

The ALU processes data elements of a fixed size each time, which is called the *word length*. Each unit of instruction is a "word". - Most microprocessors in modern computers have 64 bits.

3.1.4 Registers

Registers store data and live inside of the CPU, very close to the ALU. It holds the values that the CPU is actively working with (operands, results, addresses). It is the fastest storage the CPU can access.

For example, if we calculated $(A + B) \times C$, then in the first clock cycle, A and B are read from the register and $A + B$ is computed. In the second cycle, $(A + B)$ and C are read from the register, and their product is written to the register.

Typically, a single register stores data which is the same size as a single word. Microprocessors can contain 32 registers.

3.1.5 Input/Output

For our purposes, input and output are relatively peripheral. For a typical computer, the keyboard would be input and the display monitor would be the output. This is not too important in the von Neumann model.

3.1.6 Control Unit

The control unit is essentially a finite state machine (FSM). The current state determines which control signals to assert, and the next state depends on the current state + opcode of instruction to be executed.

To keep track of the instruction being executed, the control unit has a Program Counter (or instruction pointer), which is a register containing an address to the next instruction to be processed. - The reason it contains an address is because instructions live in memory, and there may be a lot of them (a program could be millions of lines long). The PC only needs to point to the very next one. - Furthermore, it must support non-sequential control flow. At a particular branch or jump, we need to go to an arbitrary instruction.

Then the question comes: How does the program counter know which address to point to next? The starting address is hard-coded. Since instructions have a fixed width, the next instruction is guaranteed to be a certain number of bytes away in memory, since compilers lay out instructions sequentially in memory unless there is a branch or a jump. In that case, the target address is encoded within the previous instruction itself, and the hardware can compute the new PC value from that.

The control unit steps through this general flow for each instruction: 1. Fetch 2. Decode 3. Execute 4. Writeback (update value in memory if needed)

These steps will be covered in more detail below.

memory, logic, arithmetic, muxes, state machines

programs are a set of instructions instructions are stored in memory with all the other data control unit fetches instructions and controls valves of data piping

processing unit has temporary storage and performs arithmetic/logic operations memory stores data and instructions input/output communicates with outside world

Rust/C++ -> Intermediate Representation (Low-Level Virtual Machine, but no actual virtual machine exists) -> backend ISA like RISC-V -> machine code

read section 4.1 and 4.2 of textbook

3.2 Components of the LC3

3.2.1 Bus and Tri-State Buffer

A bus is a shared set of wires and protocols that components of a computer use to move data, addresses, and control signals. Only one signal can be on the bus at a time.

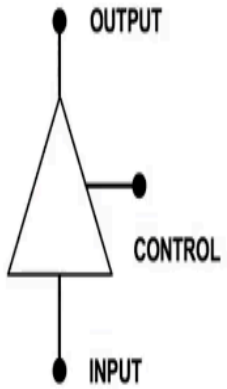
A tri-state buffer is used to control connecting/disconnecting a signal from a shared wire like the bus. There is D, the data input itself, WE, and the output.

Tristate Buffer

Very simple component that either lets the input through or blocks it. This is essential for the LC3:

Control	Input	Output
0	0	Z
0	1	Z
1	0	0
1	1	1

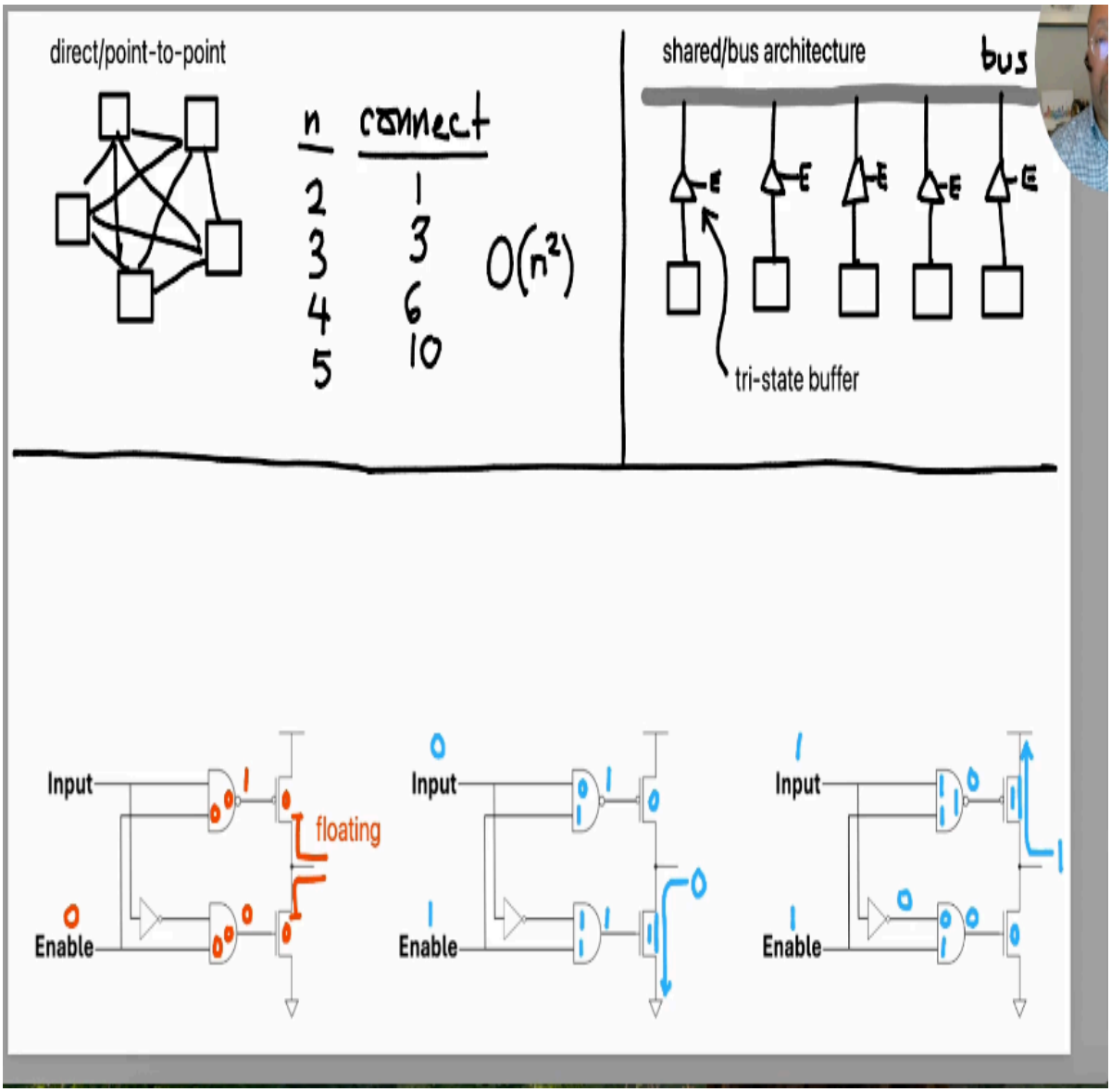
- Controls whether output of a component is asserted onto the bus
- LC3 Gates implement this (we will see later!)



The three possible output states give it its name:

E	A	Y
1	0	0
1	1	1
0	X	**Z** (high-impedance)

The significance of the tri-state buffer becomes apparent when you seek to interconnect multiple components with each other.



- Imagine if they were arranged circularly; then to join $n + 1$ components together, it takes n additional connections from that new node to the previous n nodes, which grows as $O(n^2)$.
- Compare this to a shared bus architecture where we use tri-state buffers to manage connections. This grows linearly, with one additional wire extending off of the bus for each component.

In circuitry, a tri-state buffer looks like an input and WE ANDed together, with the input propagating to become the output only if WE=1.

There are 3 separate sets of physical wires: - The address bus selects which location you want to address (RAM address). The width (n number of lines) determines address space 2^n. - The data bus holds the actual data being read/written, and width affects how many bits move per data transfer operation. - The control bus determines what kinds of operations are occurring, sending signals like READ/WRITE and CLK.

3.2.2 Program Counter (PC) and Instruction Register (IR)

Both components are registers. - The program counter holds the *address* of the next instruction to be executed. - The instruction register holds the *value* of the current instruction.

3.2.3 Register File and ALU

The register file contains all of the general purpose registers used to complete computations. In the LC3, there are 8 such registers in the register file.

The arithmetic logic unit in the LC3 is capable of ADD, AND, NOT, and PASS.

3.2.4 Memory: MAR and MDR

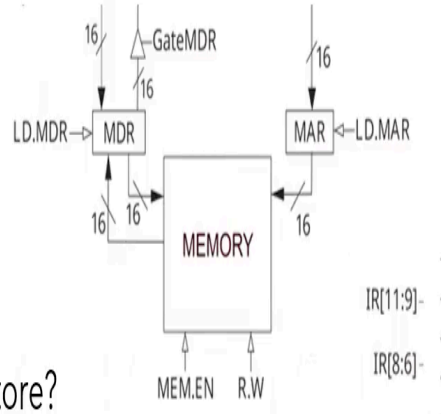
Memory consists of a sequence of Gated D-Latches while registers use D Flip-Flops.

1. In a single cycle, MAR is loaded with a particular address. Note that this is only an address for an instruction, not that actual instruction.
2. Then, the system takes some amount of time to locate that address in memory (unknown number of clock cycles).
3. After memory responds, it takes another clock cycle to load result into MDR. The size of instruction at a specific memory address also happens to be 16-bits for the LC3 (this is addressability).

Building the LC3: Memory

- What logic component is **memory** made of?

  Gated D-Latches



- What does the **MAR (Memory Address Register)** store?

  Address to read/write from

- What does the **MDR (Memory Data Register)** store?

  Data read from/written to memory

3.2.5 Condition Code (CC) Register

Normally, the program counter (PC) increments by 1 to move to the address of the next instruction (because addressability happens to be equal to instruction size). If instructions took up 4 memory addresses (i.e., larger than addressability by 4x), then the PC would increment by 4.

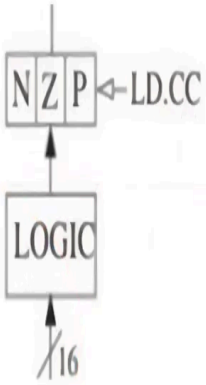
However, in a real program if-else statements and things like while loops require jumping around in the instructions, not just traveling forward contiguously.

The LC3's solution is the condition code register, a 3-bit register of flip-flops labeled N, Z, and P. For any operation that writes to a general purpose register (ADD, AND, NOT, LD, LDI, LDR) and JSR, the CC is set based on the result of the operation.

Condition Code (CC) Register

The Condition Code register stores the sign of the result computed in the most recent, CC applicable instruction.

- If **LD.CC** is set, the output will change the Condition Code Register to...
  - **N** = 100 if the value on the bus is **negative**
  - **Z** = 010 if the value on the bus is **zero**
  - **P** = 001 if the value on the bus is **positive**



*Only instructions that **write to registers** set the CC (except JSR):*

- *ADD, AND, NOT*
- *LD, LDI, LDR*

- More on this when we get to branching / control flow with BR instructions and JMPs.

3.2.6 Finite State Machine (FSM)

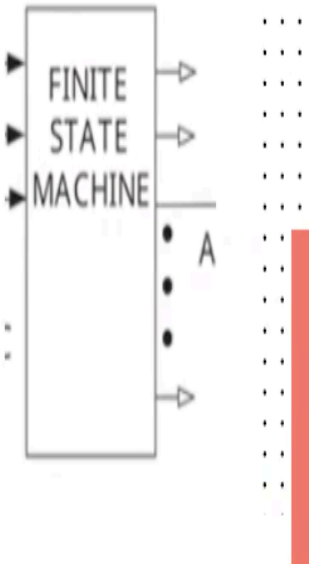
The FSM is the central orchestrator. It exists to set the signals, so all actions like loading the MAR, incrementing the PC, enabling a tri-state buffer to load a value onto the bus are controlled by the FSM.

Finite State Machine (FSM)

Recall in the last lab, we implemented a Finite State Machine to control a car's speed.

The LC3 has a much more complex FSM to control its operations:

- Controller of the LC3
  - Think of the FSM as the brain that makes all the decisions
  - The rest of the LC3 is the body that implements those decisions
- Sets control signals
- Tells LC-3 what to do and when, based on previous states & instructions



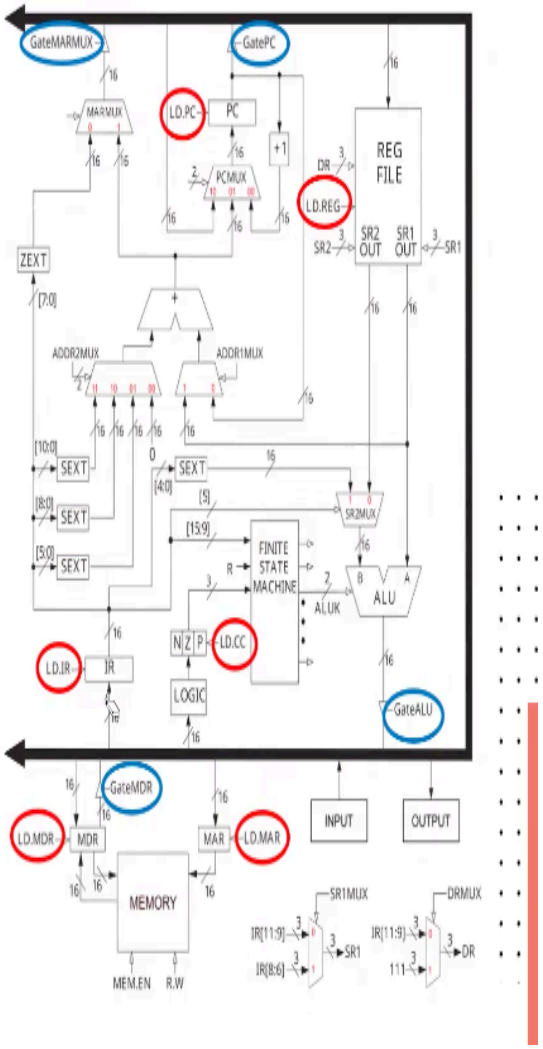
3.2.7 LD and Gate Signals

Gate signals control tri-state buffers which allow a component to pass its value onto the bus. For example, if GatePC is enabled, the PC is pushing its value onto the bus for other components to see. - At most 1 component can have an active gate signal at a time.

LD signals are essentially Write-Enable signals that allow registers to capture information from the bus. - For example, if LD.MAR is asserted, then the MAR is reading a value off the bus and loading it onto its flip-flops on the clock edge. - Clock is a bit separate from the WE signal here. You can think of it as "ANDed" so that the flip-flop doesn't see a clock edge when WE is low.

LC3: Signals

- What is the purpose of **LD** and **Gate** signals?
- What controls them, the ALUK, and the MUXes?



3.3 Machine Instructions

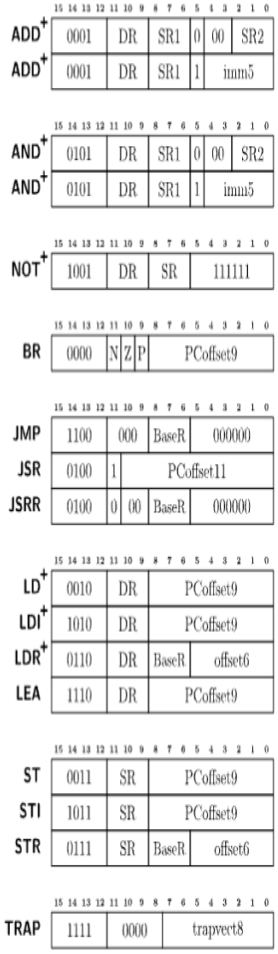
3.3.1 Instruction Set Architecture (ISA) for the LC3

A central part of the von Neumann model for computer processing is that both the program and data are stored as sequences of bits in the computer's memory. The program is executed one instruction at a time, directed by the control unit (FSM), with operations performed by the processing unit (ALU + Registers).

Instruction-set architecture is specific to the type of hardware implemented, as it is a concise way to encode the information/instructions the machine executes, working only with operations and registers, and memory values.

Note: All PC-relative offsets are relative to `PC + 1` (current instruction's address + 1, since the counter increments during the FETCH stage, occuring before the current EXECUTE stage).

CS2110 Reference Sheet



Trap Vector	Assembler Name
x20	GETC
x21	OUT
x22	PUTS
x23	IN
x25	HALT

Device Register	Addr
Keybd Status Reg	xFE00
Keybd Data Reg	xFE02
Display Status Reg	xFE04
Display Data Reg	xFE06

Instructions marked with a plus (+) set the condition codes.

3.3.2 Macrostates: FETCH, DECODE, EXECUTE

A macrostate is a larger umbrella term to group individual FSM states into higher-level phases describing their purpose.

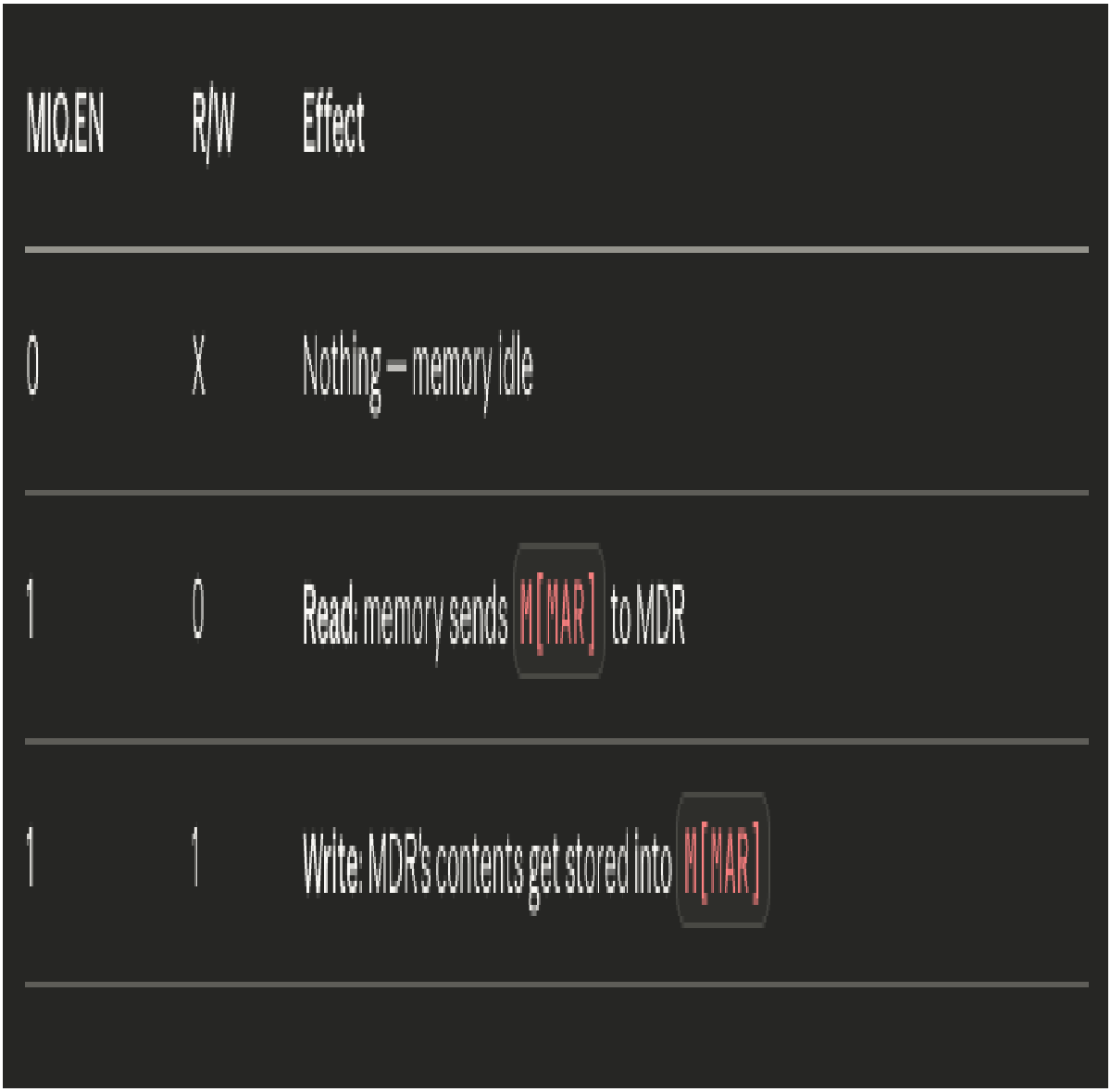
Tracing FETCH

```
Cycle 1
-------
MAR <- PC
    GatePC (tri-state buffer) => PC value onto Bus
    LD.MAR (WE=1) => MAR obtains value from Bus
PC <- PC + 1
    PCMUX selector is 00
    LD.PC (WE=1) => PC latches PCMUX output
```

PCMUX is a 3-input multiplexer with 2 selector bits. There are 3 types of inputs corresponding to `00`, `01`, and `10`: 1. PC + 1: Normal fetch when moving onto the next sequential instruction. In LC-3 compuer, since all instruction sizes are fixed, this can increment. 2. Bus: Used with JMP/RET, when a specific address computed elsewhere is read from the bus. 3. PC + SEXT Offset: Used with BR/JSR, which is the current PC address + sign-extended offset from IR.

```
Cycle 2
-------
MDR <- M[MAR]
    MEM.EN => memory performs op using address
    MEM.RW => signals op to perform
    LD.MDR (WE=1) -> MDR obtains instruction value
```

MEM.EN enables memory to perform operations. MEM.RW signals which operation to perform.




```
Cycle 3
-------
IR <- MDR
    GateMDR (tri-state buffer) => MDR value to bus
    LD.IR (WE=1) => IR obtains instruction value
```

Tracing DECODE

```
Cycle 1
-------
```

```
opcode <- IR[15:12]
BEN (Branch Enable Register) <- IR[11]&N + IR[10]&Z + IR[9]&P
```

- The FSM reads the first 4 bits of the instruction to figure out what operation the opcode matches to, which will affect how the remaining 12 bits are read and used in the EXECUTE phase.

