Design of Digital Circuits
PVW Script

Last Updated: June, 2021

Author: Sarah Kamp
Updated by Roberto Starc, João Ferreira, Anaïs Berkes, Lukas Gygi

digitech-pvw-skript@vis.ethz.ch

Disclaimer:
This script only serves as additional material for practice purposes and should not serve as a substitute for the lecture material. We can neither guarantee that this script covers all relevant topics for the exam, nor that it is correct. If an attentive reader finds any mistakes or has any suggestions on how to improve the script, they are encouraged to contact the authors under the email address indicated above or, preferably, through a GitLab issue on https://gitlab.ethz.ch/vis/luk/pvw_script_digitech.

Sources:
D. Harris and S. Harris, "Digital Design and Computer Architecture" (1st Edition).
Lecture Slides, Prof. Onur Mutlu, ETH Zurich, Spring 2018, 2019, 2020.

Contents

1 Number Systems

1.1 Binary Numbers

A *bit* (short for **b**inary dig**it**) represents one of two logical states. We will refer to these two possible states as 0 / 1 or, less commonly, `FALSE` / `TRUE` or `OFF` / `ON`. Multiple bits can be *concatenated* (i.e., put together) to express binary numbers. Because each digit in a binary number can take one of two values, these numbers are said to be expressed in base 2. An N-bit binary number represents one of 2^N sequential values, starting with 0. As a result, the largest N-bit binary number represents the value $2^N - 1$.

Bits, bytes, nibbles and words. A group of eight bits is called a *byte* and can represent $2^8 = 256$ values. A group of four bits is called a *nibble*, but this is no longer a commonly used unit. The unit of data in a microprocessor is the *word.* The size of a word therefore depends on the microarchitecture (implementation) of the processor. Most modern computers use 64-bit words, but some still use 32-bit words. Simpler microprocessors, like those you might find in smart gadgets, often use 8- or 16-bit words. You might also see references to *"half-words"*, which contain half the number of bits as a word.

Binary prefixes[1]. When used in conjunction with binary prefixes, bits are shortened as "b", and bytes are shortened as "B". 2^{10} (or 1024) bytes make up a kilobyte (KB); likewise, 1024 bits make up a kilobit (Kb or Kbit). By the same logic, 2^{20} bytes make up a megabyte (MB), and 2^{20} bits a megabit (Mb or Mbit). Going one step further, 2^{30} bytes make up a gigabyte (GB) and 2^{30} bits a gigabit (Gb or Gbit). While it is usually simpler to just work in base two, it is important to remember that 2^{10} is close to 10^3, 2^{20} is close to 10^6 and 2^{30} is close to 10^9. These are usually reasonable approximations.

Special bits and bytes. There are four important terms when talking about bits and bytes. The *least significant bit* (lsb) is the rightmost bit (the one in the 1's column) in a binary number. The bit on the leftmost position is the *most significant bit* (msb). Similarly, the rightmost byte is called the *least significant byte* (LSB) and the leftmost byte is called the *most significant byte* (MSB).

Working with binary numbers. Remembering the first few powers of two (up until 2^{10}, for example) will make your life easier. It is also useful to know that the K, M and G binary prefixes correspond to 2 to the power of 10, 20 and 30, respectively. Table 1 includes some useful powers of two for your reference.

Power	Decimal Value	Binary prefix
2^0	1	-
2^1	2	-
2^2	4	-
2^3	8	-
2^4	16	-
2^5	32	-
2^6	64	-
2^7	128	-
2^8	256	-
2^9	512	-
2^{10}	1 024	K (Kilo)
2^{20}	1 048 576	M (Mega)
2^{30}	1 073 741 824	G (Giga)

Table 1: Some useful powers of 2.

As you familiarise yourself with binary numbers, it is important that you learn how to efficiently convert numbers between different bases. Below you will find examples of how to perform this conversion between base 2 (binary) and base 10 (decimal).

[1]If you see a prefixed binary number written somewhere nowadays, you should probably interpret 1 KB as $2^{10} = 1024$ bytes, 1 MB as 2^{20} bytes, and so on. However, in certain (mostly historical) contexts, you would see 1 kB represent 10^3 bytes, 1 MB represent 10^6 bytes, and so on. This can be very confusing, and was caused by conflicting standards which were issued independently by multiple entities. Fortunately, as we discuss, making the wrong assumption will not result in a large error in most cases. If you want to learn more about binary prefixes and their confusing history, this Wikipedia article is a great place to start: `https://en.wikipedia.org/wiki/Binary_prefix`. As a general rule of thumb, unless you have a good reason not to do so, stick to the binary base: $2^{10}, 2^{20}, 2^{30}, ...$

Example. *Binary to decimal conversion.*

To convert a binary number to decimal, write out the powers of two from right to left, starting from 2^0. Multiply the digits of the binary number with their corresponding powers and add the final values. For example, consider the number 01000101_2:

$$01000101_2 = 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 69_{10}$$

Example. *Decimal to binary conversion.*

To convert from decimal to binary, repeatedly divide the number by 2. After each division, starting from the right (2^0), fill in each bit of the binary number with the remainder of the corresponding division. Suppose we want to express the number 42 in binary: $42/2 = 21$ (remainder 0), so 0 goes in the 2^0's column; $21/2 = 10$ (remainder 1), so 1 goes in the 2^1's column; $10/2 = 5$ (remainder 0), so 0 goes in the 2^2's column; $5/2 = 2$ (remainder 1), so 1 going in the 2^3's column; $2/2 = 1$ (remainder 0), so 0 goes in the 2^4's column; finally, $1/2 = 0$ (remainder 1), so 1 goes in the 2^5's column.

$$42_{10} = 101010_2$$

1.2 Hexadecimal Numbers

Just like every binary digit represents one of two values and every decimal digit one of ten values, so too does each *hexadecimal* digit represent one of sixteen values. Because sixteen is a power of two, an interesting property emerges: every four bits represent one of 16 possibilities, and can therefore be replaced by one hexadecimal digit. Because of this, converting a number from binary to hexadecimal (or vice-versa) is a very simple operation! Since writing and manipulating binary numbers can be very inconvenient and error prone (for humans), binary values are frequently written in their hexadecimal form when they are displayed by a computer. Table 2 shows all sixteen hexadecimal digits and their corresponding values in bases 10 and 2.

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Table 2: The 16 hexadecimal digits and their corresponding values in decimal and 4-bit binary representations.

Example. *Hexadecimal to Binary and Decimal conversion*

To convert a hexadecimal number to binary, simply convert each digit individually and concatenate the result.

To convert a hexadecimal number to decimal, list the powers of 16 from right to left and write the hexadecimal number below. Multiply the digits of the hexadecimal number with their corresponding powers and add the final values.

$$2_{16} = 0010_2 = 2_{10}$$
$$\mathrm{E}_{16} = 1110_2 = 14_{10}$$
$$\mathrm{D}_{16} = 1101_2 = 13_{10}$$

$$2ED_{16} = (0010\ 1110\ 1101)_2 = 2 \text{ x } 16^2 + 14 \text{ x } 16^1 + 13 \text{ x } 16^0 = 749_{10}$$

Example. *Decimal to Hexadecimal conversion*
Starting from the right, repeatedly divide the number by 16. The remainder goes in each column. 333/16 = 20, with a remainder of $13_{10} = D_{16}$, going to the 16^0 = 1's column. 20/16 = 1, with a remainder of 4, going to the 16^1 = 16's column. 1/16 = 0, with a remainder of 1, going to the 16^2 = 256's column.

$$333_{10} = 14D_{16}$$

1.3 Binary Addition

Binary addition is basically like decimal addition. Whenever a number is greater than one digit, you carry it into the next column.

Example. *Simple Addition*

```
   1 1111 111
   0 1100 1001
 + 0 1111 1111
 -------------
   1 1100 1000
```

Usually digital systems operate on a fixed number of digits and a system is said to overflow, if the result is too big to fit into the available digits.

Example. *Addition with Overflow*

```
   1 1111 111
   0 1100 1001
 + 1 1111 1111
 -------------
  10 1100 1000
```

If the above example had to be stored in a 9 digit number, the msb would be discarded, leaving us with an incorrect result.

1.4 Signed Binary Numbers

1.4.1 Sign/Magnitude Numbers

An N-bit sign/magnitude number uses the msb as the sign and the N-1 remaining bits as the magnitude. A sign bit of 0 represents a positive number and a sign bit of 1 represents a negative number. For example is $5_{10} = 101_2$ as an unsigned number and 0101_2 as a sign/magnitude number. -5_{10} would be represented as 1101_2. There are a few issues with this representation, for example that there are two ways of writing a zero in binary - a negative and a positive representation. Also, ordinary binary addition does not work in this representation. An N-bit sign/magnitude number spans the range $[-2^{N-1}+1, 2^{N-1}-1]$.

1.4.2 Two's Complement Numbers

This variant resolves the issue of having two different representations of zero - 0_{10} is written as all zeros $00...000_2$. Also ordinary binary addition works, in contrast to the sign/magnitude representation. Subtraction is performed by taking the two's complement of the second number and then adding both numbers. To write a positive number in two's complement representation, simply take the binary representation and add a zero as the msb. For example $5_{10} = 101_2$ becomes 0101_2. To represent a negative number, invert the bits and add 1. $5_{10} = 0101_2$ becomes 1010_2, and when adding 1, one gets $-5_{10} = 1011_2$. An N-bit two's complement number spans the range $[-2^{N-1}, 2^{N-1}-1]$. Sometimes a two's complement number needs to be extended to more bits. In that case, the sign bit is simply copied into the msb as many times as needed. This process is called *sign extension.* For example $-5_{10} = 1011_2$ can be extended to 8 bits as $1111\ 1011_2$.

1.5 Exercises

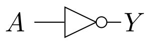
1. Convert the binary number 10110_2 to decimal.
2. Convert the decimal number 84_{10} to binary.
3. Convert the binary number 1111010_2 to hexadecimal.
4. Convert the decimal number 333_{10} to hexadecimal and binary.
5. Compute $0111_2 + 0101_2$. Does an overflow occur?
6. Compute $1101_2 + 0101_2$. Does an overflow occur?
7. Find the representation of -2_{10} as a 4-bit two's complement number.
8. Find the decimal value of the two's complement number 1001_2.
9. Compute $-2_{10} + 1_{10}$ using two's complement numbers.
10. Compute $-7_{10} + 7_{10}$ using two's complement numbers.
11. Compute $5_{10} - 3_{10}$ using 4-bit two's complement numbers.
12. Compute $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

2 Combinational Circuits

Logic gates are digital circuits that take one or more binary inputs and produce a binary output. The relation between the input and the output can be described with a truth table or a Boolean equation.

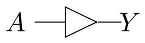
2.1 NOT Gates

A NOT gate is also called an inverter. It has one input and one output. The relation can be summarized by the Boolean equation Y = $\overline{A}$. When the input is 1, Y will be equal to 0 and when the input is 0, Y will be equal to 1. The bubble is the sign for inversion and can also be used to invert signals in other logical gates, as we will see.



2.2 Buffer

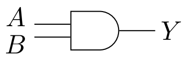
A buffer simply copies the input to the output, Y = A. From a logical point of view, this might seem useless, but from an analog point of view, there are a few applications for a buffer. For example the ability to deliver larger amounts of current to a motor or the ability to quickly send its output to many gates.



2.3 AND Gate

An AND Gate produces the output TRUE, if and only if both inputs are TRUE. This is represented by the table below or by the Boolean equation Y = AB.

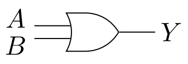
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



2.4 OR Gate

An OR Gate produces the output TRUE, if A or B or both are TRUE, and only produces FALSE, if both A and B are FALSE. This is represented by the table below or by the Boolean equation Y = A+B.

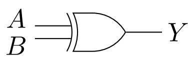
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



2.5 Other Two-Input Gates

2.5.1 XOR

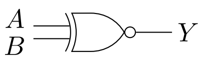
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0



$Y = A \oplus B$

2.5.2 XNOR

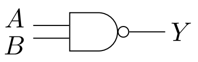
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1



$$Y = \overline{A \oplus B}$$

2.5.3 NAND

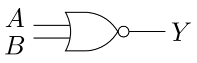
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



$$Y = \overline{AB}$$

2.5.4 NOR

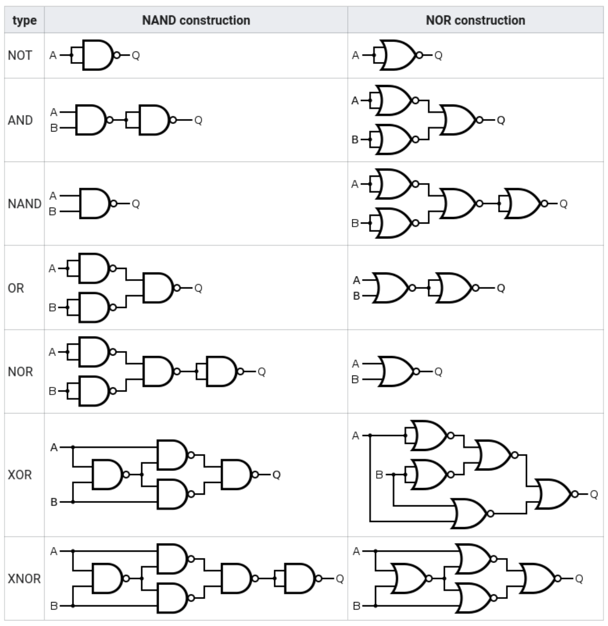
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



$$Y = \overline{A + B}$$

2.6 Universal Logic Gates

NAND and NOR gates are called universal logic gates, because they can be used to construct every other logic gate, using only this type of gate. The folloing table shows how a combination of either gate can be used to represent other logic gates:

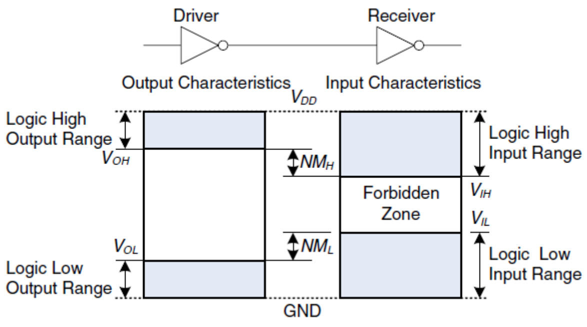


2.7 Multiple Input Gates

An N-input AND gate produces TRUE, when all of the N inputs are TRUE. An N-input OR gate outputs TRUE, when at least one input is TRUE. An N-input XOR gate produces TRUE, when an odd number of inputs is TRUE, and is also referred to as a parity gate.

2.8 Voltage And Logic Levels

The ground, or GND, is the lowest voltage in the system - let's suppose it to be 0V. The highest voltage in the system comes from the power supply and is called V_{DD}. Let's assume it to be 5V, even though it is usually lower in modern chips, in order to decrease power consumption.
Logic levels map a continuous variable onto a discrete binary variable, as shown below.

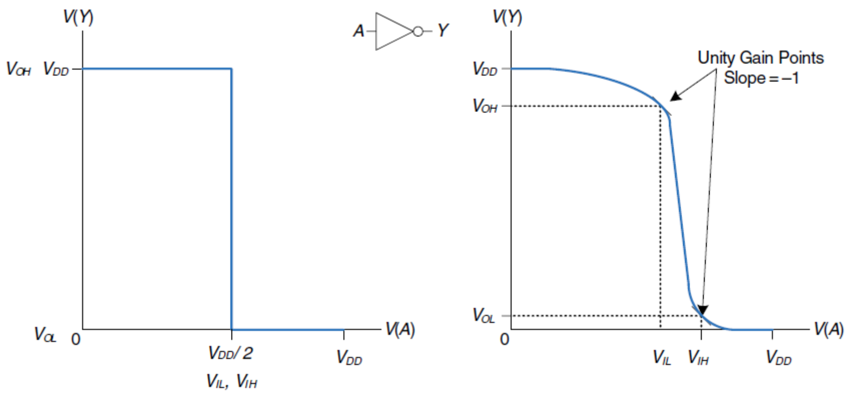


The first gate is the driver, the second is the receiver. The output of the former is connected to the input of the latter. The driver produces a LOW(0) output in the range of 0 to V_{OL} or a HIGH(1) output in the range of V_{OH} to V_{DD}.
If the receiver gets an input between 0 and V_{IL}, it will be interpreted as LOW. If the receiver gets an input between V_{IH} and V_{DD}, the receiver will interpret the input as HIGH. Anything in the forbidden zone between V_{IL} and V_{IH}, will make the behavior of the gate unpredictable.

The noise margin is the amount of noise that could be added to a worst-case output, such that the signal can still be interpreted as a valid input. One can easily read the equations below out of the diagram above:

$NM_L = V_{IL} - V_{OL}$
$NM_H = V_{OH} - V_{IH}$

An ideal inverter would have an abrupt switching threshold at $V_{DD}/2$. However, a real inverter switches more gradually between the extremes. A reasonable place to choose the logic levels is where the slope of the transfer characteristic $dV(Y)/dV(A) = -1$. These points are called unity gain points.



2.9 CMOS Transistors

Transistors are basically electrically controlled switches that turn ON or OFF when a voltage or current is applied to a control terminal. While the reality of what a transistor is is not quite as simple, this abstraction is sufficiently accurate for our purposes.

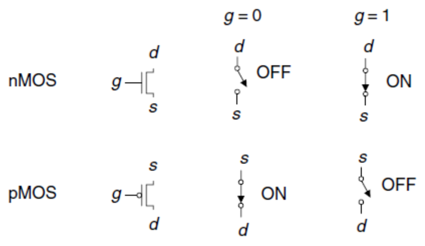
2.9.1 nMOS and pMOS Transistors

A MOS transistor is a sandwich of several layers of conducting and insulating materials. It consists of a conducting layer (the gate) on top of an insulating layer of silicon dioxide on top of the silicon wafer (the substrate).
A lack of negative charge lets a source act as a positively charged particle, those are called p-type. Electrons carrying a negative charge are called n-type.
A MOS transistor behaves as a voltage-controlled switch in which the gate voltage creates an electric field that turns ON or OFF a connection between the source and drain.

