Semester Outline

Monday, 23 August 2021 10:55 PM

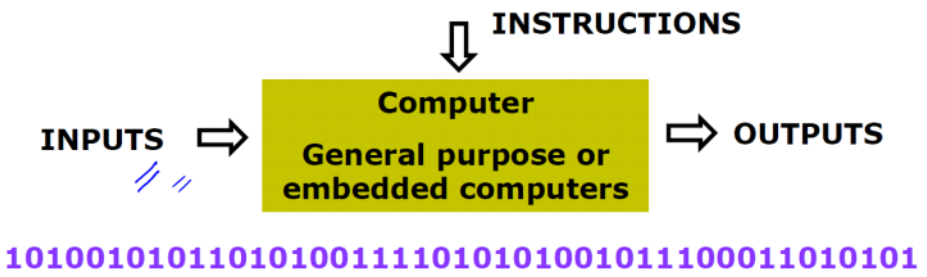
Wk	Day	Date		Lecture (date)		Learning Lab	Assessment due
1	Mon-Tue	26-27 Jul	1	Course Introduction and Binary numbers (26 Jul)		No labs (P2 sessions) on Mon-Tue on week 1	
	Wed-Fri	28-30 Jul	2	Logic gates (28 Jul)	1	Signed number representations	
2	Mon-Tue	2-3 Aug	3	Binary Arithmetic (2 Aug)	2	Logic Gates	Quiz 1 (4pm, 6 Aug)
	Wed-Fri	4-6 Aug	4	Combinational Logic (4 Aug)	3	Binary Arithmetic	
3	Mon-Tue	9-10 Aug	5	Flip flops (9 Aug)	4	Combinational Logic	Quiz 2 (4pm, 13 Aug)
	Wed-Fri	12-13 Aug	6	Sequential Circuits 1 - Shift Registers (11 Aug) Pre-recorded lecture due to public holiday on 11/08/21	5	Flip flops	
4	Mon-Tue	16-17 Aug	7	Sequential Circuits 2 - Counters (16 Aug)	6	Sequential Circuits 1	Quiz 3 (4pm, 20 Aug)
	Wed-Fri	18-20 Aug	8	Sequential Circuits 3 - State Machines (18 Aug)	7	Sequential Circuits 2	
5	Mon-Tue	23-24 Aug	9	From Digital Logic to Processors, ALU (23 Aug)	8	Sequential Circuits 3	Quiz 4 (4pm, 27 Aug)
	Wed-Fri	25-27 Aug	10A	Memory and Control Unit (25 Aug)		Cath-up labs, Assignement 1 release end of week 5	
6	Mon-Tue	30-31 Aug	10B	Introduction to C language (pre-recorded) Review session for Assignment 1 part 1 (30 Aug)		Assignment 1 working time Part 1: 1-hour timed quiz Part 2: Digital logic design	Assignment 1 - part 1 due date (4pm, 3 Sep)
	Wed-Fri	01-03 Sep	11	Introduction to AVR and Assembly Language (1 Sep)			
7	Mon-Tue	06-07 Sep	12	Instruction Set Architecture 1 (06 Sep)	9	C Programming	Assignment 1 - part 2 submission (4pm, 6 Sep) Quiz 5 (4pm, 10 Sep)
	Wed-Fri	08-10 Sep	13	Instruction Set Architecture 2 (08 Sep)	10	Atmel Assembly	
8	Mon-Tue	13-14 Sep	14	C for AVR Microcontroller (13 Sep)	11	AVR Hardware Introduction	Quiz 6 (4pm, 17 Sep)
	Wed-Fri	15-17 Sep	15	Flow Control and AVR Timers (15 Sep)	12	AVR C Programming	
9	Mon-Tue	20-21 Sep	16	Pulse Width Modulation (PWM) (20 Sep)	13	AVR Timers	Quiz 7 (4pm, 24 Sep)
	Wed-Fri	22-24 Sep	17	Interrupts (22 Sep)	14	AVR PWM	

(From lecture 9)

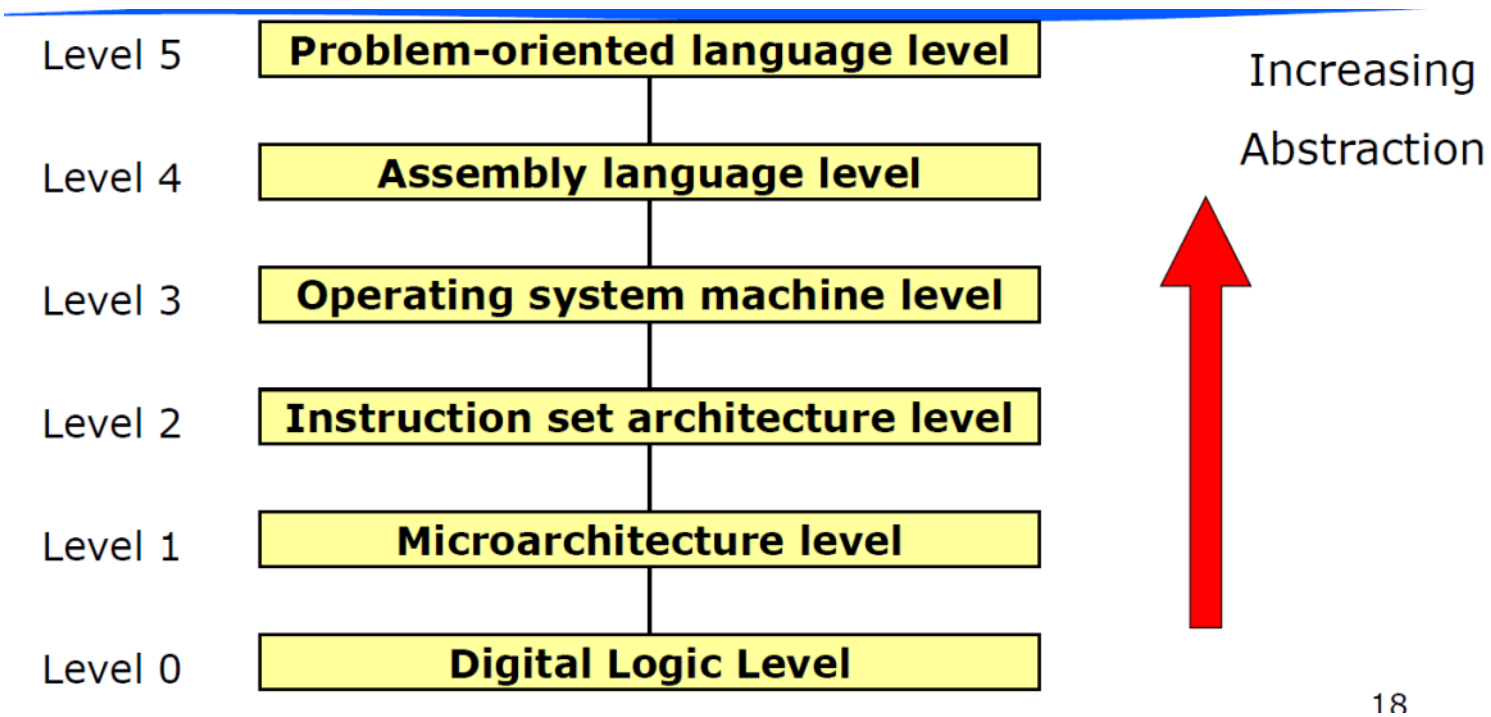
Levels of a Computer

Tuesday, 15 June 2021 11:20 AM

All inputs, instructions and outputs from a computer must be in binary because this is how a computer deals with data.



A Computer at Different Levels of Abstraction



Numbers in Binary

Saturday, 7 August 2021 8:10 PM

Bits, Bytes and Binary Numbers

- Computers represent everything in binary
- Bit = a binary digit (0 or 1)
- Byte = 8 bits
  - e.g.

0	1	0	1	0	1	1	1

- Modern computers deal with words which are usually a power of 2 number of bytes, eg
  - 1,2,4, or 8 bytes = 8, 16, 32, 64 bits.

Representing Whole (unsigned) Numbers in Binary

- Each bit position has a value:

	9	8	7	6	5	4	3	2	1	0
....	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	512	256	128	64	32	16	8	4	2	1

Bit position

Value (Powers of 2)

Converting Binary to decimal:

- Add values of each position where bit is 1

Example

7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	1

128 + 16 + 1 = 145

Converting Decimal to Binary

Convert 53 (decimal) to binary

Method 1

1. Rewrite n as sum of powers of 2 (by repeatedly subtracting the largest power of 2 not greater than n).
2. Assemble binary numbers from 1's in bit positions corresponding to those powers of 2, 0's elsewhere.

Example (Method 1)

$53 = 32 + 16 + 4 + 1.$

Therefore 53 in binary is 110101

1 1 0 1 0 1

2^5 2^4 2^3 2^2 2^1 2^0

Method 2

- Divide n by 2
- Remainder of division (0 or 1) is the next bit
- Repeat with n = quotient.

Example (Method 2)

53/2 = 26 r 1

26/2 = 13 r 0

13/2 = 6 r 1

6/2 = 3 r 0

3/2 = 1 r 1

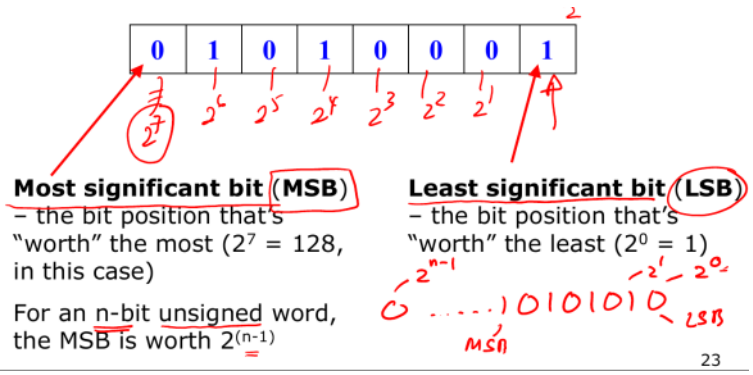
Therefore, 110101: same result.

Does not have to be 8 bits. In this case, it is 6 bits. However, in every system, there is a word length. Therefore, 53 can be made 8 bits (what the system requires) by adding 00 as a prefix.

So 53 is 00110101

Least and Most Significant Bits

- The least significant bit (LSB) in a binary number is the trailing or last bit
- The most significant bit (MSB) in a binary number is the leading or first bit



Number Range

Assuming whole (unsigned) numbers.

Smallest number is always 0
Largest number is when all bits are 1s. So in 8-bit, 11111111 would be 255

The formula for this is $2^n - 1$.

Radices (Radix singular)

Saturday, 7 August 2021 8:10 PM

Radices

- Radix is the number system base

- A radix-k number system
  - k different symbols to represent digits 0 to $k-1$;
  - Value of each digit is (from the right) $k^0, k^1, k^2, k^3, \ldots$;
- Often convenient to deal with
  - Dec: 0,1,2,3,4,5,6,7,8,9
  - Octal (radix-8) - Symbols: 0 1 2 3 4 5 6 7
  - Hexadecimal (radix-16) - Symbols 0 1 2 3 4 5 6 7 8 9 A B C D E F

A helpful little table

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Oct	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20	21
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

How do we know the Radix?

Can use subscript, e.g. 101_2 or 101_{16}.

Other options include leading or trailing syntax

- Hexadecimal
  - Leading $0x$, eg. 0x101, 0x747 C, Atmel AVR
  - Trailing h, eg. 101h, 747h Some assembly languages
  - Leading $, eg. $747 Atmel AVR Assembly
- Octal
  - Leading 0, eg. 0101, 0747 C language, Atmel AVR
  - Trailing q, eg. 101q, 747q Some assembly languages
  - Leading @, eg. @101, @747 Some assembly languages
- Binary
  - Leading 0b, eg 0b101 Atmel AVR Assembly, Some C
  - Trailing b, eg. 101b Some assembly languages
  - Leading %, eg. %101 Some assembly languages

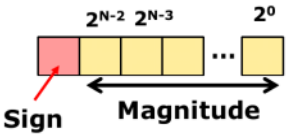
Negative Numbers (Practical)

Saturday, 7 August 2021 8:11 PM

Representing Negative Numbers

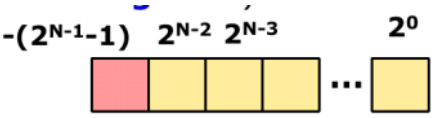
- Computers don't store + and - signs. They must use binary digits (0, 1)
- 4 different formats have been used
  - Signed magnitude
  - One's complement
  - Two's complement
  - Excess 2^{m-1}

Signed Magnitude



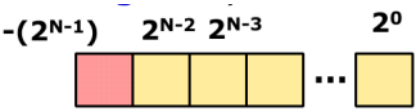
- Leftmost bit = sign bit
  - 0 = positive
  - 1 = 1 negative
- Negate number by
  - Inverting the sign bit
- The sign but does not contribute towards magnitude
- Range of numbers
  - -127 to 127 with 8-bit signed magnitude
  - $-(2^{n-1}-1)$ to $2^{n-1}-1$ with n-bit signed magnitude

One's Complement



- Leftmost bit = sign-bit (as per magnitude, but now also contributes towards magnitude in the negative direction)
  - 0 = positive
  - 1 = negative
- Bit position value changed as -->
- By having the bit position values like this allows us to find the negative number by simply inverting all the bits.
- Given x in regular unsigned binary, you can obtain its $-x$ in ones' complement form by inverting all the bits.
- Range of numbers
  - -127 to 127 with 8-bit one's complement

Two's Complement



- Leftmost bit = sign-bit (as per magnitude, but now the negative direction)
  - 0 = positive
  - 1 = negative
- Bit position value changed as -->
- By having the bit position values like this allows us to find the negative number by simply inverting all the bits and adding 1.
- Given x in regular unsigned binary, you can obtain its $-x$ in twos' complement form by inverting all the bits and adding 1.
- Range of numbers
  - -127 to 128 with 8-bit two's complement
  - -2^{n-1} to $2^{n-1}-1$ with n-bit two's complement

Excess 2^{m-1}

- Number stored as true value plus 128.
- Interestingly, same as two's complement with the sign-bit reversed
  - 0 means a value of -128, 1 means a bit value of 0.
  - Practical use: hardware becomes simple for comparing two signed numbers.
- Range of numbers
  - -128 to 127 with 8-bit one's complement

Limitations/Advantages

- Two representations of zero - undesirable
  - Signed magnitude
    - 0000 0000 = 1000 0000
  - One's complement
    - 0000 0000 = 1111 1111
  - This means we've wasted one bit pattern which could've been used to represent something else to increase the range
- Asymmetric range
  - Two's complement
  - Excess 2^{m-1}

Format Matters

- Binary data is meaningless unless you know the format
- The format is not encoded in the data. This must be known separately. e.g. through
  - A human
  - A computer program
  - Digital hardware (e.g. CPU)

Logic Diagrams

Tuesday, 27 July 2021 1:47 PM

Digital Logic

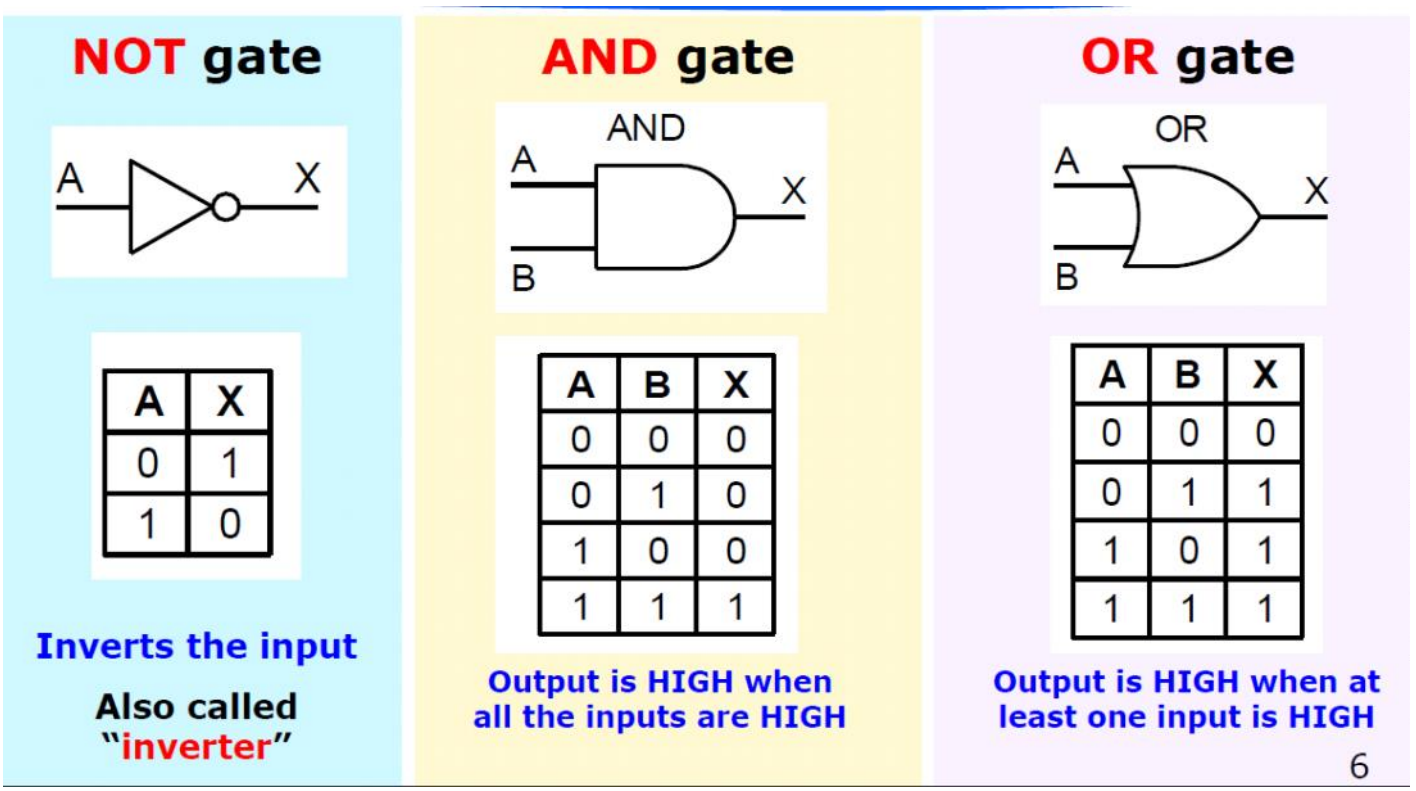
Digital Circuits

- Only two logical levels present (ie. Binary)
  - Logic '0' - usually small voltage (0 volts)
  - Logic '1' - usually larger voltage (eg. 0.8 to 5 volts depending on circuit type, type and size of transistors etc).

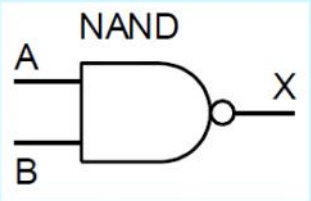
Logic Gates

- Logic gates are the building blocks of computers
- Each gate has
  - One or more inputs
  - Exactly one output
- Logic gates perform logic operations (or functions)
  - There are 7 basic types: Not, AND, OR, NAND, NOR, XOR, XNOR
  - Inputs & Outputs can have only two states, 1 & 0 - can be called "true" & "false" respectively
  - Logic symbol, truth table, Boolean expression, timing diagram

Basic Logic Gates



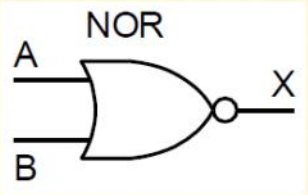
NAND gate



A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

Output is HIGH when at least one input is LOW

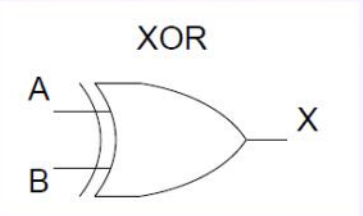
NOR gate



A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Output is HIGH when all the inputs are LOW

XOR gate

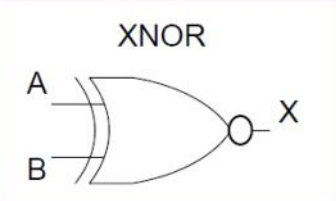


A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

Output is HIGH when exactly one input is HIGH

7

XNOR gate



A	B	X
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

Output is HIGH when the inputs are the same

It is useful to remember that XOR is the odd function and XNOR is the even function.

Boolean Algebra and Logic Expressions

Tuesday, 27 July 2021 4:32 PM

Boolean Logic Functions

- Logic functions can be expressed as expressions involving
  - Variables (literals), e.g. A B X
  - Functions, e.g. $+, \cdot, \oplus, \overline{\square}$
- Rules about how this works is called **Boolean algebra**.
- Variables and functions can only take on values **0** or **1**.

Boolean Algebra Conventions

Conventions we'll use:

- Inversion: $\overline{\square}$ overline
  - e.g. NOT(A) = $\bar{A}$ (said as "A bar")
- AND: cdot · or implied by adjacency
  - e.g. AND(A,B) = AB = A·B
- OR: plus sign
  - e.g. OR(A,B,C) = A + B + C

+ is or
· is and
$\oplus$ is exclusive or
$\overline{\square}$ is not

Other examples

- XOR(A,B)=A$\oplus$B=$\bar{A}$B + A$\bar{B}$.
- NAND(A,B,C) = $\overline{ABC}$
- NOR(A,B) = $\overline{A+B}$

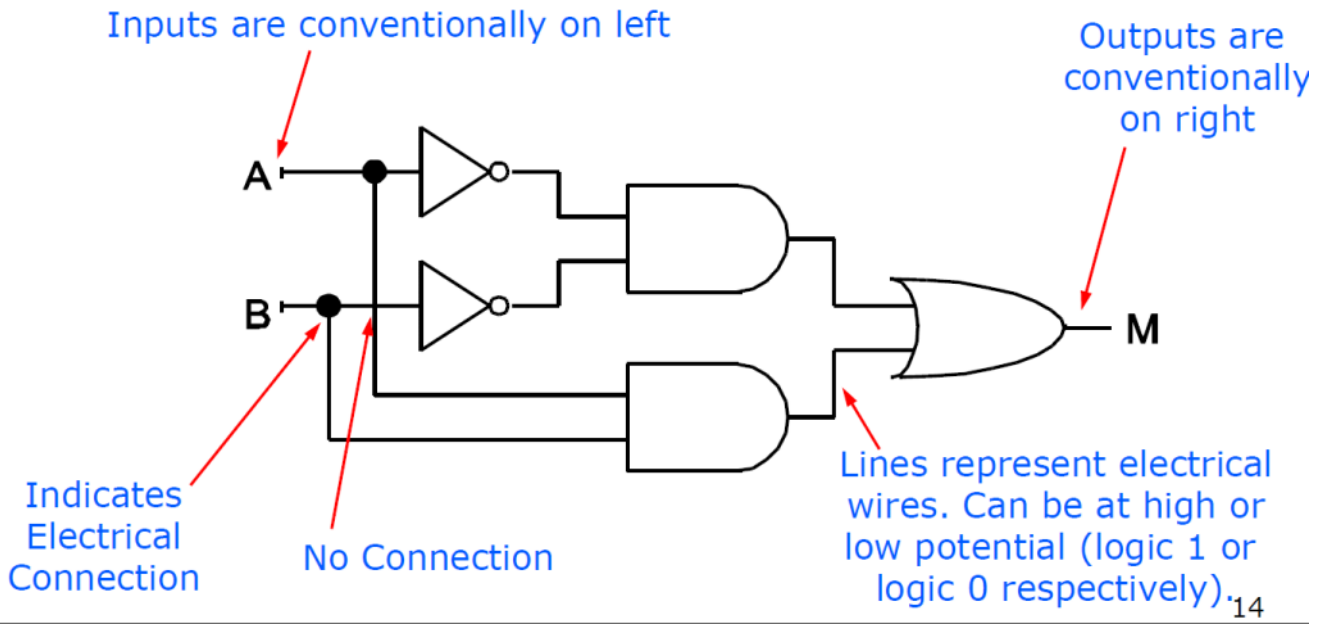
Logic Function Representations

There are four representatinos of logic functions (assume function of n inputs)

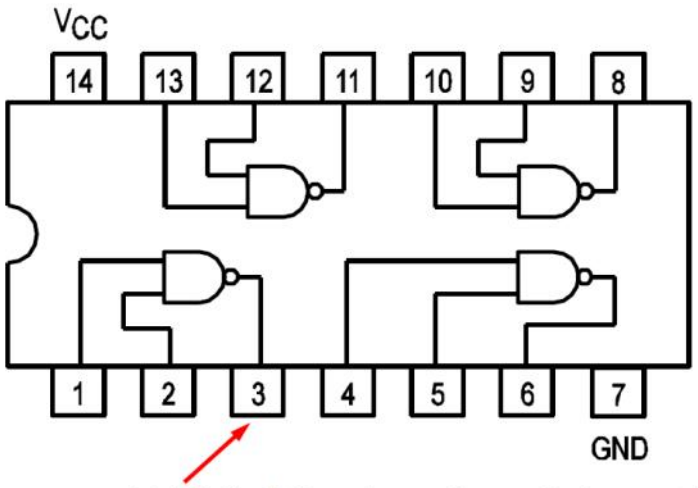
- Truth table
  - Lists output for all 2^n combinations of inputs
    - Best to list inputs in a systematic way to ensure you don't miss any
- Boolean function (or equation)
  - Describes the conditions under which the function output is 1
- Logic Diagram
  - Combination of logic symbols joined by wires
- Timing Diagram