EXECUTE is kind of a catch-all term for carrying out the specific operation identified in the DECODE step. The ISA set below specifies more details per operations.

3.3.3 Cycle Counting: Combinational vs. Sequential Logic

insert notes here

The Bus In the LC3 model, there is only a single bus wire in the datapath, and since only one value can be propagated stably across the bus at a time, if multiple different values must be transported via the bus, that's a clear sign we need to separate into multiple cycles.

In real processors, a bus may have more wires and more complex components to face this throughput challenge.

3.3.4 Tracing EXECUTE Operations

3.3.4.1 Special Codes and Selectors

```
PCMUX: Choose source for new PC value
00 -> PC + 1 (normal increment by size of instruction)
01 -> Load address from bus (JMP, RET, TRAP)
10 -> Address adder output (relative w/ SEXT)
```

```
ADDR1MUX: Select first input to address adder
0 -> PC (PC-relative addressing)
1: SR1 output (Base + offset addressing)
```

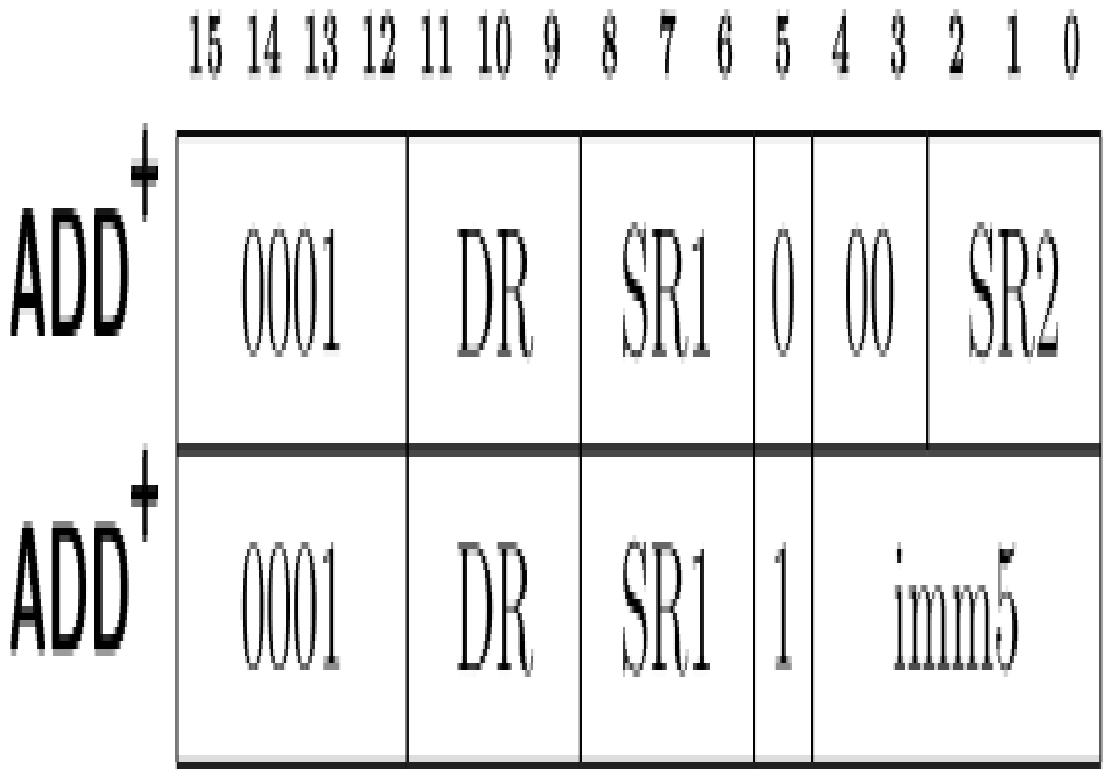
```
ADDR2MUX: Select second input to address adder
00 -> 0 (no offset)
01 -> SEXT(offset6)
10 -> SEXT(PCoffset9)
11 -> SEXT(PCoffset11)
```

```
MARMUX: Pick one of two values to load onto bus
0 -> ZEXT(IR[7:0]) for TRAP
1 -> Use address adder output (common)
```

```
ALUK: Choose ALU operation
00 -> ADD: SR1 + SR2MUX output
01 -> AND: SR1 & SR2MUX output
10 -> NOT: bitwise complement of SR1
11 -> PASS: pass SR1 through unchanged
```

3.3.4.2 Arithmetic and Logical Operations

Operate instructions read and modify data. Note the + superscripts. Since these operations write to registers, they automatically set condition codes into the condition code register, which stores NZP flags.

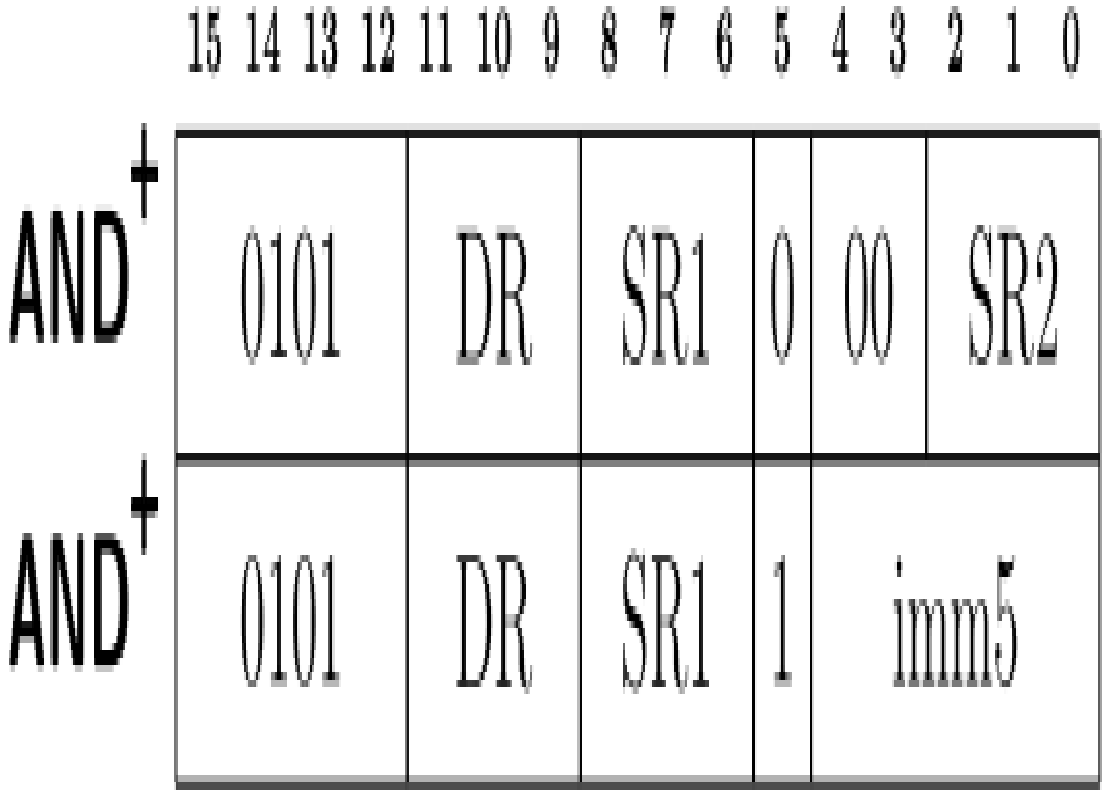


ADD Mode 0 performs: `DR <- SR1 + SR2` ADD Mode 1 performs: `DR <- SR1 + SEXT(imm5)` - Sometimes, we want to add data in a register and a small constant which doesn't need to be first loaded into the register. - However, to perform an operation between the data stored at SR1 and this two's-complement encoded integer, it must first be sign-extended. - Bit 5 acts as the flag to indicate which ADD operation is used.

```
ADD MODE 0 (IR[5] = 0)
----------------------
Cycle 1: DR <- SR1 + SR2
SR1MUX = 1: Select IR[8:6] as SR1
SR2MUX = 0: Select SR2 instead of imm5
ALUK = 00: Set ALU to ADD operation
GateALU = 1: Enable tri-state buffer to load ALU output onto bus
DRMUX = 0: Select IR[11:9] as DR
LD.REG = 1: Enable loading of value from bus to DR
LD.CC = 1: Set condition code based on output

ADD MODE 1 (IR[5] = 1)
----------------------
Cycle 1: DR <- SR1 + SEXT(imm5)
SR1MUX = 1: Select IR[8:6] as SR1
```

```
SR2MUX = 1: Select SEXT(imm5) as SR2
ALUK = 00: Set ALU to ADD operation
GateALU = 1: Enable tri-state buffer to load ALU output to bus
DRMUX = 0: Select IR[11:9] as DR
LD.REG = 1: Enable loading of value from bus to DR
LD.CC = 1: Set condition code based on output
```

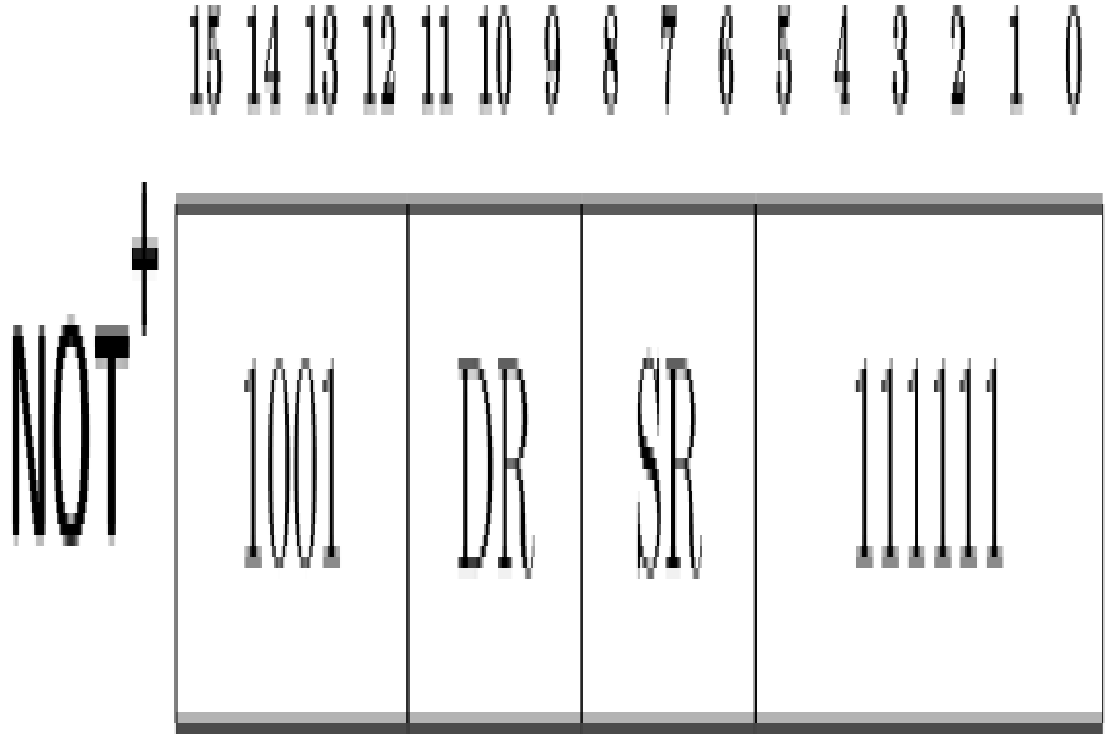
- Notice that bit 5 from IR goes to become the SR2MUX selector bit.



AND Mode 0 performs: `DR <- SR1 & SR2` AND Mode 1 performs: `DR <- SR1 & SEXT(imm5)`

```
AND MODE 0 (IR[5] = 0)
----------------------
Cycle 1:
SR1MUX = 1: Select IR[8:6] as SR1
SR2MUX = 0: Select SR2 and not imm5 as other operand
ALUK = 01: Set ALU to AND operation
GateALU = 1: Enable tri-state buffer to load ALU output onto bus
DRMUX = 0: Select IR[11:9] as DR
LD.REG = 1: Enable loading of value from bus to DR
```

```
LD.CC = 1: Enable update of condition code

AND MODE 1 (IR[5] = 1)
----------------------
Cycle 1:
SR1MUX = 1: Select IR[8:6] as SR1
SR2MUX = 1: Select SEXT(imm5) as SR2
ALUK = 01: Set ALU to AND operation
GateALU = 1: Enable tri-state buffer to load ALU output onto bus
DRMUX = 0: Select IR[11:9] as DR
LD.REG = 1: Enable loading of value from bus to DR
LD.CC = 1: Enable setting of condition code from value on bus
```

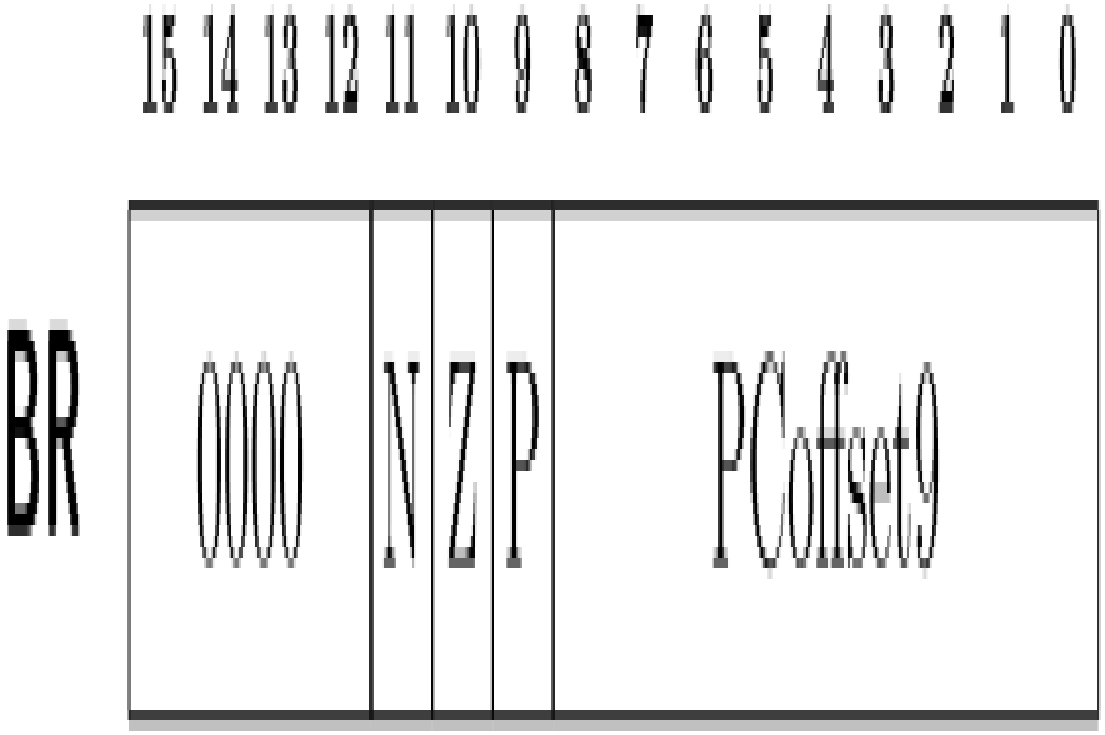


NOT performs: `DR <- ~SR`

```
Cycle 1:
SR1MUX = 1: Select IR[8:6] as SR
ALUK = 10: Set ALU operation to NOT
GateALU = 1: Enable tri-state buffer to load ALU output onto bus
DRMUX = 0: Select IR[11:9] as DR
```

```
LD.REG = 1: Enable loading of value from bus to DR
LD.CC = 1: Enable setting of condition code from value on bus
```

3.3.4.3 Branch and Jump Control flow instructions modify the normally-sequential flow of instructions processing. That is, instead of moving down in order through the sequence of instructions, we may want to jump around.