The n-type transistors, called nMOS, have regions of n-type dopants adjacent to the gate called the source and the drain and are built on a p-type semiconductor substrate. The substrate is normally tied to GND. When the gate is also at 0V, there is no path for current to flow between the source and drain, so the transistor is OFF. When the gate is raised to V_{DD}, it establishes an electric field that attracts positive charge on the top plate and negative charge to the bottom plate. When enough negative charge is attracted to the underside of the gate, the region inverts from p-type to n-type. Now the transistor has a continuous path from the n-type source to the n-type drain, so the transistor is ON.

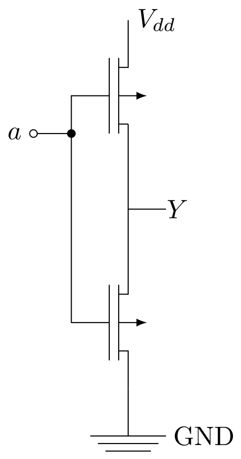
The p-type transistors, called pMOS, are the exact opposite, consisting of a p-type source and drain regions in an n-type substrate. The substrate is tied to V_{DD}, so when the gate is also at V_{DD}, the pMOS transistor is OFF. When the gate is at GND, the channel inverts to p-type and the pMOS transistor is ON.

The processes that provide both flavors of transistors on the same chip are called Complementary MOS, or CMOS. They give us two types of electrically controlled switches.



2.9.2 CMOS Gates

This is a schematic of a NOT gate built with CMOS transistors. The nMOS transistor is connected between GND and the Y output. The pMOS transistor is connected between V_{DD} and the Y output. If A = 0, the nMOS transistor is OFF and the pMOS transistor is ON. Hence, Y is connected to V_{DD}, but not to GND and is pulled up to a logic 1. If A = 1, the nMOS transistor is ON and the pMOS transistor is OFF, and Y is pulled down to a logic 0.
If both the pull-up and the pull-down networks were OFF simultaneously, the output would be connected to neither V_{DD} nor GND. Then we would say that the output floats and its value is undefined.
If both were ON, a short circuit would exist between V_{DD} and GND. The output might be in the forbidden zone and the transistors would consume large amounts of power.
To avoid both short circuits and floats, follow the rule of conduction complements: When nMOS transistors are in series, pMOS transistors must be in parallel. When nMOS transistors are in parallel, the pMOS transistors must be in series.

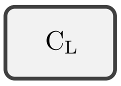


2.10 Combinational Circuits

A circuit is combinational if it consists of interconnected circuit elements such that

- Every circuit element is itself combinational
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once

Such a circuit is memoryless and the functional specification of a combinational circuit expresses the output values in terms of the current input values. For simplicity, you can express such a circuit as a black box, where the C_L in the middle indicates, that the circuit was implemented using only combinational logic.



2.11 Boolean Equations

The complement of a variable A is its inverse $\overline{A}$. The variable or its complement is called a literal. A is the true form and $\overline{A}$ is the complementary form.
The AND of one or more literals is called a product or implicant. For example, A, or $A\overline{B}$ are implicants. A minterm is a product involving all of the inputs to the function.
The OR of one or more literals is called the sum. A maxterm is a sum involving all of the inputs to the function.
The order of operations is important: the order of precedence is NOT > AND > OR.

2.11.1 Sum-Of-Products Form

A truth table with N inputs contains 2^N rows, for all the possible input values. For every row, there is a corresponding minterm, containing all the values of that row. The sum-of-products form is achieved by summing all of the minterms for which the output Y is TRUE.

A	B	Y	minterm
0	0	1	$\overline{A}\,\overline{B}$
0	1	0	$\overline{A}B$
1	0	0	$A\overline{B}$
1	1	1	AB

This truth table would have the sum-of-products form Y = $\overline{A}\,\overline{B}$ + AB

2.11.2 Product-Of-Sums Form

Each row of the truth table has a maxterm for which the output is FALSE. The product-of-sums form consists of the AND of each of the maxterms for which the output is FALSE.

A	B	Y	maxterm
0	0	1	A+B
0	1	0	A+$\overline{B}$
1	0	0	$\overline{A}$+B
1	1	1	$\overline{A}$+$\overline{B}$

This table would have the product-of-sums form Y = (A+$\overline{B}$)($\overline{A}$+B)

2.12 Boolean Algebra

2.12.1 Axioms of Boolean Algebra

Axiom		Dual		Name
A1	$B = 0$ if $B \neq 1$	A1'	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2'	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3'	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4'	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5'	$1 + 0 = 0 + 1 = 1$	AND/OR

2.12.2 Theorems of One Variable

	Axiom		Dual	Name
T1	$B \bullet 1 = B$	T1'	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2'	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3'	$B + B = B$	Idempotency
T4	$\overline{\overline{B}} = B$			Involution
T5	$B \bullet \overline{B} = 0$	T5'	$B + \overline{B} = 1$	Complements

2.12.3 Theorems of Several Variables

	Axiom		Dual	Name
T6	$B \bullet C = C \bullet B$	T6'	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7'	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8'	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9'	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10'	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11'	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 ...}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} ...)$	T12'	$\overline{B_0 + B_1 + B_2 ...}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} ...)$	De Morgan's Theorem

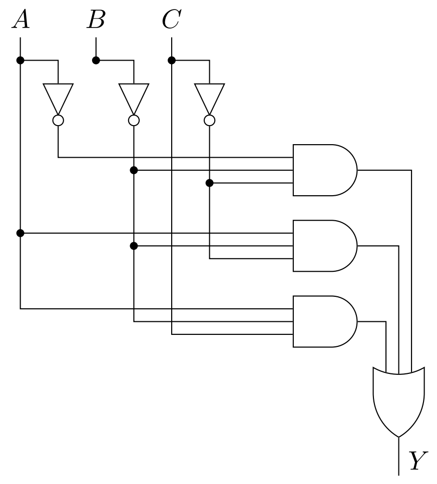
Especially De Morgan's Theorem is very important for the design for digital circuits, because it shows that for example a NAND gate can be expressed by an OR gate with inverted inputs, or likewise a NOR gate can be expressed by an AND gate with inverted inputs.
Inverting an input or output is denoted by drawing a small white circle, called a bubble. Imaging that "pushing" the bubble through the gate at one side makes it come out at the other side and flips the body of the gate from AND to OR or the other way round. Some rules for bubble pushing:

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from OR to AND or vice versa
- Pushing a bubble from the output to the inputs puts bubbles on all gate inputs
- Pushing bubbles from all inputs forward to the output puts a bubble on the gate output

2.13 Drawing Schematics

When inverters, AND gates, and OR gates are arrayed in a systematic fashion, this style is called PLA (programmable logic array). Here are some general guidelines for drawing schematics:

- Inputs are on the left (or top) side of a schematic
- Outputs are in the right (or bottom) side of a schematic
- Whenever possible, gates should flow from left to right
- Straight wires are better to use than wires with multiple corners
- Wires always connect at a T junction
- A dot where wires cross indicates a connection between the wires
- Wires crossing without a dot make no connection



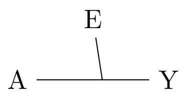
2.14 X's and Z's

2.14.1 Illegal Value: X

X indicates that the circuit node has an illegal or unknown value. This usually happens, when it is being driven to 1 and to 0 at the same time. This is called contention and is considered to be an error and you should avoid this at all times.
Be careful when using this symbol in the context of truth tables, because there it is usually used as a "Don't Care" symbol, where the value of the variable can be either 1 or 0, but always represents one of both.

2.14.2 Floating Value: Z

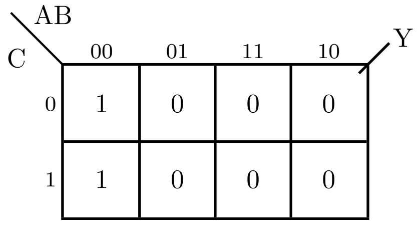
This symbol indicates that the node is driven to neither 1 nor 0. The node is said to be floating or high Z. A floating node might be 1 or 0 or some voltage in between, but it does not necessarily represent an error, as long as there is some other circuit element driving the node to a valid logic level when the value of the node is relevant to a circuit operation.
A common cause for a floating node is to forget to connect a voltage to some circuit input or to assume that an unconnected input is the same as an input of value 0.
The tristate buffer allows the output to float, when the Enable input is set to FALSE. When the Enable input is set to TRUE, the tristate buffer acts as a normal buffer and propagates the input A to the output Y. This buffer has an active high enable. It is also possible to have an active low enable, by placing a bubble at the Enable input and inverting the Enable bit. A tristate buffer with an active low enable allows floats if the Enable input is set to TRUE.



2.15 Karnaugh Maps

Karnaugh Maps, or K-Maps are a graphical method for simplifying boolean equations.

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



Each square in the K-Map corresponds to a row in the truth table and contains the value of the output Y for that row. Just like in the truth table, every square in the K-Map corresponds to a single minterm. Adjacent squares share all the same literals except one. The A and B combinations on top appear in an order called the Gray code, where adjacent entries too differ in only one variable. The K-Map also wraps around, meaning that the squares on the right hand corner are effectively adjacent to the ones on the left hand corner.
For the K-Map to help with the simplification, circle 1's in adjacent squares. In our example that would be one circle including both 1's. For each circle write the corresponding implicant (product of one or more literals). Variables whose true and complementary forms are both in the circle, are excluded from the implicant. In our case, C is present in both its true and its complementary form, so $\overline{A}\ \overline{B}$ is our implicant. Since that is the only circle, we conclude that Y = $\overline{A}\ \overline{B}$.
For any K-Map the rules you have to follow are the following:

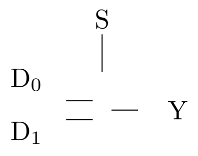
- Use the fewest circles necessary to cover all the 1's
- All the squares in each circle must contain 1's
- Each circle must span a rectangular block that is a power of 2 squares in each direction
- Each circle should be as large as possible
- A circle may wrap around the edges of the K-Map
- A 1 in a K-Map may be circled multiple times if doing so allows fewer circles to be used
- X's (Don't Cares) can be either circled or not, whatever allows for fewer circles to be used

2.16 Combinational Building Blocks

2.16.1 Multiplexers and Decoders

Multiplexers choose an output among several possible inputs, based on the value of a select signal. In the example shown on the right, the output is equal to D_0 if the select signal is FALSE. When the select signal is TRUE, the multiplexer chooses D_1 as the output.
This is an example for a 2:1 multiplexer. There are also wider multiplexers possible, where every input is encoded with more than one line.
When using a general N:1 multiplexer, $\log_2$N bits are used to encode the select signal. There, the signal chooses one of N possible outputs.

S	D_0	D_1	Y
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	1



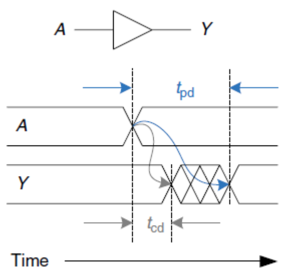
A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs, depending on the input combination. A decoder is usually depicted by a rectangle.

2.17 Timing

2.17.1 Propagation and Contamination Delay

The propagation delay t_{pd} is the maximum amount of time from when an input changes until the output(s) reach their final value (i.e. the time until a change in the input has *propagated* to all the outputs).
The contamination delay t_{cd} is the minimum amount of time from when an input changes until any output starts to change its value (i.e. the time until a change in the input *contaminates* any output).

Propagation and contamination delay are determined, among other factors, by the path a signal takes from the input to the output. The critical path is the slowest path, which usually corresponds to the longest path as well. This path is critical, because it limits the speed at which the circuit can operate.



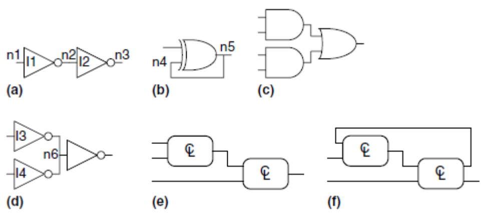
The shortest path is, as the name implies, the shortest, and therefore the fastest path through the circuit.
The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path.
The contamination delay of a combinational circuit is the sum of the contamination delays through each element on the shortest path.

2.17.2 Glitches

It is possible for a single input transition to cause multiple output transitions. This is called a glitch or hazard. Glitches usually don't cause problems. In general, they can occur when a change in a single variable crosses the boundary between two prime implicants in a K-Map. You can eliminate this glitch by adding redundant implicants to the K-Map to cover these boundaries.

2.18 Exercises

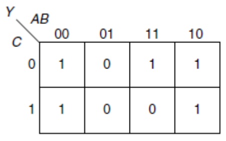
1. Draw a schematic for a three-input NAND gate using CMOS transistors.
2. Draw a schematic for a two-input NOR gate using CMOS transistors.
3. Draw a schematic for a two-input AND gate.
4. Which of these are combinational circuits?



5. Ben is having a picnic. He won't enjoy it if it rains or if there are ants. Design a circuit that will output TRUE only if Ben enjoys the picnic.
6. Using De Morgan's theorem, derive the product-of-sums canonical form of Y from the sum-of-products form of $\overline{Y}$.

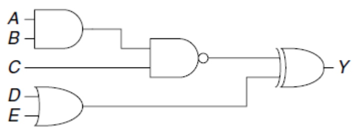
A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

7. Minimize the equation $\overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C$
8. Suppose we have the function Y = F(A, B, C). Minimize the function using the K-Map.



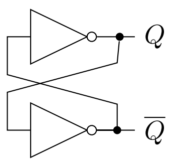
C \ AB	00	01	11	10
0	1	0	1	1
1	1	0	0	1

9. Implement a 2:4 decoder with AND, OR and NOT gates.
10. Find the propagation delay and the contamination delay of this circuit. Each gate has a propagation delay of 100 ps and a contamination delay of 60 ps.



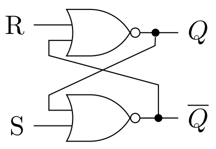
3 Sequential Circuits

3.1 Latches and Flip-Flops



On the right is a basic building block of memory, a so called bistable element, with two stable states. The inverters are cross-coupled, meaning that the output of the first inverter is connected to the input of the second and vice versa. Analyzing a sequential circuit like the one on the right is different from analyzing a combinational circuit, because it is cyclic. You have to look at two different cases.
When Q is FALSE, the upper inverter receives $\overline{Q}$ as an input, which is TRUE in this case, again producing FALSE as the output Q. This is consistent with the original assumption, making the state stable.
When Q is TRUE, the upper inverter receives $\overline{Q}$ as an input, which is FALSE in this second case, producing TRUE as the output Q. This, again, is consistent with the original assumption.
However, this bistable circuit is not practical, due to the lack of inputs, making it not possible to control the state of it.

3.1.1 SR Latch



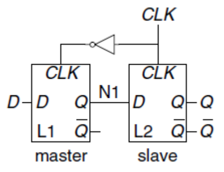
The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the two inputs S and R, which can set and reset the circuit.
When both inputs are set to FALSE, the circuit simply outputs the previous values for Q and $\overline{Q}$, having them stores, as in the cross-coupled inverter.
When only R is set to TRUE, Q is reset to FALSE, giving us TRUE as the output for $\overline{Q}$.
When only S is set to TRUE, Q is set to TRUE, leaving us with FALSE as the output for $\overline{Q}$.
When both S and R are set to TRUE, this resets the circuit, setting both the output for Q and $\overline{Q}$ to FALSE.

3.1.2 D Latch

CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Since it is kind of weird how the SR latch behaves when both S and R are set to TRUE, we have the D latch. This circuit is also very convenient, because it separates the issue of what should happen from the issue of when it should happen.
The data input D controls what the next state should be, whilst the clock input CLK controls when it should happen.
It doesn't matter what the D input is when the CLK input is FALSE, because it won't be passed on. When the CLK input is TRUE, the S value of the intern SR latch gets set to whatever value D has and R gets set to the complement of S. The rest of the D latch works the exact same way, the SR latch works.

3.1.3 D Flip-Flop



When CLK = 0, the master latch is transparent and the slave is opaque. Therefore, whatever value was at D propagates through to N1.
CLK = 1, the master becomes opaque and the slave becomes transparent. The value at N1 propagates through to Q, but N1 is cut off from D.
So, whatever value was at D before the clock rises from 0 to 1, gets copied to Q after the clock rises. At all other times, Q retains its old value, because there is always an opaque latch blocking the path between D and Q.

3.1.4 Register

An N-bit register is a bank of N flip-flops sharing a common clock, so that all bits of the register are updated at the same time.

3.1.5 Enabled Flip-Flop

An enabled flip-flop adds another input EN to determine whether data is loaded on the clock edge. When EN is TRUE, the circuit behaves like an ordinary D flip-flop, but when EN is FALSE, the enabled flip-flop ignores the clock and retains its state. This is useful when you want to load some data on the clock edge only some of the time.

3.1.6 Resettable Flip-Flop

A resettable flip-flop adds another input RESET to the circuit. When RESET is FALSE, the circuit behaves like an ordinary D flip-flop. When RESET is TRUE, the circuit ignores the current D input and resets the output to 0. This is useful when you want to force a known state into all flip-flops when you first turn it on.
There are synchronously or asynchronously resettable flip-flops. Synchronously resettable flip-flops reset themselves only on the rising edge of CLK. Asynchronously resettable flip-flops reset themselves as soon as RESET becomes TRUE, independent of CLK.