Logic Diagrams

Conventions of a logic diagram



Gates on Integrated Circuits (ICs)



Pin spacing is 0.1" x 0.3"; chip is about 15mm long

- 74HCT00 – has four 2-input NAND gates
- **Vcc** = Power (e.g. 5V), **GND** = Ground (0V)

15

Logic Function Implementation

- Any logic function can be implemented as the OR of AND combinations of the inputs
  - Called the **sum of products**
- Example:
  - Consider truth table
  - For each `1` in the output column, write down the AND combination of inputs that give that 1
  - OR these together

$M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

SOP Expression. This can be simplified using Boolean algebra as below

$M = BC + AC + AB$

This is called a simplified SOP expression

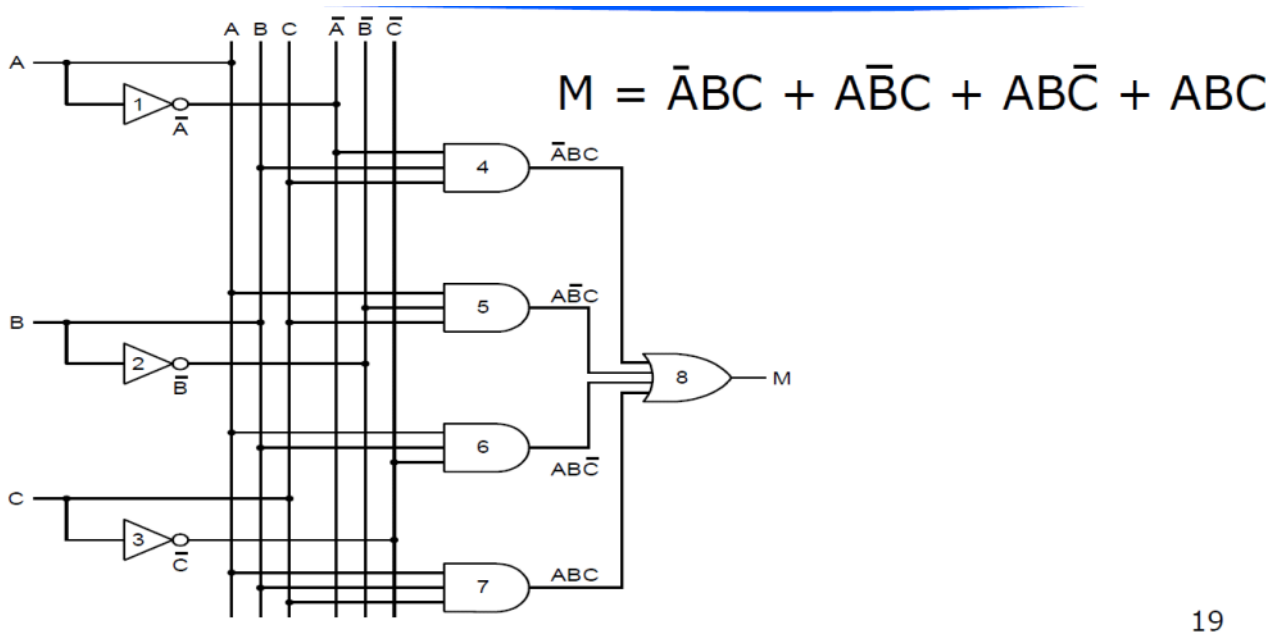
A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1

$M = BC + AC + AB$

This is called a simplified SOP expression.

1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

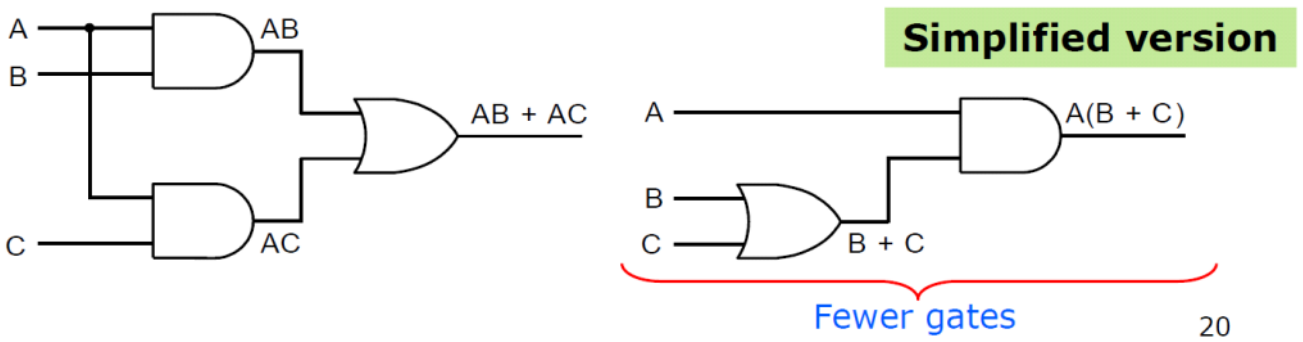
Equivalent Logic Diagram



It is important in logic diagrams to follow conventions. Vertical lines for each input, and its respective inverse. Then horizontal lines for each connection to and from a gate.

Equivalent Functions

- Sum of products does not necessarily producer a circuit with the minimum number of gates.
- It is possible to *manipulate* Boolean functions to give an equivalent function
  - Use rules of Boolean algebra to do this
- Example Z = AB + AC = A(B+C) as shown in the diagram below



★ Boolean Identities

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributative law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

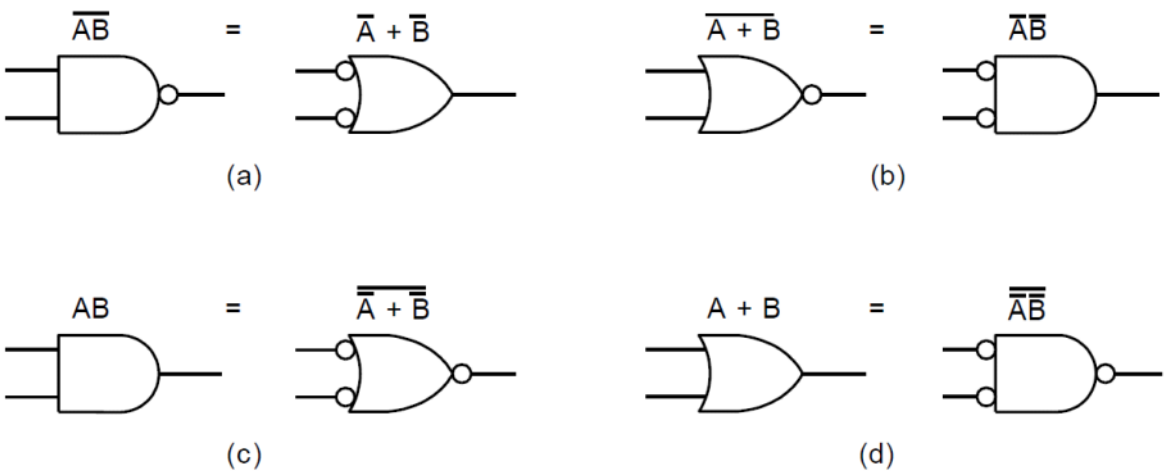
Example

Express $Z = \overline{A\left(B + C\left(\bar{A} + \bar{B}\right)\right)}$ as a sum of products

$$Z = \overline{A\left(B + C\left(\bar{A} + \bar{B}\right)\right)}$$
$$Z = \bar{A} + \overline{B + C\left(\bar{A} + \bar{B}\right)}$$
$$Z = \bar{A} + \bar{B} \cdot \overline{C\left(\bar{A} + \bar{B}\right)}$$
$$Z = \bar{A} + \bar{B} \cdot \left(\bar{C} + \overline{\bar{A} + \bar{B}}\right)$$
$$Z = \bar{A} + \bar{B} \cdot \left(\bar{C} + A \cdot B\right)$$
$$Z = \bar{A} + \bar{B}\bar{C} + AB\bar{B}$$
$$Z = \bar{A} + \bar{B}\bar{C}$$

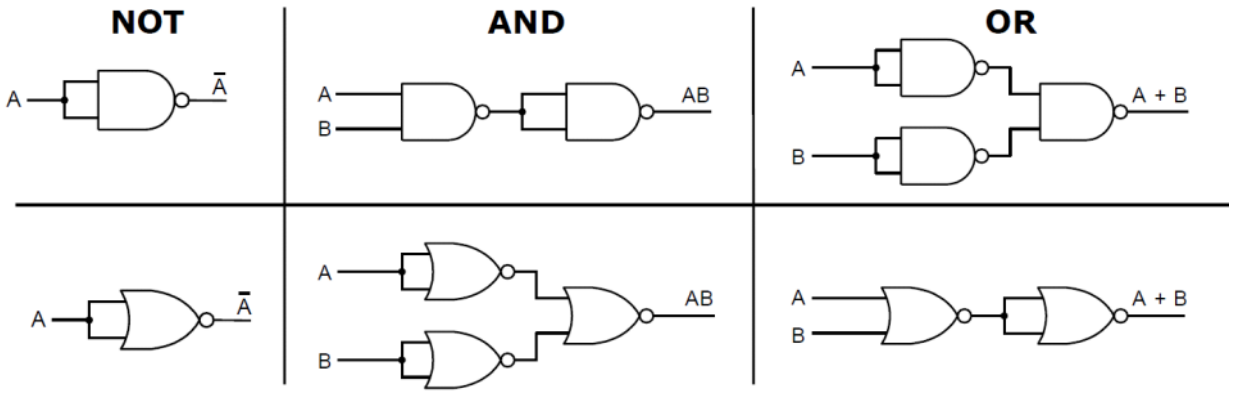
De Morgan's Laws Equivalents in Logic Gates

- AND/OR can be interchanged if you invert the inputs and outputs



Equivalent Circuits

- All circuits can be constructed from NAND and/or NOR gates
  - These are called **complete** gates



This is done because it is far easier to construct NAND and NOR gates from transistors.

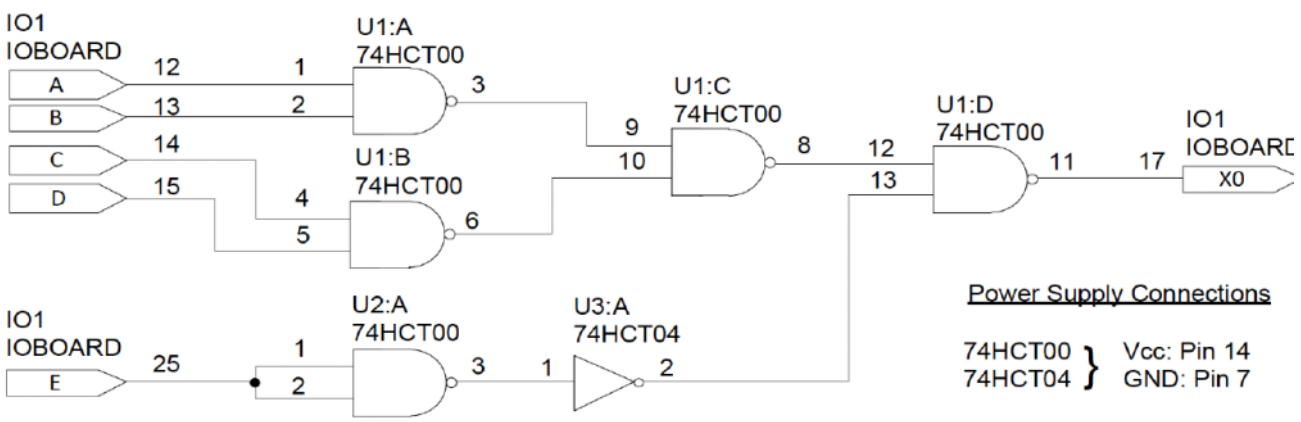
Logic Gates (Practical)

Saturday, 7 August 2021 8:37 PM

Circuit Schematic Diagram

- More than a logic diagram, a circuit schematic tells you how to **build** the circuit.
- You'll need to draw these for practicals and exams
- Schematics include
  - Labelled inputs
  - Labelled outputs
  - Labelled devices (logic chips & IO Board)
    - Logic chips: U1, U2, U3, …
    - Labelled gates within a chip (:A, :B, :C, :…)
    - IO-Board: IO1
  - Power supply connections for logic chips
  - Pin numbers

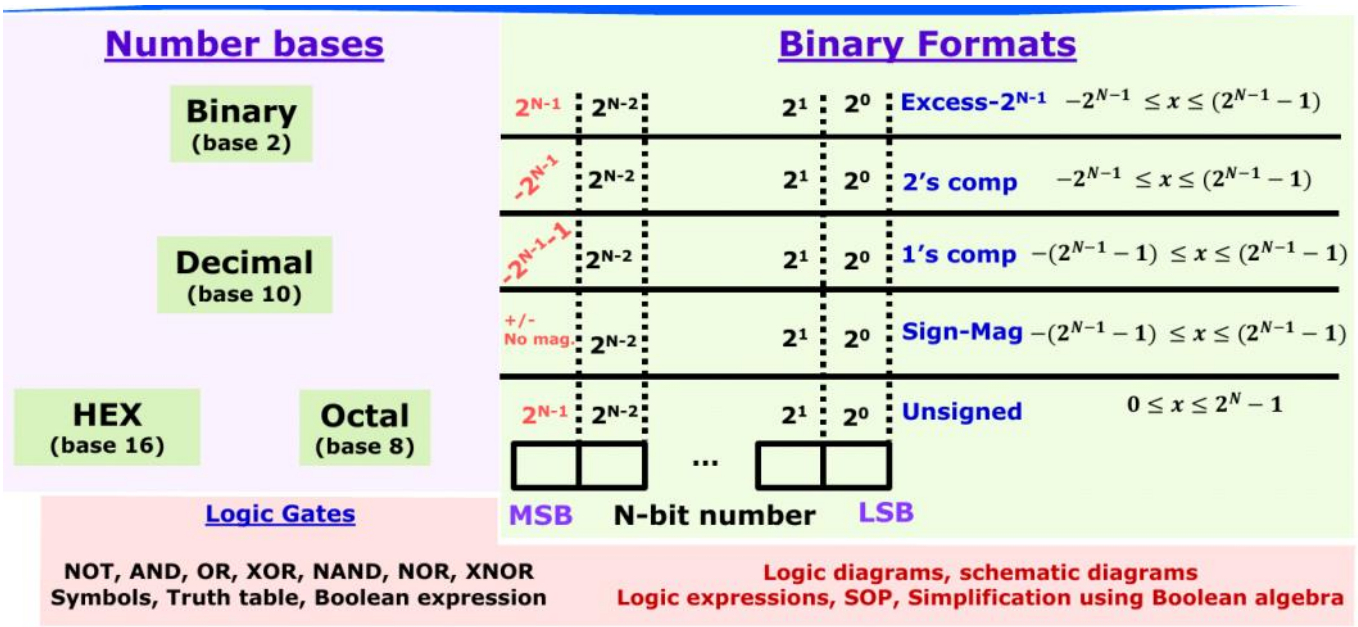
An Example



Recap of Week 1

Monday, 2 August 2021 6:06 PM

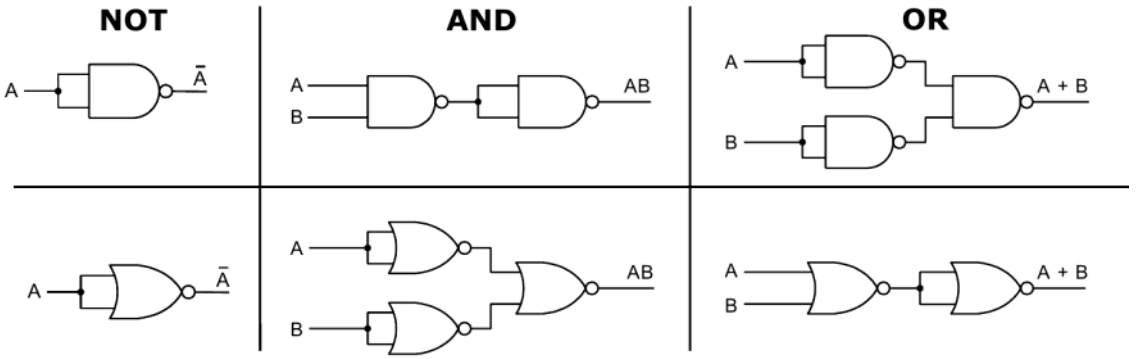
Number bases and binary formats



Equivalent Circuits (NAND and NOR)

Tuesday, 3 August 2021 11:53 AM

- All circuits can be constructed from NAND or NOR gates
  - These are called complete gates
  - This is done because it is much easier to build NAND and NOR gates from transistors.



Binary Arithmetic

Tuesday, 3 August 2021 11:54 AM

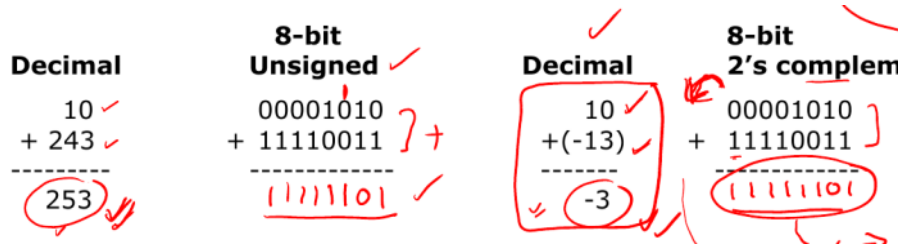
Addition

- Addition is quite simple in binary
- We shall see circuits that implement the addition operation later

Addend	0	0	1	1
Augend	+0	+1	+0	+1
Sum	0	1	1	0
Carry	0	0	0	1

Note: this ignores the carry in bit

Binary Addition



- Format matters when you interpret the numbers
- Whatever the format is, the bit-wise addition (which leads to the hardware circuit we will be looking at) is the same
- Two's complement - you don't need to do anything with the carry out from the MSB to get the correct result
- But in one's complement you will have to add the carry out from the MSB back to the result to get the correct answer. This is one drawback of one's complement representation.

Overflow in Binary Addition

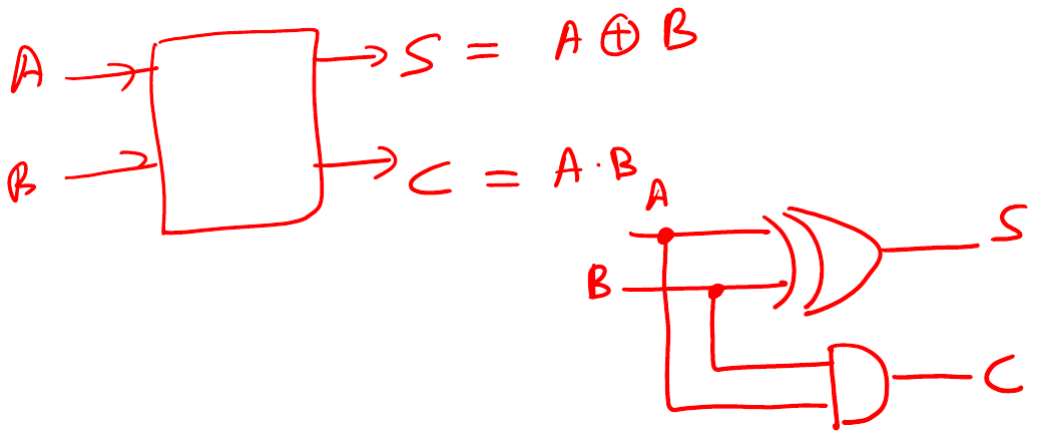
Overflow refers to the problem that arises when there is not enough bits to represent the answer. Thus the result goes out of range and the answer is incorrect.

Detecting Overflow

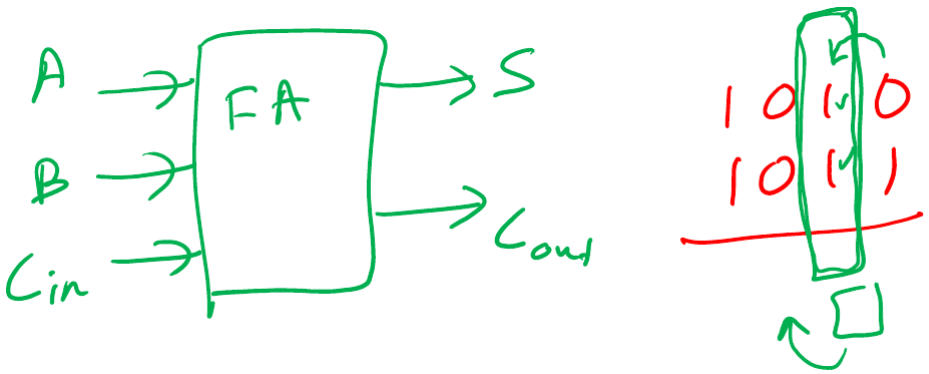
- Unsigned: carry-out from the MSB means overflow
- 2's complement: if the carry-in to the MSB and carry-out from the MSB are different (i.e. one is a 1 and one is a 0: order doesn't matter), then there is overflow
- Equivalently, overflow occurs if (in 2's comp)
  - Two negatives added together give a positive, or
  - Two positives added together give a negative.

Binary Addition

- A device which adds 2 bits (with no carry-in) is called a **half-adder**.



- However, we might have to deal with a carry-in. There may be a carry-in from the previous stage in the computer.



Addition of Binary Words

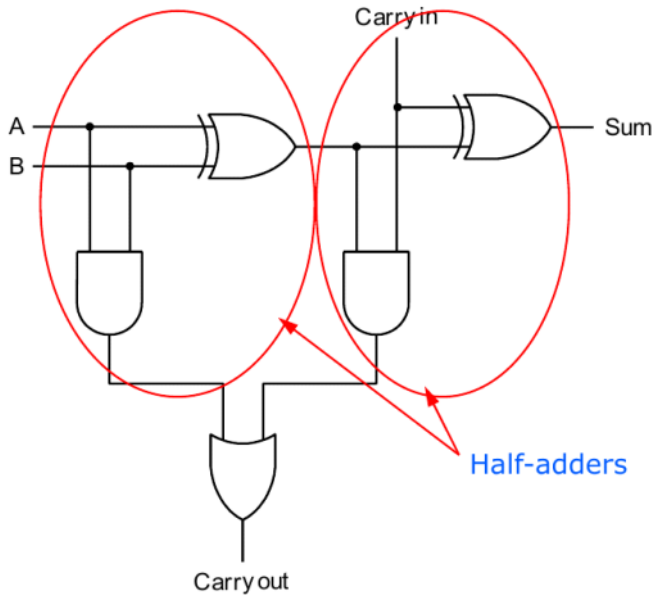
- Have to be able to deal with carry-in

A	B	Cin	Cout	Sum
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

ODD

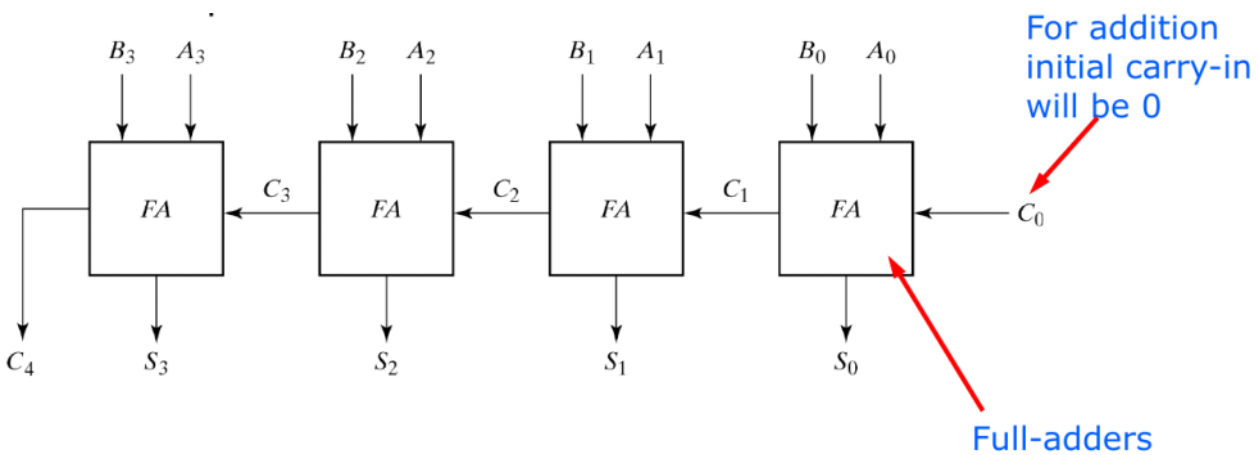
$A \oplus B \oplus C_{in}$

Full Adder



Binary Adder

- Can cascade full adders to make binary adder
  - Example: for 4 bit adding:



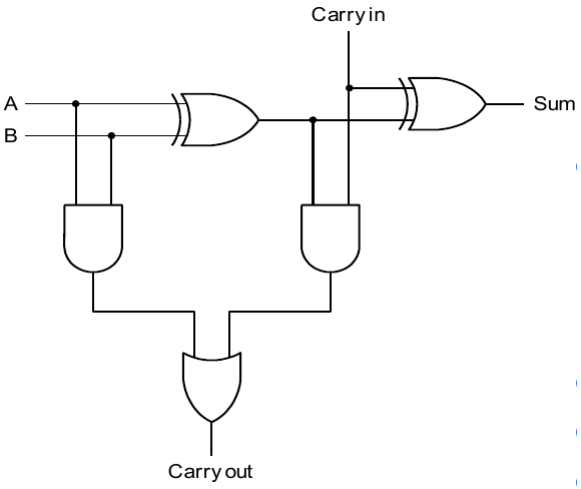
- This a **ripple-carry adder**. It is the most basic type of binary adder.