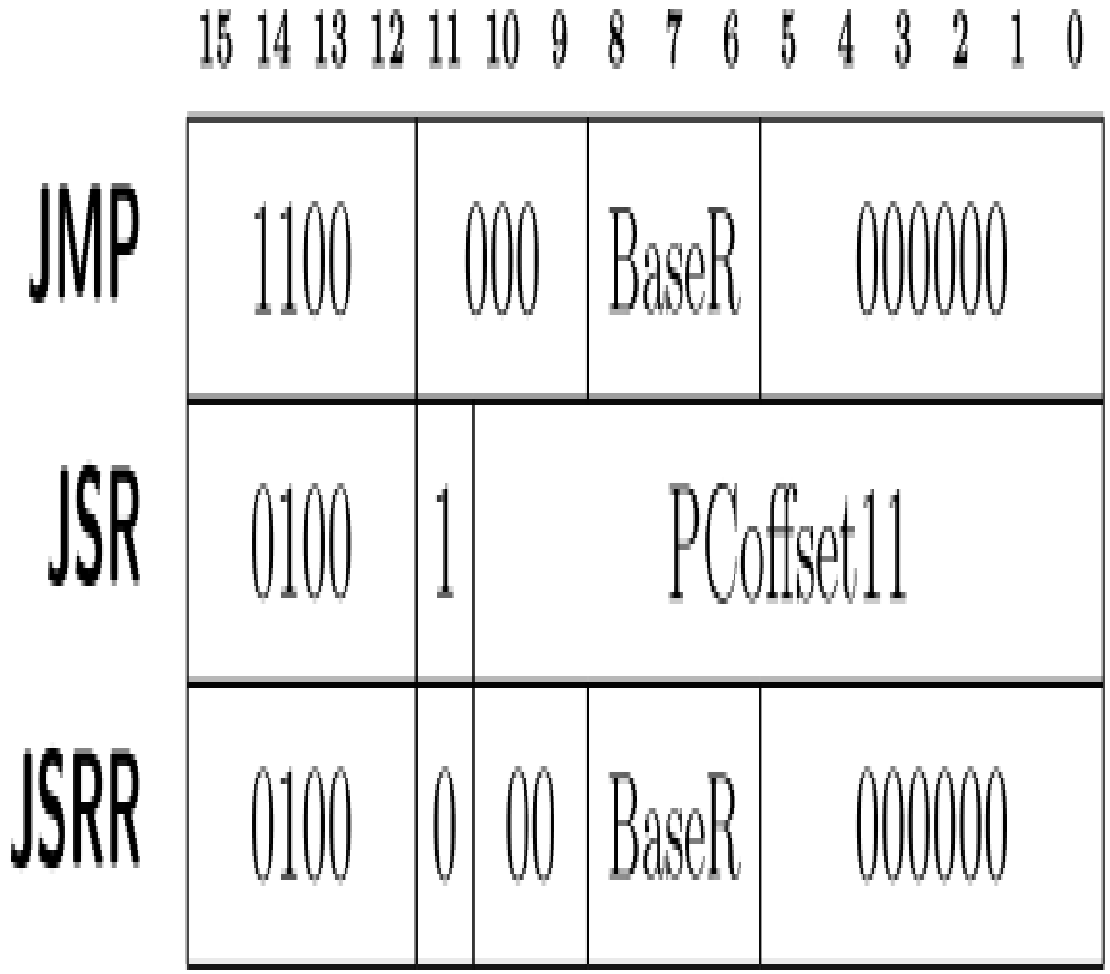


BR (branch) performs the decision-making (if statements, loops). The individual bits labeled NZP are flags for what condition code triggers the branch to be run. - If N=1, Z=1, and P=0, then the branch would run if the CC register stores a 1 in either N or Z (i.e., previous arithmetic/logic output was negative or 0). - Setting all 3 branches to 1 results in an unconditional branch (always goes to target address regardless of what happens before, similar to a `while True` statement). - Setting all 3 branches to 0 results in a No-Op, which is pointless (it's never executed).

The 9-bit offset `[8:0]` specifies a target address relative to the already-incremented PC counter (recall `PC <- PC + 1` already stores the address of the next instruction to be executed). This provides a range of -256 to +255 instructions beyond the one right after the BR.

```
Cycle 1: IF BEN is TRUE
ADDR1MUX = 0: Select PC as first input to address adder
ADDR2MUX = 10: Select SEXT(IR[8:0]) as second input to address adder
```

```
PCMUX = 10: Select address adder output for PC
LD.PC = 1: Enable loading of newly computed value onto PC
```



JMP (jump) performs an unconditional jump, similar to BRnzp, but can jump to any instruction in memory. `JMP BaseR` sets PC to whichever value is stored in the base register at BaseR address.

```
Cycle 1: PC <- BaseR
SR1MUX = 1: Select IR[8:6] as SR1
ADDR1MUX = 1: Select BaseR (SR1) as the first input for address adder
ADDR2MUX = 00: Select 0 as the second input for address adder
PCMUX = 10: Select address adder output for PC
LD.PC = 1: Enable loading of value from address adder onto PC
```

JSR (jump subroutine) first sets the return address (current PC) into R7 and then performs a PC-relative jump of `PC <- PC* + SEXT(PCoffset11)`. - Bit 11 distinguishes JSR from JSRR.

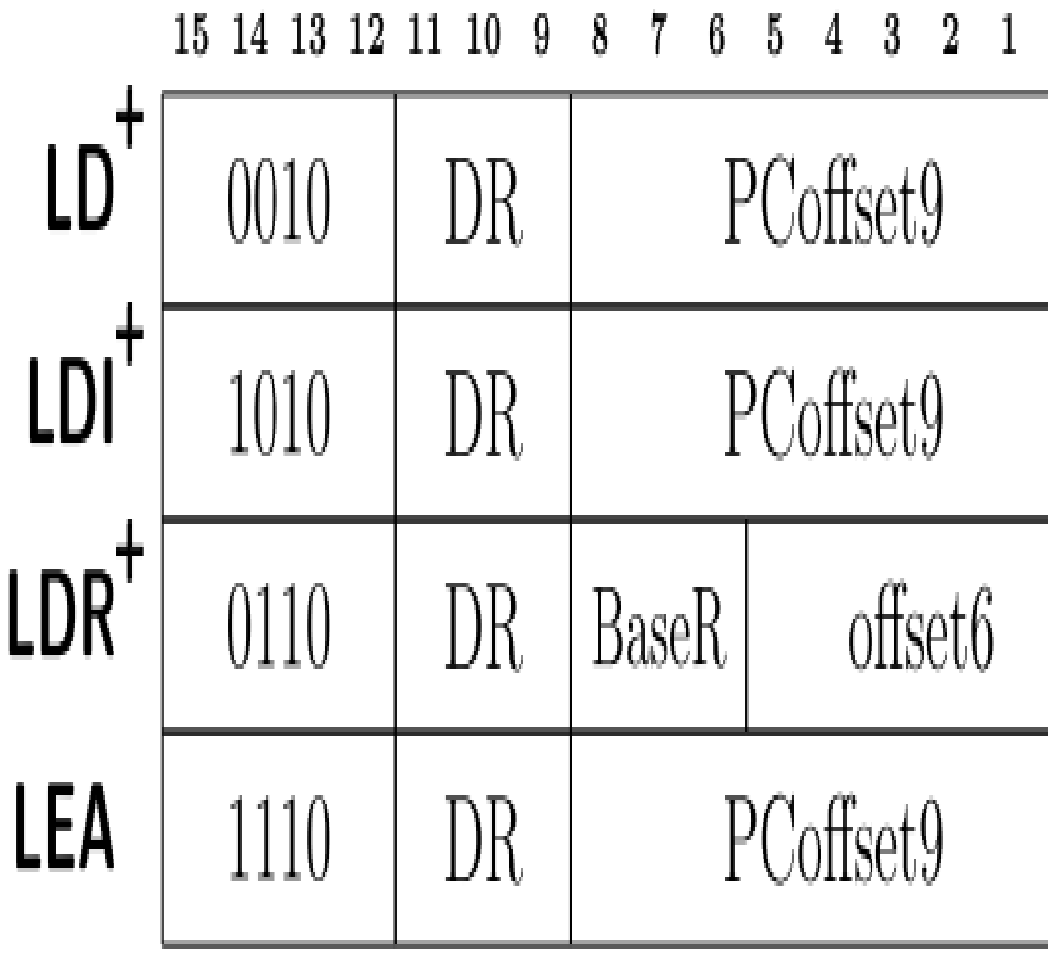
```
Cycle 1: R7 <- PC*, PC <- PC* + SEXT(PCoffset11)
DRMUX = 1: Select R7 (111) as destination register
GatePC = 1: Enable tri-state buffer to load PC* onto bus
```

```
LD.REG = 1: Enable loading of PC* into R7

ADDR1MUX = 0: Select PC as first input to address adder
ADDR2MUX = 11: Select SEXT(IR[10:0]) as second input to address adder
PCMUX = 10: Select address adder output to be new PC value
LD.PC = 1: Enable loading of new PC value
```

- Note that all logic here is combinational. There is only one thing on the bus (PC* which will be sent to R7), and both values update simulataneously at the start of the next clock cycle (PC updates to address adder output and REG updates to bus value).

JSRR (jump subroutine register) does the same thing as JSR, but uses a BaseR address to jump to the value stored in that register. The original PC is stored in R7.

```
Cycle 1: R7 <- PC*, PC <- BaseR
DRMUX = 1: Select R7 (111) as destination register
GatePC = 1: Enable tri-state buffer to load PC* to bus
LD.REG = 1: Enable writing of bus value into register

SR1MUX = 1: Select IR[8:6] as SR1
ADDR1MUX = 1: Select BaseR as first input to address adder
ADDR2MUX = 00: Select 0 as second input to address adder
PCMUX = 10: Select address adder output as new PC value
LD.PC = 1: Enable updating PC value
```



3.3.4.4 Data Access From Memory

LD (load) reads a value from memory into a register. Using PCOffset9, `MAR <- PC(incremented) + SEXT(PCoffset9)`, the value from that location is read and loaded into MDR, and then placed into destination register (DR). - Condition codes are set based on the NZP of the loaded value. This makes it easy to branch without a separate step.

```
Cycle 1: MAR <- PC* + PCoffset9
DRMUX = 0: Select IR[11:9] as DR
ADDR1MUX = 0: Select incremented PC as first input to address adder
ADDR2MUX = 10: Select SEXT(IR[8:0]) as second input to address adder
MARMUX = 1: Select address adder output to load onto bus
GateMARMUX = 1: Enable tri-state buffer to load address onto bus
LD.MAR = 1: Enable loading of value from bus onto MAR

Cycle 2: MDR <- M[MAR]
MEM.EN = 1: Enable working with memory
MEM.RW = 0: Set memory access mode to "read"
LD.MDR = 1: Enable loading of value from memory into MDR

Cycle 3: DR <- MDR
GateMDR = 1: Enable tri-state buffer to load MDR value onto bus
DRMUX = 0: Select IR[11:9] as DR
```

```
LD.REG = 1: Enable loading of MDR value from bus into register
```

LDI (load indirect) adds one layer of “indirectness” in terms of how we obtain the actual value of the data. Again, using PCoffset9, we compute `MAR <- PC(incremented) + SEXT(PCoffset9)`, which is some location in memory. But we expect the value in memory this time to be a pointer to where the data actually lives.

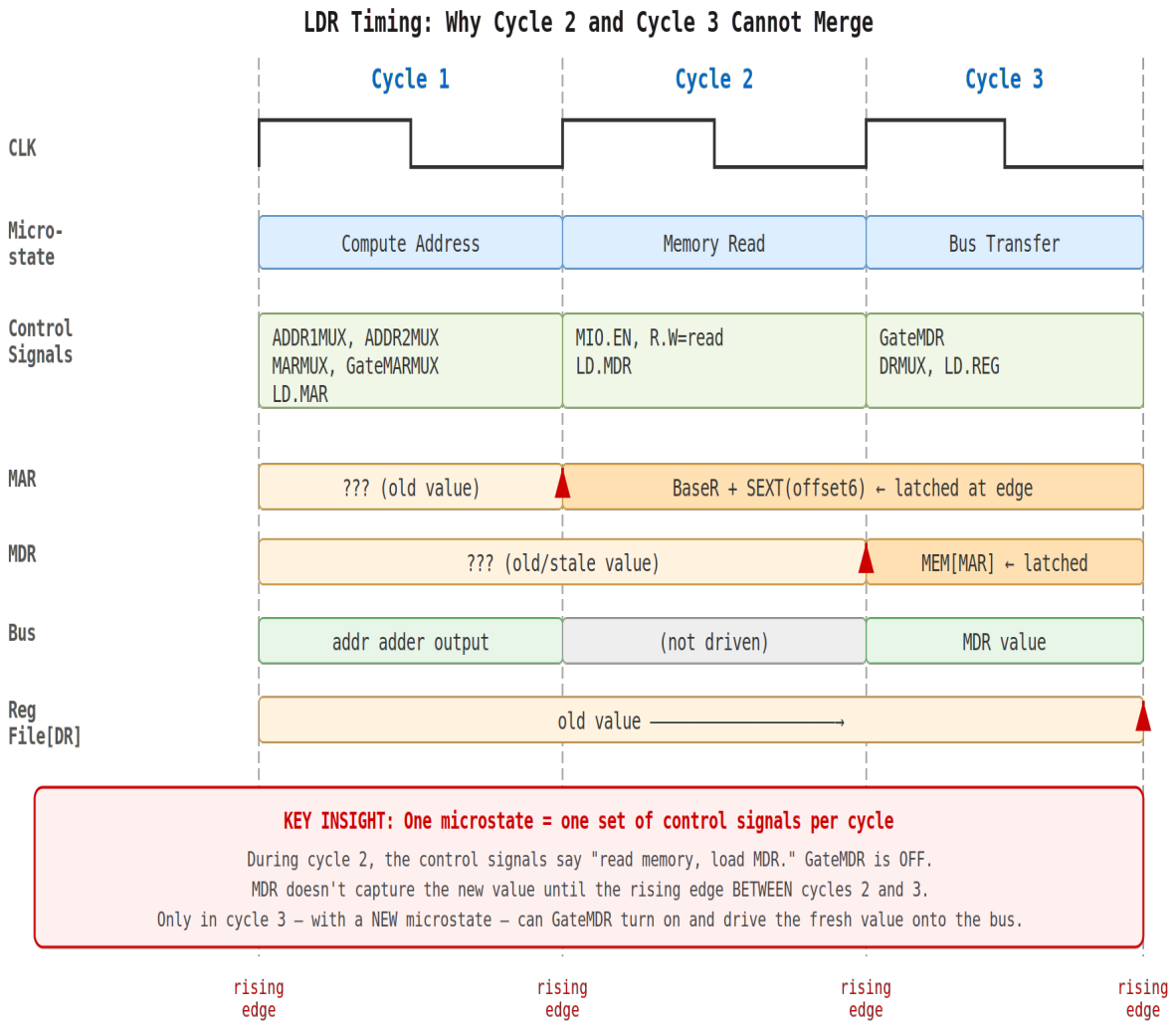
This solves the issue of being unable to access elements far away in memory, and is convenient for a few far-accesses. However, note that this perfoms additional memory accesses, which is slower.

```
Cycle 1: MAR <- PC(incremented) + SEXT(PCoffset9)
ADDR1MUX = 0: Get PC value
ADDR2MUX = 10: Get SEXT(PCOffset9) value
MARMUX = 1: Select address adder output as candidate MAR value
GateMARMUX = 1: Allow tri-state buffer to write to bus
LD.MAR = 1: Allow writing to MAR

Cycle 2: MDR <- M[MAR]
MEM.EN = 1: Enable memory access
MEM.RW = 0: Read from memory
LD.MDR = 1: Load value into MDR

Cycle 3: MAR <- MDR
GateMDR = 1: Enable tri-state buffer to write MDR onto bus
LD.MAR = 1: Load value on bus into MAR

Cycle 4: MDR <- M[MAR]
MEM.EN = 1: Enable memory access
MEM.RW = 0: Read from memory
LD.MDR = 1: Load value into MDR

Cycle 5: DR <- MDR
GateMDR = 1: Enable tri-state buffer to write MDR onto bus
LD.REG = 1: Load value from bus onto destination register
LD.CC = 1: Load value from bus onto condition code (NZP)
```

LDR (load register) allows loading elements which are within a 6-bit signed offset to a memory address stored in a base register (BaseR). This is often used as a type of “array indexing”, so that if the base register points to the start of an array, you can use the offset to specify where the next element is in memory. - This solves the issue of being able to access a local region around an arbitrary base address, which is helpful for multiple accesses around an arbitrary address at the cost of taking up a base register. - This is also useful when needing to dereference a pointer (access a value at an address stored in a register). Just set the offset6 to 0.

```
Cycle 1: MAR <- BaseR + SEXT(offset6)
SR1MUX = 1: Select IR[8:6] as BaseR
ADDR1MUX = 1: Select BaseR as first input to address adder
ADDR2MUX = Select SEXT(IR[5:0]) as second input to address adder
MARMUX = 1: Select output of address adder to load onto bus
GateMARMUX = 1: Enable tri-state buffer to load output onto bus
LD.MAR = 1: Enable loading of value from bus onto MAR

Cycle 2: MDR <- M[MAR]
MEM.EN = 1: Enable memory
MEM.RW = 0: Set read mode for memory
LD.MDR = 1: Select memory as input to MDR

Cycle 3: DR <- MDR
GateMDR = 1: Enable tri-state buffer to load MDR value onto bus
DRMUX = 0: Select IR[11:9] as DR
```

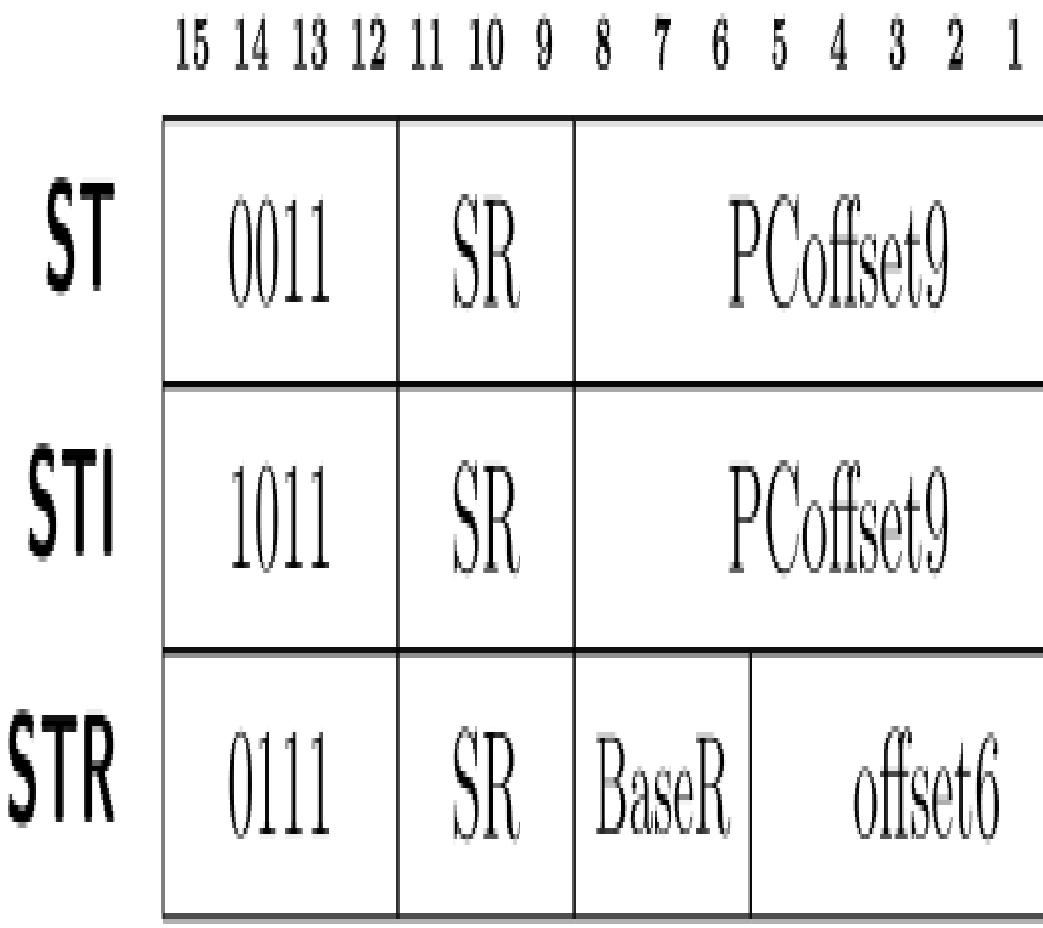
```
LD.REG = 1: Enable loading of value from bus onto DR
```



LEA (load effective address) is the distinct operation in the group of "load" operations. It does not access memory at all. LEA computes `PC + SEXT(PCoffset9)` and stores *that* into the destination register (DR).

It's useful for obtaining the starting address of a contiguous data structure in memory (array/string), and then using LDR on that to access each of the elements.

```
Cycle 1: DR <- PC* + SEXT(PCoffset9)
ADDR1MUX = 0: Select PC* as first input to address adder
ADDR2MUX = 10: Select SEXT(IR[8:0]) as second input to address adder
MARMUX = 1: Select output of address adder to load onto bus
GateMARMUX = 1: Enable tri-state buffer to load onto bus
DRMUX = 0: Select IR[11:9] as DR
LD.REG = 1: Enable loading of bus value onto DR
```



3.3.4.5 Data Storage Into Memory

ST (store) computes the address `PC* + SEXT(PCoffset9)` and writes the data stored in the source register (SR) into that location in memory.