3.2 Sequential Logic Design

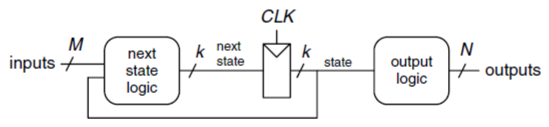
3.2.1 Synchronous Sequential Circuits

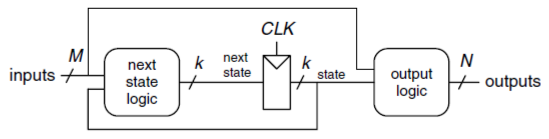
Cyclic paths can introduce race conditions and therefore undesired behavior in a sequential circuit. To avoid that, designers often insert registers, to break the cyclic paths and turn the circuit into a collection of combinational logic and registers. The registers contain the state of the system, which only changes at the clock edge, so the state is synchronized to the clock. If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated.
This leads us to these rules of synchronous sequential circuit composition:

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register

Sequential circuits that are not synchronous are asynchronous. Asynchronous sequential circuits are more general, because they can use any kind of feedback. However, they are also more difficult to design, which is why most sequential circuits are synchronous.

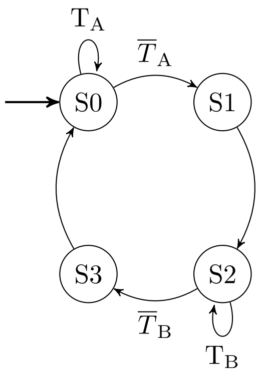
3.3 Finite State Machines





Synchronous sequential circuits like these are called finite state machines (FSM). An FSM consists of two blocks of combinational logic, next state logic and output logic, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs. There are two kinds of FSMs:
In a Moore machine, the output depends only on the current state of the machine, whilst in a Mealy machine, the output depends both on the current state of the machine and on the current inputs.

Consider the FSM on the right, depicting the states of two traffic lights at an intersection. There are two Lights, L_A and L_B, which are controlled by the two inputs T_A and T_B.
In S0, L_A is green and L_B is red.
In S1, L_A is yellow and L_B is red.
In S2, L_A is red and L_B is green.
In S3, L_A is red and L_B is yellow.
A state is encoded by two bits: S0 is encoded as 00, S1 is encoded as 01,
S2 is encoded as 10 and S3 is encoded as 11. A green light is encoded as 00, a yellow light is encoded as 01 and a red light is encoded as 10.



The so called state transition table below shows the current state, encoded as the two bits S_0 and S_1, the two inputs T_A and T_B and the next state that follows, again encoded in two bits.

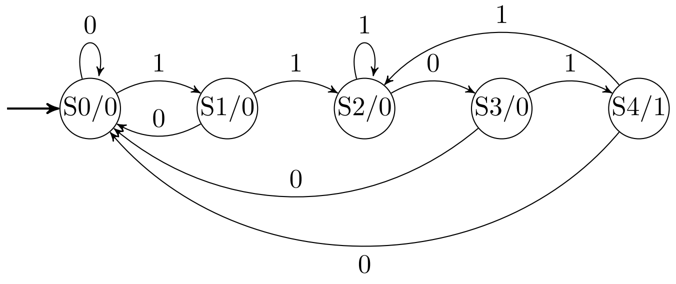
S_0	S_1	T_A	T_B	S'_0	S'_1
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

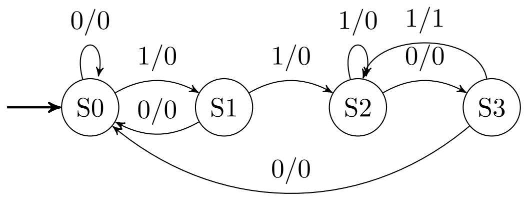
3.3.1 State Encodings

The state encoding of the traffic light was chosen arbitrarily. With this strategy, k states can be encoded with $\log_2 k$ bits. Another strategy would be one-hot encoding, where k bits are used and every state has a designated bit. A FSM with three states would for example have the three encodings 100, 010 and 001.

3.3.2 Mealy Machines

In the above example, the output only depends on the current state, which is why we could associate our states with the color the lights would have when that state was reached.
Now, when switching from a Moore to a Mealy machine, the output not only depends on the current state, but also on the current input. Here are the two in comparison (the two FSMs do the same thing):





As you can see, a Mealy is labeled in the form of A/Y, where A is the value of the input that causes the transition and Y is the output. In general, Mealy machines usually require fewer states than Moore machines.

3.3.3 Drawing a Finite State Machine

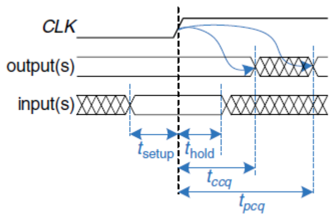
Use the following procedure to draw a FSM:

- Identify inputs and outputs
- Sketch a state transition diagram
- For a Moore machine: Write a state transition table and an output table
- For a Mealy machine: Write a combines state transition and output table
- Select state encodings
- Write Boolean equations for the next state and output logic
- Sketch the circuit schematic

3.4 Timing of Sequential Logic

The aperture of a sequential element is defined by a setup time and a hold time, before and after the clock edge. The dynamic discipline limits us to using signals that change outside the aperture time.

This is the timing specification of a FSM. When the clock rises, the output may start to change after the clock-to-Q contamination delay t_{ccq}, and must definitely settle to the final value within the clock-to-Q propagation delay t_{pcq}. For the circuit to sample its input correctly, the inputs must have stabilized at least some setup time t_{setup} before the rising edge of the clock and must remain stable for at least some hold time t_{hold}, after the rising edge of the clock. The sum of the setup and hold time is called the aperture time of the circuit.
Note that in the real world, the clock does not reach all registers at the same time. The variation is called the clock skew, t_{skew}, caused by the delay of the electrical signal in the wire.



3.4.1 System Timing

The clock period or cycle time T_c is the time between rising edges of a repetitive clock signal. $f_c = 1/T_c$ is the clock frequency. Increasing it increases the work that a digital system can accomplish per unit of time. Frequency is measured in Hertz (Hz) or cycles per second.
Here are some important constraints:

- T_c Clock period $(1/f)$
- t_{pd} Propagation delay: time from input change to correct output (longest path in circuit)
- t_{pcq} Clock-to-Q propagation delay: time from input change to correct output of a flip flop
- t_{cd} Contamination delay: time from input change to output change (shortest path in circuit)
- t_{ccq} Clock-to-Q contamination delay: time from input change to output change of a flip flop
- t_{setup} Setup time: stable input before rising clock edge
- t_{hold} Hold time: stable input after rising clock edge
- t_{skew} Clock skew: time difference between the same clock signal at different components

- $T_c \geq t_{pcq} + t_{pd} + t_{setup}$
- $t_{pd} \leq T_c - (t_{pcq} + t_{setup})$
- $t_{ccq} + t_{cd} \geq t_{hold}$
- $t_{cd} \geq t_{hold} - t_{ccq}$
- $t_{hold} \leq t_{ccq}$

- When considering $\mathrm{t_{skew}}$: $\mathrm{T_c} \geq \mathrm{t_{pcq}} + \mathrm{t_{pd}} + \mathrm{t_{setup}} + \mathrm{t_{skew}}$
- When considering $\mathrm{t_{skew}}$: $\mathrm{t_{pd}} \leq \mathrm{T_c}$ - $(\mathrm{t_{pcq}} + \mathrm{t_{setup}} + \mathrm{t_{skew}})$

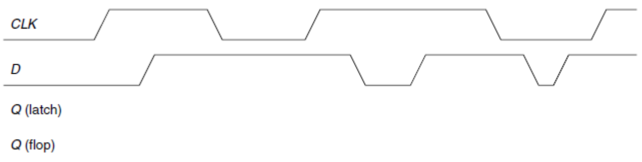
If data is changing when it is sampled, then metastability can occur. The system then needs a resolution time $\mathrm{t_{res}}$ to resolve to a stable state.
When working with asynchronous systems, metastable states are avoided by passing all asynchronous inputs through synchronizers. This is a device that receives an asynchronous input D and a clock CLK. It produces an output Q within a bounded amount of time. The output has a valid logic level with extremely high probability.

3.5 Parallelism

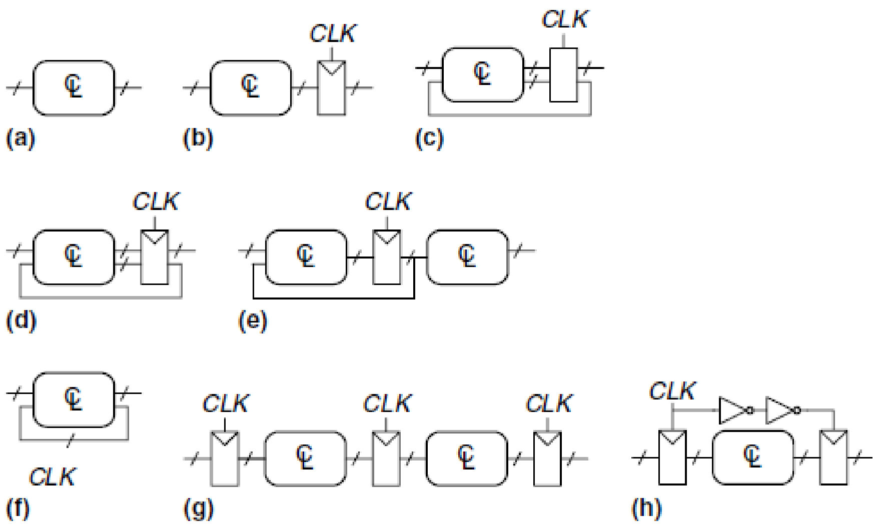
The speed of a system is measured in latency, throughput and tokens moving through a system. A token is a group of inputs that are processed to produce a group of outputs. The latency of a system is the time required for one token to pass through the system from start to end. The throughput is the number of tokens that can be produced per unit of time.
The throughput can be improved by processing several tokens at the same time, which is called parallelism. There is spatial parallelism, where multiple copies of the hardware are provided, so that multiple tasks can be done at the same time. The other form is temporal parallelism, where a task is broken into stages, like an assembly line. Although each task must pass through all stages, a different task will be in each stage at any given time so multiple tasks can overlap. This is called pipelining.
Consider a task with latency L. In a sequential system, the throughput is 1/L. In a spatially parallel system with N copies of hardware, the throughput is N/L. In a temporally parallel system, the task is ideally broken down into N stages, so the throughput can again be N/L. Pipelining is attractive, because it speeds up the system, without the need to duplicate the hardware.

3.6 Exercises

1. Apply the D and CLK input shown in the diagram and identify the output of a D latch and a D flip-flop.



2. Which of these circuits are synchronous sequential circuits? (Remember: the missing triangle in (c) means that the rectangular shape represents a latch, *not* a flip-flop.)



3. A divide-by-N counter has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. Sketch a state transition diagram for such a counter.

Current State	Next State
S0	S1
S1	S2
S2	S0

Current State	Output
S0	1
S1	0
S2	0

4. Alyssa owns a robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last four bits that it has crawled over are, from left to right, 1101. Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles. Compare Moore and Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, and output as your snail crawls along the sequence 111011010.

4 Hardware Description Languages

The two leading Hardware description languages (HDL) are VHDL and Verilog. The HDL used in the Digital Circuits course is Verilog, so this script is going to talk about the Verilog syntax.

4.1 Modules

A module is just a block of hardware with inputs and outputs. There are two common ways to describe a module. Behavioral models describe what the module does and structural models describe how a module is built from simpler pieces.

```
module sillyfunction (input a, b, c,
                             output y);

          assign y = ~a & ~b & ~c |
                      a & ~b & ~c |
                      a & ~b & c;
endmodule
```

A module in Verilog starts with the module name and a list of all inputs and outputs. The assign statement describes combinational logic. ~ describes a NOT, & is an AND and | is an OR.

4.2 Simulation and Synthesis

Simulation is the process of applying inputs to a module, and checking the outputs to verify that the module operates correctly. During synthesis, the textual description of a module is transformed into logic gates.

4.3 Combinational Logic

4.3.1 wires

Wires are used to connect different parts of the module together. They are not like variables in other programming languages, since the do not hold a value, but rather pass a signal from its input to its output(s). A wire can only be once on the left side of an assign statement, otherwise it would have multiple inputs (drivers), which is not allowed. Inputs and outputs of modules are by default wires.

Bus `[3:0] a` represents a 4-bit bus. They are arranged in a little-endian order, so the least significant bit has the smallest bit number (a[0]). An common mistake to watch out for is to declase the Bus as like (input [3:0] a) instead of (input a [3:0]). The second expression generates an array of 1 bit wires, which are handled differently by verilog. A wire of a bus can be accessed with brackets: a[2] is the third wire in the bus. Multiple wires can be accessed with a[1:0] (the two leas signifant bits in a)

Concatenation Wires and busses can be concatenated using curly braces: `assign c = {a[1:0], b};`

4.3.2 Bitwise Operators

Bitwise operators act in single-bit signals or on multi-bit busses.

```
module inv (input [3:0] a,
            output [3:0] y);

     assign y = ~a;

endmodule
```

```
module gates (input [3:0] a, b,
              output [3:0] y1, y2, y3, y4, y5);

     assign y1 = a & b; //AND
     assign y2 = a | b; //OR
     assign y3 = a ^ b; //XOR
     assign y4 = ~(a & b); //NAND
     assign y5 = ~(a | b); //NOR

endmodule
```

4.3.3 Reduction Operators

```
module and8 (input [7:0] a,
              output y);

     assign y = &a;
     // &a is easier to write than
     // assign y = a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[1] & a[0];

endmodule
```

4.3.4 Conditional Assignments

```
module mux2 (input [3:0] d0, d1,
              input s,
         output [3:0] y);

     assign y = s ? d1 : d0;

endmodule
```

The first expression, s, is the condition. If s evaluates to TRUE, the output gets assigned to the second expression, d1. If s evaluates to FALSE, the output gets assigned to the third expression, d0. This is called a ternary operator, because it takes three inputs. This can also be nested:

```
module mux4 (input [3:0] d0, d1, d2, d3,
              input [1:0] s,
         output [3:0] y);

     assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);

endmodule
```

4.3.5 Internal Variables

```
module fulladder (input a, b, cin,
              output s, cout);

     wire p, g;

     assign p = a ^ b;
     assign g = a & b;

     assign s = p ^ cin;
     assign cout = g | (p & cin);
```

```
endmodule
```

4.3.6 Precedence

Op	Meaning
A[n], A[n:m]	Bus wire selection
&A, |A, ∧A	reduction operators
!, ∼	NOT
*, /, %	MUL, DIV, MOD
+, -	PLUS, MINUS
<<, >>	Logical Left/Right Shift
<<<, >>>	Arithmetic Left/Right Shift
<, <=, >, >=	Relative Comparison
==, !=	Equality Comparison
&, ∼&	AND, NAND
∧, ∼ ∧	XOR, XNOR
|, ∼ |	OR, NOR
&&	Logical AND
||	Logical OR
?:	Conditional

4.3.7 Numbers

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010

4.3.8 Z's and X's

Floating values are indicated with a z, which is particularly useful when describing a tristate buffer, where the output is floating when the enable is 0.

```
module tristate (input [3:0] a,
                 input en,
                 output [3:0] y);

    assign y = en ? a : 4'bz;

endmodule
```

In a similar way, x is used to indicate an invalid logic level. If a bus is simultaneously driven to 1 and to 0, by two enabled tristate buffers, the result is x, indicating contention.
1 & z = 1 & x = x & x = z & z = x
0 & z = 0 & x = 0

4.3.9 Delays

```
'timescale 1ns/1ps
module example (input a, b, c,
                output y);

    wire ab, bb, cb, n1, n2, n3;

    assign #1 {ab, bb, cb} = ~ {a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;

endmodule
```

Statements of the form "timescale unit/precision" indicate the value of each time unit. The # symbol is used to indicate the number of units of delay. This is only used for simulations.

4.4 Structural Modeling

Structural modeling describes a module in terms of how it is composed of simpler modules.

```
module mux4 (input [3:0] d0, d1, d2, d3,
             input [1:0] s,
             output [3:0] y);

    wire [3:0] low, high;

    mux2 lowmux (d0, d1, s[0], low);
    mux2 highmux (d2, d3, s[0], high);
    mux2 finalmux (low, high, s[1], y);

endmodule
```

Here, the mux2 module must be defined somewhere else in the Verilog code. lowmux, highmux and finalmux are instances of the module mux2. The inputs and outputs are passed on in the round brackets.

4.5 Always Blocks

4.5.1 Registers

Registers in verilog are an alternative to wires, but they can save a value.
they are also defined in a similar way:

```
wire a; // 1-bit wire
reg b; // 1-bit register
wire [3:0] c; // 4-bit wire
reg [4:0] d; // 4-bit register

reg [7:0] mem [0:31]; // memory block of 32 8-bit values
```

Although they can be used in a similar fashion, there are some diffrences between the two:

- wires and regs can be connected to inputs of module instantiations.
- only wires can be connected to outputs of module instantiations.
- wire can not be used on the left hand side of = and ¡= in always @ blocks.

- wires are the only legal type on left side of an assign statement.
- wires need to be driven and cannot hold a value by themselves.
- wires can only be used to model combinational logic.
- regs and wires can be used as outpus within the actual module definition.
- regs cannot be used as inputs within the module definition.
- regs can be used to model combinational and sequential logic.