Adders (Practical)

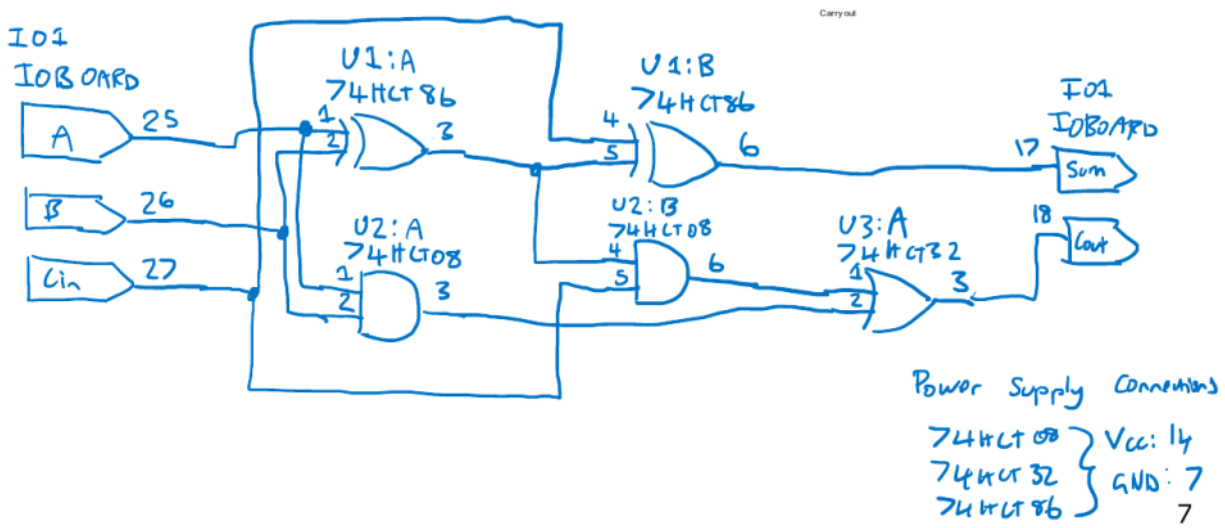
Saturday, 7 August 2021 8:42 PM

1-Bit Full Adder (from lecture)

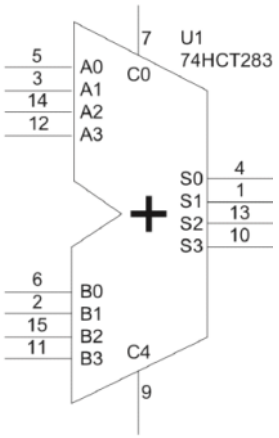
- Draw a circuit schematic diagram for a circuit which implements a full adder
  - Use switches for the three inputs and LEDs for the two outputs



Circuit Schematic



74HCT283: Single chip 4-bit adder


7
U1
74HCT283
5
A0
C0
3
A1
14
A2
12
A3
S0
4
S1
1
S2
13
S3
10
6
B0
2
B1
15
B2
11
B3
C4
9

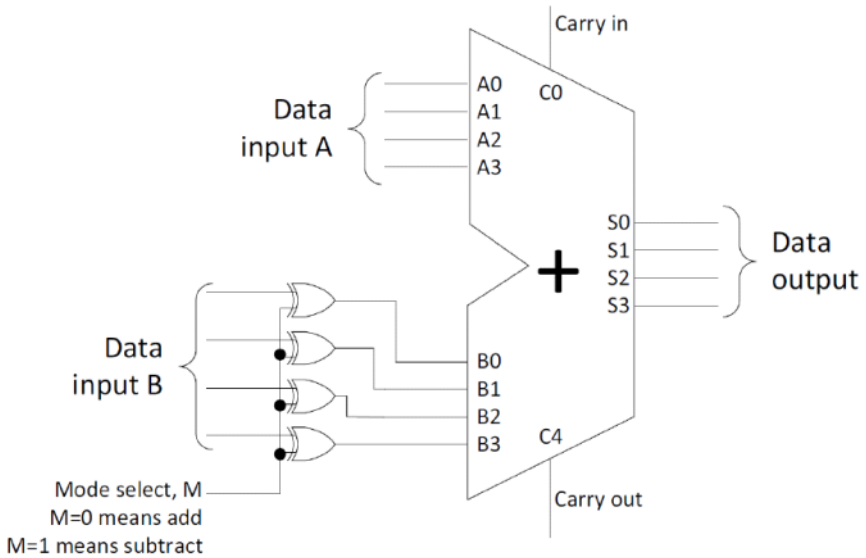
Binary Subtraction

Thursday, 5 August 2021 12:50 PM

Binary Subtraction

- $A - B$: usually implemented as $A + (-B)$
  - A and B are multi-bit quantities
  - + in this case means addition and not "or"
  - $-B$ means negative B. This can be calculated by getting the two's complement of B.
- Two's complement of B can be calculated by flipping each bit and adding one to the result.

- How can we use a gate to flip a bit - but only sometimes?



In this circuit, the carry in is connected to M. When M is 0, the circuit is in add mode and there is no carry in. When M is 1, the circuit is in subtract mode and so 1 is added to B (after its bits have been flipped (at data input B). Therefore, this results in the two's complement of B.

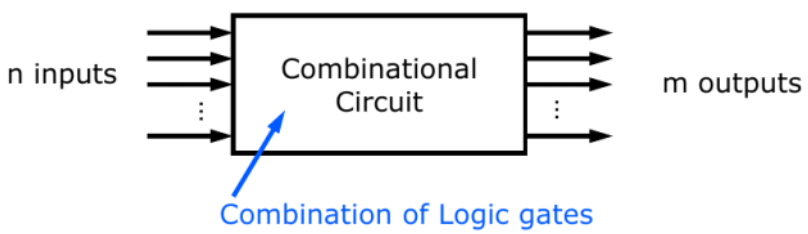
How Data input B works

For each bit of B, it is XOR'd with M. So If a bit from B is 1 and the circuit is in add mode (M=0), B will remain 1. If B is 0 and M is 0 then B will remain 0. However, if M is 1 because the circuit is in subtract mode, B=0 will become 1 and B=1 will become 0 because $0 \oplus 1 = 1$ and $1 \oplus 1 = 0$.

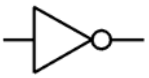
So in other words, the XOR logic gate maintains the B bit if $M = 0$ but flips the B bit if $M = 1$. This is what allows the circuit to act as an adder or a subtractor.

Combinational Circuits

- Generally



- Each output can be expressed as a function of n input variables
- Output depends on current inputs only
- Can write truth table also:
    - n input columns
    - m output columns
    - 2^n rows (i.e possible input combinations)

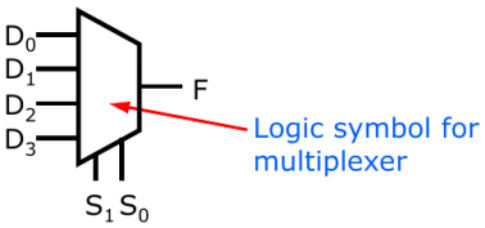


Multiplexer (Mux)

Thursday, 5 August 2021 9:44 PM

Multiplexer (or Mux)

- 2^n data inputs
- 1 output
- n control (or **select**) inputs - that *select* one of the inputs to be "sent" or "steered" to the output
- Example: 4-to-one multiplexer

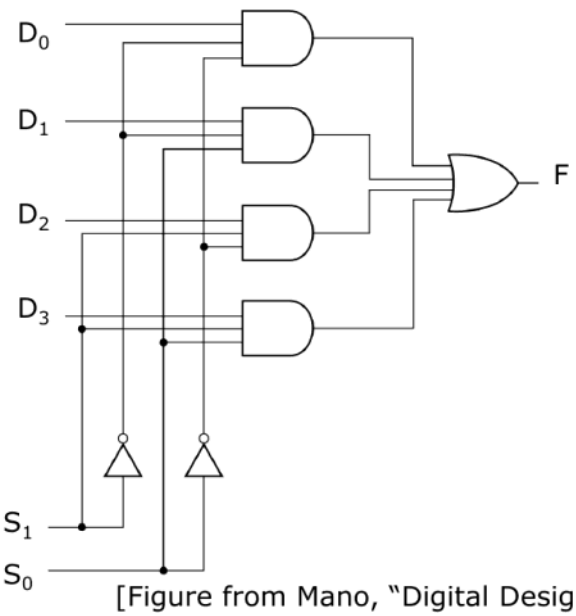


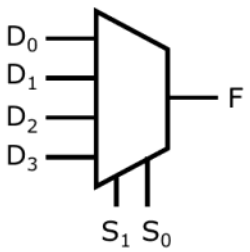
Function table

S_1	S_0	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

"Select" inputs

A 4-to-1 Multiplexer Logic Circuit Implementation





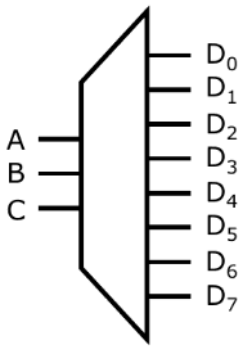
[Figure from Mano, "Digital Design", 3rd edition]

Decoder

Monday, 9 August 2021 8:38 AM

Decoder

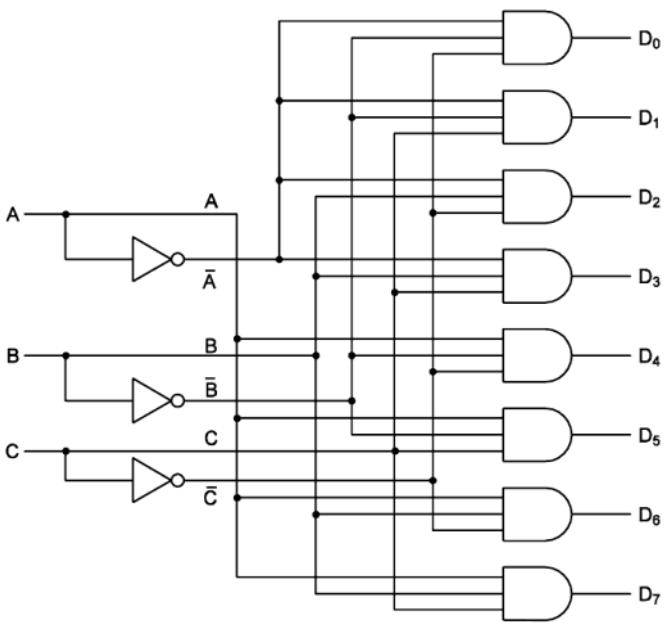
- Converts n-bit input to a logic-1 on exactly one of 2^n outputs
  This is how it is drawn in logic diagrams.



- Example: 3-to-8 decoder
  - Truth table

A	B	C	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	**1**	0	0	0	0	0	0	0
0	0	1	0	**1**	0	0	0	0	0	0
0	1	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	**1**	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...

Logic Implementation of a 3-to-8 decoder

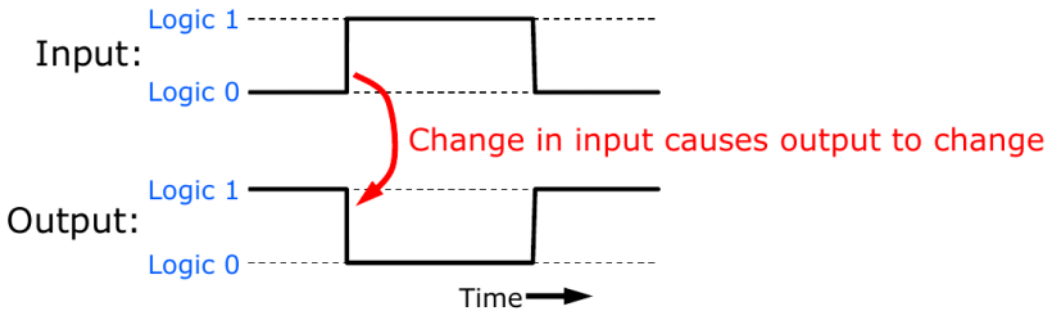


Timing Diagrams

Thursday, 5 August 2021 9:44 PM

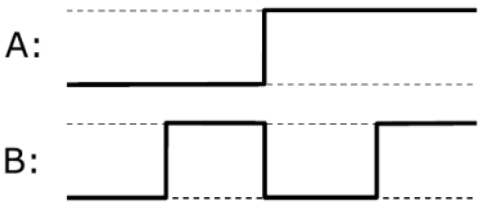
Timing Diagram

Logic Representation

- Consider an inverter
- 

- Logic can be represented graphically using a timing diagram as shown above. With logic 1 representing a 1 and logic 0 representing a 0.
- This is like a truth table, just in graphical format
  - Input waveforms show all possible combinations.

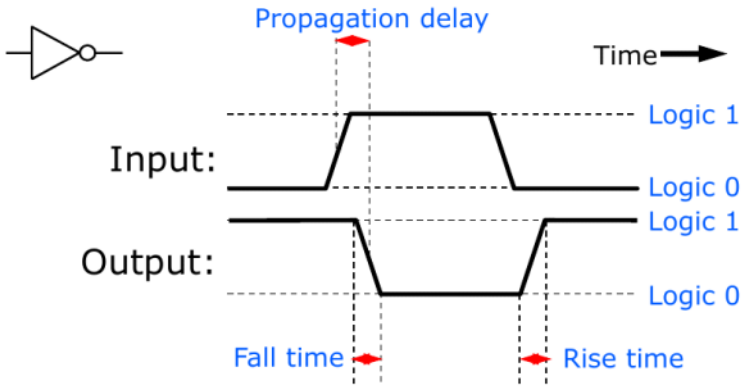
- Consider a 2-input gate



The above shows all possible combinations

Reality of Gates (delay)

In reality, gates aren't perfect: there are some delays associated with them. They are summarised below.



- **Propagation delay** - time taken for change in an input to affect the output
- **Fall time** - time taken for output to fall from 1 to 0
- **Rise time** - time taken for output to rise from 0 to 1.

Circuits that Remember Values

Saturday, 7 August 2021 8:46 PM

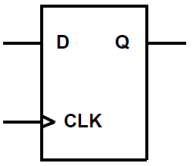
Memory in a Circuit

- The output of any logic gate or combinational circuit is dependent on the current value of inputs only
- If an input changes, the output can also change and the previous value is lost forever
- Sequential circuits: the current output depends not only on the current inputs but also on past outputs
- Circuits with memory can remember values, even if the input changes.

Flip-Flops

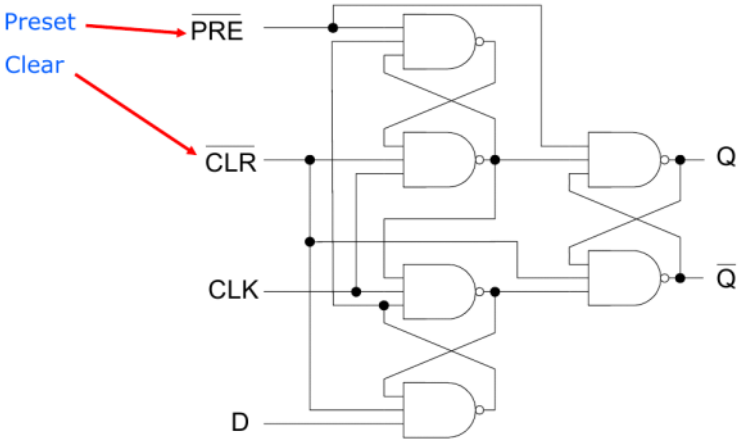
Saturday, 7 August 2021 8:52 PM

Memory Element: D Flip-Flop

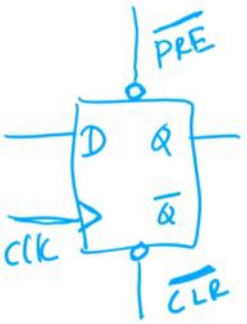


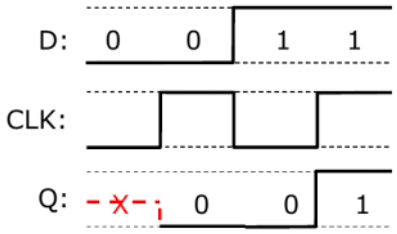
- D is input
- Q is output
- CLK (clock) is the control input
- It works by
  - Q copies the value of D (and remembers it) whenever CLK goes from 0 to 1 (rising edge)

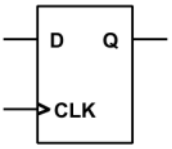
A D Flip-Flop logic Diagram



Bar above CLR and PRE together means "active-low" inputs. This is also indicated with a bubble at the connection. Remember, bubbles mean negate bit.







Remember: Q changes ONLY when CLK goes from 0 to 1

Characteristic Tables

- Characteristic table defines operation of flip-flop in tabular form
  - D flip-flop

D	Q(t+1)	
0	0	Reset
1	1	Set

Input

What the output will be on the next clock edge (i.e., CLK goes from 0 to 1).

Flip-Flops in Use

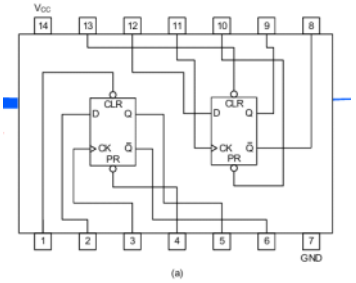
- Generally, you will need n D flip flops to remember n bits
- Definition: An n-bit register can be made using n D flip flops
- There are other types of flip flops including

- JK flip flops
- T flip flops

- Flip flops can be made out of logic gates

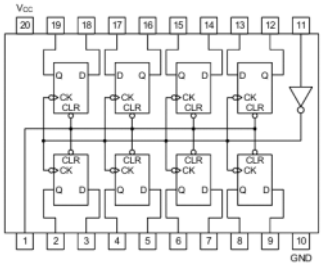
D Flip-Flop Chips

1. 74HCT74 chip (dual D flip-flop)



(a)

2. 74HCT273 (Eight D-flip flops)

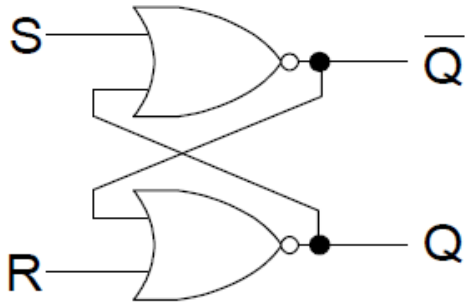


(b)

Latches

Saturday, 7 August 2021 8:51 PM

SR Latch

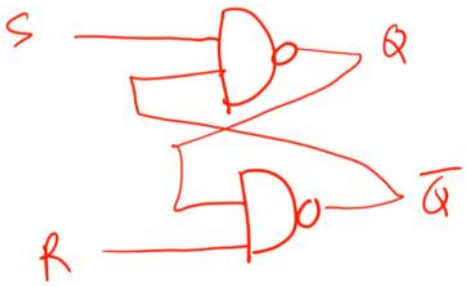


S	R	Q	$\overline{Q}$	
0	0	Q_{n-1}	$\overline{Q}_{n-1}$	← REMEMBER (STORE)
0	1	0	1	← RESET ✓
1	0	1	0	← SET ✓
1	1	0	0	← Not valid ✗

Remember, this is what a NOR gate does

A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Latches from NAND Gates

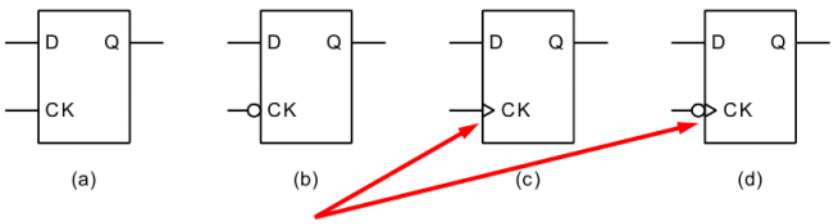


Flip-Flops vs Latches

Saturday, 7 August 2021 8:52 PM

- Latches are level triggered devices - i.e. able to latch the output and respond to changes of logic levels on the inputs
- Latch circuits can be modified such that they become sensitive to an edge (i.e. momentary transitions) of a control input (i.e. a clock signal)
- Such circuits are called "flip-flops" and a flip-flop can store 1 bit of information while being sensitive to a clock edge (i.e. a flip-flop will change its output only at the clock edges, based on the inputs)
- Latches are level triggered devices while flip-flops are edge triggered devices
- A clock signal has two edges
  - Positive edge - 0 to 1 transition
  - Negative edge - 1 to 0 transition
- There are different types of flip-flops
- So a D flip-flop can be positive edge triggered or negative edge triggered
  - Positive edge triggered D flip-flop -> Input D is copied to output Q at the positive edge of the clock. In between the clock edges, flip-flop is not responsive, thus stores the value.

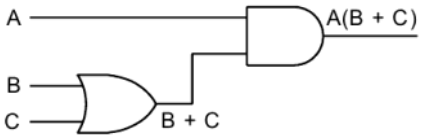
Symbols Used



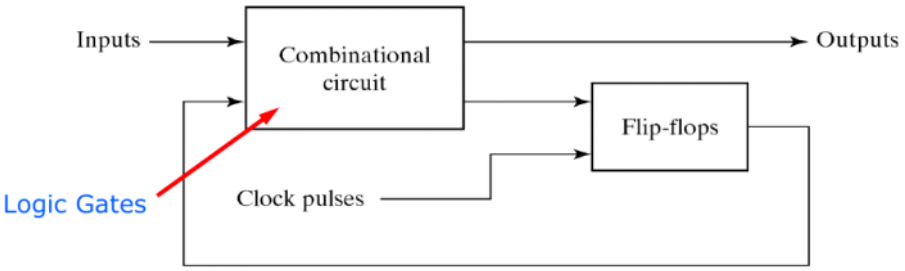
- Triangle indicates edge-triggers (therefore flip-flop)
  - (c) sensitive to rising edge of clock
  - (d) sensitive to falling edge of clock
- State of a flip-flop is the value stored
- Flip-flops more useful than latches

Combinational vs. Sequential Circuits

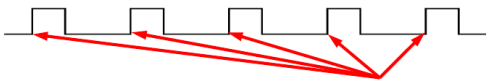
Saturday, 7 August 2021 9:04 PM

- Combinational Circuits (last week)
  - Logic gates only (no flip-flops)
  - Output is uniquely determined by the inputs
    - i.e. you'll always get the same output for a given set of inputs
  - Example:
    - 

- Sequential Circuits
  - Include flip-flops
  - Output determined by current inputs and current state (values in the flip-flops)
  - Outputs can only change when clock `ticks`
  - State = value stored in flip-flops



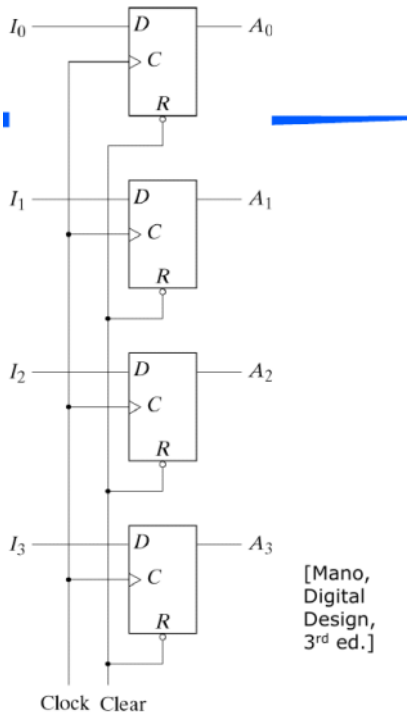
Synchronous Sequential Circuit

- Storage elements (flip-flops) can only change at discrete instants of time
- Assume
  - We have a clock signal
  - 
  - Output of storage elements change only on the edges of control signal
    - (compare with logic gates whose output changes whenever the input changes)

Registers

Saturday, 7 August 2021 9:06 PM

- A register is a group of flip-flops
  - n-bit register consists of n flip-flops capable of storing n-bits
- Example
  - 4-bit register



[Mano, Digital Design, 3rd ed.]