Notes: - Cycle 1 and cycle 2 are independent of one another and order is interchangeable. - Since the operations require two different values being on the bus, they must stay in separate cycles.

```
Cycle 1: MAR <- PC* + SEXT(PCoffset9)
ADDR1MUX = 0: Select PC* as first input to address adder
ADDR2MUX = 10: Select SEXT(PCoffset9) as second input to address adder
MARMUX = 1: Select address adder output to output to bus
GateMARMUX=1: Enable tri-state buffer to write to bus
LD.MAR = 1: Load value from bus to MAR

Cycle 2: MDR <- SR
SR1MUX = 0: Select IR[11:9] as SR
ALUK = 11: Set ALU to pass-through operation
GateALU = 1: Enable tri-state buffer to write ALU output to bus
LD.MDR = 1: Load value from bus onto MDR

Cycle 3: M[MAR] <- MDR
```

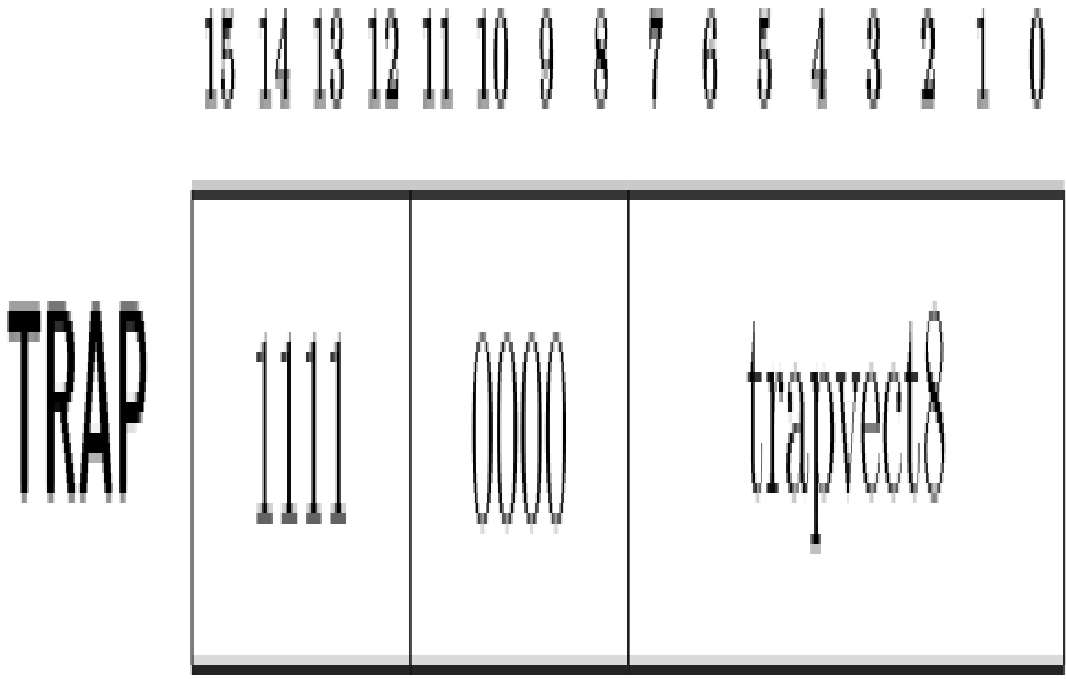
```
MEM.EN = 1: Enable memory
MEM.RW = 1: Select "write" memory operation
```

STI (store indirect) computes the address `PC(incremented) + SEXT(PCoffset9)`, goes to that address in memory, reads the pointer there, and then writes the data stored in the source register (SR) to *that* address.

```
Cycle 1: MAR <- PC* + SEXT(PCoffset9)
ADDR1MUX = 0: Select PC* as first address adder input
ADDR2MUX = 10: Select SEXT(IR[8:0]) as second address adder input
MARMUX = 1: Select address adder output to load onto bus
GateMARMUX = 1: Enable tri-state buffer to load value onto bus
LD.MAR = 1: Enable loading of value onto MAR

-- need to use loaded MAR value

Cycle 2: MDR <- M[MAR]
MEM.EN = 1: Enable memory
MEM.RW = 0: Enable read memory
LD.MDR = 1: Enable loading of value into MDR

-- need to use loaded MDR value

Cycle 3: MAR <- MDR
GateMDR = 1: Enable loading of MDR value onto bus
LD.MAR = 1: Enable receiving value from bus

-- need to transport other data on the bus

Cycle 4: MDR <- SR
SR1MUX = 0: Select IR[11:9] as SR1
ALUK = 11: Pass SR1 value through
GateALU = 1: Enable tri-state buffer to load ALU output onto bus
LD.MDR = 1: Enable loading of bus value onto MDR

-- need to use loaded MDR register value

Cycle 5: M[MAR] <- MDR
MEM.EN = 1: Enable memory
MEM.RW = 1: Enable memory write
```

- `MEM.RW = 1` triggers a hardcoded operation with MDR/memory where it uses the value stored in MAR currently as the address/destination in memory and writes its own value there.

STR (store register) computes the address `BaseR + SEXT(offset6)` and writes data stored in the source register (SR) to that memory address.



3.3.4.6 Special Operations + I/O

Trap Vector	Assembler Name
x20	GETC
x21	OUT
x22	PUTS
x23	IN
x25	HALT

The TRAP operation helps load a pre-defined OS-managed subroutine into the Program Counter.

Essentially, it takes a zero-extended 8-bit trap vector and looks in that location in memory (reserved, hence why user programs tend to start later in memory). That location contains the address of the first instruction of a particular "function", which is then loaded onto PC.

It is similar to a JSRR, where the "Register" is basically the trap vector table mapping. Each particular trap code is mapped to an address which holds the OS subroutine to be run, and so we jump to execute that subroutine at that particular address when a TRAP command is run.

```
Cycle 1: R7 <- PC*
DRMUX = 1: Select R7 as DR
GatePC = 1: Enable tri-state buffer to load PC* value on bus
LD.REG = 1: Enable loading of bus value into DR

Cycle 2: MAR <- ZEXT(trapvect8)
MARMUX = 0: Select ZEXT to load onto bus
GateMARMUX = 1: Enable tri-state buffer to load value onto bus
LD.MAR = 1: Enable loading of bus value onto MAR
```

```
Cycle 3: MDR <- M[MAR]
MEM.EN = 1: Enable memory access
MEM.RW = 0: Enable readonly access
LD.MDR = 1: Enable loading of memory value into MDR

Cycle 4: PC <- MDR
GateMDR = 1: Enable tri-state buffer to load MDR onto bus
LD.PC = 1: Enable PC to be updated
PCMUX = 01: Select bus value to update PC
```

- Cycles 1/2 are distinct cycles because two different values need to be loaded/transported via the bus.
- Cycle 2/3 are distinct since memory access always takes one or more cycles.
- Cycle 3/4 are distinct because only on the rising edge of cycle after 3 is the correct value populated/updated in MDR, so then it can be used in cycle 4 via combinational logic.

Device Register	Addr
Keybd Status Reg	xFE00
Keybd Data Reg	xFE02
Display Status Reg	xFE04
Display Data Reg	xFE06

3.3.5 Assembly Syntax for the LC3

Assembly is specific to the instruction set architecture (ISA).

Writing assembly allows us to use human-readable macros (like LD) and avoid writing numerical offsets and instead use labels (variables).

"Assemble" is the process of converting human-readable assembly language into machine binary. The assembler works in two passes. On the first pass, it goes through every line and assigns each one a memory address, starting from wherever the program starts (`.ORIG` directive).

On the second pass, it generates the binary encoding for each instruction. This is where it will raise an error if a provided label is outside of the offset range for a specific instruction, etc.

3.3.6 General Rules

Combinational logic occurs during a clock cycle's level edges Sequential logic occurs during rising clock edges (reading/writing to same register) If a signal is the result of combinational delay during the cycle, then destination register captures it at the ending clock edge of that cycle if its LD.* is asserted.

4 Assembly and Computer Programs

4.1 Assembly Basics

4.1.1 Definition and Properties

In the section, we discussed the LC3 datapath and traced different cycles in FETCH, DECODE, and EXECUTE macrostates.

For a human to be able to program a computer though, it's extremely tedious to write 16-bit instructions each time. Instead, it's easier to have a set of macros, such as `AND` or `ADD` along with other text syntax to compose these same instructions. That's where assembly language comes in.

Assembly language is a human-readable text syntax (NOT just 0's and 1's) which we can write programs in. Since it is closely coupled with the instructions machines can execute in hardware, assembly language is specific to its instruction-set architecture (ISA) in terms of what operations its EXECUTE supports, etc.

4.1.2 Assembler and Assembly Passes

Since assembly is text and computers only understand electrical signals (0's and 1's), we need some "translator" in between, which is the assembler. An assembler takes the text program instructions (.asm file) and translates it into an .obj file which can be loaded into memory and executed by a computer's processor.

The LC3 assembler makes two passes over the source file.

Pass 1 In the first pass, the assmbler builds a symbol table, which maps *labels* (similar to variables which store addresses) to their addresses.

The assembler reads the .asm file from top-down, line-by-line. Each line is assigned a particular address in memory, starting from wherever `.orig` is set.

```
.orig x3000
BRnzp SKIP ; address x3000
ADD R0, R0, #1 ; address x3001
SKIP ADD R1, R1, #1 ; address x3002
HALT ; address x3003
.end
```

The lookup table, in this scenario, would just be

```
SKIP: x3002
```

Assembly directives are used to guide the assembler in its actions. For example, `.orig x3000` isn't an instruction in the program, but rather an instruction for the assembler regarding where the program should start.

Pass 2 The second pass generates the actual binary, now that the symbol table is prepared.

When generating the .obj file, imagine that the assembler is just writing a sequence of bytes which will then be loaded into a machine's memory as instructions/data.

The assembler generates output reading both actual instructions (e.g., `ADD R0, R0, #1`), but also handles assembler directives (e.g., `.fill x1234`) and labels. - For example, if `.fill x1234` was on the 4th line, where the 1st line of the program started `.orig x3000`, that line would be address `x3002` and so the hex value `x1234` would get stored directly in output at `x3002` location. - Another example is when labels are used. If we write `X .fill x1234` and then the program uses `LD R1, X`, then the assembler computes the PCoffset9 using `X_addr - PC*` (not that this is incremented PC).

The .obj file output is a contiguous block of 16-bit values (at least in the case of the LC3) and when running the program on a machine the loader would place it into one contiguous chunk of memory starting at the `.orig` address. Instructions and data live inside of that same chunk.

4.1.3 Manually Generating a Symbol Table

For the following program:

```
.orig x3000
    LD R0, X ; R0 = a
    LD R1, Y ; R1 = b
    NOT R2, R1
    STI R3, X
    BRnz ELSE
    ST R0, Y
    ST R1, X
ELSE
    ADD R0, R0, R0
    ST R0, Y
HALT

X .fill 12 ;; example values
BLK .blkw 5
S .stringz "To be or not to be";
Y .fill 10

.end
```

The symbol table looks something like this:

```
Label: Address
ELSE: x3007
X: x300A
BLK: x300B
S: x3010
Y: x3023
```

4.1.4 Assembler Directives

4.1.4.1 **`.orig X3000`** Tells the assembler "place the following program instructions" at the specified location.

At the beginning of the program, registers are initialized with random values.

4.1.4.2 **`.end`** Tells the assembler to stop assembling the program, but does NOT stop the program from running beyond this point.

If we continue stepping through the program, the next address in PC just executes, and there will often be an illegal opcode error message.

`HALT` is required to stop the program.

4.1.4.3 **`.blkw`** `.blkw` stands for "block word" and reserves a contiguous array of memory locations. The values at these addresses through remain random upon initialization.

```
array    blkw 10 ; reserves 10 contiguous blocks of memory, starting from current line's memory address
```

4.1.4.4 **`.stringz`** `.stringz` stands for "string zero", meaning that the string is null-terminated (assembler appends a `x0000` word at the end to indicate the end of the word).

```
message    .stringz "Hello world!\n"
```

- Requires passing of a string literal in double quotes.

4.1.5 Organization of an Assembly File

```
.orig x3000

; All program instructions

HALT

; Allocate memory space for data
; Recall that .fill and .stringz fill in specified data value at the memory location of the line (
    sequentially)

.end
```

4.1.6 Arithmetic Operations in Assembly

When working with constant values representable by 5 bits, which is in the range $[-2^4, 2^4 - 1]$, we can use a AND-then-ADD combination. - This is because the initial values in the register are random, and there is no native "SET" operation to update the value atomically.

```
AND R1, R1, #0
ADD R1, R1, #10 ; some value between -16 and 15
```

With anything than a 5-bit sign-extended integer, we can access that value from memory, which is 16-bits wide (addressability of LC3).

We take advantage of assembler directives when writing assembly, because we can use text and the assember

4.1.7 Linking Assembly Files Together

Say that we want to make our assembly code more modular and split up certain functionalities or subroutines (equivalent of functions) into separate files.

We can define subroutines in separate files from the main file, and then reference those using labels.

The first step is to write the helper file. The helper file tends to start at a later address from the main file (`.orig`) in order to avoid conflicts with data/instruction storage alongside the main file.

```
        ; Helper file containing multiplication subroutine
        ; R1 <- R2 * R3

        ; Notice that the program starts at a later address
        ; Also, note that the caller of the subroutine must have deep knowledge of what it does
        ; because it may touch registers which are shared with the main file

        .orig x4100

        ; Initialize R1
mult    AND R1, R1, #0

        ; Stop when R2 is negative
        ; Use instead of checking 0 because this sets condition code immediately before BR check
loop    ADD R2, R2, #-1
        BRn done
        ADD R1, R1, R3 ; add R3 value (R2) times
        BRnzp loop
done    RET ; if this file were run by itself, R7 would contain a random value
            ; we rely on coordination with main file which calls this

        .end
```

In the main file, we use the subroutine most commonly via the `JSRR` operation. We use a register to store the address because we often cannot guarantee it will lie within a PCOffset11.

Notice that we must also use `.external <label>` where the label is from the linked file. In the linking process, this can be resolved, but to use the label address in the main file, we must also manually use `.fill` on a "local label".

```
        ; Main program performing R0 <- AX^2 + BX + C


        .orig x3000

        ; Linker determines address of this in the other file
        ; However, we must use .fill first on this label
        ; That way the assembler can compute relative offsets in later usage
        .external mult

        LD R4, lib_ptr ; lib_ptr is our local value for starting address of subroutine

        ; Call the subroutine to perfrom R1 <- R2 * R3 with A * X
        LD R2, coef_A
        LD R3, val_X
        JSRR R4

        ; R2 <- AX + B
        LD R2, coef_B
        ADD R2, R1, R2

        ; Right now R3 is still val_X
        JSRR R4 ; R1 <- AX^2 + BX

        LD R2, coef_C
        ADD R0, R1, R2

        HALT
```

```
coef_A  .fill 3
coef_B  .fill 10
coef_C  .fill 5
val_X   .fill 2
lib_ptr .fill mult

        .end
```

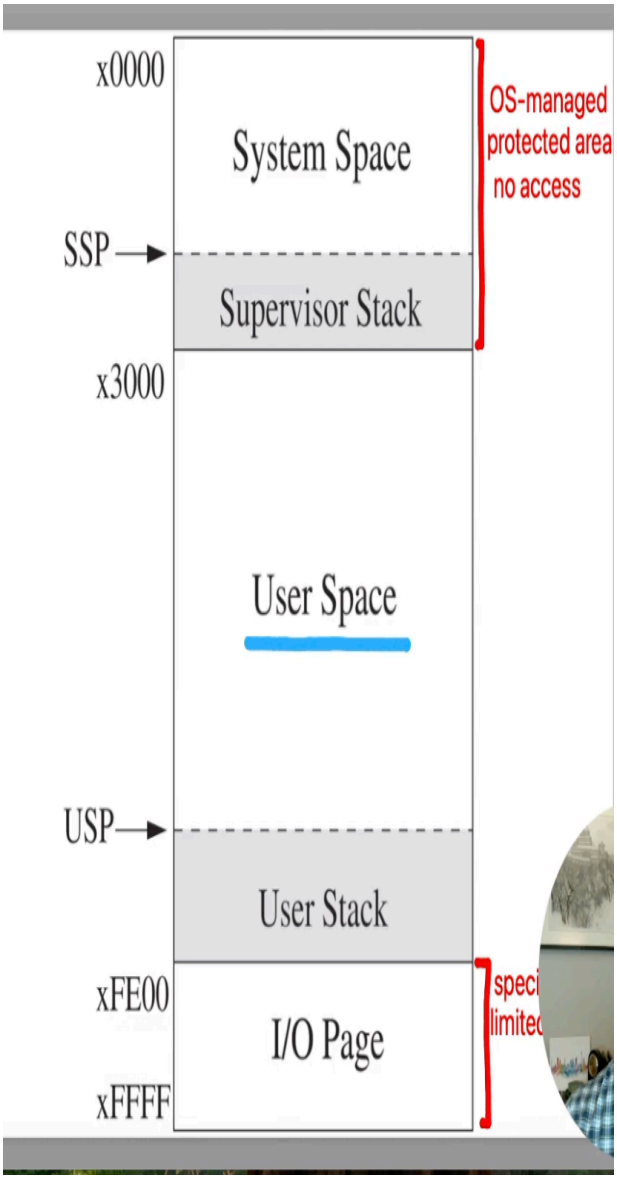
4.2 Memory Overview

4.2.1 Memory Map

Recall that the LC3 has a 16-bit address space, so there are 2^{16} memory locations, with an addressability of 16 bits, which is the size of an atomic piece of data at a particular memory location. - In our case, all memory locations can be represented with a sequence of 4-digit hexadecimal codes.

At the top of memory, at the lowest addresses is the system space. The computer OS maintains these processes and uses the first set of addresses.

Starting at x3000 is the user space. This contains data including: - program instructions - static data - heap data The user stack data starts at the end of the user space and grows upward.



4.2.2 The Call Stack

The stack is a contiguous region of memory in the LC3 which is used as a LIFO manner. In the big picture, when JSR is called the return address is stored in R7. But if that subroutine calls another subroutine (i.e., nested functions -> call stack), then R7 gets replaced and the original value is erased. The stack is used to solve that issue by storing R7 in the stack right before the nested call, hence the "call stack".

Conventionally, R6 is designated to be the stack pointer, starting at `x4000`. The stack grows moving towards zero (i.e., when elements are pushed onto the stack, they take on addresses `x3FFF`, `x3FFE`, and so on), and when elements are popped off the stack pointer moves back to `x4000`.