In Verilog, always statements, signals keep their old value until an event in the sensitivity list takes place that explicitly causes them to change. In contrast, Verilog continuous assignment statements (assign) are reevaluated anytime any of the inputs on the right hand side changes. Therefore, such code necessarily describes combinational logic.

```
module flop (input clk,
             input [3:0] d,
             output reg [3:0] q);

      always @ (posedge clk)
             q <= d;

endmodule
```

The statement is executed only when the event specified in the sensitivity list (in brackets after the @ sign) occurs. The statement in this case is $q <= d$, which is read as "q gets d".
<= is a nonblocking assignment, which is written instead of assign in an always block. All signals on the left hand side of <= or = in an always block must be declared as reg. However, this does not mean that the signal is actually the output of a register.

```
module latch (input clk,
             input [3:0] d,
             output reg [3:0] q);

      always @ (clk, d)
             if (clk) q <= d;

endmodule
```

This sensitivity list contains both clk and d, so the always statement evaluates every time either clk or d changes.

```
module inv (input [3:0] a,
             output reg [3:0] y);

      always @ (*)
             y = ~a;

endmodule
```

This sensitivity list containing the * symbol indicates, that the always statement is evaluated every time any of the signals on the right hand side of <= or = inside the always statement change. This can be used to model combinational logic using always blocks, which can be easier to write than discrete logic gates.
The = indicates a blocking assignment. A group of blocking assignments are evaluated in the order they appear in the code, whilst a group of nonblocking assignments are evaluated concurrently, before any of the statements on the left hand sides are updated. It is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic.

4.5.2 Case Statements

Case statements must always appear inside always blocks.

```
module sevenseg (input [3:0] data,
                 output reg [6:0] segments);

        always @ (*)
                case (data)
                  0: segments = 7'b111_1110;
                  1: segments = 7'b011_0000;
                  2: segments = 7'b110_1101;
                  3: segments = 7'b111_1001;
                  4: segments = 7'b011_0011;
                  5: segments = 7'b101_1011;
                  6: segments = 7'b101_1111;
                  7: segments = 7'b111_0000;
                  8: segments = 7'b111_1111;
                  9: segments = 7'b111_1011;
                  default: segments = 7'b000_0000;
                endcase

endmodule
```

4.5.3 If Statements

```
module priority (input [3:0] a,
                 output reg [3:0] y);

    always @ (*)
        if  (a[3]) y = 4'b1000;
        else if (a[2]) y = 4'b0100;
        else if (a[1]) y = 4'b0010;
        else if (a[0]) y = 4'b0001;
        else y = 4'b0000;

endmodule
```

In Verilog, if statements must appear inside always blocks.

4.5.4 Blocking and Nonblocking Assignment Guidelines

1. Use always @ (posedge clk) and nonblocking assignments to model synchronous sequential logic
2. Use continuous assignments to model simple combinational logic
3. Use always @ (*) and blocking assignments to model more complicated combinational logic where the always statement is helpful
4. Do not make assignments to the same signal in more than one always statement or continuous assignment statement.

4.6 Testbenches

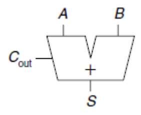
A testbench is a module that is used to test another module (the device under test, DUT). The input and desired output patterns are called test vectors. Testbenches are not synthesizable. There are different kinds of testbenches, namely simple testbenches, self-checking testbenches and self-checking testbenches with testvectors.

5 Digital Building Blocks

5.1 Arithmetic Circuits

5.1.1 Addition

A 1-bit half adder has two inputs, A and B, and two outputs, S and C_{out}. S is the sum of A and B. When both A and B are 1, the result is 0 and the carry out bit, C_{out} is set to 1. The half adder can be built from a XOR gate and an AND gate.
In a multi-bit adder, C_{out} is added, or carried in to the next most significant bit.



A full adder is very similar to a half adder, but has an additional input, C_{in} to accept C_{out} of the previous column, leading us to the following truth table:

C_{in}	A	B	C_{out}	S
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

A carry propagate adder sums two N-bit inputs, A and B, and a carry in, C_{in} to produce an N-bit result, S, and a carry out, C_{out}. It is called a carry propagate adder, because the carry out of one bit propagates into the next bit.

A ripple-carry adder is basically a N-bit carry propagate adder, where the C_{out} output of one stage acts as the C_{in} input of the next stage. So it is in principle just a long chain of full adders.
It has the advantage of being reusable for many systems, but it gets really slow when N is a large number.

A carry-lookahead adder is a faster version, because it divides the adder into blocks and provides circuitry to quickly determine the carry out of a block as soon as the carry in is known. They use generate (G) and propagate (P) signals that describe how a column or block determines a carry out. The ith column of an adder is said to generate a carry if it produces a carry out, independent of the carry in. This is guaranteed if both A and B are 1. The column is said to propagate a carry if it produces a carry out whenever there is a carry in. This is much faster than a ripple-carry adder, however, the adder delay still increases linearly with N.

5.1.2 Subtraction

Adders can add positive and negative numbers, using two's complement number representation. For subtraction, simply flip the sign of the second number and add the two. Recall that this is done by inverting the bits and then adding 1.

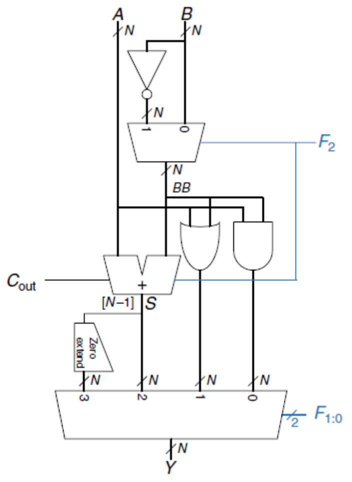
5.1.3 Comparators

A comparator determines whether two binary numbers are equal or if one is greater or less than the other. An equality comparator produces a single output indicating whether A is equal to B. A magnitude comparator produces one or more outputs indicating the relative values of A and B. This is usually done by computing A-B and looking at the sign of the result.

5.1.4 ALU

An Arithmetic/Logical Unit combines several mathematical and logical operations into a single unit. For example, it might perform addition, subtraction, magnitude comparison, AND and OR operations. An ALU receives a control signal F, which indicates which function to perform.
The SLT (set less than) function is used for magnitude comparison. When A < B, then Y = 1, otherwise Y = 0.
Some ALUs produce extra outputs, flags, that indicate information about the ALU output. For example an overflow flag indicates that the result of the adder overflowed. A zero flag indicates that the output of the ALU is 0.

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A - B
111	SLT



5.1.5 Shifters and Rotators

Shifters and rotators move bits and multiply or divide them by powers of 2.

A logical shifter (<<, >>) shifts the number to the left (LSL) or right (LSR) and fills empty spots with 0's:
11001 LSR 2 = 00110; 11001 LSL 2 = 00100

An arithmetic shifter (<<<, >>>) is the same as a logical shifter, but when the number is shifter to the right, the most significant bits are filled with a copy of the old most significant bit.
11001 ASR 2 = 11110; 11001 ASL 2 = 00100

A rotator rotates the number in a circle such that empty spots are filled with bits shifted off the other end.
11001 ROR 2 = 01110; 11001 ROL 2 = 00111

5.2 Number Systems

5.2.1 Fixed-Point Number Systems

The fixed-point notation has an implied binary point between the integer and fraction bits:
$01101100 = 0110.1100 = 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$
Signed fixed-point numbers can use either two's complement or sign/magnitude notations.
There is no way of knowing the existence of the binary point except by agreement.

5.2.2 Floating-Point Number Systems

Floating-point numbers have a sign, mantissa (M), base (B) and an exponent (E). The decimal point floats to the position right after the most significant digit. They are base 2 with a binary mantissa and

32 bits are used to represent 1 sign bit, 8 exponent bits and 23 fraction (mantissa) bits.
To show the floating point representation of 228, the number is first converted into binary: $228_{10} = 11100100_2$. Then the floating point is set to the position right after the most significant bit: $11100100_2 = 1.11001_2 \text{ x } 2^7$. The 32 bit encoding would look like this: 0 10000110 11100100000000000000000, where the first bit is the sign bit, the following 8 bits are the biased exponent and the last 23 bits are the fraction.
The leading digit in this representation is always 1 and is therefore not stored. It is called the implicit leading one. The exponent is biased, because it has to represent both negative and positive exponents. 32-bit floating-point uses a bias of 127, so in this case the biased exponent is $7+127=134=10000110_2$.

Special cases:

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

32-bit floating-point numbers are called single-precision. 64-bit numbers are called double-precision and consist of 1 sign bit, 11 exponent bits and 52 fraction bits.

5.2.3 Floating-Point Addition

Add 0 10000001 11111000000000000000000 and 0 01111100 10000000000000000000000

- Extract exponent and fraction bits
  10000001 111 1100 0000 0000 0000 0000
  01111100 100 0000 0000 0000 0000 0000
- Prepend leading 1 to form the mantissa
  10000001 1.111 1100 0000 0000 0000 0000
  01111100 1.100 0000 0000 0000 0000 0000
- Compare exponents

```
 10000001 1.111 1100 0000 0000 0000 0000
-01111100 1.100 0000 0000 0000 0000 0000
      101 (shift amount)
```

- Shift smaller mantissa if necessary
  10000001 1.111 1100 0000 0000 0000 0000
  10000001 0.000 0110 0000 0000 0000 0000 00000
- Add mantissas

```
10000001   1.111 1100 0000 0000 0000 0000
10000001 +0.000 0110 0000 0000 0000 0000
          10.000 0010 0000 0000 0000 0000
```

- Normalize mantissa and adjust exponent if necessary

```
 10000001 10.000 0010 0000 0000 0000 0000 >> 1
+       1
 10000010 1.000 0001 0000 0000 0000 0000
```

- Round result
  (no rounding necessary)
- Assemble exponent and fraction back into floating-point number
  0 10000010 000 0001 0000 0000 0000 0000

5.3 Sequential Building Blocks

5.3.1 Counters

An N-bit binary counter is a sequential arithmetic circuit witch clock and reset inputs and an N-bit output. Reset initializes the output to 0. The counter then advances through all 2^N possible outputs in binary order, incrementing on the rising edge of the clock.

5.3.2 Shift Register

A shift register has a clock, a serial input S_{in}, a serial output S_{out}, and N parallel output, $Q_{N-1:0}$. On each rising edge of the clock, a new bit is shifted in from S_{in} and all the subsequent contents are shifted forward. The last bit in the shift register is available at s_{out}.
Shift registers can also be viewed as serial-to-parallel converters. This concept can also be modified to work as a parallel-to-serial converter, taking N inputs at a time and releasing them sequentially.

5.4 Memory Arrays

Memory arrays can efficiently store large amounts of data.
A generic memory array is organized as a two-dimensional array of memory cells. The memory reads or writes the contents of one of the rows of the array. The row is specified by an address. An array with N-bit addresses and M-bit data has 2^N rows and M columns. Each row of data is a word. The depth of an array is the number of rows and the width is the number of columns, also known as the word size.

Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

Memory arrays are built as an array of bit cells, each of which stores 1 bit of data. Each bit cell is connected to a wordline (row) and a bitline (column). For each combination of address bits, the memory asserts a single wordline that activates the bit cells in that row. When the wordline is HIGH, the store bit transfers to or from the bitline. Otherwise, the bitline is disconnected from the bit cell.
To read the bit cell, the bitline is initially left floating (z). The wordline is then turned ON, allowing the stored value to drive the bitline to 0 or 1. To write a bit cell, the bitline is strongly driven to the desired value. Then the wordline is turned ON, connecting the bitline to the stored bit. The bitline then overpowers the contents of the bit cell, writing the desired value into the stored bit.

During a memory read, a wordline is asserted, and the corresponding row of bit cells drives the bitlines HIGH or LOW. During a memory write, the bitlines are driven HIGH or LOW first and then a wordline is asserted, allowing the bitline values to be stored in that row of bit cells.

All memories have one or more ports. Each port gives read and/or write access to one memory address. Multiported memories can access several addresses simultaneously.

Memories are classified based on how they store bits in the bit cell. The broadest classification is random access memory (RAM) versus read only memory (ROM).
RAM is volatile, which means that the data is lost when the power is switched off. ROM on the other hand is nonvolatile, meaning that it keeps the data indefinitely, even without a power source.
The two major types of RAM are dynamic RAM (DRAM) and static RAM (SRAM). DRAM stores data as a charge on a capacitor and SRAM stores data using a pair of cross-coupled inverters.

5.4.1 Dynamic Random Access Memory

DRAM stores a bit as the presence or absence of charge on a capacitor. The bit value is stored on a capacitor, and a nMOS transistor behaves as a switch that either connects or disconnects the capacitor from the bitline. When the wordline is asserted, the nMOS transistor turns ON, and the stored bit value transfers to or from the bitline.
Upon a read, data values are transferred from the capacitor to the bitline. Upon a write, data values are transferred from the bitline to the capacitor. Reading destroys the bit value stored on the capacitor, so the data must be rewritten after each read. Even when the DRAM is not read, the data must be rewritten after a few milliseconds anyway, because the charge of the capacitor gradually leaks over time.

5.4.2 Static Random Access Memory

SRAM is static, because stored bits do not need to be refreshed. A data bit is stored on cross-coupled inverters. Each cell has two outputs, bitline and $\overline{\textit{bitline}}$. When the wordline is asserted, two nMOS transistors turn on, and data values are transferred to or from the bitlines. If noise degrades the value of the stored bit, the cross-coupled inverters restore the value.

5.4.3 Register Files

Digital systems often use a number of registers to store temporary variables. This register file is usually built as a small, multiported SRAM array.

5.4.4 Read Only Memory

ROM stores a bit as the presence or absence of a transistor. To read the cell, the bitline is weakly pulled HIGH, and then the wordline is turned ON. If the transistor is present, it pulls the bitline LOW. If it is absent, the bitline remains HIGH.
The ROM bit cell is a combinational circuit and has no state to forget if power is turned off.
Modern ROMs are not really read only, they can be programmed (written) as well. The difference between RAM and ROM is that ROMs take a longer time to write, but are nonvolatile.

5.4.5 Logic Using Memory Arrays

Memory arrays can also perform combinational logic functions. Memory arrays used to perform logic are called lookup tables (LUTs). Using memory to perform logic, the user can look up the output value for a given input combination (address). Each address corresponds to a row in the truth table, and each data bit corresponds to an output value.

5.4.6 Memory HDL

```
module ram # (parameter N=6, M=32)
             (input clk,
             input we,
             input [N-1:0] adr,
             input [M-1:0] din,
             output [M-1:0] dout);

    reg [M-1:0] mem [2**N-1:0];

    always @ (posedge clk)
        if (we) mem [adr] <= din;

    assign dout = mem[adr];

endmodule

module rom (input [1:0] adr,
             output reg [2:0] dout);

    always @ (adr)
        case (adr)
        2'b00: dout <= 3'b011;
        2'b01: dout <= 3'b110;
        2'b10: dout <= 3'b100;
        2'b11: dout <= 3'b010;
    endcase

endmodule
```

5.5 Logic Arrays

Gates can be organized into regular arrays. These logic arrays can be configured to perform any function without the user having to connect wires in specific ways.

5.5.1 Programmable Logic Array

PLAs implement two-level combinational logic in sum-of-products form. PLAs are built from an AND array, followed by an OR array. The inputs drive an AND array, which produces implicants, which in turn are ORed together to form the outputs. An M x N x P-bit PLA has M inputs, N implicants and P outputs.

Programmable logic devices (PLDs) are souped-up PLAs that add registers and various other features to the basic AND/OR planes.

5.5.2 Field Programmable Gate Array

A FPGA is an array of reconfigurable gates. A user can implement designs on the FPGA using either an HDL or a schematic. They can implement both combinational and sequential logic.
FPGAs are built as an array of configurable logic blocks (CLBs). Each CLB can be configured to perform sequential or combinational functions. The CLBs are surrounded by Input/Output blocks (IOBs) for interfacing with external devices. The IOBs connect CLB inputs and outputs to pins on the chip package. CLBs can connect to other CLBs and IOBs through programmable routing channels. The CLB contains lookup tables (LUTs), configurable multiplexers and registers. The FPGA is configured by specifying the contents of the lookup tables and the select signals for the multiplexers.

The CLB can perform up to two combinational and/or two registered functions. All of the functions can involve at least four variables, and some can involve up to nine.

5.6 Exercises

1. Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks. Assume that each two-input gate delay is 100 ps and that a full adder delay is 300 ps.
2. Configure a 32-bit ALU for the SLT operation. Suppose A = 25_{10} and B = 32_{10}. Show the control signals and output, Y.
3. Compute 0.75 + -0.625 using fixed-point numbers.
4. Show the floating-point representation of the decimal number 228.
5. Explain how to configure a CLB to perform X = $\overline{A}\,\overline{B}$C + AB$\overline{C}$ and Y = A$\overline{B}$
6. Alyssa is building a finite state machine that must run at 200 MHz. She uses a Spartan 3 FPGA with the following specifications: t_{CLB} = 0.61 ns per CLB; t_{setup} = 0.53 ns and t_{pcq} = 0.72 ns for all flip-flops. What is the maximum number of CLBs her design can use? You can ignore interconnect delay.

6 Architecture

6.1 Assembly Language

Assembly language is the human-readable representation of the computer's native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate.

6.1.1 Instructions

Operation	High-Level Code	MIPS Assembly Code
Addition	a = b + c;	add a, b, c
Subtraction	a = b - c;	sub a, b, c
Register Operands	a = b + c;	#$s0 = a, $s1 = b, $s2 = c add $s0, $s1, $s2
Temporary Registers	a = b + c - d;	#$s0 = a, $s1 = b, $s2 = c, $s3 = d sub $t0, $s2, $s3 add $s0, $s1, $t0
Reading word-addressable memory		#read memory word 1 into $s3 lw $s3, 1($0)
Writing word-addressable memory		#write $s7 into memory word 5 sw $s7, 5($0)
Accessing byte-addressable memory		lw $s0, 0($0) #read data word 0 into $s0 lw $s1, 8($0) #read data word 2 into $s1 lw $s2, 0xC($0) #read data word 3 into $s2 sw $s3, 4($0) #write $s3 to data word 1 sw $s4, 0x20($0) #write $s4 to data word 8 sw $s5, 400($0) #write $s5 to data word 100

Immediate Operands	a = a + 4;
	b = a - 12;
		#$s0 = a, $s1 = b
		addi $s0, $s0, 4
		addi $s1, $s0, -12

The first part of the assembly instruction is called the mnemonic and is used to indicate what operation to perform. The operation is performed on the source operands, b and c, and is written to the destination operand, a.
Comments begin with the # symbol.