- A register is a sequential circuit *without* any combinational logic

Shift Register

Saturday, 7 August 2021 8:46 PM

- A shift register is a register which is capable of shifting its binary information in one or both directions
- Example

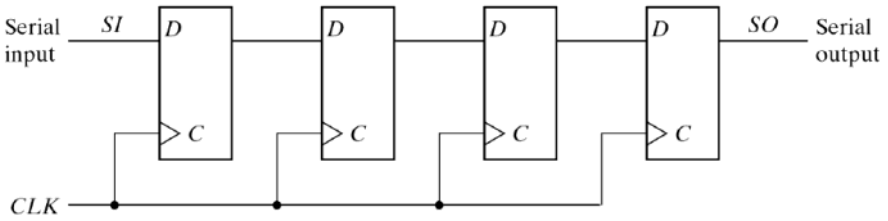
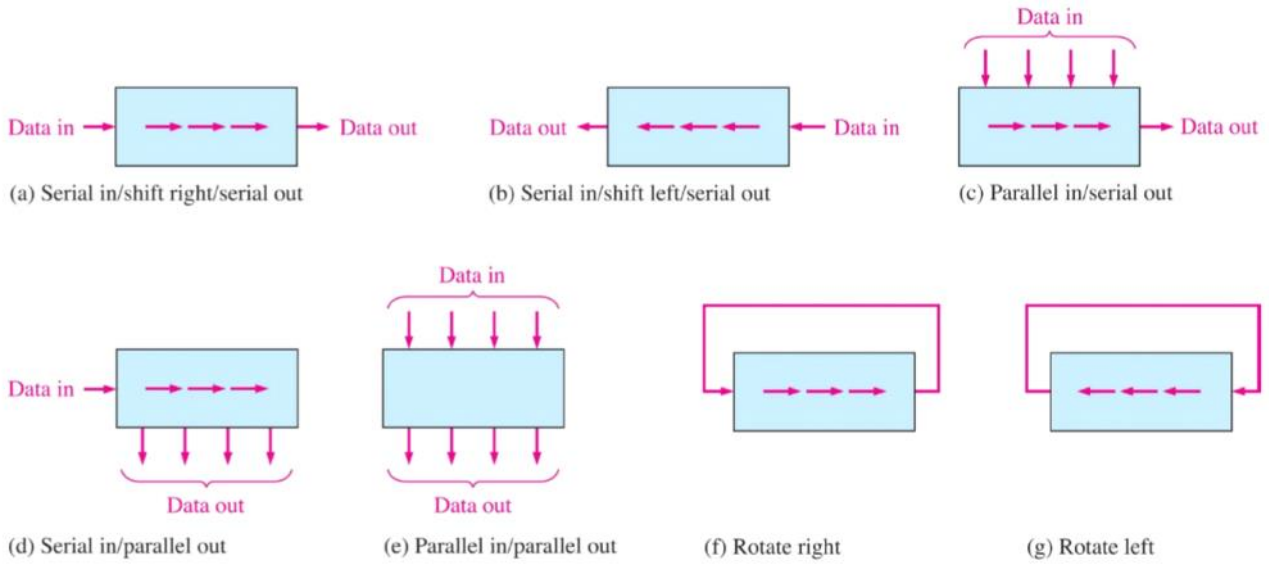
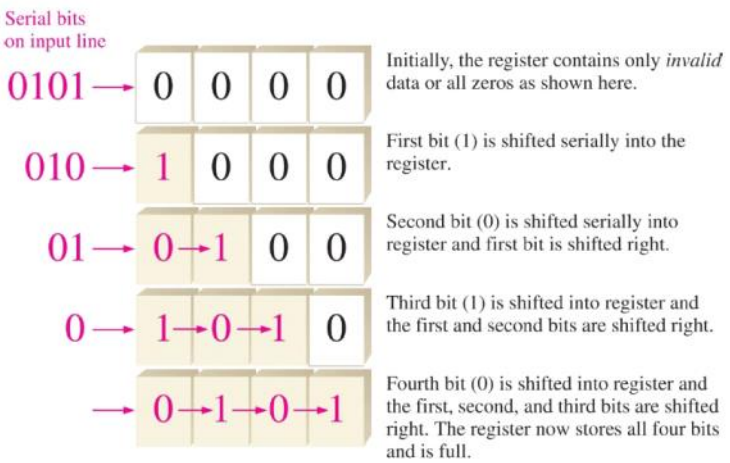


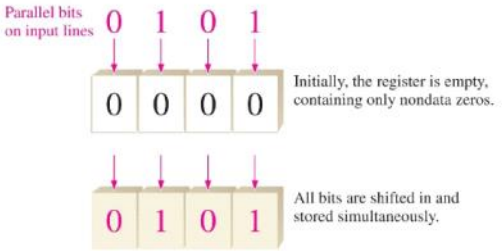
Fig. 6-3 4-Bit Shift Register

Types of Shift Registers



Loading Data in Serial and Parallel Shift Register



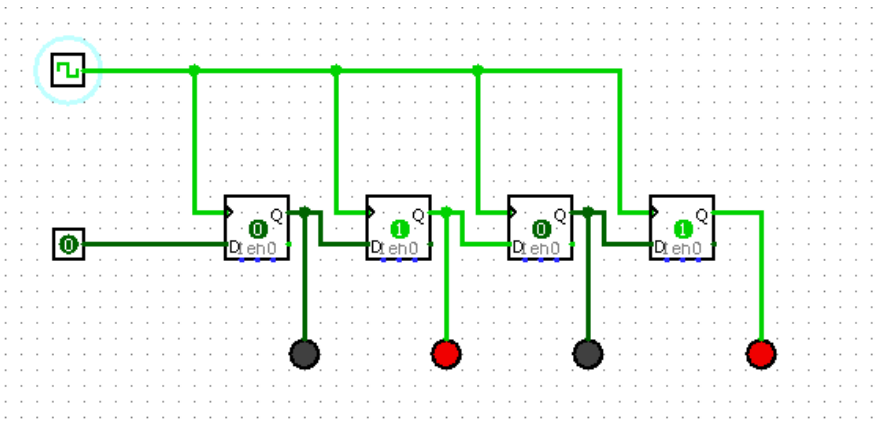


Serial ⟺ Parallel Conversion

Monday, 16 August 2021 5:23 PM

Serial ⟺ Parallel Conversion

- Shift registers can be used to do serial to parallel conversion (and vice-versa)

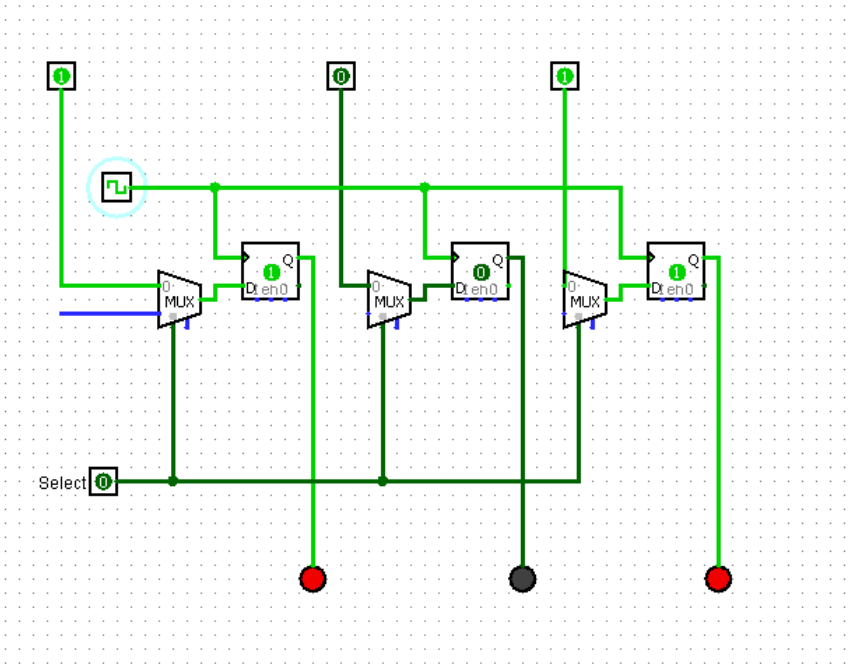


After input 0101 serially. There is a parallel output 0101.

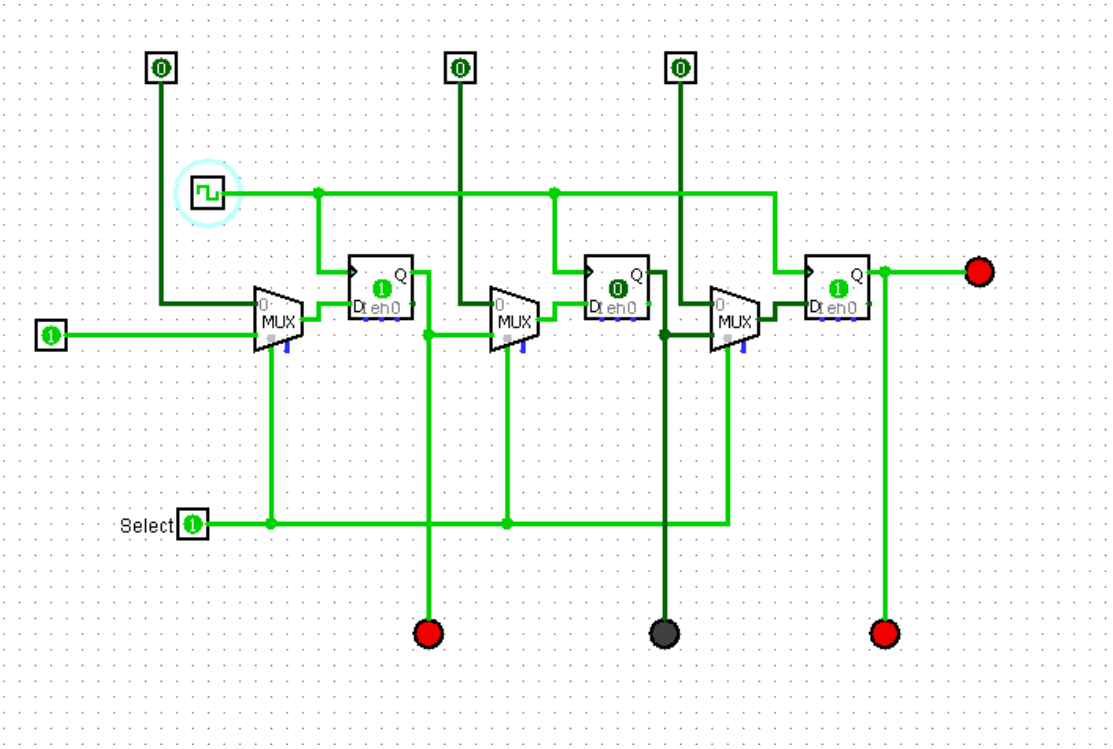
Parallel to Serial and Serial to Parallel

Select input
0 -> Parallel load.
1 -> Serial In/Shift
^^^^
Using a 2:1 Mux

Load refers to loading a parallel number (in one clock cycle). This is done by connecting D inputs in parallel



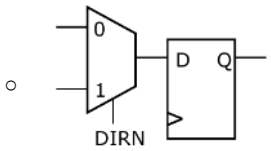
^^^^^
Parallel load of 101 into register.

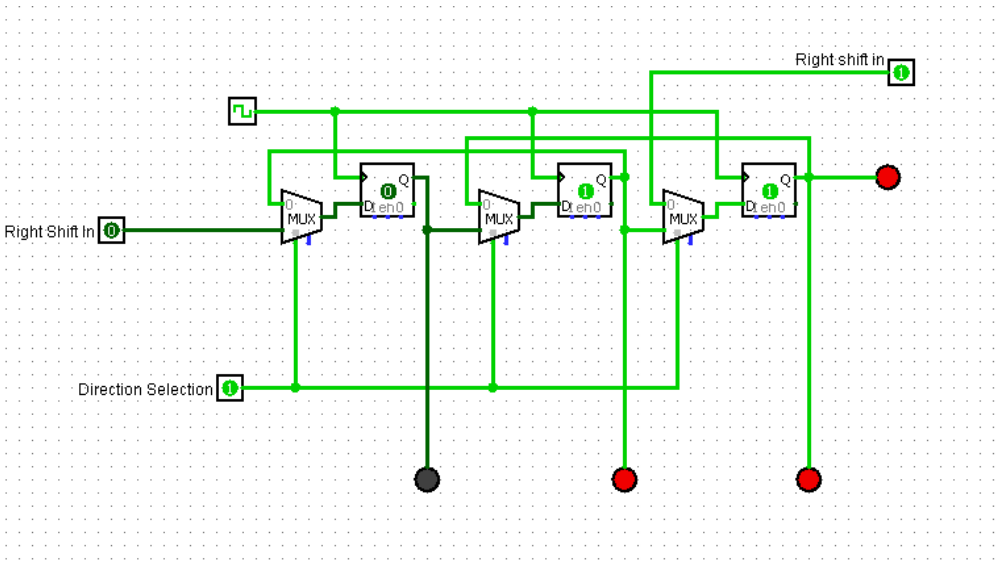


Serial load of 101 into register.

Bi-directional Shift Register

- Using same multiplexer concept, draw a 3-bit shift register which allows data to be shifted in either direction
- Hint: consider this element, where DIRN will be 0 for left shift, 1 for right shift

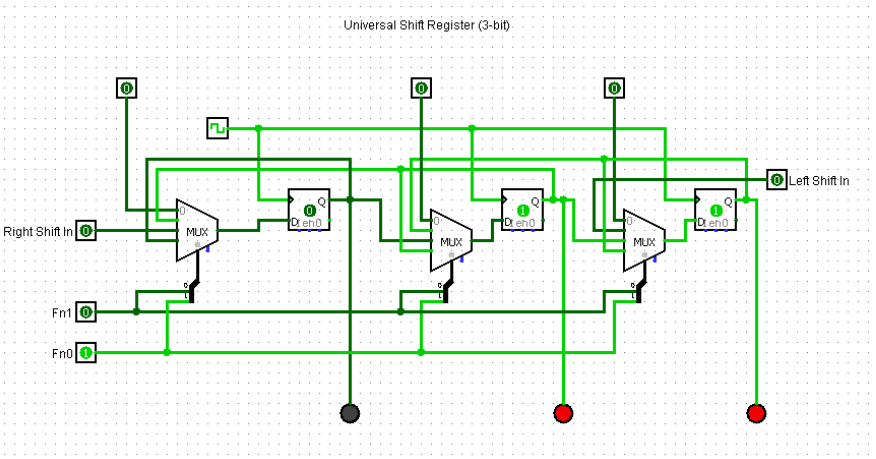




Shifting bits is very important in a register.

- Shifting to the right is equivalent to dividing by 2
- Shifting to the left is equivalent to multiplying by 2

Universal Shift Register



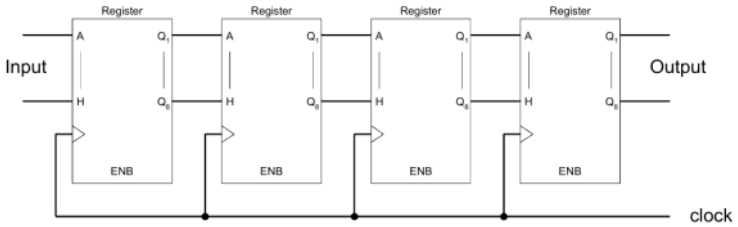
Function table

FN1	FN0	Operation
0	0	→ Parallel load
0	1	→ left Shift
1	0	→ Right shift
1	1	→ Stay same

15

8-Bit Wide Shift Register

- Multiple bits shifted at a time
- Example 4-stage 8-bit queue:



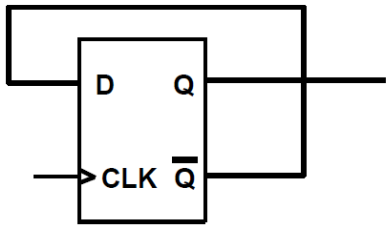
Counters

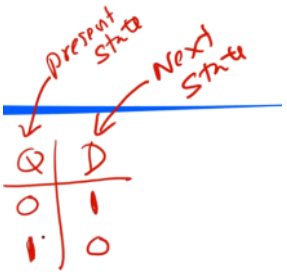
Monday, 16 August 2021 5:21 PM

- A **counter** is a multi-bit register that goes through a determined sequence of states (values) upon the application of input (clock) pulses.
- A counter which follows binary number sequence is a **binary counter**
  - n-bit binary counter
    - Has n flip-flops
    - Can count from 0 to $2^n - 1$ thus has 2^n different states
  - e.g. 2-bit binary counter will count
    - $00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00$
    - i.e. $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$

A 1-bit Counter

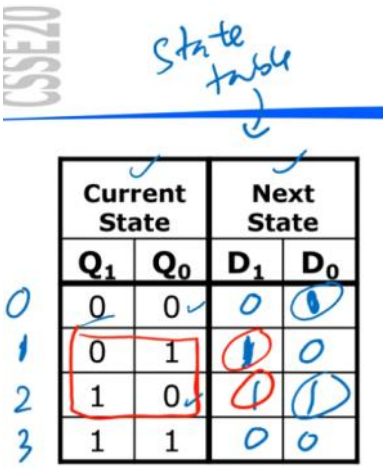
$D = \bar{Q}$



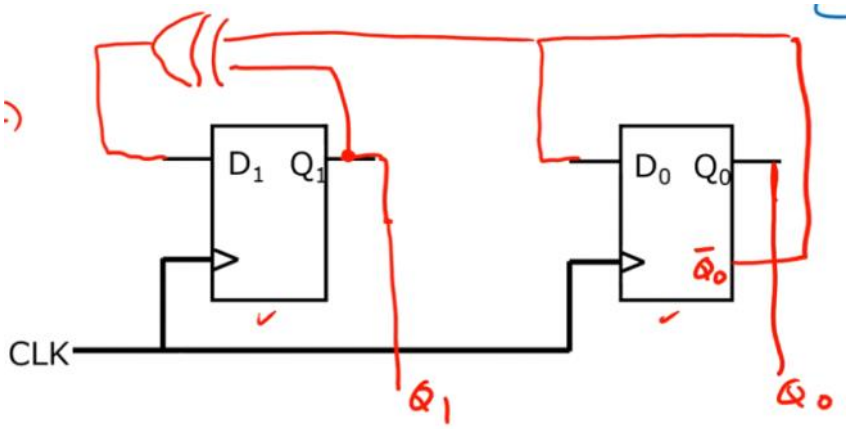


A 2-bit Counter

State Table



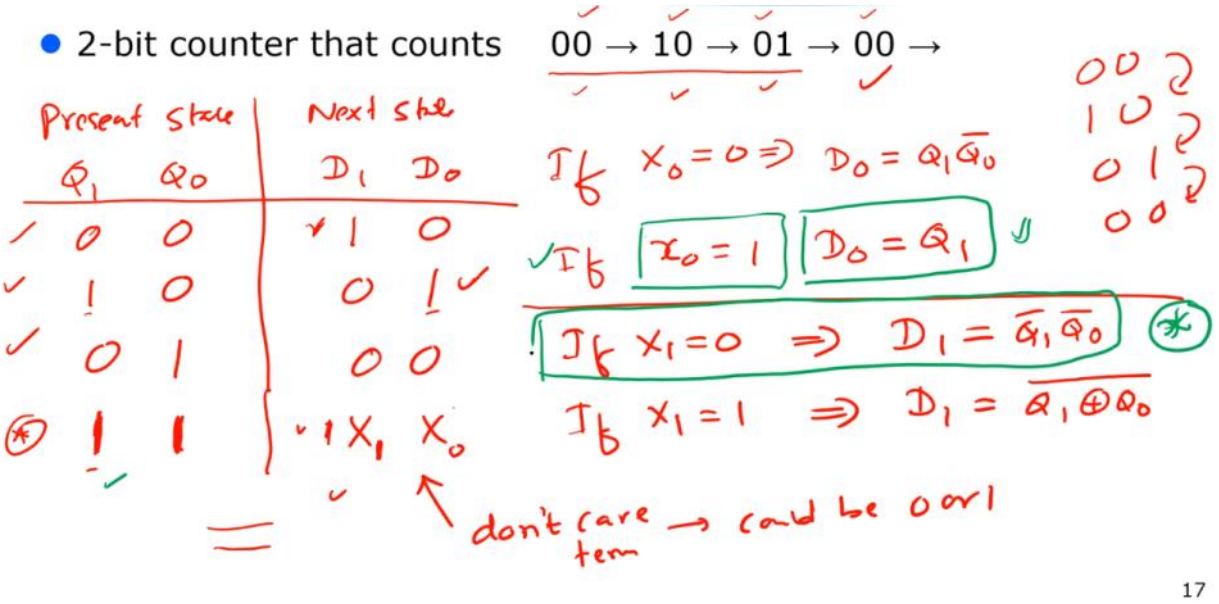
Circuit Diagram

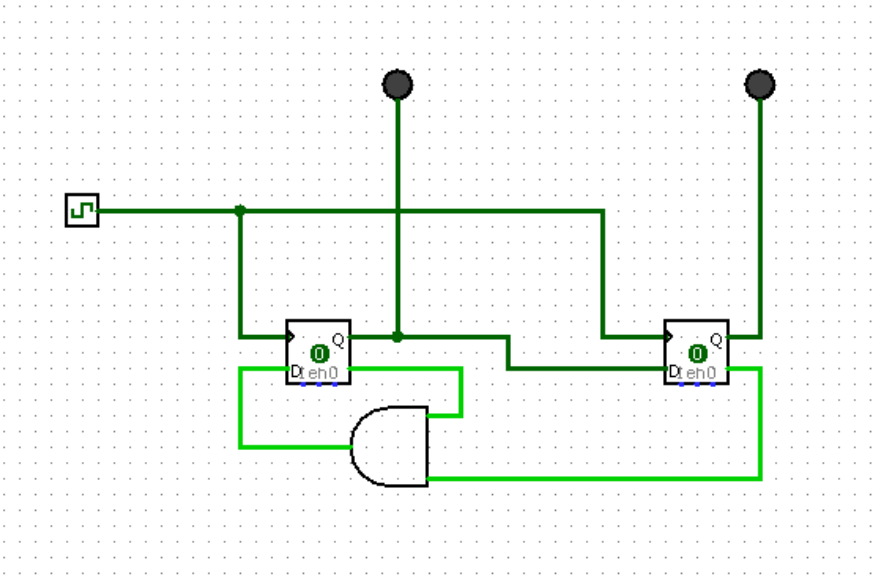


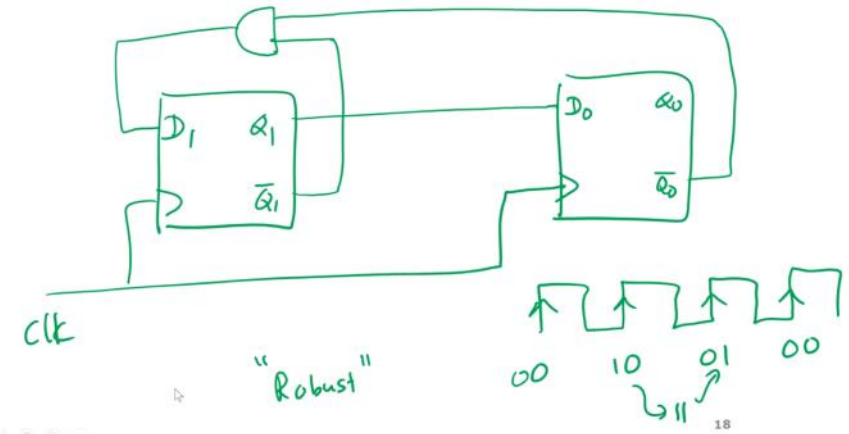
Key Points

- Next state if a function of the previous state (and possibly inputs)
- Count sequence can be binary numbers but does not have to be
  - If it is, counter is a **binary counter**
- Circuits are **synchronous**
  - All flip-flops have the same clock
- There are asynchronous counters as well - we won't discuss these however.