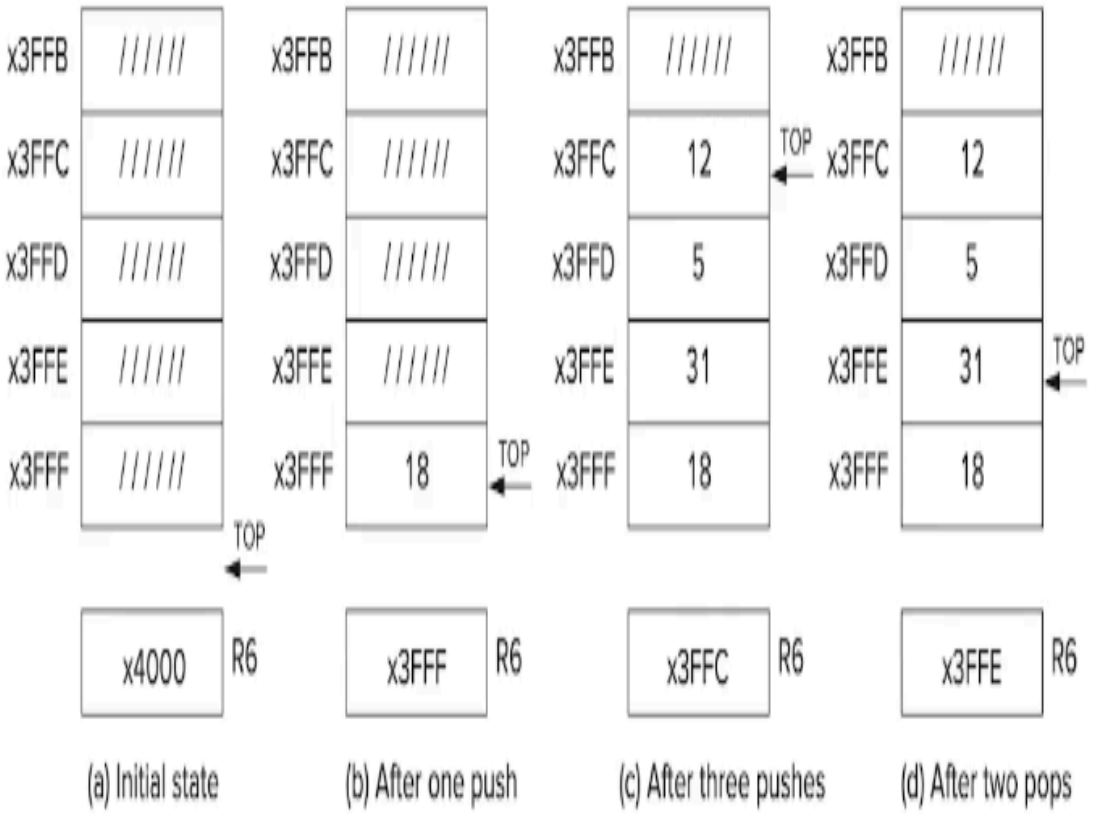


Figure 8.9 A stack, implemented in memory—data entries do not move.

When we push an element onto the stack, the stack pointer first gets decremented towards 0, and then the element is inserted at that pointer location (pointer is always pointing to the "top element" of the stack).

```
Push R0 Onto Stack
--------------------
ADD R6, R6, #-1
STR R0, R6, #0
```

When we pop from the stack, the element at the address held by the stack pointer is first stored into a register, then the stack pointer is moved upward back toward `x4000`.

```
Pop Off Stack Into R0
LDR R0, R6, #0
ADD R6, R6, #1
```

4.2.2.1 Single Subroutine Call: R7 In the most basic case, we may call upon a subroutine to perform a repetitive action for us using the following syntax:

```
JSR double
```

```
; some more code (resumes after subroutine RET)

double
    ADD R0, R0, R0
    RET
```

4.2.2.2 Multiple Subroutine Calls: R7, R6 Other times, we may call a subroutine within a subroutine. But there is an ISSUE: If only one register R7 is used to hold the return address, then with multiple nested subroutine calls all needing RET, previous return addresses get overwritten and lost, causing unexpected program behaviors.

```
JSR A
; some more code -> becomes unreachable when RET triggers in subroutine B

A
    ; some code
    JSR B
    ; some more code
    RET

B
    ; some code
    RET
```

To fix this, we use the call stack pointer at R6 to store the return address of each caller of a subroutine in that subroutine if it is not the last one in the recursive call stack (obtain latest return address from R7).

```
        JSR A
        ; caller continues here

A
        ADD R6, R6, #-1   ; push
        STR R7, R6, #0    ; save caller's return address

        ; some code
        JSR B
        ; some more code

        LDR R7, R6, #0    ; restore caller's return address
        ADD R6, R6, #1    ; pop
        RET               ; now returns to the correct return address

B
        ; some code
        RET
```

4.2.2.3 Subroutine Calls With Local Variables: R7, R6, R5 Suppose that now, there are nested subroutines (i.e., one subroutine calls another). But each subroutine also has its local variables which are also stored in stack memory. This leads to shaky references because everything is relative to R6 but depending on what local variables are pushed to the stack, offsets need to be computed over and over mentally.

```
JSR A
; more code

A
    ADD R6, R6, #-1
    STR R7, R6, #0      ; save return address
```

```
    ADD R6, R6, #-2     ; reserve 2 locals
```

At this point, the stack looks like this:

```
R6 -> local2
      local1
      saved R7
```

So far, accesses relative to R6 are ok.

```
    AND R0, R0, #0
    ADD R0, R0, #5
    STR R0, R6, #1      ; local1 = 5
```

Suppose that I also want to save R1 before calling another subroutine so that after calling that subroutine subsequent code can use R1 value.

```
    ADD R6, R6, #-1
    STR R1, R6, #0      ; push R1 temporarily
    JSR B
```

Now the stack is:

```
R6 -> saved R1
      local2
      local1
      saved R7
```

And we can no longer use `LDR R0, R6, #1` because that points to `local2` not `local1`.

The solution is to maintain a fixed anchor R5.

```
A
    ADD R6, R6, #-1
    STR R7, R6, #0

    ADD R6, R6, #-1
    STR R5, R6, #0

    ADD R5, R6, #0      ; fix frame pointer here

    ADD R6, R6, #-2     ; reserve locals
```

- `local1` = `R5 - 1`
- `local2` = `R5 - 2`

Everything relative to R5 is stable in terms of its offsets.

```
    STR R0, R5, #-1     ; store local1
    LDR R0, R5, #-1     ; load local1
```

Since R5 may also get written, it is stored in the call stack memory along with return address. For a particular caller A, there is a “stack frame” in memory of the following structure:

```
A's frame:
    higher addresses
        ...
        caller's older stack data
```

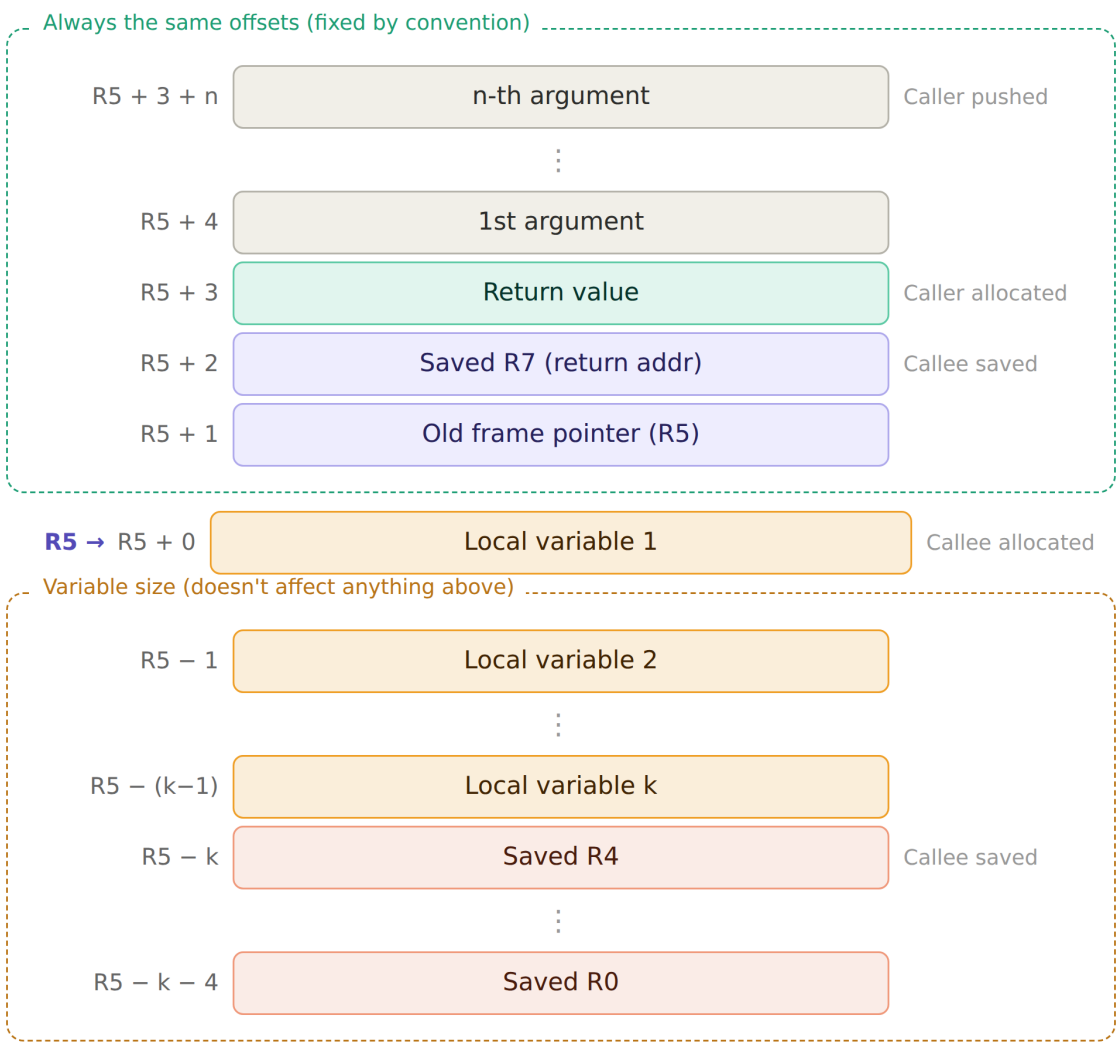
```
        return address back to caller        <- [R5 + 1]
        saved old R5 ('callers frame ptr)    <- [R5 + 0]   = [R5]
        local 1                              <- [R5 - 1]
        local 2                              <- [R5 - 2]   = [R6] after allocation
    lower addresses
```

So the full flow is: 1. We start in some outside code in the program. 2. Call `JSR A`. 3. Push R7 value (return address) to stack memory. - `ADD R6, R6, #-1` - `STR R7, R6, #0` 4. Push previous stack frame pointer to call stack. - `ADD R6, R6, #-1` - `STR R5, R6, #0` 5. Establish current frame's pointer and define local variables relative to this pointer. - `ADD R5, R6, R0` - `ADD R6, R6, #-2`: allocate two local variables - `STR R0, R5, #-1` - `STR R1, R5, #-2` 6. Call `JSR B` 7. Do the same thing in terms of storing call stack info. Perform some actions in code. 8. Move the stack pointer back to point at address containing stack frame address. - `ADD R6, R6, #_`: which is however many local variables there were 9. Restore previous caller's stack frame by popping the stack pointer. - `LDR R5, R6, #0` - `ADD R6, R6, #1` 10. Restore previous caller's return address by popping the stack pointer. - `LDR R7, R6, #0` - `ADD R6, R6, #1` 11. Call `RET` to go back to continue A's execution. 12. A does the same thing, and so on.

4.2.2.4 Build-Up And Tear-Down When the flow of execution is interrupted by a subroutine, that subroutine may use registers R0-R4 (i.e., non-reserved registers), but the original execution flow may have also had important values in those registers.

Overall: 1. The caller is responsible for pushing args in reverse order onto the stack, along with allocating one or more slots on the stack for the subroutine's return value. 2. The callee is responsible for build-up: First save R7 -> R5 -> Set R5 Frame Pointer -> allocate local vars -> save R0-R4 (states from the caller). 3. The callee is responsible for tear-down after executing its code: Pop callee-saved registers in reverse order -> deallocate local variables -> restore old frame pointer R5 -> restore old return address R7 -> RET. 4. The caller is responsible for obtaining the return value from the stack and saving it to a register (usually R0) and popping the arguments it pushed to the call stack.

The full flow then looks something like this for a function with arguments passed in via the call stack:



1. The caller pushes subroutine arguments onto the stack.

```
; Suppose there are two arguments, R0 and R1
; Push the arguments in reverse order
; That way when the callee accesses them, earlier args are at a smaller positive offset from R5
ADD R6, R6, #-1
STR R1, R6, #0
ADD R6, R6, #-1
STR R0, R6, #0

; Allocate one or more slots on the stack to store return value of the subroutine
ADD R6, R6, #-1
JSR add_two
```

At this point, the call stack is:

```
x4000: ???
x3FFF: R1
x3FFE: R0
x3FFD: ??? <- SP, saved for subroutine return val
```

2. The callee is first responsible for build-up.

```
; First push return address R7
ADD R6, R6, #-1
STR R7, R6, #0

; Then push old frame pointer
ADD R6, R6, #-1
STR R5, R6, #0

; Instantiate new frame pointer R5 NOW
; This has maximal effectiveness
; R5 + 1 is guaranteed to be old FP
; R5 + 2 is guaranteed to be return addr
; R5 + 3 is guaranteed to be subroutine return val
; R5 + 3 + i is guaranteed to be i-th arg
; R5 is guaranteed to be first local var
; R5 - i is guaranteed to be (i + 1)-th local var
ADD R5, R6, #-1

; Allocate 1 local variable
ADD R6, R6, #-1
```

At this point, we're still not done with build-up, but the stack looks like this:

```
x4000: ???
x3FFF: old R1, 2nd arg
x3FFE: old R0, 1st arg
x3FFD: ???, subroutine ret val
x3FFC: old R7, return addr
x3FFB: old R5, stack frame ptr
x3FFA: ???, local var 1 <- new R6, SP
```

- R5 <- X3FFA

3. Now we complete the build-up.

```
; LASTLY push old register states
ADD R6, R6, #-1
STR R4, R6, #0
ADD R6, R6, #-1
STR R3, R6, #0
ADD R6, R6, #-1
STR R2, R6, #0
ADD R6, R6, #-1
STR R1, R6, #0
ADD R6, R6, #-1
STR R0, R6, #0
```

The stack now looks like this:

```
x4000: ???
x3FFF: old R1, 2nd arg
x3FFE: old R0, 1st arg
x3FFD: ???, subroutine ret val
x3FFC: old R7, return addr
x3FFB: old R5, stack frame ptr
x3FFA: ???, local var 1
x3FF9: old R4
```

```
x3FF8: old R3
x3FF7: old R2
x3FF6: old R1
x3FF5: old R0 <- new R6, SP
```

4. Now that we finished build-up, we can write the actual operations. In this case, since we're just adding arg1 and arg2, there's no need to use extra storage space i.e., the local variable(s).

```
; Obtain the args in corresponding registers
; +1 is old FP, +2 is old ret addr, +3 is subroutine return val, +4 is 1st arg
LDR R0, R5, #4
LDR R1, R5, #5

; Compute result and store in stack
ADD R2, R0, R1
STR R2, R5, #3
```

5. After performing the actual subroutine operations, we tear down.

```
; Restore register states
LDR R0, R6, #0
ADD R6, R6, #1
LDR R1, R6, #0
ADD R6, R6, #1
LDR R2, R6, #0
ADD R6, R6, #1
LDR R3, R6, #0
ADD R6, R6, #1
LDR R4, R6, #0
ADD R6, R6, #1 ; This line is kinda optional

; Skip past all local variables
; No need to count because R5 is first local variable
ADD R6, R5, #1 ; currently R6 points at old frame ptr

; Restore R5 and R7
LDR R5, R6, #0
ADD R6, R6, #1
LDR R7, R6, #0
ADD R6, R6, #1

; Finally can return
RET
```

6. After the caller has called the subroutine, we clean up pushed args from call stack and use the returned value.

```
; PREVIOUSLY...
; Allocate one or more slots on the stack to store return value of the subroutine
ADD R6, R6, #-1
JSR add_two

; --------------------

; Obtain return value of subroutine
LDR R0, R6, #0
ADD R6, R6, #1 ; at this point R6 is ptr to arg2

; Clean up pushed args
ADD R6, R6, #2 ; now R6 should contain x4000 once more
```

4.3 Operating System Services

4.3.1 Program Discontinuity

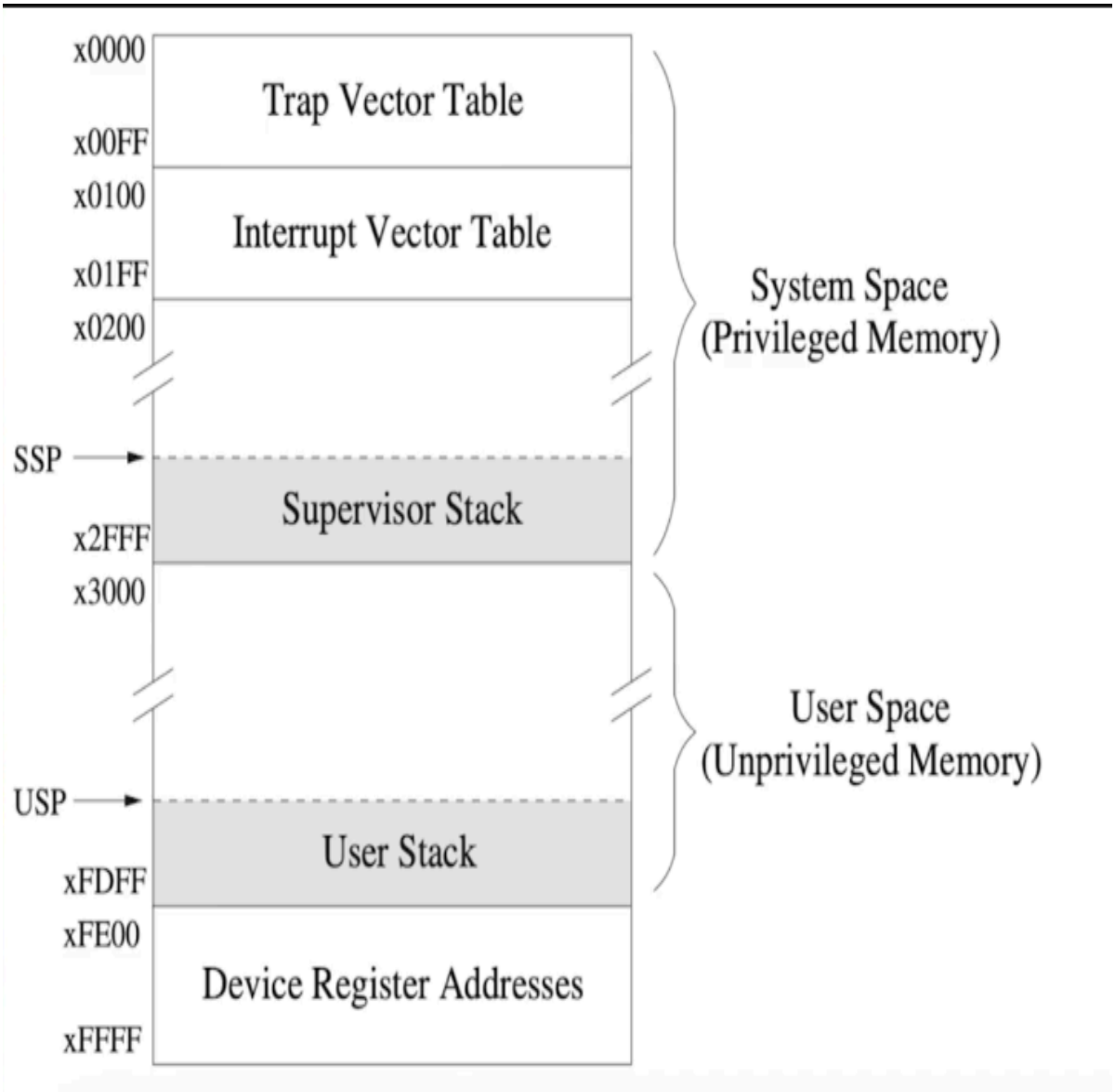
A program discontinuity is when the CPU stops following the typical “execute next instruction in sequence” flow and control jumps somewhere else for a few reasons: - Subroutine calls / TRAPs / operating system services: Program is responsible for this and this is expected behavior. - Exceptions: Program is responsible for this but this is often not expected / not desired. - Interrupts: Operating system needs to use particular resources used by the program which impacts operation of the program (not performed by program and also not planned/desired).

Subsequently, we will focus on the second type: TRAPs. A TRAP instruction is when a user program requests permission to perform a particular action from the operating system. Under the hood, a TRAP jumps to an OS subroutine that handles a particular action.