You are studying a MIPS architecture for this digital circuits course. The MIPS instruction set makes the common case fast by including only simple, commonly used instructions. MIPS is a reduced instruction set computer (RISC) architecture. Architectures with many complex instructions are complex instruction set computer (CISC) architectures.
You can find a very comprehensive overview of the MIPS assembly language here:
`https://learnxinyminutes.com/docs/mips/`

6.1.2 Operands: Registers, Memory and Constants

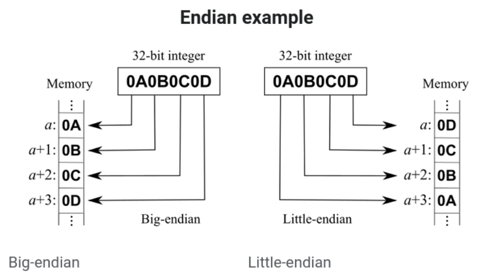
Operands can be stored in registers or memory, or as constants in the instruction itself. Operands stored as constants or in registers are accessed quickly, but can only store a small amount of data. MIPS is called a 32-bit architecture, because it operates on 32-bit data.

The MIPS architecture uses 32 registers, called the register file or register set. Common operands which have to be accessed quickly are stored there. A small register file is typically built from a small SRAM array. The SRAM array uses a small decoder and bitlines connected to relatively few memory cells. MIPS register files are preceded by the $ sign. MIPS generally stores variables in 18 of the 32 registers: $s0 - $s7, and $t0 - $t9. Registers beginning with $s are called saved registers. These store variables like a, b and c. Registers beginning with $t are called temporary registers and they are used to store temporary variables.

Name	Number	Use
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0 - \$v1	2 - 3	procedure return values
\$a0 - \$a3	4 - 7	procedure arguments
\$t0 - \$t7	8 - 15	temporary variables
\$s0 - \$s7	16 - 23	saved variables
\$t8 - \$t9	24 - 25	temporary variables
\$k0 - \$k1	26 - 27	operating system temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	procedure return address

MIPS uses a byte-addressable memory. This means that every byte in memory has a unique address. However, for the sake of explanation, first assume a word-addressable memory. This means that every 32-bit data word has a unique address. By convention, memory is drawn with low memory addresses towards the bottom and high memory addresses towards the top.
MIPS uses the load word instruction, lw, to read a data word from memory into a register. The lw instruction specifies the effective address as the sum of a base address and an offset. The base address is a register, the offset is a constant.
MIPS uses the store word instruction, sw, to write a data word from a register into memory. Word addresses for lw and sw must be word aligned, meaning that the address must be divisible by 4.

Byte-addressable memories are organized in big-endian or little-endian fashion. In both formats, the most significant byte is on the left and the least significant byte is on the right.



The constants used in MIPS instructions are called immediates. The immediate specified in an instruction is a 16-bit two's complement number. Subtracting is the same as adding a negative number, so for the sake of simplicity there is an addi instruction, but no subi instruction.

Here is a summary of the most important concepts:

- Design Principle 1: Simplicity favors regularity
- Design Principle 2: Make the common case fast
- Design Principle 3: Smaller is faster
- Design Principle 4: Good design demands good compromises

6.2 Machine Language

MIPS defines three instruction formats: R-type, I-type and J-type. R-type instructions operate on three registers, I-type instructions operate on two registers and a 16-bit immediate and J-type (jump) instructions operate on one 26-bit immediate.

R-Type is short for register type. They use three registers as operands: two as sources and one as a destination. The 32-bit instruction has 6 fields:
op (6 bits) | rs (5 bits) | rt (5 bits) | rd (5 bits) | shamt (5 bits) | funct (6 bits)

I-Type is short for immediate type. They use two register operands and one immediate operand. The 32-bit instruction has 4 fields:
op (6 bits) | rs (5 bits) | rt (5 bits) | imm (16 bits)

J-Type is short for jump type. This format is only used with jump instructions. This instruction uses a single 26-bit address operand, addr.
op (6 bits) | addr (26 bits)

6.2.1 The Power of the Stored Programm

A program written in machine language is a series of 32-bit numbers representing the instructions. These instructions can be stored in memory. This is called the stored program concept. Running a different program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing the new program to memory. Instructions in a stored program are retrieved, or fetched, from memory and executed by the processor. Even large, complex programs are simplified to a series of memory reads and instruction executions.

To run or execute the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the program counter (PC).

The architectural state of a microprocessor holds the state of a program. For MIPS, the architectural state consists of the register file and PC. If the operating system saves the architectural state at some point in the program, it can interrupt the program, do something else, then restore the state such that the program continues properly, unaware that it was ever interrupted.

6.3 Programming

6.3.1 Logical Instructions

Operation	Assembly Code
AND	and \$s3, \$s1, \$s2
OR	or \$s4, \$s1, \$s2
XOR	xor \$s5, \$s1, \$s2
NOR	nor \$s6, \$s1, \$s2
AND (I)	andi \$s2, \$s1, 0xFA34
OR (I)	ori \$s3, \$s1, 0xFA34
XOR (I)	xori \$s4, \$s1, 0xFA34

6.3.2 Shift Instructions

Shift instructions shift the value in a register left or right by up to 31 bits. Shift operations multiply or divide by powers of two. MIPS shift operations are sll (shift left logical), srl (shift right logical), and sra (shift right arithmetic). Shifting a value left by N is equivalent to multiplying it by 2^N. Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N
MIPS also has variable-shift instructions: sllv (shift left logical variable), srlv (shift right logical variable), and srav (shift right arithmetic variable).

Operation	Assembly Code
SLL	sll $t0, $s1, 4
SRL	srl $s2, $s1, 4
SRA	sra $s3, $s1, 4
SLLV	sllv $s3, $s1, $s2
SRLV	srlv $s4, $s1, $s2
SRAV	srav $s5, $s1, $s2

6.3.3 Generating Constants

To assign 32-bit constants, use a load upper immediate instruction (lui) followed by an or immediate (ori) instruction. lui loads a 16-bit immediate into the upper half of a register and sets the lower half to 0. ori merges a 16-bit immediate into the lower half.

Operation	High-Level Code	MIPS Assembly Code
16-Bit Constant	int a = 0x4f3c;	#$s0 = a addi $s0, $0, 0x4f3c
32-Bit Constant	int a = 0x6d5e4f3c;	#$s0 = a lui $s0, 0x6d5e #a 0x6d5e0000 ori $s0, $s0, 0x4f3c #a 0x6d5e4f3c

6.3.4 Multiplication and Division Instructions

Multiplication and division are different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. Dividing two 32-bit numbers produces a 32-bit quotient and a 32-bit remainder. The MIPS architecture has two special-purpose registers, hi and lo, which are used to hold the results of multiplication and division. mult $s0$,s1 multiplies the values in $s0and$s1. The 32 most significant bits are placed in hi and the 32 least significant bits are placed in lo. Similarly, div $s0$,s1 computes $s0$/s1. The quotient is placed in lo and the remainder is placed in hi.

6.3.5 Branching

To sequentially execute instructions, the program counter increments by 4 after each instruction. Branch instructions modify the program counter to skip over sections of code or to go back to repeat previous code. Conditional branch instructions perform a test and branch only if the test is TRUE. Unconditional branch instructions, called jumps, always branch.

The MIPS instruction set has two conditional branch instructions: branch if equal (beq) and branch if not equal (bne). Assembly code uses labels to indicate instruction locations in the program:

```
addi $s0, $0, 4          #$s0 = 0 + 4 = 4
addi $s1, $0, 1          #$s1 = 0 + 1 = 1
sll $s1, $s1, 2          #$s1 = 1 + 2 = 4
beq $s0, $s1, target     #$s0 == $s1, so branch is taken
addi $s1, $s1, 1         #not executed
sub $s1, $s1, $s0        #not executed
target:
add $s1, $s1, $s0        #$s1 = 4 + 4 = 8
```

A program can unconditionally branch, or jump, using the three types of jump instructions: jump (j), jump and link (jal), and jump register (jr). Jump (j) jumps directly to the instruction at the specified label. Jump and link (jal) is similar to j but is used by procedures to save a return address. Jump register (jr) jumps to the address held in a register:

```
addi $s0, $0, 4          #$s0 = 4
addi $s1, $0, 1          #$s1 = 1
j target                 #jump to target
addi $s1, $s1, 1         #not executed
sub $s1, $s1, $s0        #not executed
target:
add $s1, $s1, $s0        #$s1 = 1 + 4 = 5
```

6.3.6 Switch/Case Statements

```
# $s0 = amount, $s1 = fee
case20:
addi $t0, $0, 20           #$t0 = 20
bne $s0, $t0, case50       #i = 20? if not, skip to case50
addi $s1, $0, 2            #if so, fee = 2
j done                     #and break out of case
case50:
addi $t0, $0, 50           #$t0 = 50
bne $s0, $t0, case100      #i = 50? if not, skip to case100
addi $s1, $0, 3            #if so, fee = 3
j done                     #and break out of case
case100:
addi $t0, $0, 100          #$t0 = 100
bne $s0, $t0, default      #i = 100? if not, skip to default
addi $s1, $0, 5            #if so, fee = 5
j done                     #and break out of case
default:
add $s1, $0, $0            #charge = 0
done:
```

6.3.7 Loops

```
# $s0 = pow, $s1 = x
addi $s0, $0, 1            #pow = 1
addi $s1, $0, 0            #x = 0
addi $t0, $0, 128          #t0 = 128 for comparison
while:
beq $s0, $t0, done         #if pow = 128, exit while
sll $s0, $s0, 1            #pow = pow * 2
addi $s1, $s1, 1           #x = x + 1
j while
done:

# $s0 = i, $s1 = sum
add $s1, $0, $0            #sum = 0
addi $s0, $0, 0            #i = 0
addi $t0, $0, 10           #$t0 = 10
for:
beq $s0, $t0, done         #if i = 10, branch to done
add $s1, $s1, $s0          #sum = sum + i
addi $s0, $s0, 1           #increment i
j for done:
```

6.3.8 Arrays

```
# $s0 = base address of array
lui $s0, 0x1000              #$s0 = 0x10000000
ori $s0, $s0, 0x7000         #$s0 = 0x10007000
lw $t1, 0($s0)               #$t1 = array[0]
sll $t1, $t1, 3              #$t1 $t1 << 3 = $t1 * 8
sw $t1, 0($s0)               #array[0] = $t1
lw $t1, 4($s0)               #$t1 = array[1]
sll $t1, $t1, 3              #$t1 = $t1 << 3 = $t1 * 8
sw $t1, 4($s0)               #array[1] = $t1
```

6.3.9 The Stack

The stack is memory that is used to save local variables within a procedure. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. The stack is a last-in-first-out (LIFO) queue. Each procedure may allocate stack space to store local variables but must deallocate it before returning. The top of the stack, is the most recently allocated space. The stack expands to lower memory addresses when a program needs more scratch space. The stack pointer, $sp, is a special MIPS register that points to the top of the stack. A pointer is a fancy name for a memory address. It points to (gives the address of) data.

One of the important uses of the stack is to save and restore registers that are used by a procedure. A procedure should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides the one containing the return value, $v0. To avoid that, a procedure saves registers on the stack before it modifies them, then restores them from the stack before it returns. Specifically, it performs the following steps:

- Makes space on the stack to store the values of one or more registers
- Stores the values of the registers on the stack
- Executes the procedure using the registers
- Restores the original values of the registers from the stack
- Deallocates space on the stack

When the procedure returns, $v0 holds the result, but there are no other side effects: $s0, $t0, $t1, and $sp have the same values as they did before the procedure call. The stack space that a procedure allocates for itself is called its stack frame.

6.3.10 Preserved Registers

MIPS divides registers into preserved and nonpreserved categories. The preserved registers include $s0–$s7. The nonpreserved registers include $t0–$t9. A procedure must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Preserved	Nonpreserved
Saved Registers: $s0 - $s7	Temporary Registers: $t0 - $t9
Return Address: $ra	Argument Registers: $a0 - $a3
Stack Pointer: $sp	Return Value Registers: $v0 - $v1
Stack above the Stack Pointer	Stack below the Stack Pointer

6.4 Addressing Modes

Register-only addressing uses registers for all source and destination operands. All R-type instructions use register-only addressing.

Immediate addressing uses the 16-bit immediate along with registers as operands. Some I-type instructions, such as add immediate (addi) and load upper immediate (lui), use immediate addressing.

Memory access instructions, such as load word (lw) and store word (sw), use base addressing. The effective address of the memory operand is found by adding the base address in register rs to the sign-extended 16-bit offset found in the immediate field.

Conditional branch instructions use PC-relative addressing to specify the new value of the PC if the branch is taken. The signed offset in the immediate field is added to the PC to obtain the new PC; hence, the branch destination address is said to be relative to the current PC.

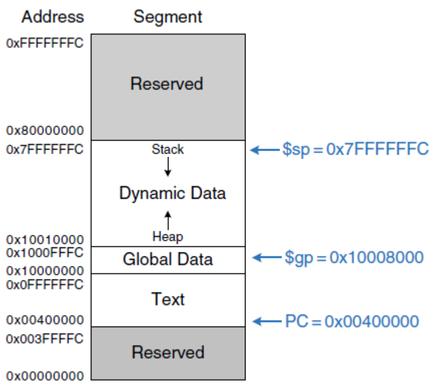
6.5 The Memory Map

With 32-bit addresses, the MIPS address space spans 2^{32} bytes 4 gigabytes (GB). Word addresses are divisible by 4 and range from 0 to 0xFFFFFFFC.

The text segment stores the machine language program. It is large enough to accommodate almost 256 MB of code. Note that the four most significant bits of the address in the text space are all 0, so the j instruction can directly jump to any address in the program.

The global data segment stores global variables that, in contrast to local variables, can be seen by all procedures in a program. Global variables are defined at start-up, before the program begins executing. These variables are declared outside the main procedure in a C program and can be accessed by any procedure. The global data segment is large enough to store 64 KB of global variables. Global variables are accessed using the global pointer ($gp), which is initialized to 0x100080000. Unlike the stack pointer ($sp), $gp does not change during program execution. Any global variable can be accessed with a 16-bit positive or negative offset from $gp. The offset is known at assembly time, so the variables can be efficiently accessed using base addressing mode with constant offsets.

The dynamic data segment holds the stack and the heap. The data in this segment are not known at start-up but are dynamically allocated and deallocated throughout the execution of the program. This is the largest segment of memory used by a program, spanning almost 2 GB of the address space. The heap stores data that is allocated by the program during runtime. The heap grows upward from the bottom of the dynamic data segment. If the stack and heap ever grow into each other, the program's data can become corrupted. The memory allocator tries to ensure that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data. The reserved segments are used by the operating system and cannot directly be used by the program.



6.6 Very Long Instruction Word (VLIW)

Very long instruction word is an architecture optimization employed in some ISAs that allows the exploitation of instruction-level parallelism (ILP) by combining multiple *independent* instructions into a single,

”very long”, instruction word. This combined instruction word, which may contain a variable number of instructions, is then parsed as a unit by the processor, which typically contains multiple functional units and/or memory ports to be able to handle the multiple concurrent instructions.
One benefit of VLIW is that multiple instructions can be grouped together explicitly by the compiler, enabling high performance with low complexity.

6.7 Exercises

1. Translate the following high-level code into assembly language. Assume variables a–c are held in registers \$s0–\$s2 and f–j are in \$s3–\$s7.

   a = b - c;

   f = (g + h) - (i + j);

2. Suppose that \$s0 initially contains 0x23456789. After the following program is run on a big-endian system, what value does \$s0 contain? In a little-endian system? lb \$s0, 1(\$0) loads the data at byte address (1 + \$0) = 1 into the least significant byte of \$s0.

   sw \$s0, 0(\$0)

   lb \$s0, 1(\$0)

3. The following high-level code converts a ten-entry array of characters from lower-case to upper-case by subtracting 32 from each array entry. Translate it into MIPS assembly language. Remember that the address difference between array elements is now 1 byte, not 4 bytes. Assume that \$s0 already holds the base address of chararray.

   char chararray[10];

   int i;

   for (i = 0; i != 10; i = i + 1) {chararray[i] = chararray[i] - 32;}

7 Microarchitecture

7.1 Performance Analysis

A particular processor architecture can have many microarchitectures with different cost and performance trade-offs.
The only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The computer that executes your program fastest has the highest performance. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run; this may be necessary if you haven't written your program yet or if somebody else who doesn't have your program is making the measurements. Such collections of programs are called benchmarks, and the execution times of these programs are commonly published to give some indication of how a processor performs.

The execution time of a program is given by:

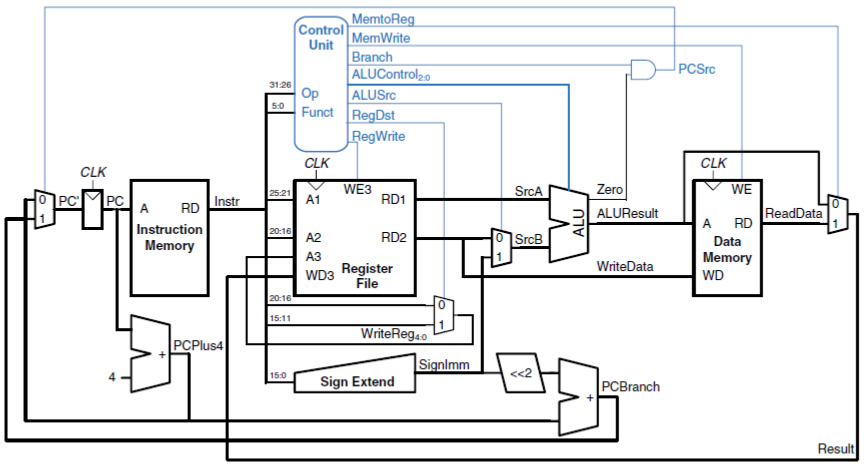
$$ExecutionTime = (\#instructions)(\frac{cycles}{instructions})(\frac{seconds}{cycle}) \tag{1}$$

The number of cycles per instruction, often called CPI, is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (instructions per cycle, or IPC).
The number of seconds per cycle is the clock period, T_c. The clock period is determined by the critical path through the logic on the processor.