A 2-bit counter that counts 00 -> 10 -> 01 -> 00 ->




Q
D
en
0
Q
D
en
0


D1
Q1
Q̄1
D0
Q0
Q̄0
clk
"Robust"
00
10
01
00
11
18

State of Circuit

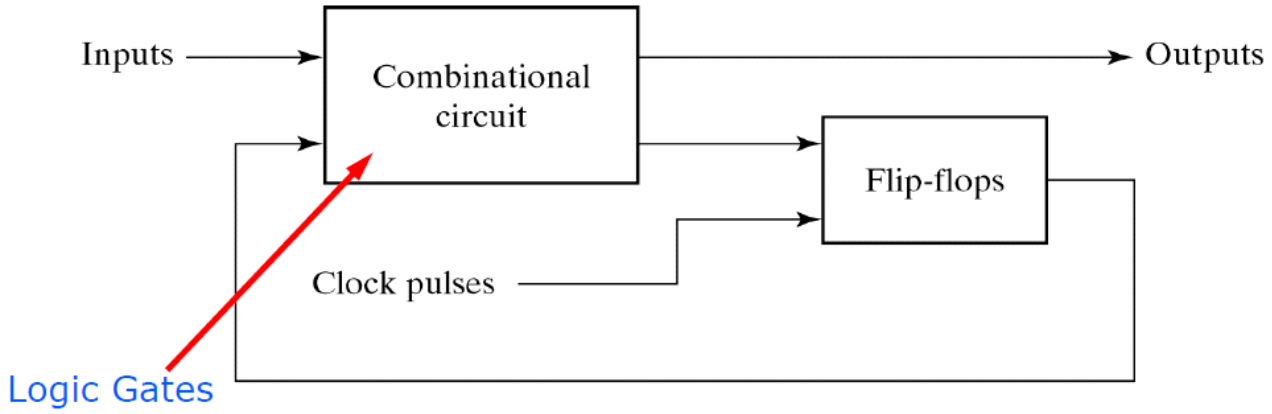
Wednesday, 18 August 2021 9:24 AM

- Values stored in the flip-flops can be considered the **current state** of the circuit
- D inputs to the flip-flops are the **next state**.
- D inputs are some function of the current state and inputs.

Sequential Circuits

Monday, 16 August 2021 5:21 PM

- **State** = value stored in flip-flops
- Output depends on input and state
- Next state depends on inputs and state



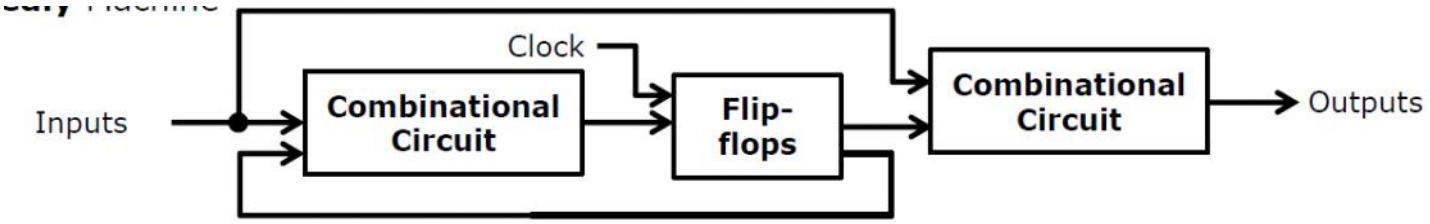
State Machines

Wednesday, 18 August 2021 9:27 AM

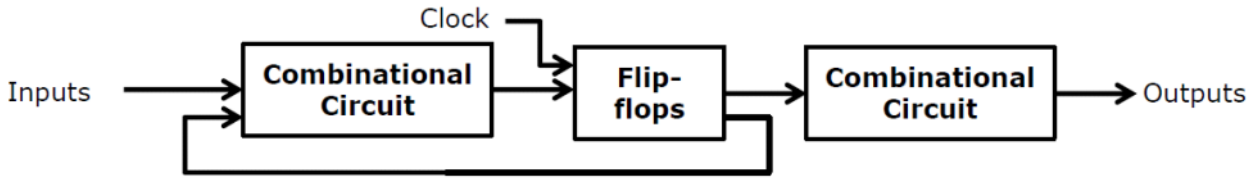
State Machines

- Sequential circuits can also be called
  - State machines
  - Finite State Machines (FSMs)
- State machine has
  - Finite number of possible states
  - Only one **current state**
  - Can **transition** to other states based on inputs and current state
- The states can be defined based on the following problem
  - E.g. a vending machine accepts 5 cents and 10 cents coins to dispense a candy when it receives 15 cents in total. What can be the different states?
- Two types of state machines
  a. Mealy machines
     - Outputs depend on current state and inputs
  b. Moore machines
     - Outputs depend only on current state (flip-flop values)
       - Outputs can only change when state changes

Mealy Machine



Moore Machine

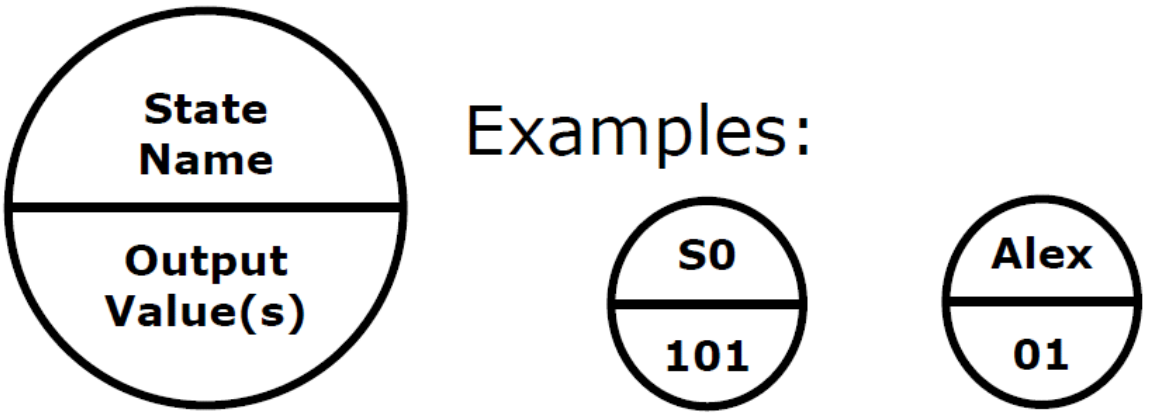


This course will predominantly use Moore machines in this course.

State Diagrams

Notation

- State



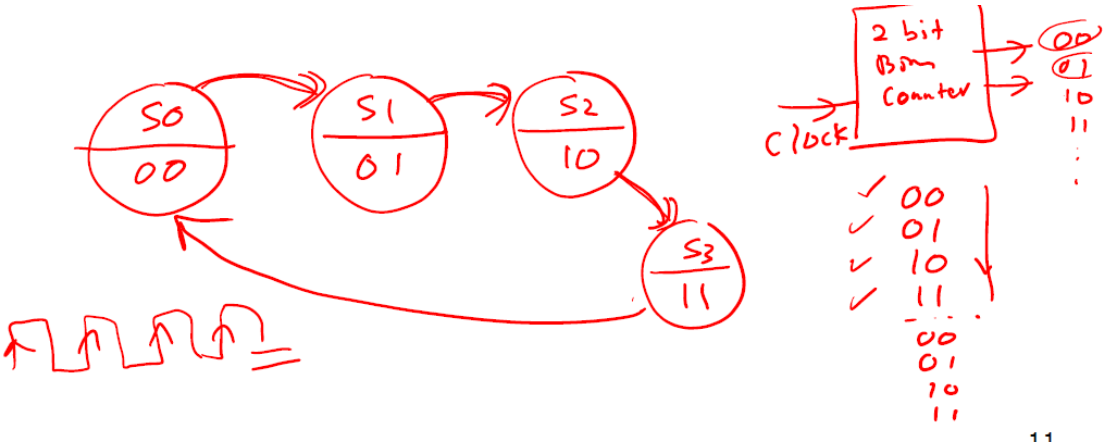
- Transition

- Logic expression → or → 0 → 1 → Special case for only one input
- If expression is true, state transition is made
- No label means "true" - i.e. always transition
- Examples

$\xrightarrow{Z}$ $\xrightarrow{\overline{RESET}}$ $\xrightarrow{A+B.\overline{C}}$ $\longrightarrow$

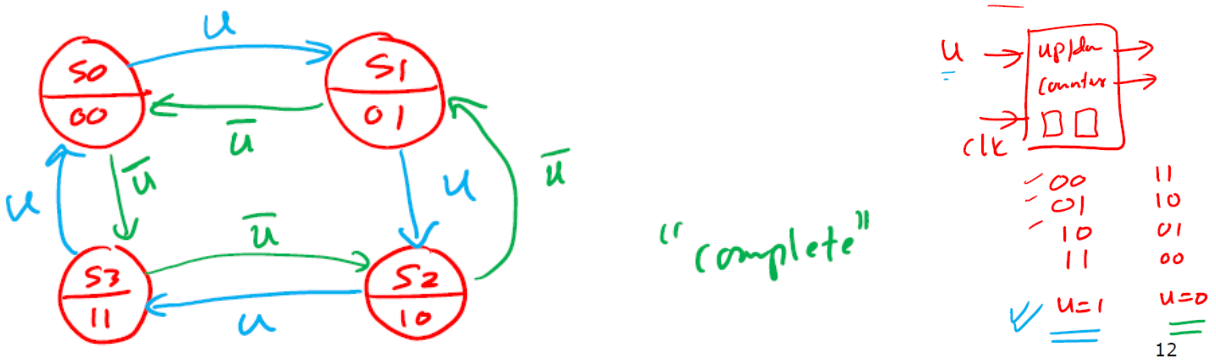
Example 1: State Diagram

- Binary counter with 2-bit output



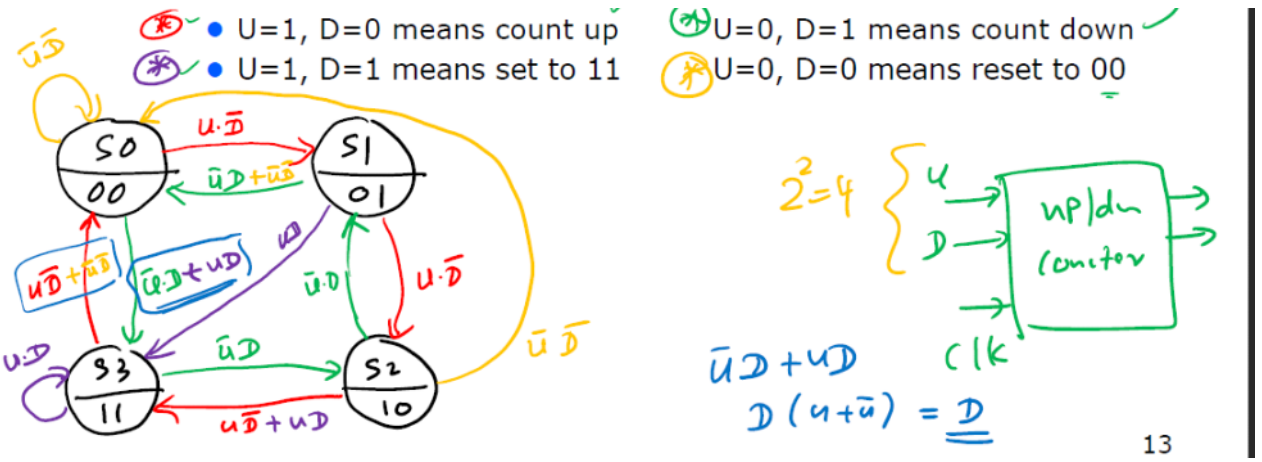
Example 2: State Diagram

- Binary up/down counter with 2-bit output
  - One input: U
    - 1 -> count up
    - 0 -> count down



Example 3: State Diagram

- Binary up/down counter with 2-bit output
  - Two Inputs: U, D
    - U = 1, D = 0 -> count up
    - U = 0, D = 1 -> count down
    - U = 1, D = 1 means set to 11
    - U = 0, D = 0 means reset to 00



Completeness (in state diagrams)

- Each possible combination of inputs should be addressed exactly once for each state
  - i.e. transition arrows from each state must encompass all possibilities (exactly once)
- Example

State Tables

- State diagrams can also be represented in a state table
- Example - binary up/down counter with 2-bit output and single input U (1 means up, 0 means down)
- One dimensional state table:

1D

Every combination of state and inputs

Current State	Input U	Next State	Outputs	
			Q1	Q0
S0	0	S3	0	0
S0	1	S1	0	0
S1	0	S0	0	1
S1	1	S2 ~~S1~~	0	1
S2	0	S1	1	0
S2	1	S3	1	0
S3	0	S2	1	1
S3	1	S0	1	1

00 01 10 11

17

- Two dimensional state table

Every combination of input

- Same example

2D Formal

One row per state

Current State	Next State		Outputs	
	$\bar{U}$	U	Q1	Q0
S0	S3	S1	0	0
S1	S0	S2	0	1
S2	S1	S3	1	0
S3	S2	S0	1	1

11

18

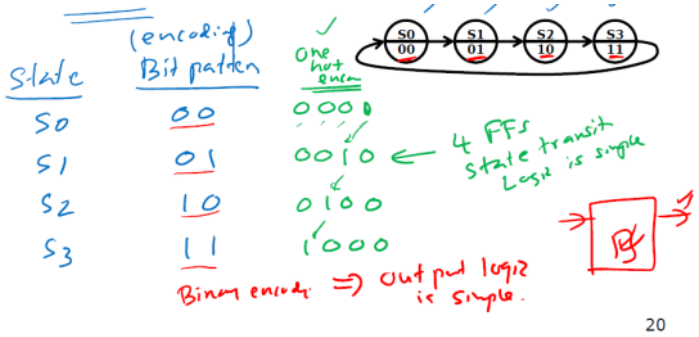
State Encoding

- Must encode each state into flip-flop values
- Choose
  - Number of flip-flops
  - Bit patterns that represent each state
- Ideally
  - Choose state encoding to make combinational logic simple, for both
    - Output logic
    - Next state logic

In basic terms, state encoding simply refers to the process of setting a bit pattern for each state of the circuit (as specified by the state diagram and state tables).

Example

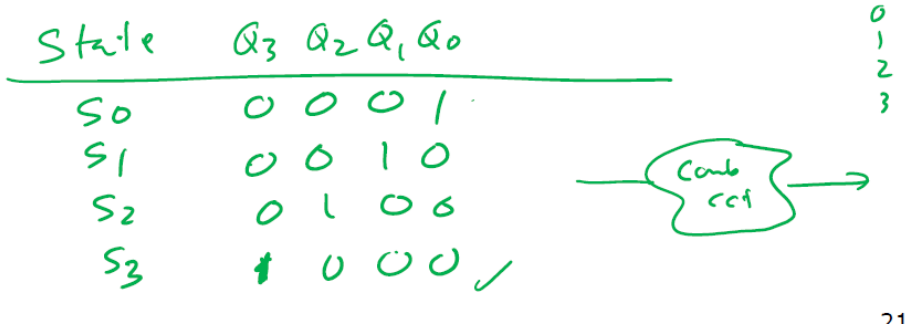
- Binary counter with 2-bit output



One-hot Encoding

A type of state encoding where exactly 1 bit is a 1 and the rest are 0.

- Use one flip-flop per state
- Only one flip-flop has 1 at any time
- Example - binary counter with 2-bit output



Sequence Detector Example

Wednesday, 18 August 2021 9:36 AM

- Design a state machine which detects the pattern 101 in an incoming bit stream and outputs 1 whenever it detects this pattern
  - Output is 1 for one clock cycle after the third digit is clocked in
- Example
  - Input: 00<u>101</u><u>1010</u>100
  - Output: 0000100010100

1 → defining the states

2 → Draw the state diagram (Moore)

3 → State table < 1D, 2D } still have state names e.g S_0, S_1.

4 → State encoding → i.e binary, one-hot

5 → State table again with encoding.

6 → get the expressions for state trans & output & draw the logic diagram

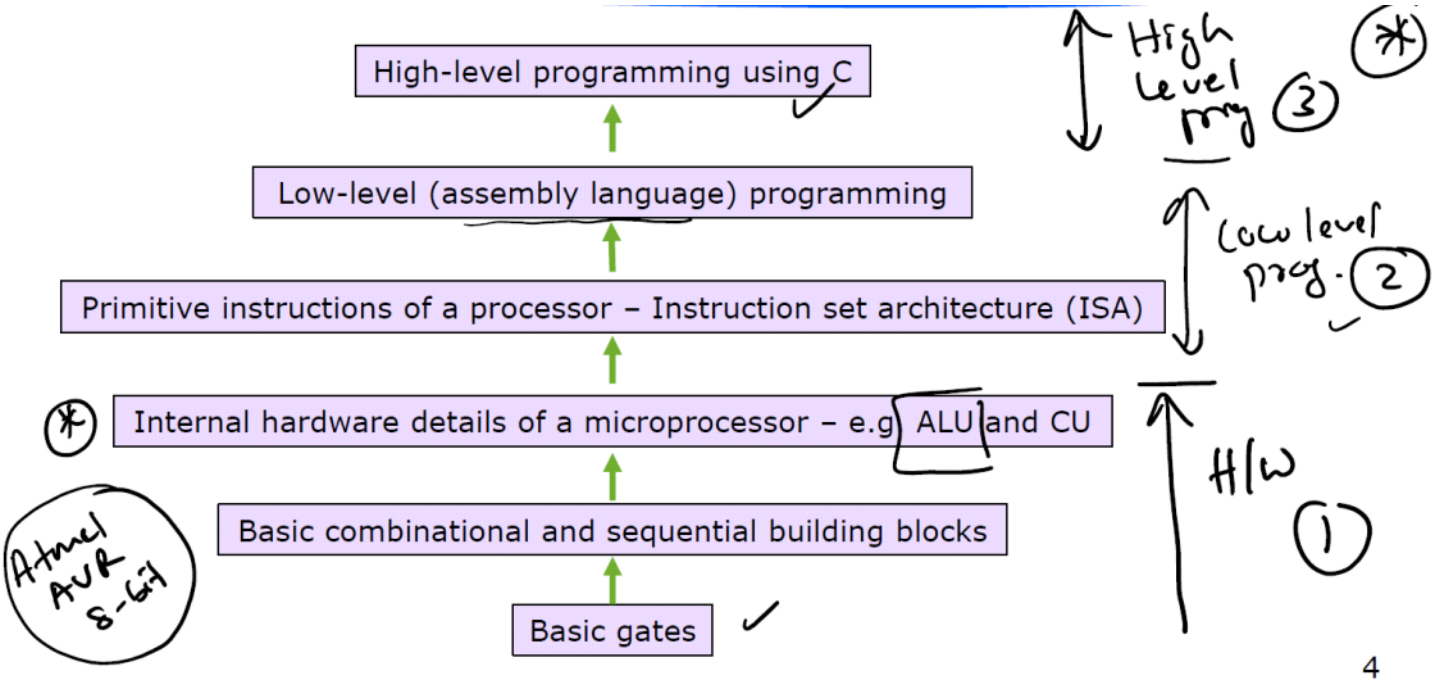
⟹ watch the example for steps 5 & 6.

A pre-recorded example will be provided on Blackboard

23

Hierarchy of a Computer

Monday, 23 August 2021 10:53 PM

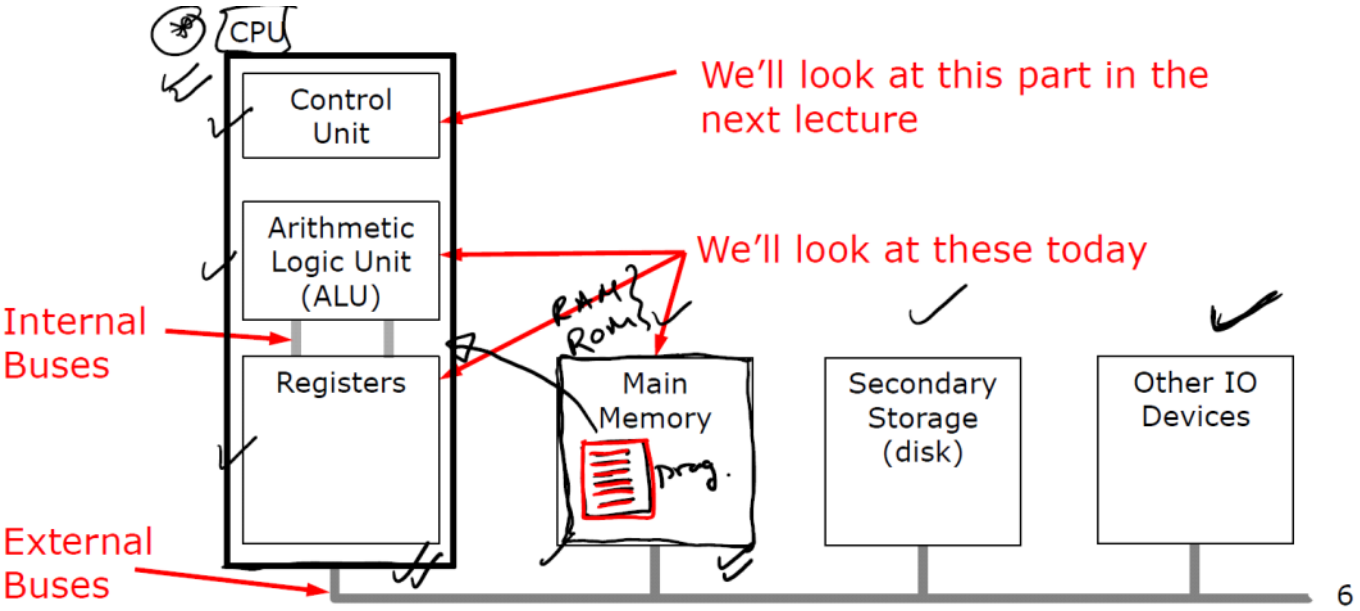


4

Computer Organisation

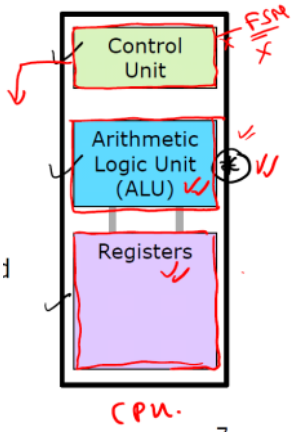
Monday, 23 August 2021 10:55 PM

Simplified view:



Parts of a CPU

- Control unit
  - Fetches instructions from memory. Then tells the ALU and the registers perform said instruction.
  - => Lecture 10
- ALU
  - Performs arithmetic and logical operations
- Registers
  - High speed memory - stores temporary results and control information
  - e.g. small CPU may have 8×8-bit registers
  - Large CPU may have 64×32-bit registers



Registers

- Registers are temporary storage locations inside the CPU.
- Two types of registers
  - **General Purpose Registers**
    - Contains data to be operated on (e.g. data read from memory), results of operations,
    - Also known as register file. Width is the CPU word size
  - **Special Purpose Registers**
    - Used by the system
    - Program Counter (PC) - stores the address of the next instruction to be fetched from the memory
    - Instruction Register (IR) - stores the current instruction.