In LC-3, the types of TRAPs involve: - `TRAP x20` -> GETC: Read one character from the keyboard and store in R0. - `TRAP x21` -> PUTC/OUT: Print one character to the console by taking the ASCII value in R0. - `TRAP x22` -> PUTS: Print a string where R0 contains the address of the first character, terminating when it reaches `x0000` null terminator. - `TRAP x23` -> IN: Reads a character, stores that into R0, and echoes that character to the console (Basically GETC + PUTC). - `TRAP x25` -> HALT: Stops program execution.

4.3.2 A Review of LC3 Memory

From `x0000` to `x2FFF` is system space, which is reserved for the operating system and device I/O.



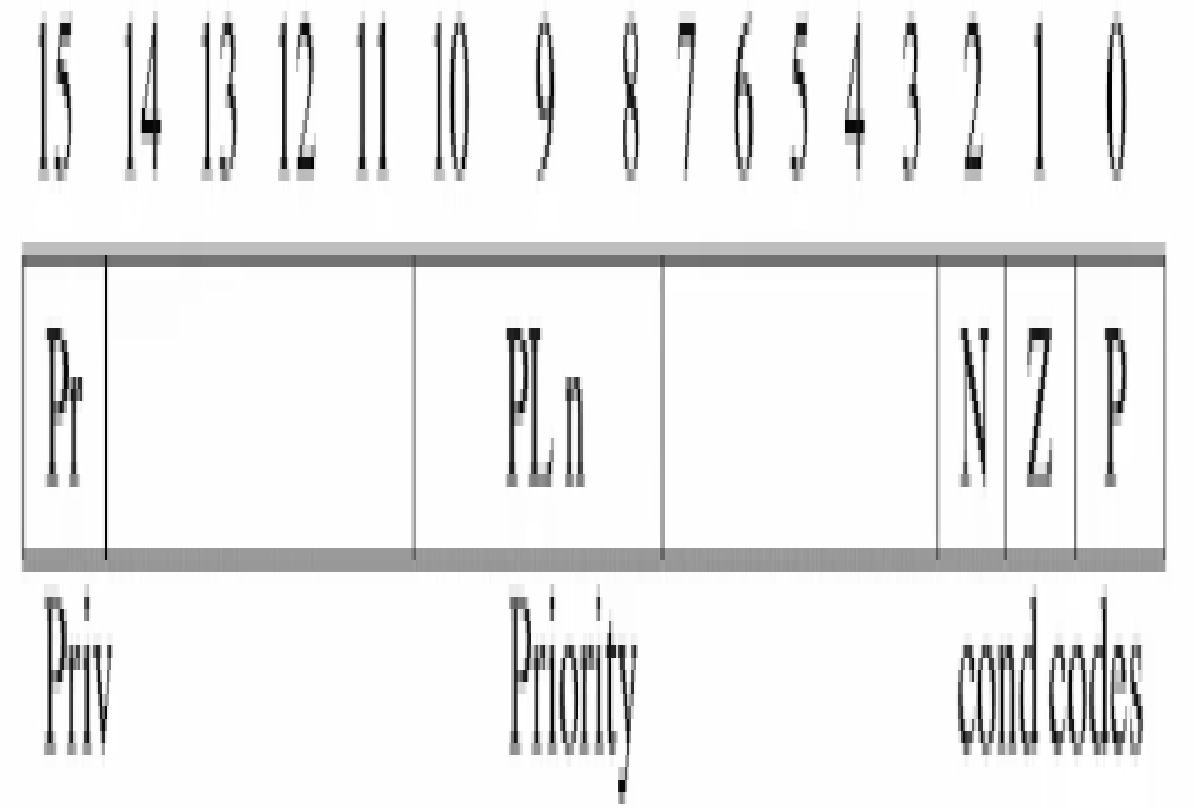
4.3.3 Program Execution: Processes

Recall that a program is like a series of instructions or a recipe to execute a particular task. A process is the running instance of a program, and a thread is a particular path of execution within the process.

There is a distinction between process identity (i.e., which program is running) and CPU privilege mode: user vs. supervisor. The majority of a user's program runs with the CPU in user mode, unless it makes a system call via a TRAP, takes an exception, or is interrupted. During those special execution stages, the CPU goes in supervisor mode.

4.3.4 Process Status Register

Each process has a special register called Process Status Register (PSR) that stores status info about the currently-executing context. For example, when a program is running in supervisor mode, the Priv Bit 15 switches to become 0.

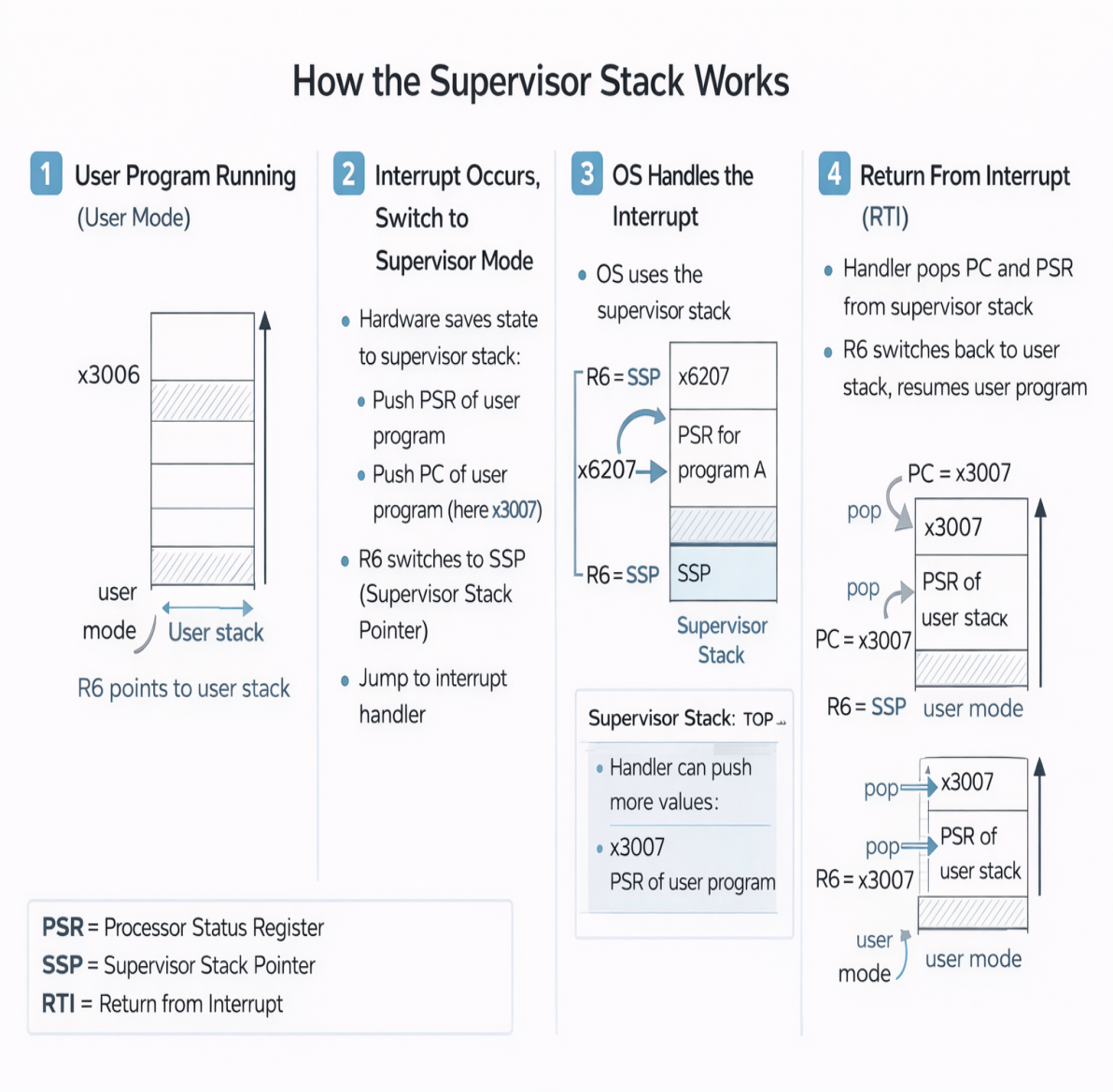


- Bit 15 indicates whether the machine is in user mode (restricted, 1) vs. supervisor mode (privileged OS mode, 0).
- Bits 10-8 represent an integer which represents the priority of the current process. Lower-priority interruptes cannot pause high-priority work.
- NZP is the same NZP as what is used for BR statements. It holds the current state of the program so it's easy to resume at any moment.

4.3.5 User and Supervisor Stacks (RTI)

The user stack is essentially the same thing as the call stack of the program. Whereas a supervisor stack is a separate stack used while the CPU is the supervisor mode.

First push old PSR, then push old PC (address ot next instruction to be executed). RTI restores the most recently saved PC and PSR to resume the interrupted code.



RTI stands for “Return from Trap or Interrupt” and requires supervisor privilege to execute, performing the action of returning from supervisor-level subroutines. It does 3 things: 1. Restores program counter using return address saved on supervisor stack. 2. Restore Processor Status Register from supervisor stack. 3. Swap back to user stack pointer if needed.

4.3.6 I/O Synchronization (Polling)

The CPU processor runs much more quickly than keyboard input (e.g., have to wait for keyboard press). This means that we need to ensure, for example, input is only read when it’s ready / when there is actually input, and display is also displaying the fresh/intended content.

To manage this, there are four new registers storing relevant information: - KBSR (Keyboard Status Register) - KBDR (Keyboard Data Register) - DSR (Display Status Register) - DDR (Display Data Register)

Each of these registers are 16-bit memory-mapped registers. This means that they are located at fixed addresses in memory address space. The CPU can use memory-style operations to access I/O; i.e. most addresses point to RAM but a few point to devices. - For example, if we say `LDI R1, KBDR_PTR` and `KBDR_PTR` is a label for `xFE02`, then we’re not reading from memory but rather the device.

Device Register	Addr
Keybd Status Reg	xFE00
Keybd Data Reg	xFE02
Display Status Reg	xFE04
Display Data Reg	xFE06

The program/OS will poll Keyboard Status Register (KBSR) until the "ready" bit says input is available. Then read Keyboard Data Register (KBDR). For display, we will poll DSR (Display Status Register) until the "ready" bit says display is free, then write to Display Data Register (DDR).

```
POLL
    LDI Rx, status_reg_addr
    BRzp POLL ; not ready (bit 15 indicates sign; 0 indicates not ready)
    LDI/STI Rx, data_reg_addr ; LDI if keybd, STI if display
    BRnzp NEXT_TASK

status_reg_addr .fill xFE__
data_reg_addr .fill xFE__
```

5 C Programming

5.1 Translating Between C and Assembly

5.1.1 Compilation Stages

C uses ahead-of-time compilation, so source code is translated into machine code before it ever runs. The transformation happens in 4 stages: 1. Preprocessing: A text-substitution engine runs to copy-paste in `#include` files, expand `#define` macros, and strip comments. The output is a `.i` text file called a translation unit. 2. Compilation: A compiler parses C syntax, checks types, resolves scopes, and runs optimizations. This emits assembly code `.s` for the target architecture. Each `.c` file is compiled independently. 3. Assembly: A translator turns human-readable assembly into binary object code `.o`, which is actual machine instructions but with some placeholders for symbols defined elsewhere. 4. Linking: Takes all of the compiled/assembled `.o` files and resolves cross-file references to produce the final executable.

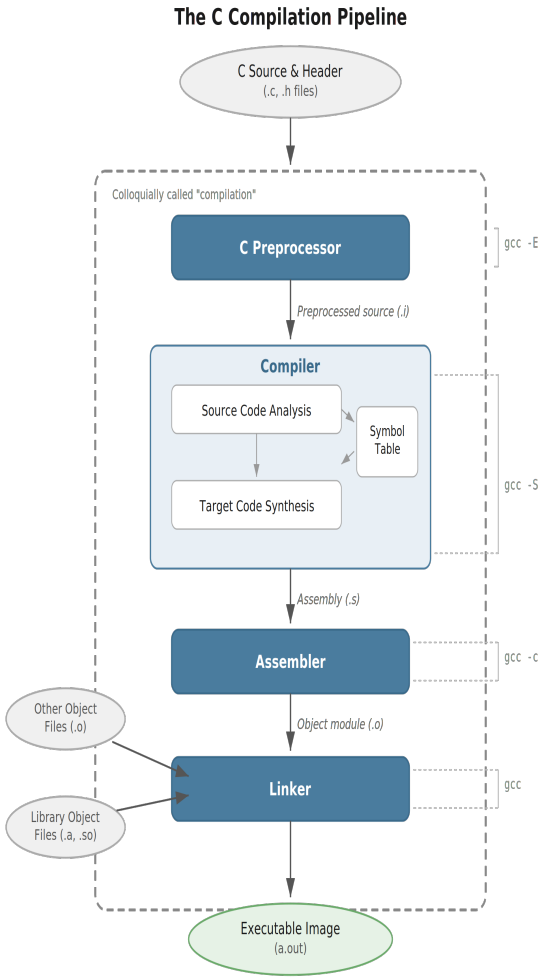


Figure: The C Compilation Pipeline

The dotted box encloses the entire process colloquially called "compilation," even though the compiler is only one stage. Inputs are C source/header files and library object files. The output is an executable image.

5.1.2 Compiling Code

After writing a program like the following:

```
// 01_hello_world.c
```

```c
#include <stdio.h>

int main() {
    printf("hello, world\n");
}
```

We compile it using the `gcc` command. To compile a specific file, run `gcc filename.c`, which generates a file called `a.out`. To name it something more informative, do:

```
gcc 01_hello_world.c -o hello_world
```

You may also use a path to specify it to a build directory, such as `build/hello_world`.

5.1.3 Builds and Makefiles

To make the process of going from source files to executable easier, there is a tool called make, which is a build automation tool. Instead of typing `gcc 01_hello_world.c -o build/01_hello_world` every time, we describe *rules* and *dependencies* and Make figures the rest out.

There are a few motivations for this: 1. Reproducible builds that work for others. Make also acts as a macro to avoid typing out compilation commands over and over again. 2. Make selectively recompiles portions of the program that have changed since the last build (incremental builds). This saves time for large projects. 3. A uniform interface abstracting away the individual recipe commands.

A rule consists of the following:

```
target: prerequisites
    recipe
```

For example, a Makefile performing `gcc 01_hello_world.c -o build/hello_world` would contain the following rule:

```
build/hello_world: 01_hello_world.c
    gcc 01_hello_world.c -o build/hello_world
```

5.1.4 The Preprocessor and Header Files

The preprocessor is the first stage of compilation. It does pure text substitution before the compiler ever sees the code, it doesn't understand C syntax, types, or scopes.

`#include` directives `#include` tells the preprocessor to literally paste the contents of another file into the current one. There are two forms that differ in search order: - `#include <filename.h>`: Used for system/standard library files like `<stdio.h>`. The preprocessor searches only system/standard library directories. - `#include "filename.h"`: Used for user-defined headers. The preprocessor searches the local directory first, then falls back to the system/stdlib directories.

`#define` macros Macros perform text replacement before compilation. They come in two flavors:

Constants

```c
#define PI 3.14
#define BUFSIZE 1024
```

Function-style

```c
#define MULT(a, b) ((a) * (b))
```

The aggressive parentheses in `MULT` matter a lot. Because the preprocessor is a dumb text substitutor, forgetting parentheses can silently change the semantics of your code.

Tricky: macro expansion pitfalls Consider this program, which looks like it should print `8`:

```c
#define SUB(a, b) (a - b)
#define MULT(x) (x * 2)

int main() {
    int result = MULT(SUB(5, 2 - 1));
    printf("%d", result);
    return 0;
}
```

After text substitution, `MULT(SUB(5, 2 - 1))` becomes `(SUB(5, 2 - 1) * 2)`, which expands to `((5 - 2 - 1) * 2) = (2 * 2) = 4`. The `2 - 1` argument is never parenthesized before the subtraction inside `SUB` fires, so precedence quietly eats you. This is exactly why the `MULT(a, b) ((a) * (b))` template wraps each argument and the whole expression.

Another classic: `#define SQUARE(x) x * x` applied to `SQUARE(1 + 2)` expands to `1 + 2 * 1 + 2 = 5`, not `9`. Always wrap both arguments and the full expansion.

NOTE: Assignment inside a macro also produces chaos. `#define foo = 5` followed by `foo = 4;` expands to `= 5 = 4;`, a syntax error. `#define` does not create a variable, it creates a text substitution, so the result is never assignable.

5.2 C Fundamentals

5.2.1 Data Types and Sizes

In C, there are only a few data types: - `char`: a single byte, storing one character in the character set. May be declared as either `signed` or `unsigned`. On a MacBook, the default is signed, so values range from -128 to 127. - `short`: 2-byte signed integer. - `int`: 4-byte signed integer. - `long`: 8-byte signed integer. `long long` is the same thing but guaranteed across platforms. - `float`: 4-byte floating-point number with ~7 decimal places of precision. - `double`: 8-byte floating-point number with ~15 decimal places of precision.

When we say "precision," that means you can reliably store a value with that many decimal places (converted to binary when stored) and recover the same value up to said decimal place. Any further beyond that and it may not round-trip.

5.2.2 Declarations and Definitions

A C declaration is how an identifier (variable) is introduced in a program. There are 3 pieces to each C declaration: storage class specifier, base type, and declarators.

When we have a statement like the following:

```c
static volatile long int i, *j, k[10];
```

- `static` is the storage class specifier, indicating linkage and lifetime.
- `volatile long int` is the base type.
- `i, *j, k[10]` are the declarators.

In C, `*` and `[]` belong to individual declarators. So the above initializes `i`, a `volatile long int`; `j`, a pointer to a `volatile long int`; and `k`, an array of 10 `volatile long int` values. All of these are `static`.

When reading a declarator, move from right to left, inside-out, prioritizing parentheses:

```c
// First, look within the (**f). This is a pointer to a pointer.
// Then step outside those parentheses. It's a pointer to a pointer of a function which returns a pointer
    to an int.
int *(**f) ()
```

Variable declaration and initialization in practice:

```c
// Declaration then initialization
int x;
x = 5;

// Both
int y = 10;

// Notice that only b is initialized to a value.
// a and c are random values (whatever's at that memory location).
int a, b = 5, c;
```

5.2.3 Arithmetic and Logical Operators

C distinguishes between logical and bitwise variants: - `&&` and `||` are logical AND and OR. - `&` and `|` are bitwise AND and OR. - `>>` and `<<` are right and left bit shifts. Right-shifting by `k` divides by 2^k (for non-negative values).

Truthy and Falsy Values Traditionally, C had no boolean type. Anything evaluating to 0 is false, and all else is true. There is no null value except `NULL` for pointers. Variables that are declared but not initialized hold random values (undefined behavior if read).

Tricky: assignment returns the assigned value, so a statement like `if (u = 5) {}` would always run because `(u = 5)` evaluates to `5`, which is truthy. This is a common bug, the intended code was `if (u == 5)`.

Tricky: evaluating C expressions Given `int x = 3; int y = 6;`, work through these carefully: - `(x < y)` evaluates to `1` (true). - `(x == y) ? 10 : 20` evaluates to `20` because `x != y`. - `(y & x)` is bitwise AND: `0b110 & 0b011 = 0b010 = 2`. - `(y >> 1)` shifts `6` right by 1, giving `3` (integer division by 2). - `(y++) + (--x)` evaluates to `6 + 2 = 8`. `y++` is post-increment (yields the old value, then increments), while `--x` is pre-decrement (decrements first, then yields the new value). After the expression, `y == 7` and `x == 2`.

5.3 Pointers

5.3.1 Definition and Operators

Pointers are variables that store memory addresses. A pointer to an integer is a variable whose contents are the address of some integer in memory.