7.2 Single-Cycle Processor

The program counter (PC) register contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. The PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or fetches, the 32-bit instruction, labeled Instr.
For a lw instruction, the next step is to read the source register containing the base address. This register is specified in the rs field of the instruction, $Instr_{25:21}$. These bits of the instruction are connected to the address input of one of the register file read ports, A1. The register file reads the register value onto RD1.
The lw instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{15:0}$. Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits. The 32-bit sign-extended value is called SignImm. The processor must add the base address to the offset to find the address to read from memory. The ALU receives two operands, SrcA and SrcB. SrcA comes from the register file, and SrcB comes from the sign-extended immediate. The ALU can perform many operations. The 3-bit ALUControl signal specifies the operation. The ALU generates a 32-bit ALUResult and a Zero flag, that indicates whether ALUResult = 0. For a lw instruction, the ALUControl signal should be set to 010 to add the base address and offset. ALUResult is sent to the data memory as the address for the load instruction. The data is read from the data memory onto the ReadData bus, then written back to the destination register in the register file at the end of the cycle. Port 3 of the register file is the write port. The destination register for the lw instruction is specified in the rt field, $Instr_{20:16}$, which is connected to the port 3 address input, A3, of the register file. The Read-Data bus is connected to the port 3 write data input, WD3, of the register file. A control signal called RegWrite is connected to the port 3 write enable input, WE3, and is asserted during a lw instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle. While the instruction is being executed, the processor must compute the address of the next instruction, PC. Because instructions are 32 bits = 4 bytes, the next instruction is at PC = 4. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the lw instruction.

Like the lw instruction, the sw instruction reads a base address from port 1 of the register and sign-extends an immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by the datapath. The sw instruction also reads a second register from the register file and writes it to the data memory. The register is specified in the rt field, $Instr_{20:16}$. These bits of the instruction are connected to the second register file read port, A2. The register value is read onto the RD2 port. It is connected to the write data port of the data memory. The write enable port of the data memory, WE, is controlled by MemWrite. For a sw instruction, MemWrite = 1, to write the data to memory; ALUControl = 010, to add the base address and offset; and RegWrite = 0, because nothing should be written to the register file. Note that data is still read from the address given to the data memory, but that this ReadData is ignored because RegWrite = 0. Next, consider extending the datapath to handle the R-type instructions add, sub, and, or, and slt. All of these instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register file. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware, using different ALUControl signals. The register file reads two registers. The ALU performs an operation on these two registers. the ALU always received its SrcB operand from the sign-extended immediate (SignImm). Now, we add a multiplexer to choose SrcB from either the register file RD2 port or SignImm. The multiplexer is controlled by a new signal, ALUSrc. ALUSrc is 0 for R-type instructions to choose SrcB from the register file; it is 1 for lw and sw to choose SignImm. The register file always got its write data from the data memory. However, R-type instructions write the ALUResult to the register file. Therefore, we add another multiplexer to choose between ReadData and ALUResult. We call its output Result. This multiplexer is controlled by another new signal, MemtoReg. MemtoReg is 0 for R-type instructions to choose Result from the ALUResult; it is 1 for lw to choose ReadData. We don't care about the value of MemtoReg for sw, because sw does not write to the register file. The register to write was specified by the rt field of the instruction, Instr$_{20:16}$. However, for R-type instructions, the register is specified by the rd field, Instr$_{15:11}$. Thus, we add a third multiplexer to choose WriteReg from the appropriate field of the instruction. The multiplexer is controlled by RegDst. RegDst is 1 for R-type instructions to choose WriteReg from the rd field, Instr$_{15:11}$; it is 0 for lw to choose the rt field, Instr$_{20:16}$. We don't care about the value of RegDst for sw, because sw does not write to the register file. Finally, let us extend the datapath to handle beq. beq compares two registers. If they are equal, it takes the branch by adding the branch offset to the program counter. Recall that the offset is a positive or negative number, stored in the imm field of the instruction, Instr$_{31:26}$. The offset indicates the number of instructions to branch past. Hence, the immediate must be sign-extended and multiplied by 4 to get the new program counter value: PC' = PC + 4 + SignImm x 4. The next PC value for a taken branch, PCBranch, is computed by shifting SignImm left by 2 bits, then adding it to PCPlus4. The left shift by 2 is an easy way to multiply by 4, because a shift by a constant amount involves just wires. The two registers are compared by computing SrcA - SrcB using the ALU. If ALUResult is 0, as indicated by the Zero flag from the ALU, the registers are equal. We add a multiplexer to choose PC from either PCPlus4 or PCBranch. PCBranch is selected if the instruction is a branch and the Zero flag is asserted. Hence, Branch is 1 for beq and 0 for other instructions. For beq, ALUControl = 110, so the ALU performs a subtraction. ALUSrc = 0 to choose SrcB from the register file. RegWrite and MemWrite are 0, because a branch does not write to the register file or memory. We don't care about the values of RegDst and MemtoReg, because the register file is not written.



The control unit computes the control signals based on the opcode and funct fields of the instruction, Instr$_{31:26}$ and Instr$_{5:0}$. The main decoder computes most of the outputs from the opcode. It also determines a 2-bit ALUOp signal. The ALU decoder uses this ALUOp signal in conjunction with the funct field to compute ALUControl.

Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1. The cycle time is given by

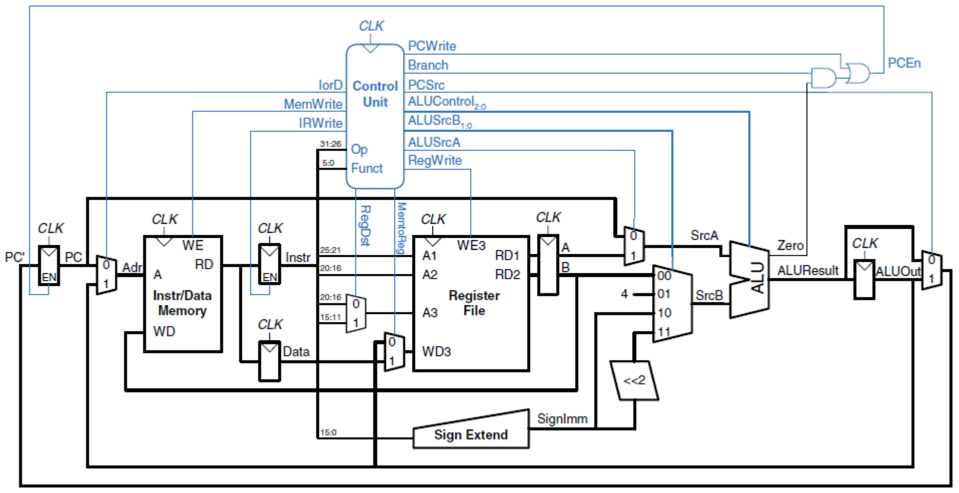
$$T_c = t_{\text{pcq\_PC}} + t_{\text{mem}} + max[t_{\text{RFread}}, t_{\text{sext}}] + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{\text{RFsetup}} \tag{2}$$

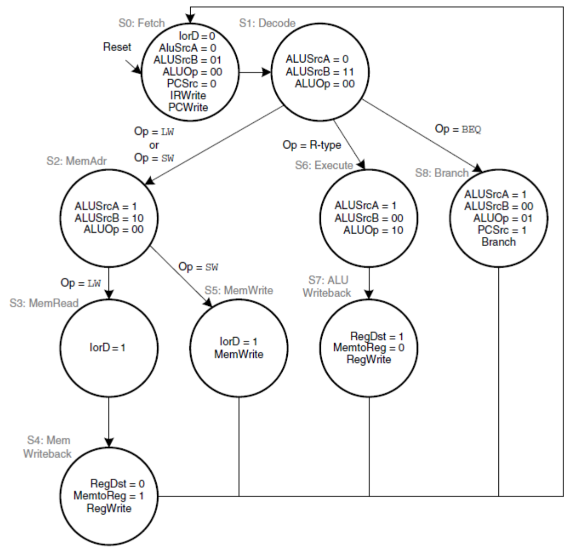
In most implementation technologies, the ALU, memory, and register file accesses are substantially slower than other operations. Therefore, the cycle time simplifies to

$$T_c = t_{\text{pcq\_PC}} + 2t_{\text{mem}} + t_{\text{RFread}} + 2t_{\text{mux}} + t_{\text{ALU}} + t_{\text{RFsetup}} \tag{3}$$

7.3 Multicycle Processors

The multicycle processor breaks an instruction into multiple shorter steps. In each short step, the processor can read or write the memory or register file or use the ALU. Different instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones. The processor needs only one adder; this adder is reused for different purposes on various steps. And the processor uses a combined memory for instructions and data. The instruction is fetched from memory on the first step, and data may be read or written on later steps.





The execution time of an instruction depends on both the number of cycles it uses and the cycle time. Whereas the single-cycle processor performed all instructions in one cycle, the multicycle processor uses varying numbers of cycles for the various instructions. However, the multicycle processor does less work in a single cycle and, thus, has a shorter cycle time:

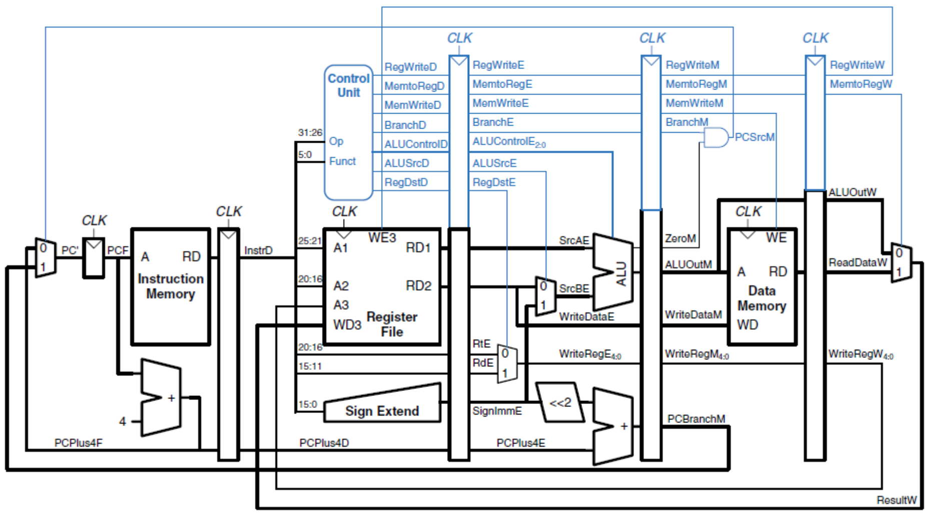
$$T_c = t_{\text{pcq}} + t_{\text{mux}} + max(t_{\text{ALU}} + t_{\text{mux}}, t_{\text{mem}}) + t_{\text{setup}} \tag{4}$$

7.4 Pipelined Processors

You design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is almost five times faster. Hence, the latency of each instruction is ideally unchanged, but the throughput is ideally five times better. Pipelining introduces some overhead, so the throughput will not be quite as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

Reading and writing the memory and register file and using the ALU typically constitute the biggest delays in the processor. We choose five pipeline stages so that each stage involves exactly one of these slow steps. Specifically, we call the five stages Fetch, Decode, Execute, Memory, and Writeback. They are similar to the five steps that the multicycle processor used to perform lw. In the Fetch stage, the processor reads the instruction from instruction memory. In the Decode stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the Execute stage, the processor performs a computation with the ALU. In the Memory stage, the processor reads or writes data memory. Finally, in the Writeback stage, the processor writes the result to the register file, when applicable. We assume that in the pipelined processor, the register file is written in the first part of a cycle and read in the second part, as suggested by the shading. This way, data can be written and read back within a single cycle.

The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers. The register file is peculiar because it is read in the Decode stage and written in the Writeback stage. It is drawn in the Decode stage, but the write address and data come from the Writeback stage. This feedback will lead to pipeline hazards.



The pipelined processor takes the same control signals as the single-cycle processor and therefore uses the same control unit.

7.4.1 Hazards

In a pipelined system, multiple instructions are handled concurrently. When one instruction is dependent on the results of another that has not yet completed, a hazard occurs. The register file can be read and

written in the same cycle. Let us assume that the write takes place during the first half of the cycle and the read takes place during the second half of the cycle, so that a register can be written and read back in the same cycle without introducing a hazard.
When one instruction writes a register and subsequent instructions read this register, it is called a read after write (RAW) hazard.
Hazards are classified as data hazards or control hazards. A data hazard occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A control hazard occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place.

Some data hazards can be solved by forwarding (also called bypassing) a result from the Memory or Writeback stage to a dependent instruction in the Execute stage. This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage. Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage.
In summary, the function of the forwarding logic for SrcA is given below. The forwarding logic for SrcB (ForwardBE) is identical except that it checks rt rather than rs.

```
if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then ForwardAE == 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then ForwardAE == 01
else ForwardAE == 00
```

Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction, because its result can then be forwarded to the Execute stage of the next instruction. Unfortunately, the lw instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the lw instruction has a two-cycle latency, because a dependent instruction cannot use its result until two cycles later. The alternative solution is to stall the pipeline, holding up operation until the data is available.
In summary, stalling a stage is performed by disabling the pipeline register, so that the contents do not change. When a stage is stalled, all previous stages must also be stalled, so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared to prevent bogus information from propagating forward. Stalls degrade performance, so they should only be used when necessary. The logic to compute the stalls and flushes is:

```
lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE
StallF = StallD = FlushE = lwstall
```

The beq instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched. A solution is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor can throw out the instructions if the prediction was wrong. In particular, suppose that we predict that branches are not taken and simply continue executing the program in order. If the branch should have been taken, the three instructions following the branch must be flushed (discarded) by clearing the pipeline registers for those instructions. These wasted instruction cycles are called the branch misprediction penalty.

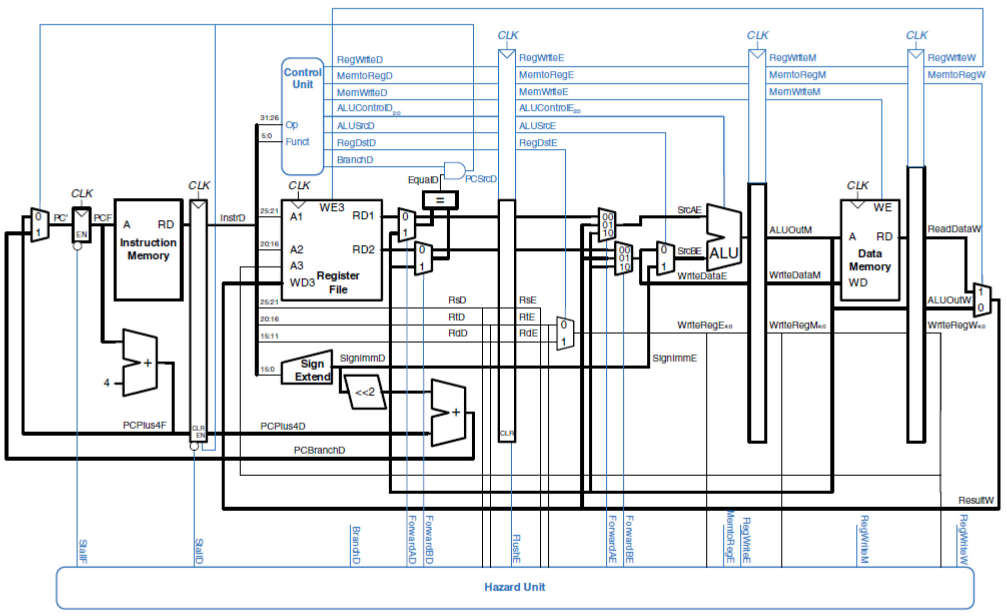


Figure 7.58 Pipelined processor with full hazard handling

7.5 Branch Prediction

Branches are costly operations: each time a branch occurs, the pipeline must be stalled until the branch is resolved. For this reason, a lot of effort has been dedicated to designing *branch predictors*, which attempt to predict the outcome of a given branch. If the prediction mechanism is sufficiently accurate, the processor's performance will improve.
The rationale is as follows: whenever the processor encounters a branch, it queries the branch predictor. The prediction is often given in the same cycle, as opposed to the usual two-cycle latency from branch resolution (under the assumption that branches are resolved in the Execute stage — note that this assumption does not hold for all microarchitectures). Since branches are relatively common instructions, the benefits quickly add up!
If the branch predictor predicts that the branch *will* be taken, the instruction indicated by the offset is fetched; if it predicts that the branch *will not* be taken, the next instruction is fetched from PC + 4 (i.e., the memory location consecutive to the branch).
When the branch is eventually resolved, the processor *verifies* whether the prediction was correct. If the prediction was correct, program execution continues as normal. If the prediction was incorrect, it is necessary to 1) flush from the pipeline the instructions that should not have been fetched, 2) fetch the correct next instruction, and 3) continue the execution of the program from there as normal.
Below are some important branch prediction strategies:

- **Static Branch Predictor:** always predict all branches as *either* taken, or not taken.
- **Last Time Branch Predictor:** keep track of the resolution status of the previously-seen branch. Predict that the same will happen for the current branch. There are two variants to this approach: the history may be *local* (when each PC address has its own history) or *global* (when you consider the last branch, regardless of its PC address).
- **Backward Taken, Forward Not Taken (BTFN):** as the name implies, this predictor always predicts that branches whose target is an earlier position in the code (backward = lower PC) will be taken, and that branches whose target is a later position (forward = higher PC) will not be taken. If your code makes heavy use of loops, this strategy will perform well (note that, during a loop, branches that point backward will often be taken).