Buses

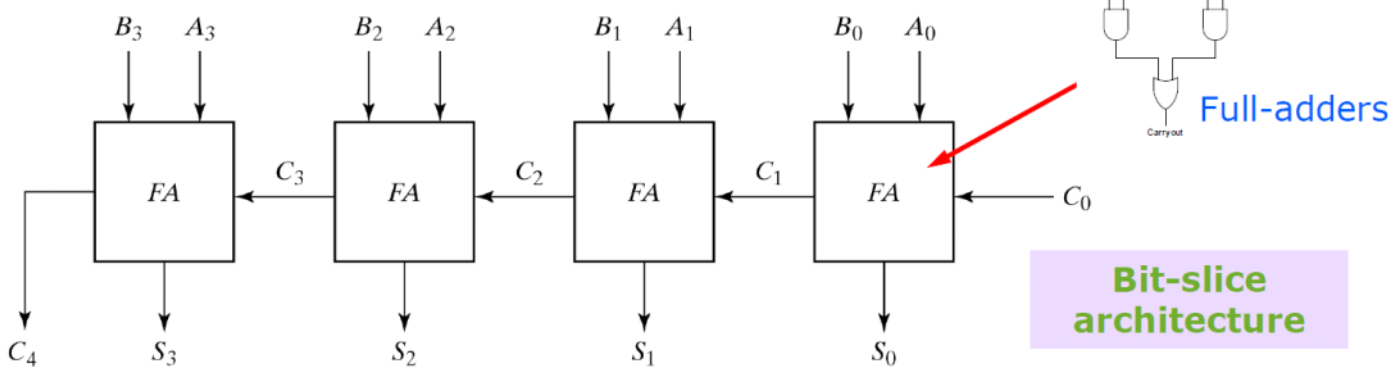
- **Bus** = Common pathway (collection of "wires") connecting parts of a computer
- Characteristics

- Can be **internal** to CPU (e.g. ALU to registers)
- Can be **external** to CPU (e.g. CPU to GPU)
- Buses have a **width** - number of bits that can be transferred together over a bus
    - e.g. 1-bit, 8-bit or 32-bit
    - May not always be the same as the word size of the computer

Binary Adder (Revisited)

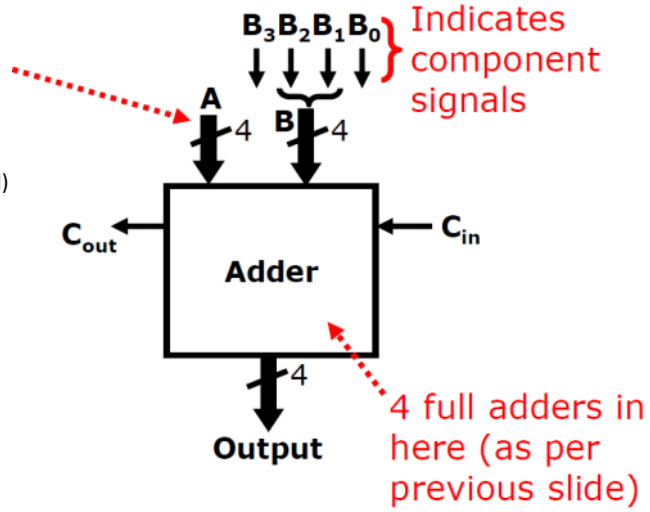
Monday, 23 August 2021 11:23 PM

- Can cascade full adders to make binary adder
  - Example: for 4 bits…



Simplifying the Representation

- To Simplify representation we group bits together
- **Bus**
  - Collection of wires
- **Bus width**
  - Number of bits transferred together (in parallel)
  - 4 in this example

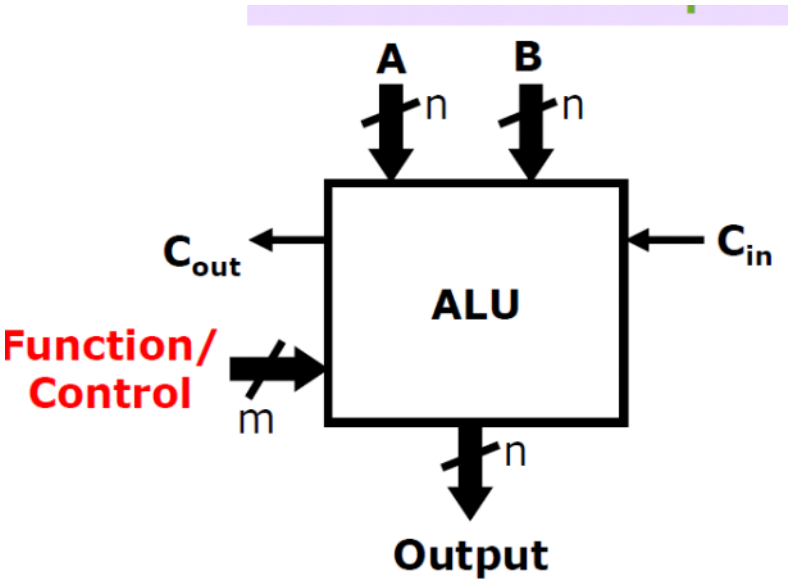


Arithmetic Logic Unit (ALU)

Monday, 23 August 2021 11:54 PM

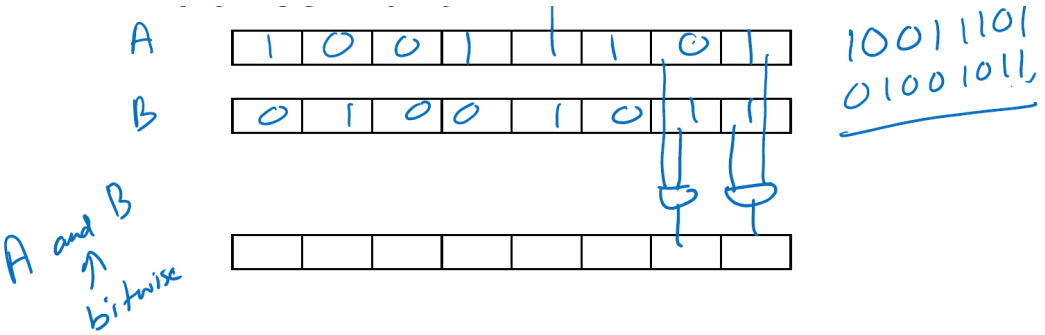
- Does more than adding
- **Function/control** input indicates the operation that the ALU is to perform e.g.
  - Addition
  - Subtraction
  - Increment
  - Bitwise AND
  - Bitwise OR
  - …
- Like adders, ALUs can be made up from 1-bit slices

ALU: Versatile hardware which can perform various operations based on some control inputs (control signals)



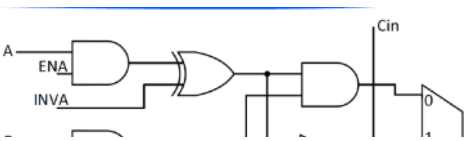
Bitwise Operations

- Operate on corresponding bits, e.g.
  - Bitwise A and B
  - Remember that A and B are multi-bit. The outputs are also multibit
  - Bitwise means you do the operation on each bit

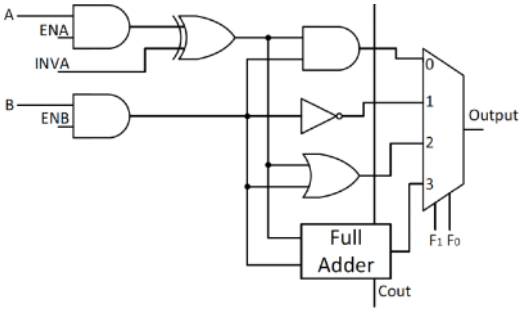


1-bit ALU (bit slice)

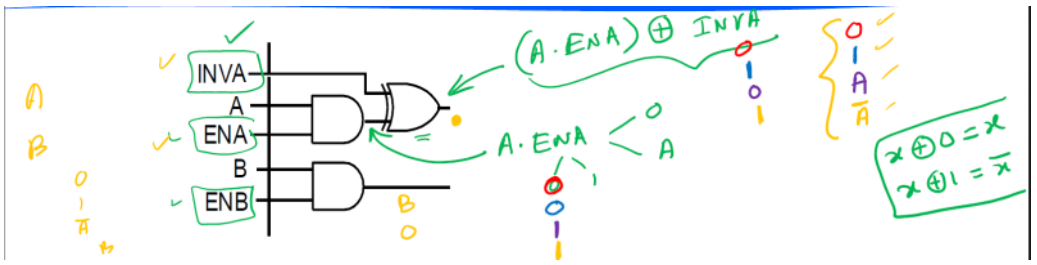
- F_1, F_0 select ALU function
  - 00 -> A and B
  - 01 -> invert B



- F_1, F_0 select ALU function
  - 00 -> A and B
  - 01 -> invert B
  - 10 -> A or B
  - 11 -> A plus B



1-bit ALU - Components



Data inputs - A, B

Control inputs

- INVA - invert A - inverts the corresponding input it set to 1
- ENA, ENB - enable A, enable B - enable the data input if set to 1
- Also some other function select control inputs (not shown here)

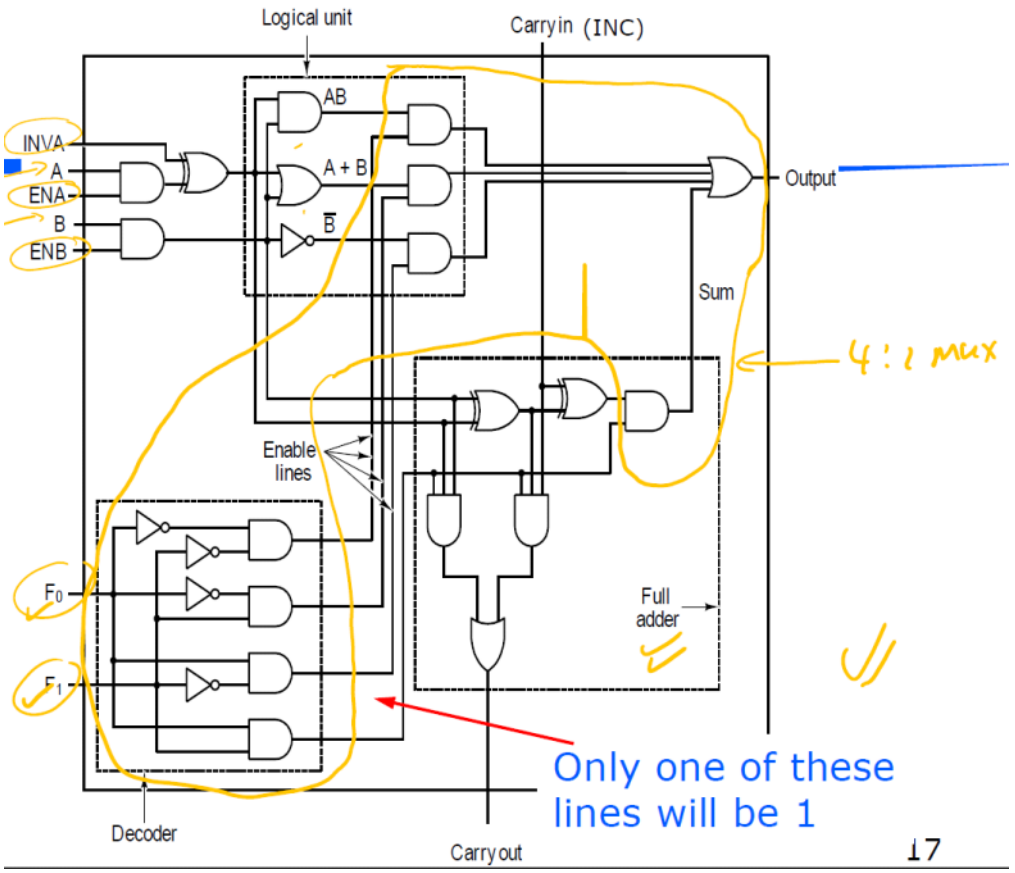
Two levels of control

- Permutations of data inputs
- Function (operation)

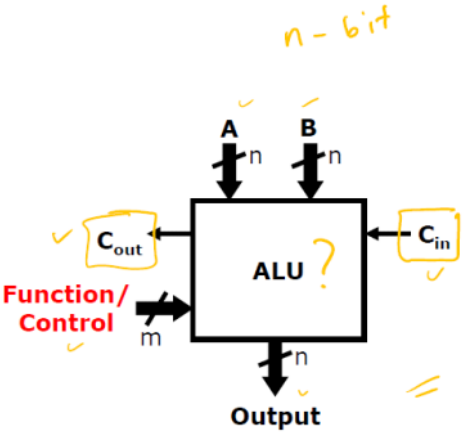
Building the ALU

4 functions

- A and B
- $\bar{B}$
- A or B
- A plus B

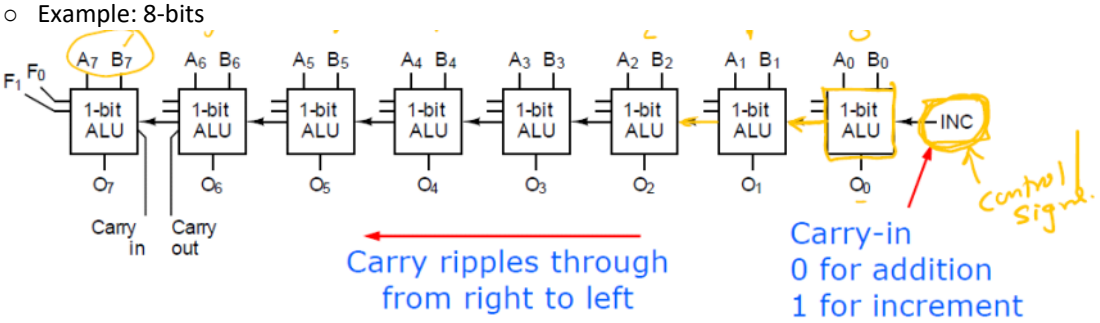


- Remember, machine word = n bits
  - Often 8, 16, 32
- n-bit ALU can be built up of n identical circuits for the individual bit positions



ALU Bit Slice Architecture (8-bit)

- Can be cascaded to make an n-bit ALU
  - Example: 8-bits



- F_1, F_0, ENA, ENB, INVA inputs are shared between all bit slices

How can we calculate $B - A$ using this ALU

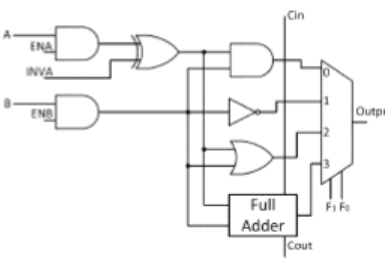
- i.e. what are the values of F_1, F_0, ENA, ENB, INVA?
- Note: B-A = B+(-A) using twos complement of A

$= B + (-A)$
$= B + (\bar{A} + 1)$
$= B + \bar{A} + 1$

INVA = 1
ENA = 1
ENB = 1
$F_1 = 1$
$F_0 = 1$
INC = 1 (carry in for adding one as required in two's comp)

Some Useful Combinations of ALU Inputs

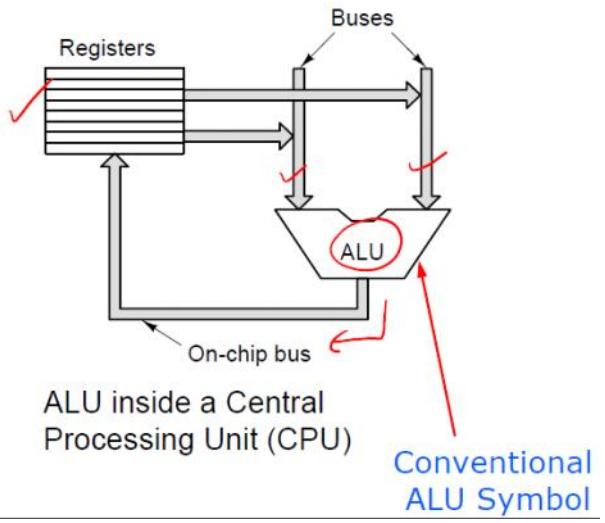
F_1	F_0	ENA	ENB	INVA	INC	Function
1	0	1	0	0	0	A
1	0	0	1	0	0	B
1	0	1	0	1	0	$\bar{A}$
0	1	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
						B + 1
1	1	1	1	1	1	B − A
1	1	0	1	1	0	B − 1
1	1	1	0	1	1	− A
0	0	1	1	0	0	A AND B
1	0	1	1	0	0	A OR B
1	0	0	0	0	0	0
1	1	0	0	0	1	1
1	0	0	0	1	0	− 1



22

How the ALU is used … Data Path

- Operands come from register file
- Result written to register file
- Implements routine instructions such as arithmetic, logical shift
- Width of registers/buses is the CPU word size



Data Path (Explanation)

- ALU performs operations on data from the register file
  - CPU control unit determines which registers the operands come from and which operation is performed.
- Result returned to a register within the register file
  - Could overwrite one of the operands
- Register ⇔ memory transfers also possible
  - Read: load a register with data from a memory address
  - Write: store a register's contents to memory address

ALU: In Reality

- Microprocessors may have a number of processing units: e.g.
  - One or more integer processing units
  - Floating point unit(s)
- Units may be capable of operating in parallel

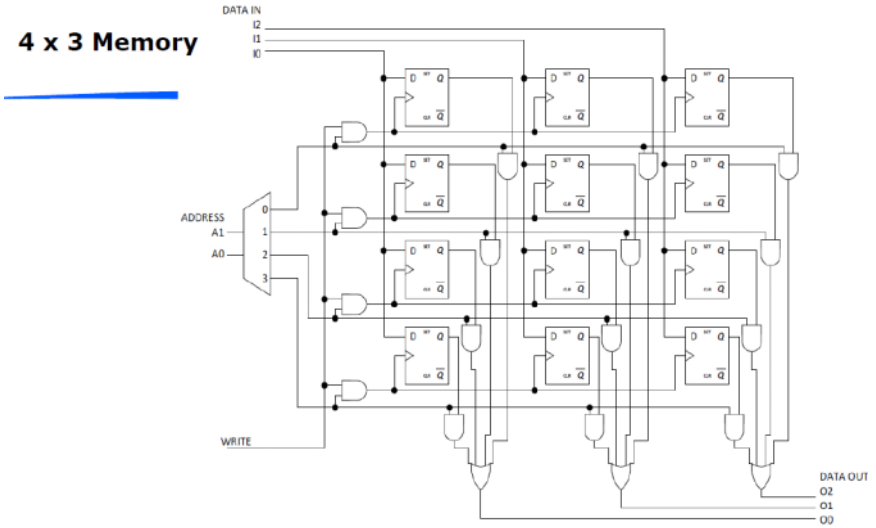
Computer Memory

Monday, 23 August 2021 10:53 PM

Memory in a Computer

- Computers need memory to
  - Store temporary results
  - Store programs
  - Remember settings when power off
- Many different types
  - Volatile vs. non-volatile
  - Random access
  - Read only
  - Static vs dynamic
  - Primary vs secondary
- We'll return to different types of memory in more detail in week 12

Memory Organisation

- Computer memory is conceptually a set of addressable cells
  - Cells usually 8-bits wide (i.e. hold one byte)
  - Various cell widths have been used in the past
- Example: 4 x 3 memory
  - 4 addresses, 3 bit cells
  - 

Memory Cycle

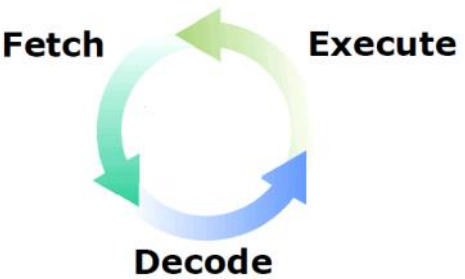
Fetch-Decode-Execute Cycle

- Control Unit operates in continuous cycle on the Datapath

Fetch: fetch an instruction from memory and bring it inside the CPU. Two special purpose registers involved - PC and IR

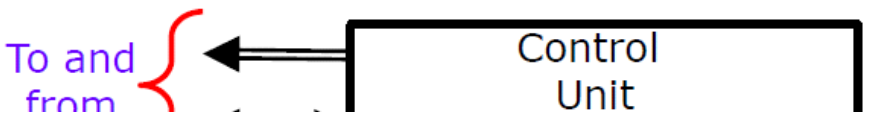
Decode: resolve the operation (OpCode) and operands (variables)

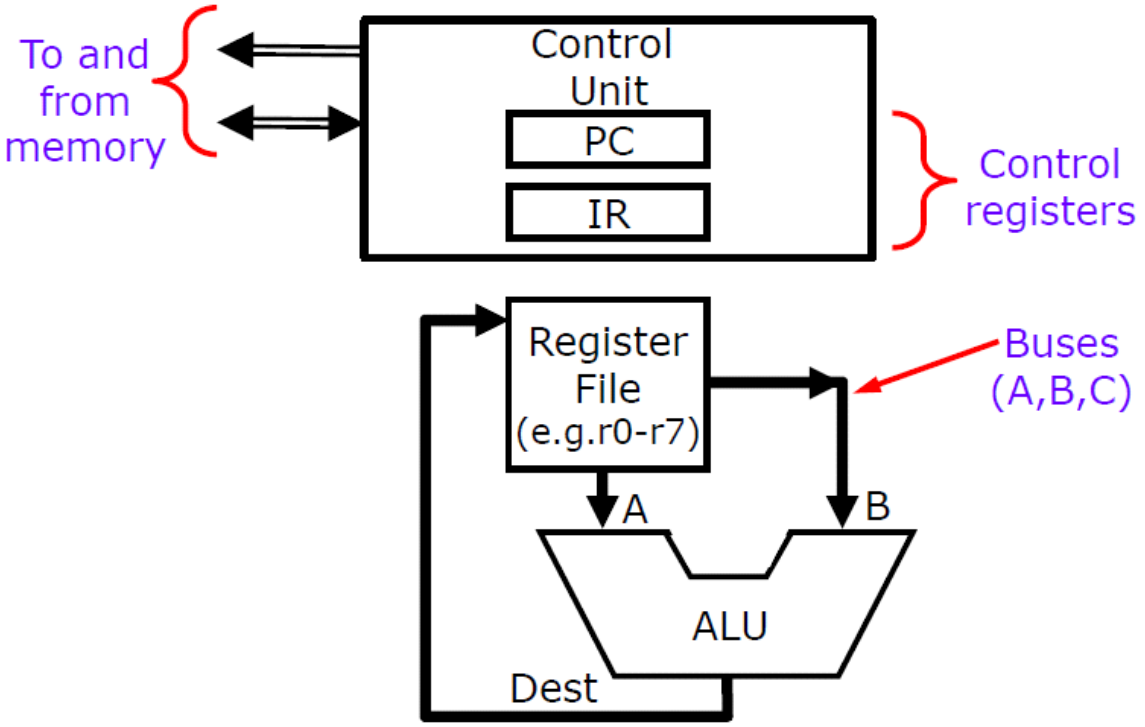
Execute: Control signals are issued and operation takes place in data path.



1. Fetch next instruction from memory into Instruction Register (IR)
2. Change Program Counter (PC) to point to the next instruction
3. Determine type of (decode) instruction just fetched (i.e. look at OpCode)
4. Execute the instruction
5. Return to step 1

Explanatory diagram





How do we control this?

Specifically

- How do we tell the ALU what to do
- How do we choose what register values are placed on the A and B buses
- How do we choose which register receives the result from the Destination bus?
- How do we know when to read or write memory?

Annotated Diagram (in class)

Control Unit

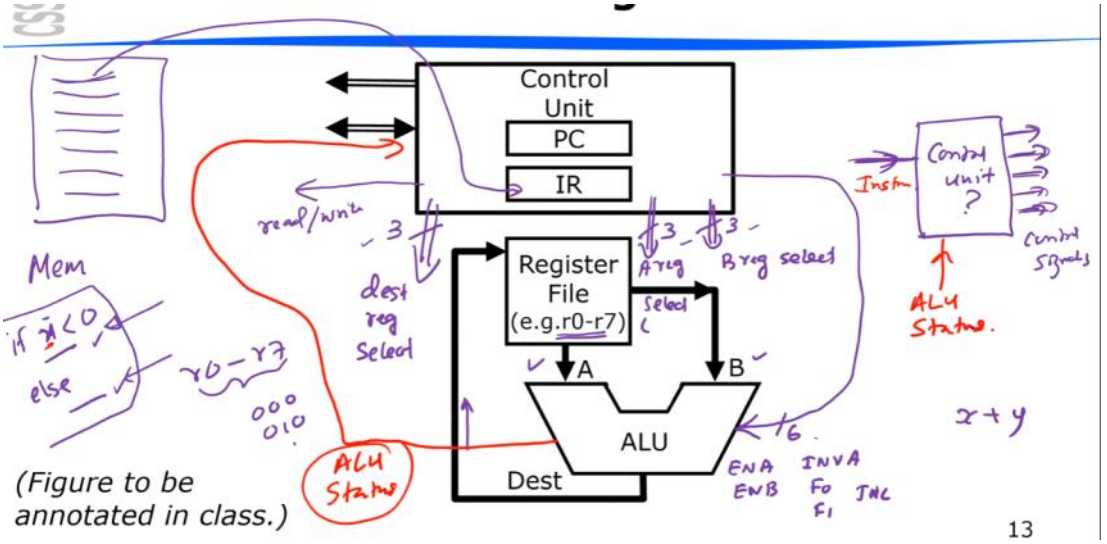
- Control signals come from the **control unit**
- Control unit must generate the control signals in the right order for a given instruction
  - Instruction determined by contents of the Instruction Register (IR)

Data Influencing Control

- Need to know more than what's in the instruction register
- Control unit can't operate without knowing something about the data
  - e.g. if a for-loop variable is supposed to go from 10 to 0, how does the control unit know when it gets there?

⇒ Status Register
  - Determined by the ALU operations
  - Stores status of last ALU operation

ALU Status



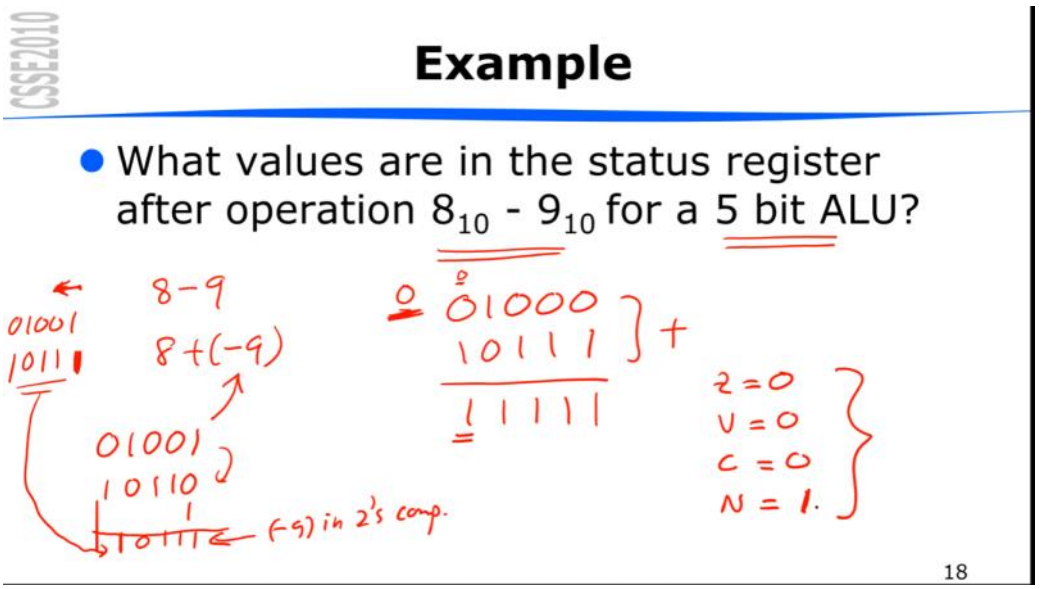
Status Register

The Status Register typically includes

- Z - zero bit - was the last result 0?
- V - overflow bit - did the last addition/subtraction operation overflow (assuming it was a two's complement operation)?
- C - carry bit - was a carry out generated?
- N - negative bit - was the result negative if considered as two's complement (i.e. what's the sign bit)

These are also called the ALU status flags. They are typically stored in a special purpose register called the STATUS register which is available to the control unit

Examples



What values are in the status register after operation 110 + 010 (binary addition) for a 3-bit ALU?