Two core operators: - `&` (address-of): Returns the memory address of its operand. - `*` (dereference / declaration): In a declaration, marks the variable as a pointer. Everywhere else, it's the dereference operator, going to the address stored in the pointer and accessing the value there.

```c
int num = 10;
int *ptr = &num; // & returns the memory address of its operand

printf("ptr     = %p\n", ptr);     // e.g., 0x1000
printf("ptr + 1 = %p\n", ptr + 1); // 0x1004 (advances by sizeof(int) = 4 bytes)

double d = 3.14;
double *dp = &d;

printf("dp      = %p\n", dp);      // e.g., 0x2000
printf("dp + 1  = %p\n", dp + 1);  // 0x2008 (advances by sizeof(double) = 8 bytes)

char c = 'A';
char *cp = &c;

printf("cp      = %p\n", cp);      // e.g., 0x3000
printf("cp + 1  = %p\n", cp + 1);  // 0x3001 (advances by sizeof(char) = 1 byte)
```

NOTE: `&my_var` does *not* evaluate to the data stored inside `my_var`, it evaluates to the address of `my_var`. The value itself is `my_var` (or `*(&my_var)` if you want to round-trip through a pointer).

Recall that `&` obtains the memory address of any lvalue (variable, array, struct member). It does NOT work for rvalues, which have no storage location (literals, arbitrary expression results).

5.3.2 Pointer Dereferencing

Dereferencing `*ptr` goes to the address stored in `ptr` and reads the value there:

```
int num = 10;
int *ptr = &num;     // ptr stores the address of num
int num_val = *ptr;  // num_val stores value at address of num
```

This is equally powerful for writes; dereferencing on the left side of = changes the value at that address:

```
*ptr = 42;             // Go to addr stored in ptr and change the value there to 42
printf("%d\n", num);   // 42
```

It also works for multiple levels of indirection:

```
int num = 10;
int *ptr = &num;
int **dptr = &ptr;   // pointer to a pointer

**dptr = 99;          // follow dptr to ptr, follow ptr to num, write 99 there
printf("%d\n", num);  // 99

*dptr = &some_other_int;  // follow dptr to ptr, overwrite ptr itself with a new address

// If we did *dptr = 1 instead (after the reassignment), some_other_int becomes 1
```

5.3.3 Pass by Reference via Pointers

C is a pass-by-value language. Arguments passed to a function are copied into the function's local parameter slots. Modifying a parameter inside a function cannot affect the caller's variable.

To simulate pass-by-reference, pass a pointer to the variable. Inside the function, dereference the pointer to reach the caller's memory:

```
// Expects a pointer (memory address)
void addTwo(int *num) {
    *num = *num + 2;  // Dereference to modify the caller's value
}

int main() {
    int x = 5;
    addTwo(&x);           // Pass the ADDRESS of x. Passing just 'x' would be a type error.
    printf("%d", x);      // Prints 7
    return 0;
}
```

Tricky: passing a value to a function that expects a pointer. If you forget the `&`, the compiler will complain (type mismatch) or, if types happen to line up (e.g., passing an integer where a pointer is expected with a cast), the program will try to dereference a garbage address and segfault.

Worked example: counter reset. To reset an `int` to 0 inside a helper, the helper must receive an `int *`:

```c
void reset_counter(int *c) {
    *c = 0;
}

int main() {
    int my_counter = 10;
    reset_counter(&my_counter);   // pass the address
    return 0;
}
```

Writing `reset_counter(my_counter)` passes the *value* `10`, which cannot be written back.

Worked example: swap. Any swap function in C must take pointers, otherwise it swaps local copies that die at `return`:

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5;
    int y = 10;
    swap(&x, &y);
}
```

5.3.4 Pointers vs Arrays

The value stored in a pointer (a memory address) can change. But the memory address of a pointer itself is a different thing from its contents: `&ptr != ptr`.

```c
int arr[3] = {10, 20, 30};
int *ptr = arr;
```

```
Stack address   Name     Contents
-----------     ----     --------
0x100           arr[0]   10          ┐
0x104           arr[1]   20          │ arr IS these 12 bytes. nothing else.
0x108           arr[2]   30          ┘
0x110           ptr      0x100       <- ptr is a SEPARATE 8-byte variable holding an address
```

Reassigning `ptr` just changes the value stored at `0x110`:

```c
int other[3] = {40, 50, 60};
ptr = other;   // change the value stored at 0x110 from 0x100 to other's address
```

HOWEVER, something like:

```c
arr = other;   // ILLEGAL
```

will cause a compile-time error. Even though arrays *act* like a pointer to a memory address, it is better to treat them as a literal memory address; literals cannot be changed/reassigned. `arr` is a non-modifiable lvalue.

5.3.5 The Danger of Dereferencing: Const Overwrites

Consider an array of constant integers:

```
const int arr[3] = {10, 20, 30};
```

If we directly do `arr[0] = 30`, the compiler throws an error because each int at its location is `const`; the value at the memory address of `arr[0]` should not be changed.

HOWEVER, if we instead do:

```
int *ptr = arr;
*ptr = 99;    // Compiles with a warning and runs
```

When we wrote `int *ptr = arr`, `arr` decays to `const int *` (pointer to constant integers), and then assigning this to `int *` (pointer to integer) performs a cast that drops the const-ness. The write technically works, but this is undefined behavior depending on compiler optimizations and where `arr` lives in memory.

5.3.6 Common Mistakes

- Operator Precedence

5.4 Arrays and Strings

5.4.1 Arrays

An array is a fixed-size sequence of identical types, stored contiguously in memory. The type must be identical because different types have different sizes, and traversing memory requires knowing how far to jump each step.

Initialization

```
int arr[3] = {1, 2, 3};
int arr[] = {1, 2, 3};  // size inferred from initializer

int zeroed[5];          // at file scope: zero-initialized
void f(void) {
    int stack_arr[5];       // uninitialized -> random values on the stack
    static int s_arr[5];    // zero-initialized (static storage)
}
```

When an array is partially initialized but not all elements are covered, the remaining elements are zero-initialized regardless of scope:

```
void f(void) {
    int arr[5] = {1, 2};   // arr = {1, 2, 0, 0, 0}
}
```

Pointer decay and element access Array names decay into pointers to the first element in almost every expression. So `arr` (as a pointer) is equivalent to `&arr[0]`, and `arr[i]` is exactly equivalent to `*(arr + i)` (take address of first element, step `i` times by the element size, dereference).

Because of decay, arrays are effectively pass-by-reference when handed to a function:

```
void modifyArray(int my_arr[]) {
    my_arr[0] = 99;   // modifies the original array!
}

int main() {
    int arr[3] = {1, 2, 3};
    modifyArray(arr);     // decays to int*, modifyArray writes to caller's memory
}
```

Tricky: **`sizeof`** is the one place decay does NOT happen. Inside the same scope where the array was declared, `sizeof(arr)` returns the total bytes (`num_elements * sizeof(element)`). But the moment the array is passed to a function as `int arr[]` (or `int *arr`), it has already decayed to a pointer, and `sizeof(arr)` inside the function returns `sizeof(int *)` (the pointer size, typically 8 bytes on a 64-bit machine). So given a pointer `char *p` pointing to the same string as a local `char s[]`, `sizeof(s)` and `sizeof(p)` are different.

Tricky: **`&arr`** vs **`&arr[0]`**. For `int arr[10]`: - `&arr[0]` has type `int *` (pointer to one int), so incrementing it advances by `sizeof(int)`. - `&arr` has type `int (*)[10]` (pointer to an array of 10 ints), so incrementing it advances by `10 * sizeof(int)`. Both hold the same numeric address, but pointer arithmetic on them does completely different things.

5.4.2 Arrays Are Non-Modifiable Lvalues

Arrays are effectively constant pointers. Several operations that look reasonable are forbidden: - `arr++` is illegal (you cannot reassign an array name). - `arr = other_arr` is illegal (no wholesale array assignment). - `arr = {0, 1, 2, 3}` is illegal *after* declaration. The curly-brace syntax is only valid at the point of declaration.

After declaration, elements can only be assigned one at a time via bracket access (which is pointer arithmetic + dereference under the hood).

C performs zero runtime bounds checking on array access. Writing past the end of an array (a buffer overflow) silently corrupts adjacent memory and is one of the most common sources of security vulnerabilities in C. The compiler does NOT catch this.

5.4.3 Strings

Strings in C are character arrays ending with a null terminator (`'\0'`). There are three common ways to create one:

```
// 1. Explicit character array with manual null terminator
char s1[] = {'M', 'y', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0'};

// 2. String-literal initializer for a stack array
char s2[] = "My string";

// 3. Pointer to a string literal (lives in read-only memory)
char *s3 = "My String";
```

- The integer 0 corresponds to ascii value for null terminator '\0'.

Curly-brace and literal-into-array initialization (forms 1 and 2): - Normal array on the stack, mutable. - Form 1 requires you to manually include `'\0'`. - Form 2 automatically appends `'\0'`.

Literal-pointer initialization (form 3): - The string literal lives in read-only (`.rodata`) memory. - `'\0'` is automatically appended. - Attempting to write through the pointer is undefined behavior, typically a segfault.

```
char s1[] = "hello";
char *s2 = "hi";

s1[0] = 'H';  // fine: s1 is a mutable stack array
s2[0] = 'H';  // SEGFAULT: "hi" lives in read-only memory
```

Tricky: fixed-size arrays don't accept string-literal assignment after declaration.

```
char s[10];
s = "hello";        // illegal
strcpy(s, "hello"); // use strcpy instead
```

The string-literal form only works at the point of declaration.

5.4.4 Common String/Array Functions

Know the distinction between the following three:

`sizeof(x)`: A compile-time operator returning the total size in bytes of `x`. For arrays (in the scope where they were declared), this is the full array size including the null terminator. For pointers, it's just the pointer size.

`strlen(s)`: A runtime function that walks the string counting characters until it hits `'\0'`. Excludes **`'\0'`** from the count. If the string lacks a null terminator, `strlen` reads past the end of the buffer (undefined behavior).

`strcpy(dest, src)`: Copies bytes from `src` to `dest` until it hits `'\0'` in `src`. Does not check if `dest` has enough room, buffer overflows are on you. Requires `src` to be null-terminated.

Worked example:

```
char str1[] = {'h', 'i', '\0'};
char *str2 = "hi";

sizeof(str1);  // 3: the array occupies 3 bytes ('h', 'i', '\0')
strlen(str2);  // 2: 'h' and 'i', '\0' excluded
```

`sizeof(str2)` would be `8` (or whatever the pointer size is), not `3`, because `str2` is a pointer, not an array.

Tricky: **`strcpy`** with a non-null-terminated source.

```
char source[] = {'c', 'a', 't'};   // no '\0' at the end!
char dest[10];
strcpy(dest, source);              // UB: strcpy walks past 't' looking for '\0'
```

The curly-brace initializer does NOT add a null terminator. `strcpy` reads garbage past the end of `source` until it happens to hit a zero byte, copying all of it into `dest`. This is a classic undefined-behavior bug.

5.4.5 Shorthand **`strcpy`**

A very compact way to copy a string — basically what `strcpy` does under the hood:

```
char local_s[10];
char *p = ip_s, *q = local_s;
while (*q++ = *p++);
```

Three things going on:

1. `a = b` is an expression, and it evaluates to `b`. So `*q = *p` returns the character that just got copied, and `while` checks that character.
2. `*p++` means `*(p++)`: read the char at `p`'s current address, then bump `p` forward. Same on the `*q++` side — write, then bump.
3. `'\0'` is 0, which is false. Once we copy the null terminator, the whole expression is 0 and the loop stops. The null terminator gets copied first, then detected, so `local_s` ends up properly terminated.

Trace for `p` pointing to `"hi"`:

Step	read `*p`	write `*q`	both pointers advance	loop value	continue?
1	`'h'`	`'h'`	yes	`'h'`	yes
2	`'i'`	`'i'`	yes	`'i'`	yes
3	`'\0'`	`'\0'`	yes	`0`	stop

The loop body is empty — the `;` right after `while (...)` *is* the body. Everything happens in the condition.

Gotchas: - `*p++` is NOT `(*p)++`. The `++` attaches to `p`, not to the char. `(*p)++` would increment the character itself. - It's `=`, not `==`. The single `=` is intentional — that's how we both copy and test. gcc/clang will warn about this; the usual fix is extra parens: `while ((*q++ = *p++))`. - No bounds check on `q`. If the source is longer than the destination buffer, same overflow as `strcpy`. - Source must be null-terminated, or the loop runs off the end.

5.4.6 Common Mistakes

1. Arrays of chars are modifiable via pointer arithmetic. Pointers to string literals are not.
   - `char str[] = "hello"` and then `*(str + 1) = 'c'` is allowed because "hello" is copied over onto the stack/global data from the read-only portion of memory.
   - `char *str = "hello"` and then `*(str + 1) = 'c'` would result in undefined behavior (often a segfault error) because we have a pointer that points directly to read-only memory, which we cannot modify.
2. Arrays are non-modifiable lvalues
   - `char str[5]` and then `str = "hello"` would ALSO result in an error since they cannot appear on the left side of `=` directly (non-modifiable). The only time arrays can be assigned values directly instead of using pointer arithmetic is during initialization.
3. Char arrays need extra space for null terminator
   - Even if `strlen(s) == k` for some char array `s`, the actual amount of space for `s` would have to be at least `k + 1` with a null character at index `s[k]`.
4. Sizeof behavior changes for string variables of type array vs. pointer
   - `char str1[] = "hi"` and `char *str2 = "hi"` have different `sizeof` values: `sizeof(str1) == 3` (includes null terminator) while `sizeof(str2) == 8` (size of pointer).
   - Arrays have a fixed size upon declaration. So if we said `str[10] = "hi"`, `sizeof(str) == 10` and not 3. In this scenario, since it's initialized from a string literal, the rest of the spaces are zero-initialized (null terminator char).

5.5 Structs and Typedefs

5.5.1 Struct Basics

A struct groups fields of different types into a single object. This is the primary mechanism for building data structures in C.

```
struct Point {
    double x;    // fields are public by default
    double y;    // use dot notation to access/modify fields
};

struct Point p = {1.0, 2.0};       // positional initialization
printf("Point p is at (%f, %f)\n", p.x, p.y);
```

- At the local scope, if it is declared but not initialized, values are garbage.

5.5.2 Accessing Struct Members

Two operators, depending on whether you have a struct value or a pointer to one: - Dot operator `.`: For struct values. `myDog.age = 2;` - Arrow operator `->`: For pointers to structs. `my_dog_ptr->age = 2;`

`my_dog_ptr->name` is just shorthand for `(*my_dog_ptr).name`.

```
struct Dog {
    char *name;
    int age;
};

int main() {
    struct Dog myDog = {"Rufus", 2};
```

```
    printf("Age: %d", myDog.age);         // dot: direct access

    struct Dog *my_dog_ptr = &myDog;
    my_dog_ptr->name = "Spot";            // arrow: through a pointer
    // equivalent to (*my_dog_ptr).name = "Spot";

    return 0;
}
```

Tricky: operator precedence. The dot operator has among the highest precedence in C, higher than `*`. That's why `(*my_dog_ptr).name` needs parentheses, without them, `*my_dog_ptr.name` would be parsed as `*(my_dog_ptr.name)`, which tries to apply `.name` to the pointer itself (a type error).

Initializing a struct through a heap pointer:

```
struct Player *p = malloc(sizeof(struct Player));
p->score = 10;            // arrow operator
(*p).score = 10;          // equivalent, using dot through explicit dereference
```

Both lines above are valid ways to assign to the `score` field.

5.5.3 Struct Padding (Alignment)

In structs, fields are laid out in declaration order with possible padding inserted between them.

Because the CPU reads memory in chunks, most architectures require a data type's address to be evenly divisible by that data type's size. However, structs may also be stored in a contiguous array. This introduces the need for padding, which guarantees that each field within a struct, regardless of whether multiple structs are packed together in continuous memory, lies in a memory address divisible by its size.

For example, for the following struct:

```
typedef struct {
    char *z; // alignment=8
    char a[10]; // alignment=1
} i
```

The char pointer `z` needs to be at an address divisible by 8, while the array of chars `a` can pretty much be anywhere (`sizeof(char) == 1`).

This means that there needs to be padding at the end of `a` of size 6, making `sizeof(i) == 24`.

The general rule is that the struct is padded to a multiple of its strictest alignment size.

IMPORTANT: struct size is implementation-defined. The C standard does not fix the alignment of basic types, nor does it forbid a compiler from inserting extra padding beyond the minimum. All the standard guarantees is: - The first member is at offset 0. - Members appear in declaration order, each properly aligned. - `sizeof(struct X)` is at least the sum of its members' sizes.

So the only thing you can state about a struct's size without running the code is the lower bound. The exact value depends on the platform's alignment rules and the compiler's choices. For this reason, always use **`sizeof(struct X)`**, never hand-computed numbers, when calling `malloc` or doing any memory-level reasoning.

5.5.4 Typedef

`typedef` creates aliases for existing types. Most commonly used with structs to avoid repeating `struct` everywhere:

```
typedef struct {
    double x;
    double y;
} Point;

Point p = {1.0, 2.0};   // no "struct" keyword needed
```

5.5.5 Modeling a Struct: Worked Example

Consider modeling a Quiz with these fields: - Quiz number (integer) - Version (a single character) - Score (not necessarily an integer, so floating-point) - Student name (any length, so we use a pointer to a string)

```
struct Quiz {
    int number;
    char version;
    double score;
    char *student_name;
};

struct Quiz q4 = {4, 'A', 100.0, "Review Session Attendee"};
```

The pointer-to-string choice for `student_name` is what handles arbitrary-length names; a fixed-size `char name[N]` would cap the length.

5.6 Union

5.6.1 Definition and Properties

A union is a data structure where all members share the same data storage. Its size is equivalent to the padded value of its largest member.

```
struct S { int a; char b; double c; };  // sizeof ~ 24 (padding between b and c)
union  U { int a; char b; double c; };  // sizeof == 8 (the double's size)
```

To define a union object, we can do one of the following:

```
// Form 1: tag name only. Must use "union point_un" everywhere.
union point_un {
    int  x;
    char x_tag[5];
    int  y;
    char y_tag[3];
};

// Form 2: typedef. Now you can just write Point.
typedef union {
    int  x;
    char x_tag[5];
    int  y;
    char y_tag[3];
} Point;

// Form 3: both (idiomatic, lets you forward-declare).
typedef union point_un {
    int  x;
    char x_tag[5];
    int  y;
```

```
    char y_tag[3];
} Point;
```

- The 3rd form is useful for if we need to reference the type itself in one of the inner fields. For example, consider defining a `Node` struct with a `Node * next` field.
- If we did `typedef struct Node {...} Node;` then we would be able to do that, vs. if we only did `typedef struct {...} Node`.

5.6.2 Initialization

5.6.2.1 Declaration + Initialization

The first way to initialize is default initialization:

```
union point_un p = { 42 };   // initializes p.x = 42
```

- The remainder of the shared bytes are zero-initialized.

The other way is to – unlike structs – pick out a specific field using dot notation to assign a value to. One union should get one thing assigned at a time; if assigning to the same union but a different field, previous assignments will get overwritten.

```
union point_un p = { .x = 42 };
union point_un q = { .x_tag = "buzz" };
union point_un r = { .y_tag = "gt" };
```

- Performing `p = { .x = 42 };` along will result in a syntax error.

5.6.2.2 Reassignment After Declaration

Normally, we can write directly to the field that we want to change:

```
p.x = 4; // this is ok
```

- Changes only the bytes for `.x`, leaving the other bytes untouched.

The other alternative is to use a compound literal `(type) {...}`, which allows us to use a brace list a type and turn it into an expression that can be assigned as an rvalue.

```
p = (union point_un){ .x = 4 };
```

- Zeros out the rest of the bytes after assignment. Basically creates a fresh union from scratch and then assigns to `p`.

5.6.3 Motivation

Beyond using less memory than structs, a union's true purpose is to express that a particular variable/value may have multiple different possible types.

Due to the limitation of packing different types/values into a single area in memory, we "tag" unions by packing it within a struct along with an enum describing which "type" is active.

5.6.4 Tagging Unions

Say that we have an object, and depending on the type of shape it is we want to contain different information about it.