- **Forward Taken, Backward Not Taken (FTBN):** this is similar to the previous policy, but the directions are flipped: here, you predict all branches that point to later positions in the code as taken, and branches that point to earlier positions as not taken.

- **Two-bit-Counter-Based Prediction:** use a saturating two-bit counter to keep track of the tendency of a branch to either be taken or not taken, based on the recent branching history. The counter can hold one of the following values: 00 (strongly not taken), 01 (weakly not taken), 10 (weakly taken) or 11 (strongly taken). When a branch is taken, you add 1 to its corresponding counter. Analogously, when a branch is not taken, you subtract 1 from the counter. Because the counter is *saturating*, adding 1 to 11 results in 11 and subtracting 1 from 00 results in 00. Everything else operates normally. The idea here is that if a branch has a preferential tendency to be either taken or not taken, that tendency will be captured by this counter, and it will be possible to make more accurate predictions over time.

7.6 Out-of-Order Processor

To cope with the problem of dependencies, an out-of-order processor looks ahead across many instructions to issue, or begin executing, independent instructions as rapidly as possible. The instructions can be issued in a different order than that written by the programmer, as long as dependencies are honored so that the program produces the intended result.
To enable OoO execution, you need to link the consumer of a value to the producer (register renaming: associate a tag with each data value), then you need to buffer instructions until they are ready to execute. Instructions need to keep track of readiness of source values (broadcast the tag when the value is produced) and finally when all source values of an instruction are ready, you need to dispatch the instruction to its functional unit (FU).

7.7 Tomasulo's Algorithm

Tomasulo's algorithm is used to implement dynamic scheduling in processors.
If a reservation station (rest areas for dependent instructions) is available before renaming, the instruction and renamed operands (source value/tag) are inserted into the reservation station.
Else stall.
While in the reservation station, each instruction watches the common data bus (CDB) for the tag of its sources and when the tag is seen, grabs the value for the source and keeps it in the reservation station. When both operands are available, instructions are ready to be dispatched.
Dispatch the instruction to the functional unit when the instruction is ready and after the instruction finishes in the functional unit, arbitrate for CDB and put the tagged value onto the CDB (tag broadcast).
The register file is then connected to the CDB, containing a tag indicating the latest writer to the register.
If the tag in the register file matches the broadcast tag, write the broadcast value into the register and set the valid bit. Finally reclaim the rename tag, so there is no valid copy of the tag in the system.

7.8 Single Instruction Multiple Data

Flynn's Taxonomy:

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data streams (SIMD)
3. Multiple instruction streams, single data stream (MISD)
4. Multiple instruction streams, multiple data streams (MIMD)

The term SIMD stands for single instruction multiple data, in which a single instruction acts on multiple pieces of data in parallel. A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing. This is also called packed arithmetic. For example, a 32-bit microprocessor might pack four 8-bit data elements into one 32-bit word. Packed add and subtract instructions operate on all four data elements within the word in parallel.
In an array processor, the instruction operates on multiple data elements at the same time using different spaces.
In a vector processor, the instruction operates on multiple data elements in consecutive time steps using the same space. This allows deeper pipelines.
GPUs are SIMD engines underneath. The instruction pipeline operates like a SIMD pipeline (e.g. an array processor). However, the programming is done using threads, not SIMD instructions. Each thread executes the same code but operates a different piece of data. Each thread has its own context (i.e. can be treated/restarted/executed independently). A set of threads executing the same instruction are dynamically grouped into a warp by the hardware.
An important metric when evaluating the performance of a program on a GPU is the *utilization* that is achieved. Utilization is defined as the ratio between the number of useful slots in a warp (i.e., where there are threads performing useful work) and the total available number of thread slots where work *could have been performed* for the warp. Utilization is usually calculated for the entire execution of a workload/application.

8 Memory Systems

8.1 Memory Systems Performance Analysis

Memory system performance metrics are miss rate or hit rate and average memory access time. Miss and hit rates are calculated as:

$$MissRate = \frac{NumberOfMisses}{NumberOfTotalMemoryAccesses} = 1 - HitRate \tag{5}$$

$$HitRate = \frac{NumberOfHits}{NumberOfTotalMemoryAccesses} = 1 - MissRate \tag{6}$$

Average memory access time (AMAT) is the average time a processor must wait for memory per load or store instruction. In the typical computer system, the processor first looks for the data in the cache. If the cache misses, the processor then looks in main memory. If the main memory misses, the processor accesses virtual memory on the hard disk. Thus, AMAT is calculated as:

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{\text{MM}} + MR_{\text{MM}}t_{\text{VM}}) \tag{7}$$

where $\mathrm{t}_{\text{cache}}$, t_{MM}, and t_{VM} are the access times of the cache, main memory, and virtual memory, and $\mathrm{MR}_{\text{cache}}$ and MR_{MM} are the cache and main memory miss rates, respectively.

8.2 Caches

A cache holds commonly used memory data. The number of data words that it can hold is called the capacity, C. Because the capacity of the cache is smaller than that of main memory, the computer system designer must choose what subset of the main memory is kept in the cache. When the processor attempts to access data, it first checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use. To accommodate the new data, the cache must replace old data. Caches use spatial and temporal locality to predict what data will be needed next. If a program accesses data in a random order, it would not benefit from a cache. In the following explanations, caches are specified by their capacity (C), number of sets (S), block size (b), number of blocks (B), and degree of associativity (N).

Recall that temporal locality means that the processor is likely to access a piece of data again soon if it has accessed that data recently. Therefore, when the processor loads or stores data that is not in the cache, the data is copied from main memory into the cache. Subsequent requests for that data hit in the cache. Recall that spatial locality means that, when the processor accesses a piece of data, it is also likely to access data in nearby memory locations. Therefore, when the cache fetches one word from memory, it may also fetch several adjacent words. This group of words is called a cache block. The number of words in the cache block, b, is called the block size. A cache of capacity C contains B = C/b blocks.

A cache is organized into S sets, each of which holds one or more blocks of data. The relationship between the address of data in main memory and the location of that data in the cache is called the mapping. Each memory address maps to exactly one set in the cache. Some of the address bits are used to determine which cache set contains the data. If the set contains more than one block, the data may be kept in any of the blocks in the set.
Caches are categorized based on the number of blocks in a set. In a direct mapped cache, each set contains exactly one block, so the cache has S = B sets. Thus, a particular main memory address maps to a unique block in the cache. In an N-way set associative cache, each set contains N blocks. The address still maps to a unique set, with S = B/N sets. But the data from that address can go in any of the N blocks in that set. A fully associative cache has only S 1 set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B-way set associative cache.
To illustrate these cache organizations, we will consider a MIPS memory system with 32-bit addresses and 32-bit words. The memory is byte-addressable, and each word is four bytes, so the memory consists of 2^{30} words aligned on word boundaries. We analyze caches with an eightword capacity (C) for the sake of simplicity. We begin with a one-word block size (b), then generalize later to larger blocks.

8.2.1 Direct Mapped Caches

A direct mapped cache has one block in each set, so it is organized into S = B sets. To understand the mapping of memory addresses onto cache blocks, imagine main memory as being mapped into b-word blocks, just as the cache is. An address in block 0 of main memory maps to set 0 of the cache. An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block B 1 of main memory maps to block B 1 of the cache. There are no more blocks of the cache, so the mapping wraps around, such that block B of main memory maps to block 0 of the cache. Because many addresses map to a single set, the cache must also keep track of the address of the data actually contained in each set. The least significant bits of the address specify which set holds the data. The remaining most significant bits are called the tag and indicate which of the many possible addresses is held in that set. The two least significant bits of the 32-bit address are called the byte offset, because they indicate the byte within the word. The next three bits are called the set bits, because they indicate the set to which the address maps. (In general, the number of set bits is $\log_2 S$.) The remaining 27 tag bits indicate the memory address of the data stored in a given cache set.
Sometimes, such as when the computer first starts up, the cache sets contain no data at all. The cache uses a valid bit for each set to indicate whether the set holds meaningful data. If the valid bit is 0, the contents are meaningless.
When two recently accessed addresses map to the same cache block, a conflict occurs, and the most recently accessed address evicts the previous one from the block. Direct mapped caches have only one block in each set, so two addresses that map to the same set always cause a conflict.

8.2.2 Multi-way Set Associative Caches

An N-way set associative cache reduces conflicts by providing N blocks in each set where data mapping to that set might be found. Each memory address still maps to a specific set, but it can map to any one of the N blocks in the set. Hence, a direct mapped cache is another name for a one-way set associative cache. N is also called the degree of associativity of the cache. Each way consists of a data block and the valid and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from that way. Set associative caches generally have lower miss rates than direct mapped caches of the same capacity, because they have fewer conflicts. However, set associative caches are usually slower and somewhat more expensive to build because of the output multiplexer and additional comparators. They also raise the question of which way to replace when both ways are full.

8.2.3 Fully Associative Caches

A fully associative cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set associative cache with one set.

8.2.4 Block Size

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to more conflicts, increasing the miss rate. Moreover, it takes more time to fetch the missing cache block after a miss, because more than one data word is fetched from main memory. The time required to load the missing block into the cache is called the miss penalty. If the adjacent words in the block are not accessed later, the effort of fetching them is wasted. Nevertheless, most real programs benefit from larger block sizes.

8.2.5 Data Replacement

In a direct mapped cache, each address maps to a unique block and set. If a set is full when new data must be loaded, the block in that set is replaced with the new data. In set associative and fully associative caches, the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block, because it is least likely to be used again soon. Hence, most associative caches have a least recently used (LRU) replacement policy. In a two-way set associative cache, a use bit, U, indicates which way within a set was least recently used. Each time one of the ways is used, U is adjusted to indicate the other way. For set associative caches with more than two ways, tracking the least recently used way becomes complicated. To simplify the problem, the ways are often divided into two groups and U indicates which group of ways was least recently used. Upon replacement, the new block replaces a random block within the least recently used group. Such a policy is called pseudo-LRU and is good enough in practice.

8.2.6 Types of Misses

The first request to a cache block is called a compulsory miss, because the block must be read from memory regardless of the cache design. Capacity misses occur when the cache is too small to hold all concurrently used data. Conflict misses are caused when several addresses map to the same set and evict blocks that are still needed.
Changing cache parameters can affect one or more type of cache miss. For example, increasing cache capacity can reduce conflict and capacity misses, but it does not affect compulsory misses. On the other hand, increasing block size could reduce compulsory misses (due to spatial locality) but might actually increase conflict misses (because more addresses would map to the same set and could conflict).

8.2.7 Write Policy

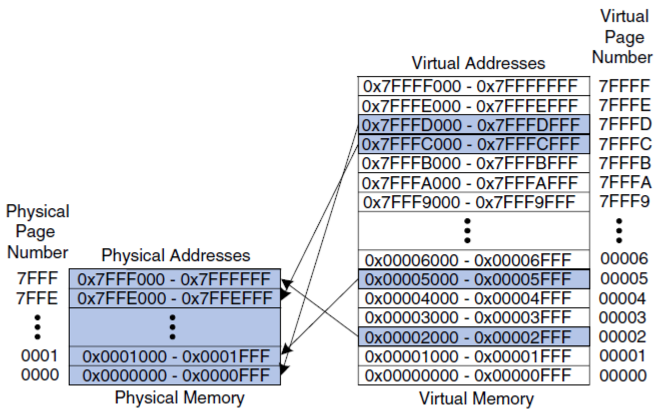
Caches are classified as either write-through or write-back. In a write-through cache, the data written to a cache block is simultaneously written to main memory. In a write-back cache, a dirty bit (D) is associated with each cache block. D is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory only when they are evicted from the cache. A write-through cache requires no dirty bit but usually requires more main memory writes than a write-back cache.

8.3 Virtual Memory

The objective of adding a hard disk to the memory hierarchy is to inexpensively give the illusion of a very large memory while still providing the speed of faster memory for most accesses. A computer with only 128 MB of DRAM, for example, could effectively provide 2 GB of memory using the hard disk. This larger 2-GB memory is called virtual memory, and the smaller 128-MB main memory is called physical memory.
Programs can access data anywhere in virtual memory, so they must use virtual addresses that specify the location in virtual memory. The physical memory holds a subset of most recently accessed virtual memory. In this way, physical memory acts as a cache for virtual memory.Thus, most accesses hit in physical memory at the speed of DRAM, yet the program enjoys the capacity of the larger virtual memory.
Virtual memory is divided into virtual pages. Physical memory is likewise divided into physical pages of the same size. A virtual page may be located in physical memory (DRAM) or on the disk.
The process of determining the physical address from the virtual address is called address translation. If the processor attempts to access a virtual address that is not in physical memory, a page fault occurs, and the operating system loads the page from the hard disk into physical memory. To avoid page faults caused by conflicts, any virtual page can map to any physical page. In other words, physical memory behaves as a fully associative cache for virtual memory. In a conventional fully associative cache, every cache block has a comparator that checks the most significant address bits against a tag to determine whether the request hits in the block. In an analogous virtual memory system, each physical page would need a comparator to check the most significant virtual address bits against a tag to determine whether the virtual page maps to that physical page.
The virtual memory system uses a page table to perform address translation. A page table contains an entry for each virtual page, indicating its location in physical memory or that it is on the disk. Each load or store instruction requires a page table access followed by a physical memory access. The page table access translates the virtual address used by the program to a physical address. The physical address is then used to actually read or write the data. The page table is usually so large that it is located in physical memory. Hence, each load or store involves two physical memory accesses: a page table access, and a data access. To speed up address translation, a translation lookaside buffer (TLB) caches the most commonly used page table entries.

8.3.1 Address Translation

Recall that virtual memory and physical memory are divided into pages. The most significant bits of the virtual or physical address specify the virtual or physical page number. The least significant bits specify the word within the page and are called the page offset.



Here is the page organization of a virtual memory system with 2 GB of virtual memory and 128 MB of physical memory divided into 4-KB pages. MIPS accommodates 32-bit addresses. With a 2-GB 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used; the 32nd bit is always 0. Similarly, with a 128-MB 2^{27}-byte physical memory, only the least significant 27 physical address bits are used; the upper 5 bits are always 0. Because the page size is 4 KB 2^{12} bytes, there are $2^{31}/2^{12} = 2^{19}$ virtual pages and $2^{27}/2^{12} = 2^{15}$ physical pages. Thus, the virtual and physical page numbers are 19 and 15 bits, respectively. Physical memory can only hold up to 1/16th of the virtual pages at any given time. The rest of the virtual pages are kept on disk.
This is the translation of a virtual address to a physical address. The least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the virtual page number (VPN) and are translated to a 15-bit physical page number (PPN).

8.3.2 The Page Table

The processor uses a page table to translate virtual addresses to physical addresses. Recall that the page table contains an entry for each virtual page. This entry contains a physical page number and a valid bit. If the valid bit is 1, the virtual page maps to the physical page specified in the entry. Otherwise, the virtual page is found on disk. Because the page table is so large, it is stored in physical memory. The page table is indexed with the virtual page number (VPN). For example, entry 5 specifies that virtual page 5 maps to physical page 1. Entry 6 is invalid (V = 0), so virtual page 6 is located on disk.
The page table can be stored anywhere in physical memory, at the discretion of the OS. The processor typically uses a dedicated register, called the page table register, to store the base address of the page table in physical memory. To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and adds it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally, it reads or writes data at this physical address. Because the page table is stored in physical memory, each load or store involves two physical memory accesses.

8.3.3 The Translation Lookaside Buffer

In general, the processor can keep the last several page table entries in a small cache called a translation lookaside buffer (TLB). The processor "looks aside" to find the translation in the TLB before having to access the page table in physical memory. A TLB is organized as a fully associative cache and typically holds 16 to 512 entries. Each TLB entry holds a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If the TLB hits, it returns the corresponding physical page number. Otherwise, the processor must read the page table in physical memory. The TLB is designed to be small enough that it can be accessed in less than one cycle.

8.3.4 Memory Protection

As you probably know, modern computers typically run several programs or processes at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can crash or hijack another program. Specifically, no program should be able to access another program's memory without permission. This is called memory protection. Virtual memory systems provide memory protection by giving each program its own virtual address space.

8.3.5 Memory Mapped I/O

Processors also use the memory interface to communicate with input/output (I/O) devices such as keyboards, monitors, and printers. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory. A portion of the address space is dedicated to I/O devices rather than memory. This method of communicating with I/O devices is called memory-mapped I/O. An address decoder determines which device communicates with the processor. It uses the Address and MemWrite signals to generate control signals for the rest of the hardware. The ReadData multiplexer selects between memory and the various I/O devices. Write-enabled registers hold the values written to the I/O devices.
Software that communicates with an I/O device is called a device driver.

8.4 Exercises

1. Suppose a program has 2000 data access instructions (loads or stores), and 1250 of these requested data values are found in the cache. The other 750 data values are supplied to the processor by main memory or disk memory. What are the miss and hit rates for the cache?

2. Suppose a computer system has a memory organization with only two levels of hierarchy, a cache and main memory. What is the average memory access time given the access times and miss rates given this table?

Memory Level	Access Time (cycles)	Miss Rate
Cache	1	10%
Main Memory	100	0%

3. Find the number of set and tag bits for a direct mapped cache with 1024 (2^{10}) sets and a one-word block size. The address size is 32 bits.

4. Show the contents of an eight-word two-way set associative cache after executing the following code. Assume LRU replacement, a block size of one word, and an initially empty cache.

   lw $t0, 0x04($0)

   lw $t1, 0x24($0)

   lw $t2, 0x54($0)

5. Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: writethrough or write-back?

   sw $t0, 0x0($0)

   sw $t0, 0xC($0)

   sw $t0, 0x8($0)

   sw $t0, 0x4($0)

9 Solutions

9.1 Chapter 1

1. $10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$

2. Determine whether each column of the binary result has a 1 or a 0. We can do this starting at either the left or the right column.