1. Z=0 V=0 **C=1** **N=1**
2. Z=0 **V=1** **C=1** N=0
3. Z=0 **V=1** C=0 **N=1**
4. **Z=1** V=0 **C=1** N=0
5. **Z=1** **V=1** C=0 N=0
6. **Z=1** V=0 C=0 N=0
7. **Z=1** **V=1** **C=1** N=0

```
 1 1
  110
  010
  ---
  000
```

Z=1
V=0
C=1
N=0

10

19

Control Unit

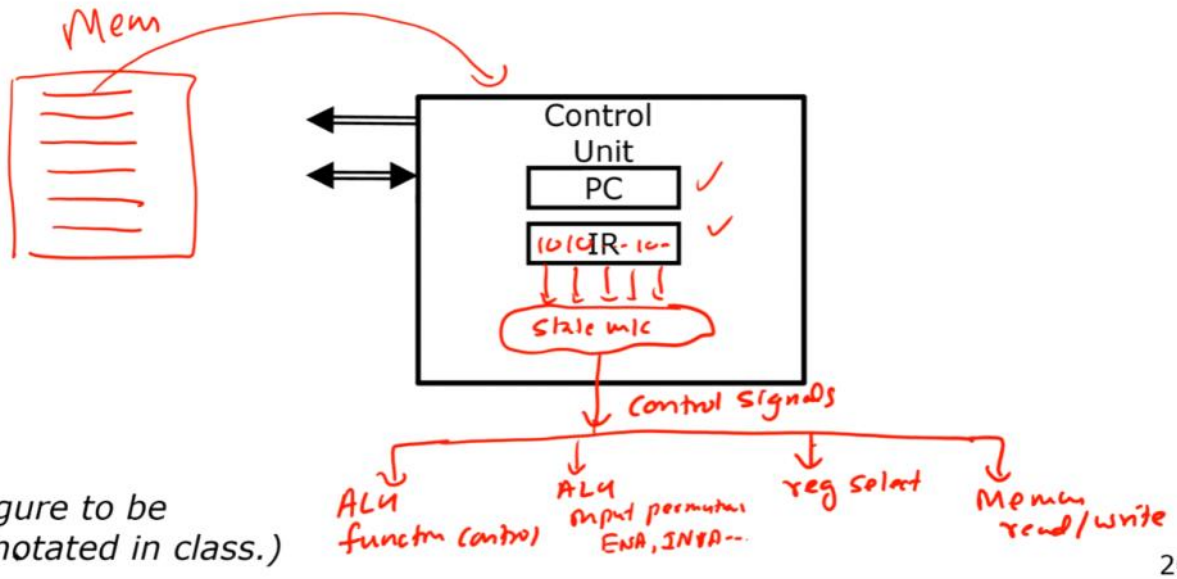


Figure to be annotated in class.)

20

C Programming Introduction

Tuesday, 31 August 2021 2:23 PM

- In this course we expect you to
  - Be able to write small C programs and functions from scratch
  - Understand the meaning of C programs
  - Be able to modify C programs
- Lectures won't be teaching the programming. So you need to get hands on and practice
  - Learning lab 9 continues the introduction to C programming
  - C tutorials and other resources are available on Blackboard.

Basic Structure of C

- There must be a function called main
  - This function is executed when the program starts
- Blocks of code are enclosed with braces {}
- C statements must end with a semicolon ;
- C statements are case sensitive
  - Variable is different from variable.
- Comments are
  - Within /* … */
  - From // to end of line

This is all the same as Java

Coding in C

Tuesday, 31 August 2021 2:27 PM

Declaring Variables

- Declaration
  - `type-name variable name, variable-name`

- Examples
  - `char c;`
  - `unsigned char x;`
  - `int day;`
  - `unsigned int count;`

- `char, int, float, double` are among data types supported by C
- Any value can be treated as a Boolean
  - zero means false, non-zero means true.

Functions

return-type function-name (arg-decl) {
    Variable-declarations;

    Statements;
}

Example Function

```
/*
    Return the average of two integers
    (result will be rounded towards 0)
 */
int average(int a, int b) {
    int avge;

    avge = (a+b)/2;

    return avge;
}
```

This is an assignment operator

These are expressions. Outer expression is a statement

Return avge

C Constants

- Character constants
  - Use single quotes, e.g. 'a', 'b', '1', etc
  - Some special characters - backslash escaped
    - '\n' = newline, '\'' = single quote, '\t' = tab, '\\' = backslash
- String constants
  - Use double quotes (can include backslash escapes)
  - e.g. "abc \n \" hello\t"

- Integer constants
  - Decimal - e.g. 3, -27, 65535, +5
  - Hexadecimal (leading 0x), e.g. 0x5F, 0xFFFF, 0xDEADBEEF
- Floating point characters
  - Include decimal point (.) and/or "e" for exponent
  - Examples: 3.14159, -7, 6.02e23, -5.2e-2
  - Note 7 is an integer, 7. is a floating point

C Operators

- Binary operators

+	addition
-	subtraction
*	multiplication
/	division
%	remainder (integer)
>	greater than
>=	greater than or equal
==	equal
!=	not equal
<	less than
<=	less than or equals
&	bitwise AND
|	bitwise OR
^	bitwise XOR
&&	logical AND
||	logical OR
=	assignment

- Unary operators

!	logical not
~	one's complement (invert)
-	two's complement (negate)
++	increment (prefix or postfix)
- -	decrement (prefix or postfix)

More Operators: Bit-shifter an assignment

- `a << b` means a shifted left by b bits
- `a >> b` means a shifted right by b bits
- `a = b` means a is assigned the value of b
- `a += b` is shorthand for `a=a+b`
- Similarly -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- Examples

  `1 << 5` is $1 * 2^5 = 32$

  `3 << 4` is $3 * 2^4 = 48$

  `a += 1` same as a++

Postfix/prefix
Increment and Decrement

- Example

```
int a,b,c,d,e;
a = 4;
b = a++;      /* b=a; a=a+1; */
```

Postfix – change happens after the value used

```
a = 4;
b = a++;      /* b=a; a=a+1; */
c = --a;      /* a=a-1; c=a; */
d = ++a;      /* a=a+1; d=a; */
e = a--;      /* e=a; a=a-1; */
```

Prefix – change happens before the value used

- After these statements, values are
  - a=4, b=5, c=4, d=5, e=5

If Statements

- `If (expression) stmt else stmt`
- Else clause is optional
- Note on expressions
  - C interprets any 0 value as false, anything else as true
  - `if(a)` means `if (a != 0)`
  - `if (!a)` means `if (a == 0)`
- `stmt` can be replaced with multiple statements enclosed in braces `{ }`

Machine Instructions

Wednesday, 1 September 2021 3:36 PM

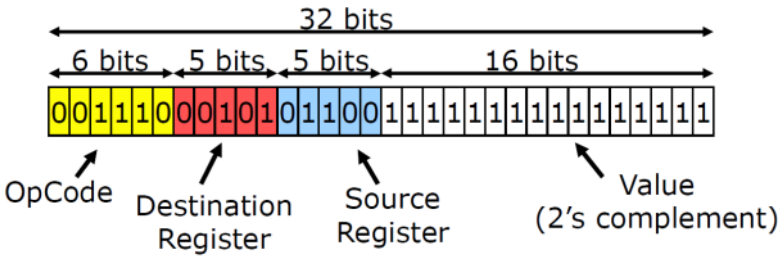
Machine Code

- Just 1's and 0's
  - The 1's and 0's specify the instruction to be executed
- PowerPC Example
  - `00111000101011001111111111111111`
- What does this example mean?

Instructions

- Instructions typically consist of
  - Opcode (Operation Code)
    - Defines the operations (e.g. addition)
  - Operands
    - What's being operated on (e.g. particular registers or memory address)

PowerPC Instruction Example



- How do we know this
- A machines Instruction Set Architecture (ISA) defines how binary instructions are interpreted.
- OpCode (001110_2 or 14) tells us that this instructions is an integer addition and has the format shown:
  - Destination-register = source-register + value
  - `R5 = r12 + (-1)`

Instruction Set Architecture

- Is a specification
  - i.e. a document
- Specifies the interface between hardware and software
- We return to ISA in lectures 12 and 13

Microarchitecture

Wednesday, 1 September 2021 3:47 PM

- Microarchitecture = The architecture of the control unit
  - i.e. the state machine (gates and flip-flops etc) that implements the control unit

ISA vs Microarchitecture

- An instruction set architecture (ISA) can be implemented by many different microarchitectures
- Examples
  - 8086 ISA is implemented by many processors - in different ways
  - Pentium ISA is implemented by
    - Pentium, Pentium IV, Core2Duo, i7
      - In different ways
    - Various AMD devices
    - Other manufactures too.

Assembly Language

Wednesday, 1 September 2021 3:49 PM

- Machine code is too hard to program in
- Use assembly language instead
  - Human readable (just!)
  - Converted into machine code by software called an assembler
- Atmel AVR examples
  - `Add r3, r4` (means `r3 <- r3 + r4`)
  - `Inc r22` (means `r22 <- r22 + 1`)

Introduction to Atmel AVR

Wednesday, 1 September 2021 3:50 PM

- Atmel ACR devices are microcontrollers
- Microcontrollers include lots of components on one chip, e.g.,
  - CPU
  - Memory (RAM)
  - Program Memory
  - Input/Output devices
- Some datasheet extracts to follow and discuss

ATmega324A Architecture

Grab annotated screenshots from notes

AVR Instructions

AVR assembly language instructions - examples

- MOV rd, rr
  - "move" (but actually means copy) contents of register rr (source) to register rd (destination)
  - Example: mov r3, r14

- `Add rd, rr`
  - Add contents of register rr to register rd
  - Example add r5, r6
- `AND rd, rr`
- `OR rd, rr`
- `EOR rd, rr`
  - Bitwise operations on give to registers, result goes into rd
- `LDI rd, number`
  - "Load immediate" - put the specified number into the given register
  - Registers r16 to r31 only
  - 8-bit numbers (`00000000` to `11111111`)
    - Range in decimal `-128` to `255`)
  - Example `ldi r16, 73`
- We'll see many more instructions coming in later weeks.

AVR Status Register

- `8`-bits

AVR Machine Code

- `Add` instruction
  - `0000 11rd dddd rrrr`
- Example: `add r23, r7`

- `COM` instruction
  - Flip-all bits (one's complement)
  - `1001 010d dddd 0000`
- Example: `com r19`

- LDI instruction
  - Load immediate
  - Only for registers r16 to r31
  - Machine code: `1110 KKKK dddd KKKK`
    - `K = 8`-bit constant
    - `D =` register number - 16

Instruction Set Architecture (ISA)

Monday, 6 September 2021 8:12 PM

- **ISA** defines the interface between the hardware and the software
  - ISA is a specification
  - **Microarchitecture** is how the control unit is built
- For hardware (microarchitecture) designers
  - Don't need to know about the high level software
  - They just need to build a microarchitecture that implements the ISA
- For software writers (machine language programmers and compiler writers)
  - Don't need to know (much) about the microarchitecture
  - They just need to write or generate instructions that match the ISA specification

What makes an ISA?

1. Memory models
2. Registers
3. Instructions
4. Data types

1. Memory Models

- Memory model: how does memory look to the CPU
- Issues
  a. Addressable cell size
  b. Alignment
  c. Address space
  d. Endianness
     - Big endian
     - Little endian

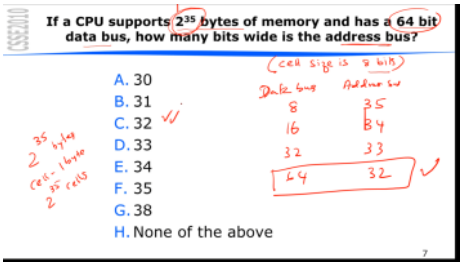
a. Addressable cell size

- Memory has cells, each of which has a unique address (or cell number)
- Most common cell size is 8 bits (1 byte)
  - But not always!
    - AVR instruction memory has 16 bit cells
- A bus is used to transport the content of a memory cell, but sometimes the data bus may be of a different width
  - i.e. the bus may be wider in order to be able to retrieve several cells at once.
    - The address bus does not need to be as wide

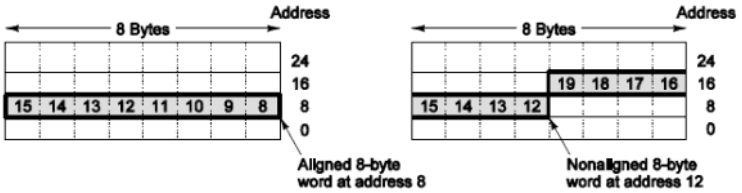
Bus Sizes

- For every doubling of the data bus width
  - Remove least significant bit of address *bus*
- Example 2^n cells $\times$ 8-bit cell size
  - Address size: n-bits
- Bus configurations
  - Data bus: 8-bits, address bus n bits
    - One cell transferred at a time
  - Data bus: 16-bits, address bus $n-1$ bits
    - Two cells transferred at a time
  - Data bus: 32-bits, address bus $n-2$ bits
    - Four cells transferred at a time.

Address bus example question



b. Alignment

- Many architectures require natural alignment, e.g.
  - 4-byte words starting at addresses 0, 4, 8, …
  - 8-byte words starting at addresses 0, 8, 16, …
- For example, with an 8-byte word, it can be stored as follows
  - 

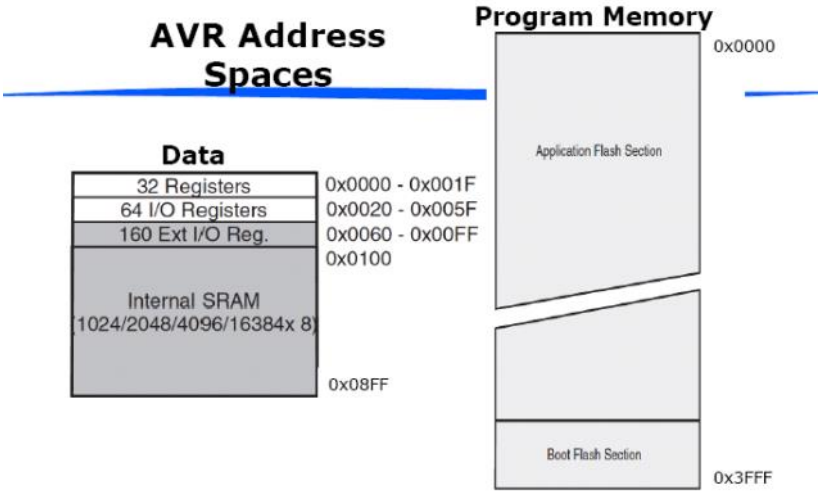
- Alignment is often required because it is more efficient
- Example - Pentium IV
  - Fetches 8 bytes of memory at a time (64-bit wide data bus)
  - Alignment is NOT required (so the processor is backwards compatible with earlier processors and instructions
    - e.g. 4-byte word stored at address 4 takes 1 read (bytes 0 to 7)
    - e.g. 4 byte word stored at address 6:
      - Read bytes 0 to 7
      - Read bytes 8 to 15
      - Extract required bytes from the 16 bytes read.

c. Address Spaces

- Many microprocessors have a single linear memory address space (**von Neumann** architecture)
  - e.g. 2^{32} bytes numbered from 0 to $2^{32}-1$
    - May not be bytes. This depends on addressable cell size.
- However, **Harvard architecture** is:
  - Separate address spaces for instructions and data

- AVR ATmega 324A
  - Data address space: 2^{11} (2048) bytes
  - Instruction address space: 2^{14} 16-bit words
    - 16,384 instruction words = 32,768 bytes

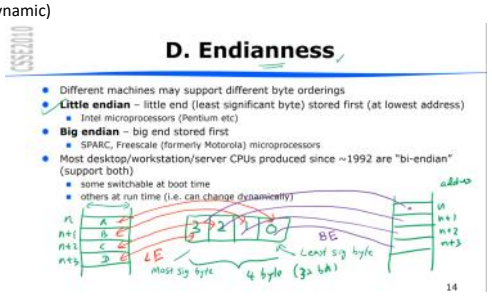
AVR ATmega 324A Address Spaces



[From ATmega324A datasheet]

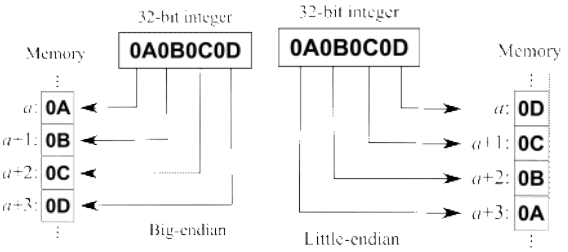
d. Endianness

- Different machines may support different byte orderings
- **Little endian** - little end (least significant byte) stored first (at lowest address)
- **Big endian** - big end stored first
  - SPARC, Freescale (formerly Motorola) microprocessors
- Most desktop/workstation/server CPUs produced since ~1992 are "bi-endian" - meaning they support both
  - Some are switchable at boot time
  - Others can be switched at run time (dynamic)



Here's a little diagram to remember.

- In big endianness, the most significant bit *0A* is stored at position a in memory with the least significant *bit 0D* being stored at position $a + 3$
- In little endianness, the least significant bit *0D* is stored at position a in memory with the most significant *bit 0A* being stored at position $a + 3$



2. Registers

- Two types of registers
  - General Purpose
    - Used for temporary results etc.
  - Special purpose, e.g.
    - Program Counter (PC)
    - Stack pointer (SP)
    - Input/Output Registers
    - Status Register
      - (Tanenbaum calls this the Program Status Word)
- Some other registers are part of the microarchitecture and are NOT a part of the ISA
  - e.g. Instruction Register (IR)
  - The programmer doesn't really need to know about these because they can't directly change or use them.

AVR Registers

- General purpose registers (0-31) are quite regular and can be used interchangeably
  - Exception: a few instructions work on half the registers (16-31)
    - Only 4 bits used to represent operand in instruction
  - X, Y, Z registers
- There are many I/O Registers
  - Not to be confused with General Purpose Registers
  - Some instructions work with these, others with general purpose registers
  - Don't confuse them

X, Y, Z Registers in the AVR

R0	0x00	
R1	0x01	
R2	0x02	
...		
R13	0x0D	
R14	0x0E	
R15	0x0F	
R16	0x10	
R17	0x11	
...		
R26	0x1A	X-register Low Byte
R27	0x1B	X-register High Byte
R28	0x1C	Y-register Low Byte
R29	0x1D	Y-register High Byte
R30	0x1E	Z-register Low Byte
R31	0x1F	Z-register High Byte

AVR ATmega 324A has 224 I/O register addresses to control peripherals and get data to/from them, e.g.

- Timers and counters
- Analog to Digital converters
- Serial input/output
- General purpose input/output
- Three registers associated with each (learning lab 11)
  - DDRx - Data direction register
  - PORTx - values to output
  - PINx - values on the pins
- Some addresses reserved (not actually available on this device)

3. Instructions

- This is the main feature of an ISA
- Instruction types include
  - **Input/Output** - communicate with I/O devices
  - **Load/Store** - move data between registers
  - **Arithmetic** - addition, subtraction, …
  - **Logical** - Boolean operators
  - **Branching** - for deciding which instruction to perform next.

Recall - AVR Instructions

- **MOV rd, rr**
  - "Move" (but actually means copy) contents of register rr (source) to register rd (destination)
  - Example: mov r3, r14
- **ADD rd, rr**
  - Add contents of register rr to register rd
  - Example: add r5, r6
- **AND rd, rr; OR rd, rr; EOR rd, rr**
  - Bitwise operations on given two registers, result goes into rd
- **LDI rd, K** (where K is a constant)
  - Load constant value into a register (16 to 31)

Input/Output Instructions

- AVR Atmega324A has 224 I/O registers to control peripherals
- I/O Registers 0 to 223
  - Available at memory address 32 to 255 ($20 to $FF)
    - Use load and store instructions to access
- First 64 I/O registers (0 to 63) are special
  - Can also be accessed using "in" and "out" instructions
- Other 160 I/O registers (64 to 223) are called "extended" I/O registers
  - Can only be accessed using load and store instructions

IN and OUT Instructions

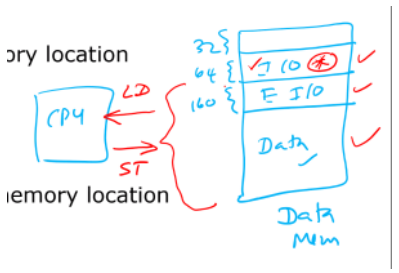
- **In rd, P**
  - Load I/O register value into general purpose register
  - rd: r0 to r31
  - P = I/O register number (0 to 63, 0 to 0x3F)
  - Example
- **out P, rr**
  - Store value from general purpose register into I/O register
  - rr: r0 to r31, P = 0 to 63 (0 to 0x3F)
  - Example

Load and Store Instructions

Wednesday, 8 September 2021 9:21 AM

Load/Store Instructions

- Load
  - Load register from memory location
  - AVR Examples: 1d, 1ds
  - 1d is indirect addressing
  - 1ds is direct addressing
- Store
  - Store register value in memory location
  - AVR Examples: st, sts
  - St is indirect addressing
  - Sts is direct addressing



Direct and Indirect Addressing

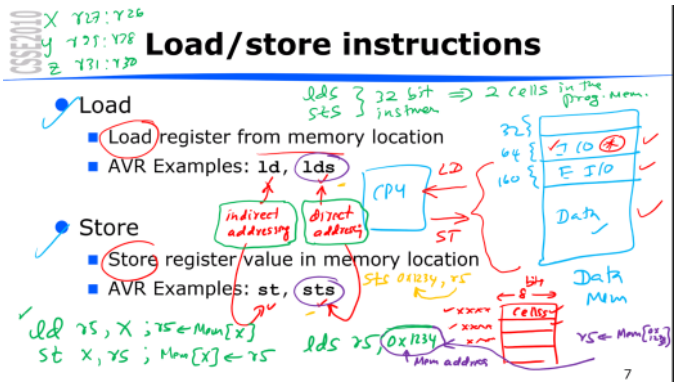
Direct addressing refers to when you directly give the address along with the instruction. For example

- Load: lds r5, 0x1234
- Store: sts 0x1234, r5
- Direct addressing is a 32-bit instruction in AVR. Therefore, 2 cells are required in the program memory.

Indirect addressing refers to when the memory address is specified in the register rather than in the instruction. This requires a 16 bit register: X, Y, Z in the AVR (r26 - r31)

Note

Load and store instructions can be used in I/O, Ext I/O and Data memory regions. However, mostly used in Ext I/O and Data memory regions because I/O has it's own IN and OUT instructions.



Data Movement Instructions

- Data is usually **copied**, not moved
- Types of data movement
  - Register -> register (move)
    - mov
  - Memory -> register (load)
    - sd, lds
  - Register -> memory (store)
    - st sts
  - I/O register -> register (input)
    - IN
  - Register -> I/O register (output)
    - OUT

Dyadic Instructions

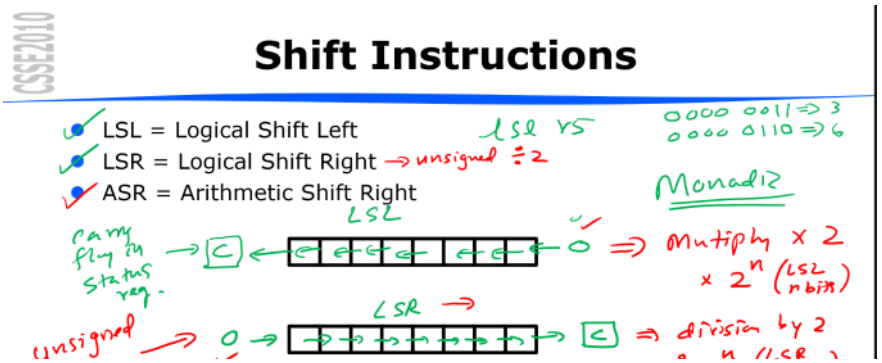
- Combine 2 operands to produce a result
- Usually two types
  - Arithmetic
  - Logical
    - Bitwise

Monadic Instructions

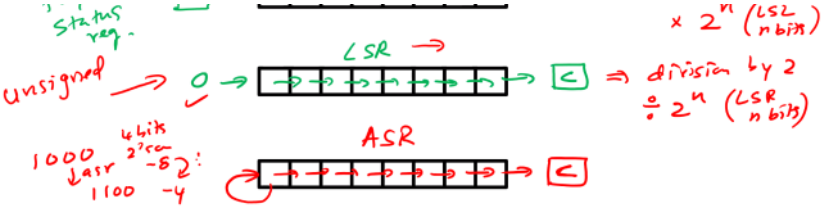
- One operand -> one result
- Examples
  - Setting a register
    - Set r16 -> ldi r16 0xFF
  - Clearing a register
    - Clr r5
  - Negation (2's complement)
    - Neg r5
  - Inversion (1's complement negation)
    - Com r5
  - Shifting & Rotation.