```
typedef enum {
    SHAPE_CIRCLE,
    SHAPE_SQUARE
} ShapeKind;

typedef union {
    double radius;
    double side_length;
```

```
} ShapeData;

typedef struct {
    ShapeKind kind; // enum
    ShapeData data; // union
} Shape;

// ---- Functions ----

double area(Shape s) {
    switch (s.kind) {
        case SHAPE_CIRCLE: return 3.14159 * s.data.radius * s.data.radius;
        case SHAPE_SQUARE: return s.data.side_length * s.data.side_length;
    }
    return 0.0;  // unreachable if all cases handled
}
```

5.7 Storage Classes, Type Qualifiers, and Scope

5.7.1 Memory Regions

A running C program's memory is divided into distinct regions. You must be able to look at a variable and say which region its data lives in.

Code (text): Executable machine instructions, including all function bodies. Read-only.

Data: Global, static, and string-literal storage that persists for the program's lifetime. Split into two sub-regions: - Read-only data (`.rodata`): String literals (like `"hello"` in `char *p = "hello"`), `const` globals. - Read/write data (`.data` / `.bss`): Global variables, `static` variables, whether inside or outside a function.

Stack: Local variables and function arguments. A new stack frame is pushed on every function call and popped on return.

Heap: Dynamically allocated memory (`malloc`, `calloc`, `realloc`). Persists until explicitly `free`d.

Worked example: classify every item.

```
#include <stdlib.h>

extern int global_counter;      // 1: data (read/write), actual definition in another file
static int file_limit = 500;    // 2: data (read/write), file-scoped
int max_score = 100;            // 3: data (read/write), global
const double PI = 3.14159;      // 4: data (read-only)

// 5: code region, the executable instructions for calculate_sum()
void calculate_sum() {
    static int sum_calls = 0;        // 6: data (read/write), persists across calls
    char local_str[] = "Hello";      // 7: stack (local array copy of the literal)
                                     // 8: "Hello" literal itself is in data (read-only),
                                     //    even though it was copied into a stack array
}

int main() {
    int index = 0;                   // 9: stack (local variable)
    char *msg = "Welcome!";
    int *buffer = malloc(20);        // 10: heap (the 20 bytes pointed to by buffer)
    calculate_sum();
    free(buffer);
    return 0;
}
```

Key subtleties: `sum_calls` is a local variable by syntax but `static` lifts its storage into the data segment. `local_str[]` copies the literal `"Hello"` into a stack array, but the literal itself originally lived in read-only data. `buffer` itself is a pointer on the stack (8 bytes), but the 20 bytes it points to are on the heap.

5.7.2 Storage Classes

Storage classes determine where an object lives, how long it lives, and who can see it. Four keywords:

`auto`: The default for local variables. Object is created when execution enters a block and destroyed when it leaves. You almost never write this out; `auto int x = 5;` is the same as `int x = 5;` inside a function.

`register`: Hints to the compiler that this variable is used frequently and should be kept in a CPU register. Same storage duration as `auto`; the only strict rule is that you cannot write `&x` for a `register` variable, because it might not have a memory address.

`static`: Two different effects depending on where it's applied. - On a local variable, it extends the variable's lifetime to the entire program run. The variable is only initialized once and retains its value between calls. It does NOT live on the stack; it lives in the data region. - On a file-scope variable or function, it changes linkage from external to internal, so the object is only visible within that same translation unit (`.c` file).

`extern`: This is used in files when we want to use a global variable which is defined in a different file. It basically tells the compiler to trust the user and use a placeholder which will get resolved during the linking stage. - If the linker cannot resolve it there is a linker error. - `extern` needs to refer to a global / file-scoped object. But `extern` itself can be used within a locally-scoped function in the file using it. - Top-level functions (all C functions are file-scoped because there are no nested functions) have external linkage enabled by default.

NOTE: File-scope variables without an initializer get a default zero-initialization. Block-scoped variables without an initializer contain garbage (undefined behavior to read).

5.7.3 Type Qualifiers

Type qualifiers constrain how variable values can be accessed: - **`const`**: The object's value cannot be modified through the variable name. Note that a `const`-qualified variable cannot be used where a *constant expression* is required (such as an array size in some contexts). - **`volatile`**: Normally, if we read a variable multiple times with no intervening writes, the compiler assumes the value hasn't changed and reuses the cached value in a register. `volatile` disables this optimization, telling the compiler the value may change for reasons it can't see (hardware registers, signal handlers, other threads). - **`restrict`**: Qualifies pointers only. A promise that no other pointer in the current scope accesses the same memory, enabling vectorization optimizations. - **`_Atomic`**: Reads/writes on this variable happen as indivisible units, avoiding race conditions between threads. Out of scope for this class.

The most notable in practice are `const` and `static`.

More on **`static`** locals. A static local variable persists for the lifetime of the program regardless of being defined inside a function. This means: - It is initialized exactly once. - It retains its value between calls. - If initialized, it must be initialized to a compile-time constant expression (a literal or a constant expression over literals). At compile time, the bytes for that constant are placed in the data segment. - If declared but not initialized, it is zero-initialized by the compiler.

Tricky: **`static`** inside vs. outside a function.

```
static int x = 0;     // outside any function: file-scope, visible throughout this .c file, lifetime of
    program
void f(void) {
    static int x = 0; // inside a function: visible only inside f, but lifetime of program
}
```

- Outside a function: restricts *visibility* from global (cross-file) to file-scope; lifetime is already the whole program (as it would be for any file-scope variable).
- Inside a function: keeps visibility at function-scope, but lifts *lifetime* from the stack frame to the whole program.

5.7.4 Global Scope and Linkage

Any top-level variable, by default, has external linkage and is visible to other files. Other files access it using `extern`.

```c
// ---- file1.c ----
int count = 42;           // definition: allocates storage, initializes to 42
                          // external linkage (default for file-scope)

void increment(void) {
    count++;
}
```

```c
// ---- file2.c ----
#include <stdio.h>

extern int count;         // declaration only: no storage allocated
                          // "count exists elsewhere, it's an int"

void print_count(void) {
    printf("%d\n", count); // uses the same count from file1.c
}
```

```c
// ---- main.c ----
extern int count;         // same declaration
void increment(void);
void print_count(void);

int main(void) {
    print_count();        // 42
    increment();
    print_count();        // 43
    count = 100;          // direct access works too
    print_count();        // 100
    return 0;
}
```

Tentative definition rule. If we write `int count;` at file scope with no initializer and no `extern` specifier, the compiler treats it as a *tentative definition*. It waits through the rest of the file for a "real" definition. If none appears by end of translation unit, the tentative definition is promoted to a zero-initialized definition.

When using functions defined in a separate file, you can add `extern` before the prototype, but it's the default behavior for functions. `extern` is primarily needed on variables because of the tentative-definition rule.

```c
// utils.c
static int internal_func(int x) {
    return x * 2;
}
int public_func(int x) {
    return x + 2;
}
```

```c
// main.c — function prototype at top, equivalent to including a "utils.h"
int public_func(int);      // parameter names are optional
int public_func(int x);    // including can be helpful for readers
int public_func(int num);  // parameter name need not match the definition's
```

5.7.5 Local Scope and Shadowing

Any pair of enclosing `{}` creates a local scope block, not just within a function. Storage is allocated on the stack when execution enters the block and deallocated when execution leaves.

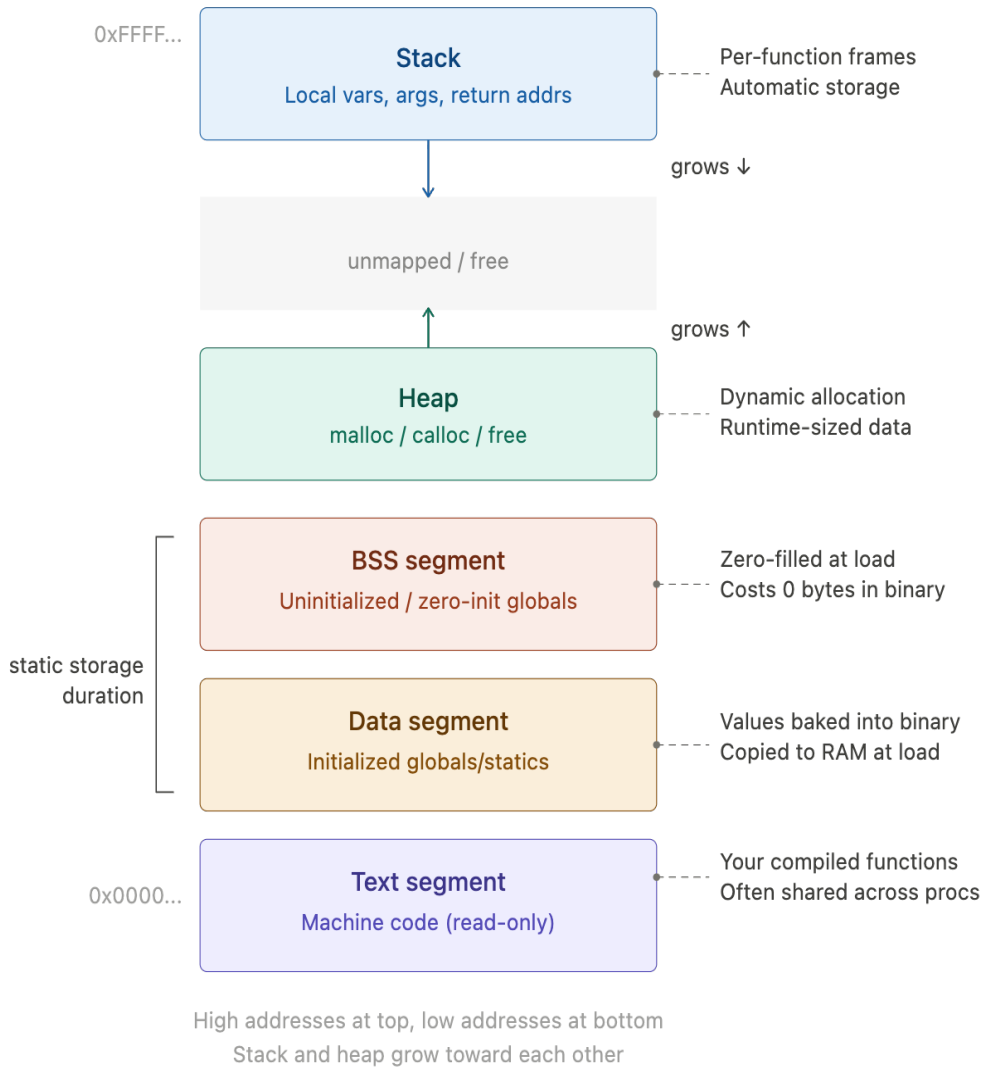
When an inner scope declares a variable with the same name as an outer scope, the inner shadows the outer. The outer variable is untouched; it just becomes unreachable by name for the duration of the inner scope.

```
void shadow_demo(void) {
    int x = 10;
    printf("%d\n", x);       // 10

    {
        int x = 99;          // new x, shadows outer x
        printf("%d\n", x);   // 99
    }

    printf("%d\n", x);       // 10 — outer x was never touched
}
```

The same principle applies to function parameters and to variables declared in `for`-loop headers.

NOTE: Shadowing only kicks in at the point where the inner variable is *declared*. Before that point, the outer variable is still in scope.

```
int i = 0;                              // (A) global

int main(int argc, char *argv[]) {
    i = 1;                              // (A) global -> now 1
    int i = 0;                          // (B) main-local, shadows A
    i = 2;                              // (B) -> now 2
    for (int i = 0; i < 10; i++) {      // (C) for-loop, shadows B
        f(i);                           // (C) -> prints 0, 1, 2, ... per iteration
        int i = 2;                      // (D) inner block, shadows C
        f(i);                           // (D) -> always prints 2
    }
    f(i);                               // (B) -> prints 2
}
```



5.8 Dynamic Memory

5.8.1 malloc and calloc

The heap holds memory that outlives the scope in which it was allocated, which is essential for building data structures whose size isn't known at compile time.

`void *malloc(size_t size)`: Allocates `size` uninitialized bytes on the heap and returns a `void *` to the first byte. The bytes contain whatever garbage happened to be there.

`void *calloc(size_t num, size_t size)`: Allocates `num * size` bytes on the heap and zero-initializes them. Useful for arrays that you want pre-zeroed.

```
int *scores = calloc(20, sizeof(int));   // array of 20 zeroed ints
```

Allocation can fail if the heap is out of room. Both functions return `NULL` on failure. Every allocation should be null-checked before use:

```
int *arr = malloc(5 * sizeof(int));
if (arr == NULL) {
```

```
    // handle failure (typically return an error code)
    return 1;
}
```

5.8.2 realloc

`void *realloc(void *ptr, size_t size)`: Resizes the existing heap block pointed to by `ptr` to `size` bytes. May return a different address (if the block had to move).

Two special cases worth knowing: - **`realloc(NULL, size)`** acts exactly like **`malloc(size)`**. (Useful for a single code path that grows a buffer starting from empty.) - **`realloc(ptr, 0)`** acts similarly to **`free(ptr)`** (implementation-defined in modern C, but the common behavior is free).

Tricky: realloc failure. If `realloc` fails, it returns `NULL` but does NOT free the original block. If you write `arr = realloc(arr, ...)` and it fails, you've overwritten the only pointer to the old block and leaked it. The safe pattern uses a temporary pointer:

```
int *temp = realloc(arr, 10 * sizeof(int));
if (temp == NULL) {
    free(arr);    // old block is still live; free it before bailing
    return 1;
}
arr = temp;       // safe to reassign now
```

5.8.3 free

`void free(void *ptr)`: Returns the heap block pointed to by `ptr` to the OS. - `free(NULL)` is a no-op (safe). - Accessing a pointer after **`free`** is use-after-free, undefined behavior. - Calling **`free`** twice on the same block is double-free, undefined behavior.

Good practice: set the pointer to `NULL` immediately after freeing, so any stray use is a null-pointer dereference (an immediate, reproducible crash) rather than a subtle use-after-free.

5.8.4 Should You free() Everything?

Answer: yes, you should free everything you malloc/calloc/realloc. Every dynamic allocation creates heap storage that is NOT reclaimed when the owning function returns (unlike stack variables). Forgetting to free is a memory leak, the block sits around, claimed but unreachable, for the rest of the program's lifetime.

Pedantic caveat: when the program exits, the OS reclaims all of its memory anyway, so short-lived programs often "leak" their final state without observable harm. But this is not a habit to cultivate, long-running programs (servers, GUIs) will bleed memory and eventually crash. Tools like Valgrind and AddressSanitizer flag unfreed allocations for exactly this reason.

5.8.5 Common Memory Bugs

Memory leak: Losing the pointer to a heap block before freeing it. Classic example:

```
int *my_arr = create_array(5);    // allocation 1
my_arr = create_array(10);        // allocation 2, allocation 1 is now unreachable: LEAK
```

Use-after-free: Reading or writing through a pointer after `free`:

```
free(my_arr);
printf("%d\n", my_arr[0]);        // UNDEFINED BEHAVIOR
```

Double-free: Calling `free` on the same pointer twice.

Reading uninitialized **`malloc`** memory: The bytes are garbage; treat them as garbage until you've written a real value.

Worked debugging example. Identify the bugs in:

```
int *create_array(int size) {
    int *arr = malloc(size * sizeof(int));
    arr[0] = 10;                          // bug 1: malloc can fail, arr could be NULL
    return arr;
}

int main() {
    int *my_arr = create_array(5);
    my_arr = create_array(10);            // bug 2: leaks the first 5-int allocation
    free(my_arr);
    printf("First element: %d\n", my_arr[0]);  // bug 3: use-after-free
    return 0;
}
```

At minimum: (1) no null-check after `malloc`, (2) leaked allocation from the first `create_array(5)` call, (3) dereferencing `my_arr` after `free`. A fix would null-check, keep/free both allocations properly, and either refrain from using `my_arr` after `free` or NULL it out.

5.9 Linked Lists

5.9.1 Definition

A linked list is a dynamic, node-based data structure. Each node holds data and a pointer to the next node, so unlike an array, a linked list doesn't need contiguous memory and can grow one node at a time.

```
struct node {
    int data;
    struct node *next;
};
```

New nodes are created with `malloc` and destroyed with `free`.

5.9.2 Traversing and Freeing

Tricky: use **`==`** (comparison), not **`=`** (assignment), in your loop condition. A very common bug:

```
struct node *curr = head;
while (curr = NULL) {    // BUG: assigns NULL to curr, loop never runs
    free(curr);
    curr = curr->next;
}
```

This has several problems stacked together: the assignment `curr = NULL` in the condition both (a) destroys the original pointer and (b) evaluates to `0` (false), so the loop body never executes. Even if you fix the `=` to `==`, the loop as written is still broken: `free(curr)` happens before saving `curr->next`, so after `free` the next line dereferences freed memory (use-after-free).

Correct pattern: save `next` into a temporary *before* freeing the current node:

```
void free_list(struct node *head) {
    struct node *curr = head;
    while (curr != NULL) {
        struct node *temp = curr->next;   // SAVE before free
```

```
        free(curr);
        curr = temp;                    // advance using saved pointer
    }
}
```

5.9.3 Insertion at the Front

When prepending a new node to a linked list, the order of pointer updates matters. You must connect the new node's `next` to the current head before updating the list's `head` to point to the new node, otherwise you lose the rest of the list.

Question: which must happen first? - A) `list->head = new_node;` - B) `new_node->next = list->head;`

Answer: B first. If you do A first, `list->head` already points to `new_node`, and reading `list->head` on the right-hand side of B now gives you `new_node` itself, so `new_node->next` points to itself and the rest of the list is orphaned (leaked).

```
new_node->next = list->head;   // (B) new node points to old first node
list->head = new_node;         // (A) list now starts with new node
```

5.9.4 Deletion of the Last Node

To safely delete the last node of a linked list, walk until `curr->next->next == NULL` (so `curr` is the second-to-last node), then free `curr->next` and set `curr->next = NULL`.

```
void delete_last(struct linked_list *list) {
    struct node *curr = list->head;
    while (curr->next->next != NULL) {
        curr = curr->next;
    }
    free(curr->next);      // free the last node
    curr->next = NULL;     // the new last node has no successor
}
```

Order matters again: after `free(curr->next)`, `curr->next` still holds the old (now-dangling) address. Setting it to `NULL` both marks the end of the list and avoids a dangling pointer.

5.9.5 Stack as a Linked List

A stack is a LIFO (last-in, first-out) structure. Implementing it with a linked list is natural: push = insert at head, pop = remove at head. Both operations are $O(1)$ because they only touch the first node.

Assume `SUCCESS = 0` and `FAILURE = -1`:

```
struct node {
    int data;
    struct node *next;
};

struct stack {
    struct node *top;
};
```

Push: allocate a node, check for allocation failure, point its `next` at the old top, and make it the new top.

```
int push(struct stack *s, int value) {
    if (s == NULL) return FAILURE;

    struct node *new_node = malloc(sizeof(struct node));
    if (new_node == NULL) return FAILURE;   // allocation failure
```

```
    new_node->data = value;
    new_node->next = s->top;    // link to old top (may be NULL if empty)
    s->top = new_node;          // new node becomes the top
    return SUCCESS;
}
```

Pop: save the data, save the pointer to the top node, advance `top`, then free the old top. The critical detail is that you must save the data BEFORE freeing the node; reading `old_top->data` after `free(old_top)` is a use-after-free.

```
int pop(struct stack *s) {
    if (s == NULL || s->top == NULL) return FAILURE;

    struct node *old_top = s->top;
    int value = old_top->data;    // SAVE data before freeing
    s->top = old_top->next;       // advance top pointer
    free(old_top);                // safe to free now
    return value;
}
```