   Working from the left, start with the largest power of 2 less than the number (in this case, 64). $84 \geq 64$, so there is a 1 in the 64's column, leaving 84 - 64 = 20. $20 < 32$, so there is a 0 in the 32's column. $20 \geq 16$, so there is a 1 in the 16's column, leaving 20 - 16 = 4. $4 < 8$, so there is a 0 in the 8's column. $4 \geq 4$, so there is a 1 in the 4's column, leaving 4 - 4 = 0. Thus there must be 0's in the 2's and 1's column. Putting this all together, $84_{10} = 1010100_2$.

3. Start reading from the right. The four least significant bits are $1010_2 = \mathrm{A}_{16}$. The next bits are $111_2 = 716$. Hence $1111010_2 = 7\mathrm{A}_{16}$.

4. Like decimal to binary conversion, decimal to hexadecimal conversion can be done from the left or the right.

   Working from the right, repeatedly divide the number by 16. The remainder goes in each column. 333/16 = 20 with a remainder of 13_{10} = D16 going in the 1's column. 20/16 = 1 with a remainder of 4 going in the 16's column. 1/16 = 0 with a remainder of 1 going in the 256's column. Again, the result is $14\mathrm{D}_{16}$.

5. There is no overflow.

$$\begin{array}{r} {\scriptstyle 111} \\ 0111 \\ +\,0101 \\ \hline 1100 \end{array}$$

6. There is overflow.

$$\begin{array}{r} {\scriptstyle 1\;\,1} \\ 1101 \\ +\,0101 \\ \hline 10010 \end{array}$$

7. Start with $2_{10} = 0010_2$. To get -2_{10}, invert the bits and add 1. Inverting 0010_2 produces 1101_2. $1101_2 + 1 = 1110_2$. So -2_{10} is 1110_2.

8. 1001_2 has a leading 1, so it must be negative. To find its magnitude, invert the bits and add 1. Inverting $1001_2 = 0110_2$. $0110_2 - 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.

9. $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$.

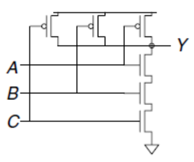
10. $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2.

11. $3_{10} = 0011_2$. Take its two's complement to obtain $-3_{10} = 1101_2$. Now add $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Note that the carry out of the most significant position is discarded because the result is stored in four bits.

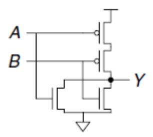
12. Take the two's complement of 5_{10} to obtain $-5_{10} = 1011$. Now add $3_{10} + (-5_{10}) = 0011_2 = 1011_2 = 1110_2 = -2_{10}$.

9.2 Chapter 2

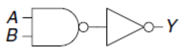
1. The NAND gate should produce a 0 output only when all three inputs are 1. Hence, the pull-down network should have three nMOS transistors in series. By the conduction complements rule, the pMOS transistors must be in parallel.



2. The NOR gate should produce a 0 output if either input is 1. Hence, the pull-down network should have two nMOS transistors in parallel. By the conduction complements rule, the pMOS transistors must be in series.

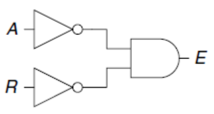


3. It is impossible to build an AND gate with a single CMOS gate. However, building NAND and NOT gates is easy. Thus, the best way to build an AND gate using CMOS transistors is to use a NAND followed by a NOT.



4. Circuit (a) is combinational. It is constructed from two combinational circuit elements (inverters I1 and I2). It has three nodes: n1, n2, and n3. n1 is an input to the circuit and to I1; n2 is an internal node, which is the output of I1 and the input to I2; n3 is the output of the circuit and of I2. (b) is not combinational, because there is a cyclic path: the output of the XOR feeds back to one of its inputs. Hence, a cyclic path starting at n4 passes through the XOR to n5, which returns to n4. (c) is combinational. (d) is not combinational, because node n6 connects to the output terminals of both I3 and I4. (e) is combinational, illustrating two combinational circuits connected to form a larger combinational circuit. (f) does not obey the rules of combinational composition because it has a cyclic path through the two elements. Depending on the functions of the elements, it may or may not be a combinational circuit.

5. First define the inputs and outputs. The inputs are A and R, which indicate if there are ants and if it rains. A is TRUE when there are ants and FALSE when there are no ants. Likewise, R is TRUE when it rains and FALSE when the sun smiles on Ben. The output is E, Ben's enjoyment of the picnic. E is TRUE if Ben enjoys the picnic and FALSE if he suffers.



6. $\overline{Y} = \overline{A}\,\overline{B} + \overline{A}\mathrm{B}$

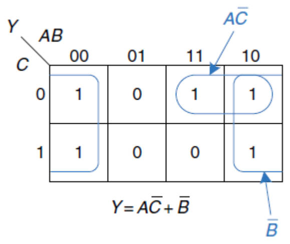
   Taking the complement of both sides and applying De Morgan's Theorem twice, we get:

   $\overline{\overline{Y}} = \mathrm{Y} = \overline{\overline{A}\overline{B} + \overline{A}B} = (\overline{\overline{A}\overline{B}}) + \overline{\overline{A}B}) = (\mathrm{A} + \mathrm{B})(\mathrm{A} + \overline{B})$

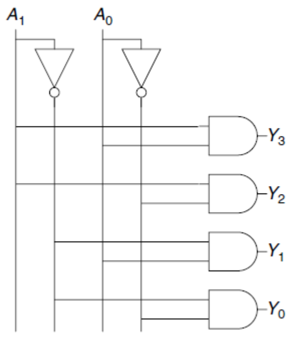
7.

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C} + \mathrm{A}\,\overline{B}\,\overline{C} + \mathrm{A}\,\overline{B}\,\mathrm{C}$	
1	$\overline{A}\,\overline{B}\,\overline{C} + \mathrm{A}\,\overline{B}\,\overline{C} + \mathrm{A}\,\overline{B}\,\overline{C} + \mathrm{A}\,\overline{B}\,\mathrm{C}$	T3: Idempotency
2	$\overline{B}\,\overline{C}(\overline{A} + \mathrm{A}) + \mathrm{A}\overline{B}(\overline{C} + \mathrm{C})$	T8: Distributivity
3	$\overline{B}\,\overline{C}(1) + \mathrm{A}\overline{B}(1)$	T5: Complements
4	$\overline{B}\,\overline{C} + \mathrm{A}\overline{B}$	T1: Identity

8. Circle the 1's in the K-map using as few circles as possible. Each circle in the K-map represents a prime implicant, and the dimension of each circle is a power of two. Form the prime implicant for each circle by writing those variables that appear in the circle only in true or only in complementary form.



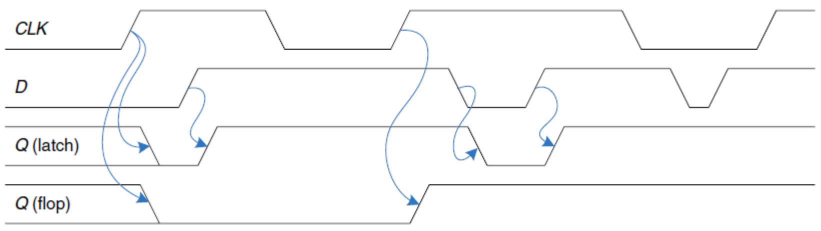
9.



10. Ben begins by finding the critical path and the shortest path through the circuit. The critical path is from input A or B through three gates to the output, Y. Hence, t_{pd} is three times the propagation delay of a single gate, or 300 ps. The shortest path is from input C, D, or E through two gates to the output, Y. There are only two gates in the shortest path, so t_{cd} is 120 ps.

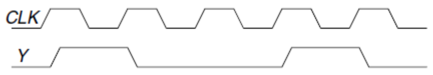
9.3 Chapter 3

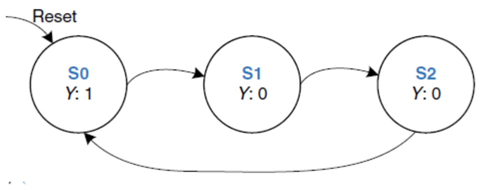
1. The arrows indicate the cause of an output change. The initial value of Q is unknown and could be 0 or 1, as indicated by the pair of horizontal lines. First consider the latch. On the first rising edge of CLK, D = 0, so Q definitely becomes 0. Each time D changes while CLK = 1, Q also follows. When D changes while CLK = 0, it is ignored. Now consider the flip-flop. On each rising edge of CLK, D is copied to Q. At all other times, Q retains its state.



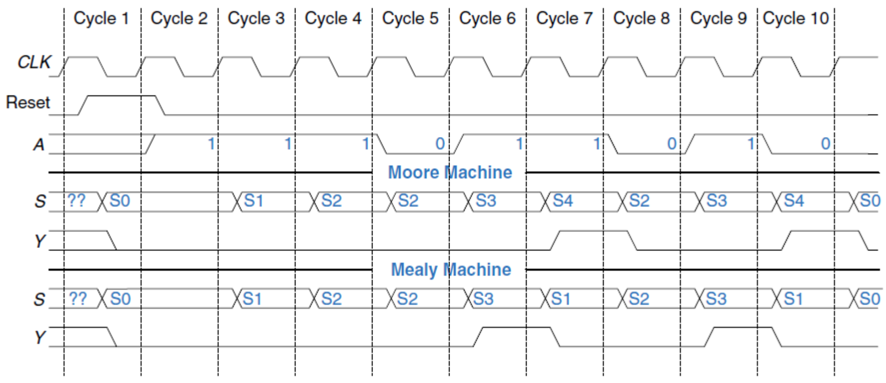
2. Circuit (a) is combinational, not sequential, because it has no registers. (b) is a simple sequential circuit with no feedback. (c) is neither a combinational circuit nor a synchronous sequential circuit, because it has a latch that is neither a register nor a combinational circuit. (d) and (e) are synchronous sequential logic; they are two forms of finite state machines. (f) is neither combinational nor synchronous sequential, because it has a cyclic path from the output of the combinational logic back to the input of the same logic but no register in the path. (g) is synchronous sequential logic in the form of a pipeline. (h) is not, strictly speaking, a synchronous sequential circuit, because the second register receives a different clock signal than the first, delayed by two inverter delays.

3. Here are the waveform and the state transition diagram:





4. The Moore machine requires five states. Convince yourself that the state transition diagram is correct. In particular, why is there an arc from S4 to S2 when the input is 1? In comparison, the Mealy machine requires only four states. Each arc is labeled as A/Y. A is the value of the input that causes that transition, and Y is the corresponding output.



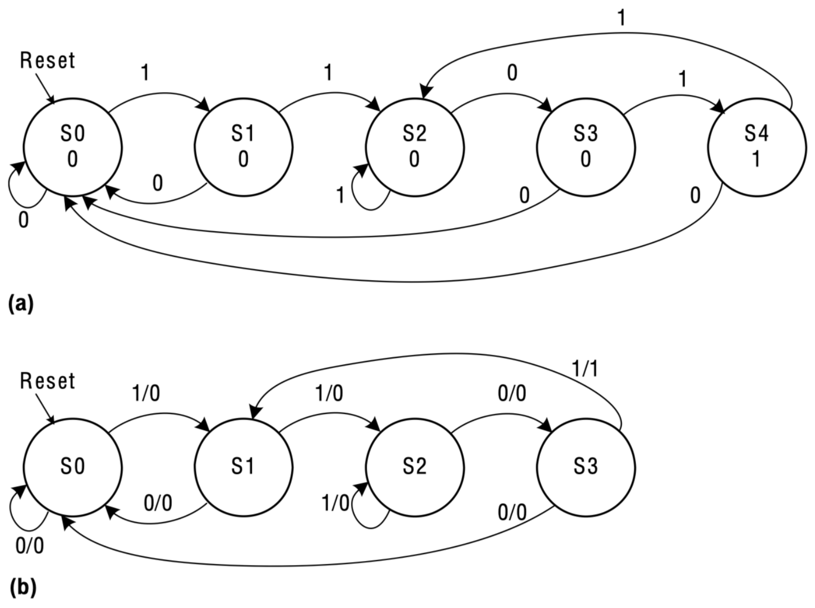


Figure 3.30 **FSM state transition diagrams: (a) Moore machine, (b) Mealy machine**

9.4 Chapter 5

1. The propagation delay of the 32-bit ripplecarry adder is 32 x 300 ps = 9.6 ns.

   The CLA has t_{pg} = 100 ps, t_{pg_block} = 6 x 100 ps = 600 ps, and t_{AND_OR} = 2 x 100 ps = 200 ps.

   The propagation delay of the 32-bit carry-lookahead adder with 4-bit blocks is thus 100 ps + 600 ps + (32/4 - 1) x 200 ps + (4 x 300 ps) = 3.3 ns, almost three times faster than the ripple-carry adder.

2. Because A < B, we expect Y to be 1. For SLT, $F_{2:0}$ = 111. With F_2 = 1, this configures the adder unit as a subtractor with an output, S, of 25_{10} - 32_{10} = -7_{10} = $1111...1001_2$. With $F_{1:0}$ = 11, the final multiplexer sets Y = S_{31} = 1.

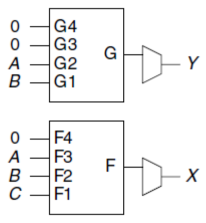
3. First convert 0.625, the magnitude of the second number, to fixedpoint binary notation. 0.625 $\geq 2^{-1}$, so there is a 1 in the 2^{-1} column, leaving 0.625 - 0.5 = 0.125. Because 0.125 $< 2^{-2}$, there is a 0 in the 2^2 column. Because 0.125 $\geq 2^{-3}$, there is a 1 in the 2^{-3} column, leaving 0.125 - 0.125 = 0. Thus, there must be a 0 in the 2^{-4} column. Putting this all together, 0.625_{10} = 0000.1010_2.

4. First convert the decimal number into binary: 228_{10} = 11100100_2 = 1.11001_2 x 2^7. The sign bit is positive (0), the 8 exponent bits give the value 7, and the remaining 23 bits are the mantissa.

5. Configure the F-LUT to compute X and the G-LUT to compute Y. Inputs F3, F2, and F1 are A, B, and C, respectively (these connections are set by the routing channels). Inputs G2 and G1 are A and B. F4, G4, and G3 are don't cares (and may be connected to 0). Configure the final multiplexers to select X from the F-LUT and Y from the G-LUT. In general, a CLB can compute any two functions, of up to four variables each, in this fashion.

	(A)	(B)	(C)	(X)
F4	F3	F2	F1	F
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

		(A)	(B)	(Y)
G4	G3	G2	G1	G
X	X	0	0	0
X	X	0	1	0
X	X	1	0	1
X	X	1	1	0



6. Alyssa solves for the maximum propagation delay of the logic: $t_{pd} \leq T_c - (t_{pcq} + t_{setup})$.

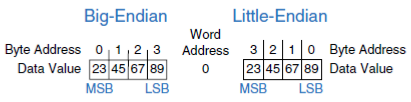
   Thus, $t_{pd} \leq$ 5ns - (0.72ns + 0.53ns), so $t_{pd} \leq$ 3.75ns. The delay of each CLB, t_{CLB}, is 0.61ns, and the maximum number of CLBs, N, is $Nt_{CLB} \leq$ 3.75ns. Thus, N = 6.

9.5 Chapter 6

1. # MIPS assembly code

   ```
   # $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 = g, $s5 = h,
   # $s6 = i, $s7 = j
   sub $s0, $s1, $s2        # a = b - c
   add $t0, $s4, $s5        # $t0 = g + h
   add $t1, $s6, $s7        # $t1 = i + j
   sub $s3, $t0, $t1        # f = (g + h) - (i + j)
   ```

2. After the load byte instruction, lb $s0, 1($0), $s0 would contain 0x00000045 on a big-endian system and 0x00000067 on a little-endian system.

3. # MIPS assembly code

   ```
   # $s0 = base address of chararray, $s1 = i
   addi $s1, $0, 0         # i = 0
   addi $t0, $0, 10        # $t0 = 10
   loop:
   beq $t0, $s1, done      # if i == 10, exit loop
   add $t1, $s1, $s0       # $t1 = address of chararray[i]
   lb $t2, 0($t1)          # $t2 = array[i]
   addi $t2, $t2, 32       # convert to upper case: $t1 = $t1 - 32
   sb $t2, 0($t1)          # store new value in array: chararray[i] = $t1
   addi $s1, $s1, 1        # i = i + 1
   j loop                  # repeat
   done:
   ```

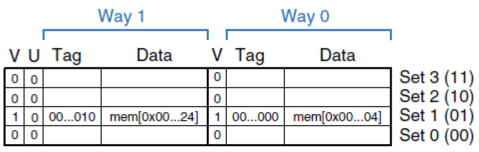
9.6 Chapter 8

1. The miss rate is 750/2000 = 0.375 = 37.5%. The hit rate is 1250/2000 = 0.625 = 1 - 0.375 = 62.5%.

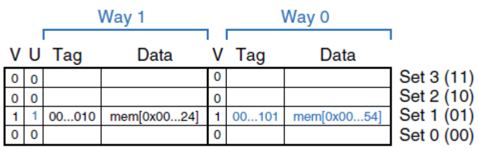
2. The average memory access time is 1 + 0.1(100) = 11 cycles.

3. A cache with 2^{10} sets requires $\log_2(2^{10}) = 10$ set bits. The two least significant bits of the address are the byte offset, and the remaining 32 - 10 - 2 = 20 bits form the tag.

4. The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache. U = 0 indicates that data in way 0 was the least recently used. The next memory access, to address 0x54, also maps to set 1 and replaces the least recently used data in way 0. The use bit, U, is set to 1 to indicate that data in way 1 was the least recently used.



	Way 1			Way 0			
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]	Set 1 (01)
0	0			0			Set 0 (00)



	Way 1			Way 0			
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]	Set 1 (01)
0	0			0			Set 0 (00)

5. All four store instructions write to the same cache block. With a writethrough cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.