Shift Instructions

- LSL = Logical shift Left
  - Multiply by 2
  - Shift n times: multiply by 2^n
- LSR = Logical Shift Right -> unsigned
  - Divide by 2
  - Shift n times: divide by 2^n
  - Must be unsigned because it inserts a 0 at the MSB
- ASR = Arithmetic Shift Right
  - Divide by 2
  - Shift n times: divide by 2^n
  - Sign bit is replicated. So works for signed bytes



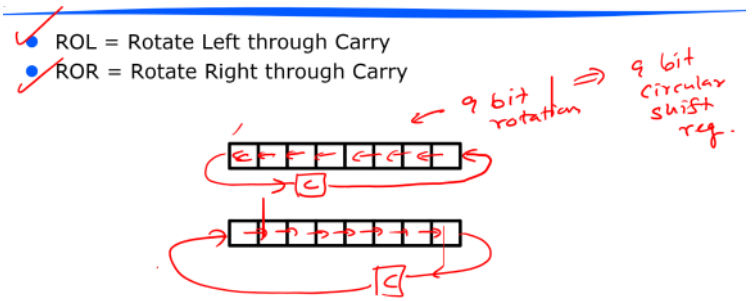
- Sign bit is replicated. So works for signed bytes



11

Rotate Instructions

- ROL = Rotate Left through carry
  - Rotates the MSB to the carry flag and then back to the LSB
  - Actually a 9-bit circular shift register
- ROR = Rotate Right through carry.
  - Rotates the LSB to the carry flag and then back to the MSB
  - Also actually a 9-bit circular shift register



Comparison Instructions

- Same as subtraction
  - Result not saved
  - Sets status register flags (Z, N, C, V, ...)
- Instruction reference specifies which status register bits are set by which instructions

Branch Instructions

- Based on status register values
- Example
  - Breq label
    - Branch if Z = 1
  - Brne label
    - Branch is Z = 0

Addressing Modes

Instruction operands can come from different places

- **Immediate** addressing
  - An operand value is in the instruction
    - AVR Example: andi r17, 0xBA
- **Register** addressing
  - Only register numbers in instruction
    - AVR Example: and r18, r19
- **Direct** addressing
  - Memory address is in instruction
    - AVR example: lds r15, $1234
  - Note, these AVR instructions occupy two instruction words (32 bits)
- **Indirect** addressing
  - Memory address in register
  - Register number is in instruction
    - AVR example: ld r5, X

- These addressing mode names are as per Tanenmaum "Structured Computer Organization"
- AVR instruction set manual uses different names

4. Data Types

- Numeric
  - Integers of different lengths (8, 16, 32, 64 bits)
    - Possibly signed or unsigned
  - Floating point numbers, e.g. 32 bits (single precision) or 64 bits (double precision)
  - Many others not listed here
- Non-numeric
  - Boolean (0 means false, 1 means true) - stored in a whole byte or word
  - Bit-map (collection of Booleans, e.g. 8 Booleans in a byte)
  - Characters
  - Pointers (memory addresses)

- Different machines provide *hardware support* for different data types.
  - e.g. Intel-64 Architecture

Data Type	8 bits	16 bits	32 bits	64 bits	128 bits
Signed integer	✓	✓	✓	✓	
Unsigned integer	✓	✓	✓	✓	
BCD integer	✓				
Floating point			✓	✓	

  - e.g. Atmel AVR

Data Type	8 bits	16 bits	32 bits	64 bits	128 bits
Signed integer	✓				
Unsigned integer	✓				
BCD integer					
Floating point					

- Other data types can be supported through software!
  - e.g. 16 bit integer operations can be built out of 8-bit operations, using a

sequence of instructions. Though this will come at the cost of performance because many instructions have to be executed rather than just one with a higher bit capable CPU

More Data Types in C

Sunday, 12 September 2021 3:06 PM

Bit Shifts in C

- Recall `lsl`, `lsr`, `asr` from assembly language

There are equivalents in C

- `a << b` means a shifted left by b bits
- `a >> b` means a shifted right by b bits

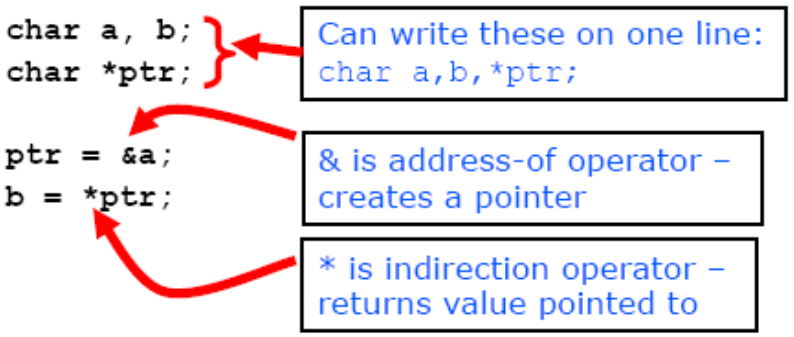
- Performs logical shift right if a is unsigned
- Performs arithmetic shift right if a is signed.

Arrays in C

- Declaring an array
  - Type `variable_name[size];`
  - Examples
    - `char message[16];`
    - `int values[10];`
- Accessing elements within an array
  - `variable_name[index]`
  - `index = 0 … size-1`
    - This is called zero-based indexing
  - Examples
    - `message[0] = 'c';`
    - `values[9] = values[8]++;`

Pointers

- C has the concept of pointers
- Pointer declaration
  - `type * variable_name;`
  - `variable_name` is a pointer to something of given type
    - How? - pointer variables store memory addresses
- Example



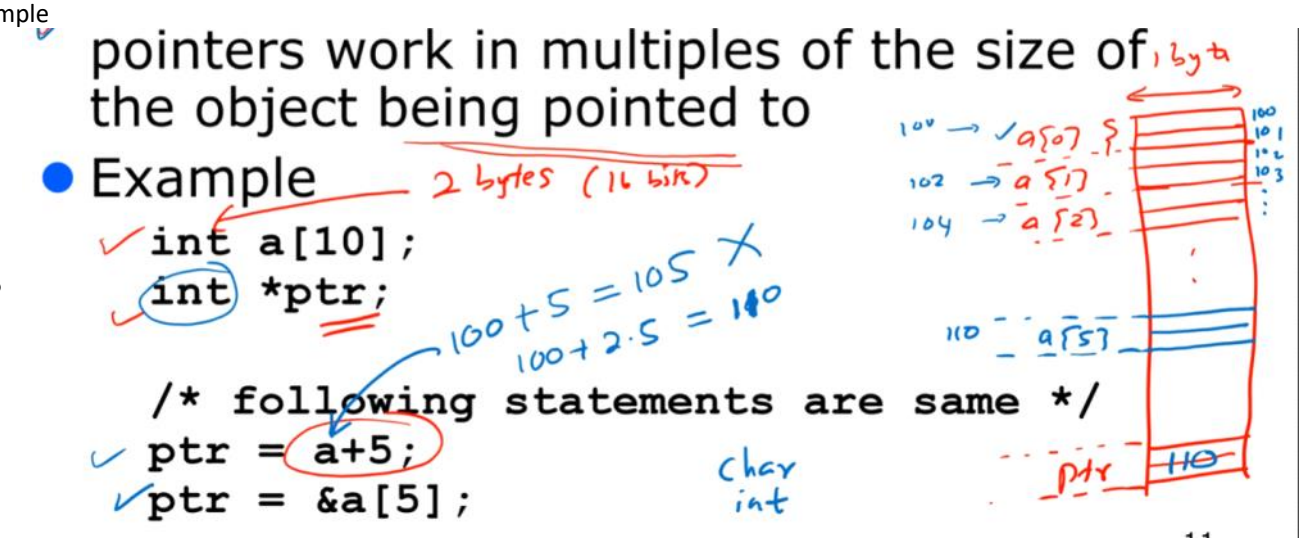
Pointers and Arrays

- Array name can be treated as a pointer to the first element
  - i.e. address of first element
- Example

```
int a[10];
int *ptr;

/* following statements are same */
ptr = a;
ptr = &a[0];
```

Operations on Pointers

- Addition and subtraction on pointers work in multiples of the size of the object being pointed to
- Example



Traversing an Array

- Two examples of clearing an array.
- Using **index**

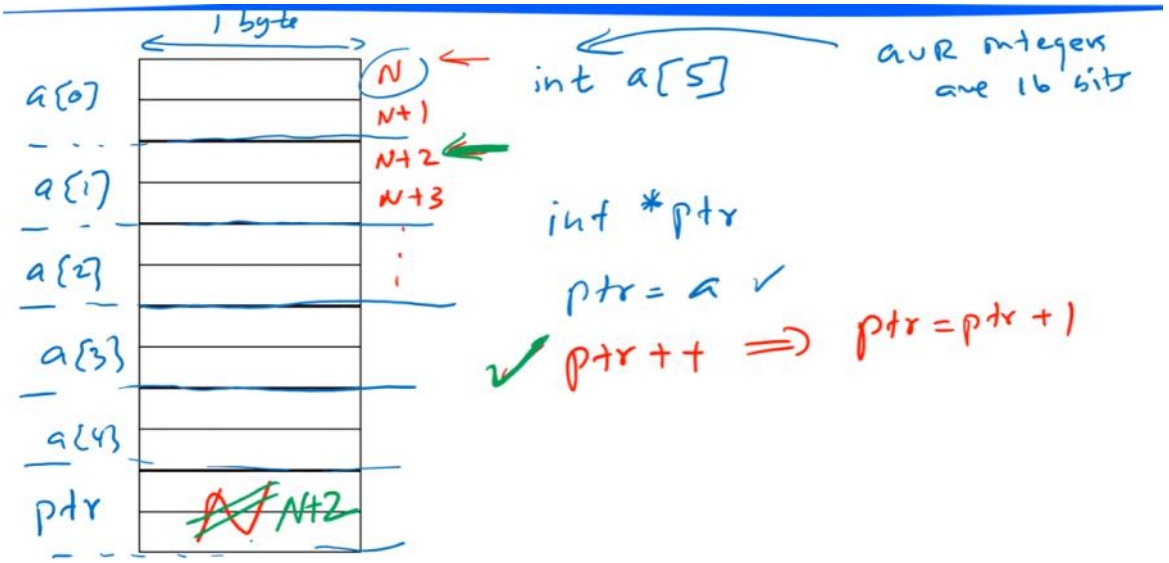
```
int a[10];
int index;
for(index=0; index<10; index++) {
        a[index] = 0;
}
```

- Using **pointer**

```
int a[10], *ptr;
for(ptr=a; ptr < a+10; ptr++) {
        *ptr = 0;
}
```

Adding one to an array pointer makes it point to next element in array

Arrays in Memory


1 byte
a[0]
a[1]
a[2]
a[3]
a[4]
ptr
N
N+1
N+2
N+3
int a[5]
our integers are 16 bits
int *ptr
ptr = a
ptr++ ⟹ ptr = ptr + 1

Miscellaneous Programming in C

Wednesday, 15 September 2021 8:44 AM

Pre-processor

- C programs are first parsed by a pre-processor
- Pre-processor replaces pre-processor directives with substitute text
- Example
  - `#include` …
    `#define` … (macros)
  - Many others
    - `#if`
      `#ifdef`
      `#else`
      `#undef`
      …

Header (.h) files

- Header files (`.h` files) are used for common macros, function prototypes etc
  - Like `m324Adef.inc` for assembly
- Examples
  - `stdio.h` - includes prototypes of `printf()` etc
    - AVR has a simplified version of this (e.g. no files)
      - `printf()` will only work if you specify where the characters go, e.g. sent via serial port
  - `avr/io.h` - AVR has this to define useful constants and functions/macros
    - Definitions depend on the actual device targeted

Pre-processor Examples

#include<filename.h> <> means system header file

#include"filename.h" "" means user defined files.

Macros – textual substitutions

```
#define pi 3.1415
#define EVER ; ;
#define sum(a,b) (a+b)
```

For (; ;) replaced by
For (EVER). This makes more sense.

Pre-processor will replace any of the definitions with the value on the right
So PI will be replaced with 3.1415.
This is not a variable. The compiler does not ever see pi.

AVR C Macro Examples

```
#define _MMIO_BYTE(mem_addr) \
                (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) \
                _MMIO_BYTE((io_addr) + 0x20)

/* Input Pins, Port A */
#define PINA      _SFR_IO8(0x00)

/* Data Direction Register, Port A */
#define DDRA      _SFR_IO8(0x01)

/* Data Register, Port A */
#define PORTA     _SFR_IO8(0x02)
```

Each port has 3 I/O registers associated with it:

- PORTx
- PINx
- DDRx

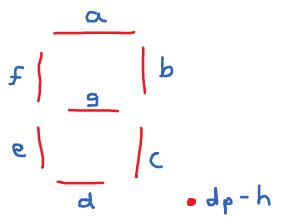
Where x can be any of A, B, C, D
e.g. PORTA, PINB, DDRC

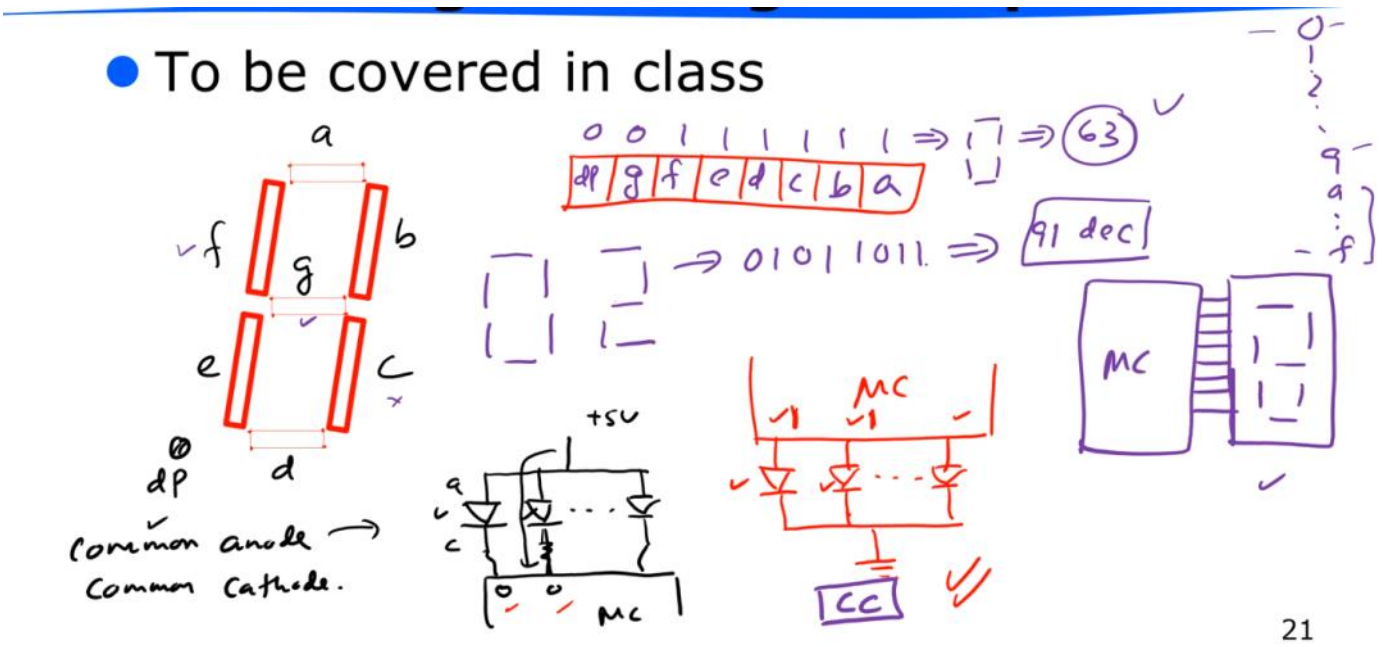
- Upshot of these macro definitions
  - I/O register names can be used as variables, e.g.
    - `DDRA = 0xFF;`
    - `PORTA = PINC | 0xF0;`

Seven Segment Display Programming

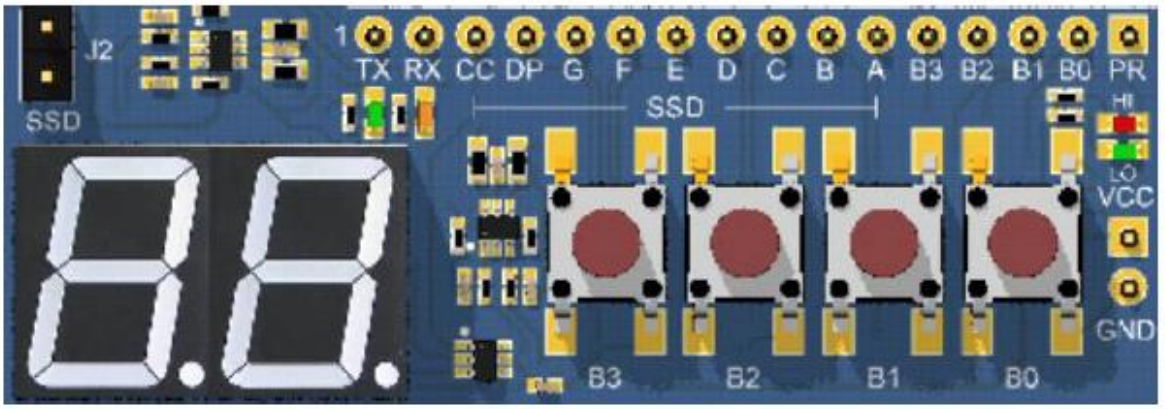
Wednesday, 15 September 2021 4:39 PM

Seven Segment Display Programming Example





Seven Segment Display on CSSE2010 Kit IO Board



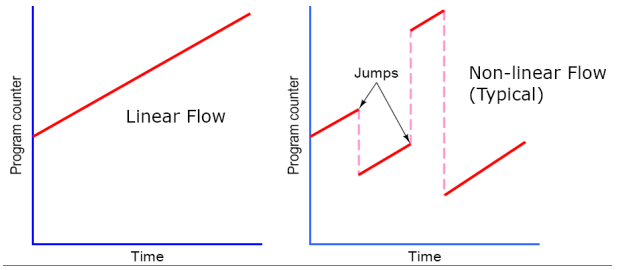
```c
#include <avr/io.h>

/* Seven segment display values */
uint8_t seven_seg[10] = { 63,6,91,79,102,109,125,7,127,111};

int main(void) {
   uint8_t digit;

   /* Set port A pins to be outputs, port C pins to be inputs */
   DDRA = 0xFF;
   DDRC = 0;     /* This is the default, could omit. */
   while(1) {
      /* Read in a digit from lower half of port C pins */
      /* We read the whole byte and mask out upper bits */
      digit = PINC & 0x0F;
      /* Write out seven segment display value to port A */
      if(digit < 10) {
         PORTA = seven_seg[digit];
      } else {
         PORTA = 0;
      }
   }
}
```

Flow of Control

Monday, 13 September 2021 10:53 PM

What is it?

- The sequence that instructions actually get executed in
- Linear (i.e. successive instructions) unless we have
  - Branches/jumps
  - Procedure calls
  - Etc



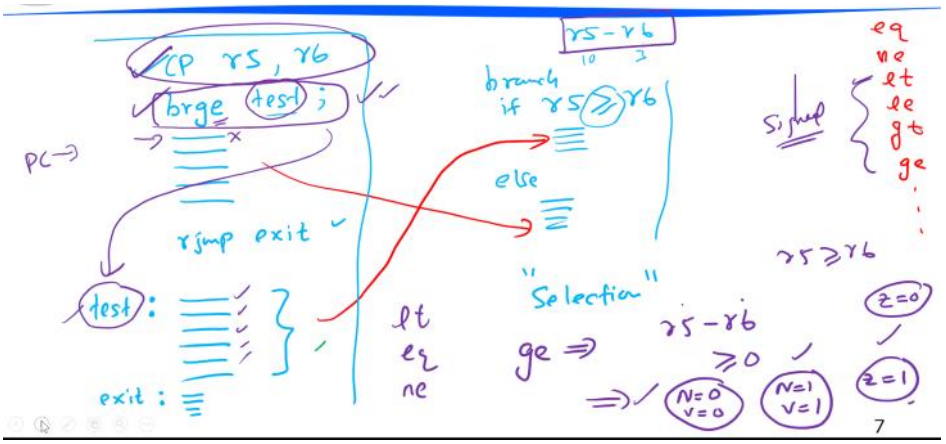
- Most instructions result in the PC being incremented (sequential execution)
  - i.e. PC <-- PC + 1
    - This is part of the fetch, decode, execute cycle
- Instructions aren't executed sequentially when we have
  - Branches - conditional if/else
  - Jumps - unconditional branch (loops)
  - Procedures - functional call
  - Interrupts / Traps

These operations change the value of the program counter. So instead of $PC = PC + 1$
$PC = PC + K$

Branches

- AVR branch instructions are typically conditional
- Useful in implementing if-else and loops in assembly language
- `breq` and `brne` mentioned previously
- Example: "Greater than or equal to"
  - Instruction used depends on whether you're comparing unsigned or signed (two's complement) numbers.

Branches - AVR Assembly Example.



AVR "S" Status Bit

Recall from earlier lectures

- N bit = negative (1 if sign bit of result is 1)
- V bit = two's complement overflow

AVR also has the "S" bit

- S = $N \oplus V$
  - Sign-bit corrected for overflow
    - If overflow happens, sign bit will be wrong so instead of checking N bit, check S bit.

Branch Instructions (Conditional)

Branch if status bit is equal to

	C	N	V	S=N⊕V	Z
0	brsh (=brcc)	brpl	brvc	brge	brne
1	brlo (=brcs)	brmi	brvs	brlt	breq

sh lo → unsigned

ge lt → signed

C = 0 | brsh ← same or higher } unsigned

C = 1 | brlo ← lower

pl → positive

mi → Negative

br ge, br lt } signed comparisons

cp r5, r6

r5 < r6 r5 > r6

r5: −1 1111 1111; 2 0000 0010 r6

9

Comparing Signed vs Comparing Unsigned numbers

There are different instructions when comparing signed numbers as opposed to comparing unsigned numbers.

For signed numbers (two's complement)

- `brge` - branch if greater or equal to
- `brlt` - branch if less than

For unsigned numbers

- `brsh` - branch if same or higher
- `brlo` - branch if lower

Some other important branch instructions

These are all outlined in the instruction set summary document for the AVR 324A

- `brpl` - branch if positive
- `brmi` - branch if negative
- `brvs` - branch if overflow set (i.e. there is overflow; $V = 1$)
- `brvc` - branch if overflow clear (i.e. there is no overflow; $V = 0$)
- `brne` - branch if not equal
- `breq` - branch if equal

Jumps (unconditional)

2 cells in program memory.

- AVR has three jump instructions
  - `jmp label` - absolute jump, 32-bit instruction, 3 cycles
  - `rjmp label` - relative jump, 16-bit instruction, 2 cycles
  - `ijump` - indirect jump, 16-bit instruction, 2 cycles, jump target is given in `Z` register.

1 cell in program memory

Indirect jump means you specify the address you want to jump to in a register.

Procedures

- Procedure
  - Group of instructions that perform some task
  - May have operands (**arguments** or **parameters**)
  - May produce result(s)
  - Can be **invoked** (called) from multiple places in program
    - Even within itself - **recursive** procedure
  - After invocation, control returns to the statement after the call
- Good use of procedures helps structure a program well
- Other names
  - **subroutine**, **function** (C, python), **method** (Java)

Procedure Example - AVR Assembly

Directly giving the memory address of the function or subroutine. 32-bit instruction. Very slow (4 cycles)

- `call label` - direct subroutine call, 4 cycles, 32-bit instruction
- `rcall label` - relative subroutine call, 3 cycles, 16-bit instruction
- `icall` - indirect call, 3 cycles, 16-bit instruction.

Adding some value to the program counter to get to the function or subroutine. Faster (3 cycles). Use wherever possible. If function is too far away, cannot use. (i.e. if 16-bits is not enough).

Specify address to go to in a register rather as a label

Procedure Issues

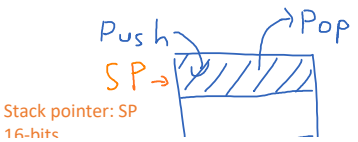
- After procedure finishes, control returns to the statement after the call.
  - How does the procedure know where to return to?
- How to we specify operands (i.e. pass in arguments)?
- How do we return the result?

Return Address

- Where can we store the return address
  - Single fixed memory address or register for all procedure calls
    - But, procedures can't call other procedures
  - Memory location per procedure
    - Doesn't allow for recursion
  - Use a stack...

Procedure Call - Use of Stack

Example



Stack pointer: SP
16-bits

Procedure Call - Use of Stack

Example

Stack - Last in, first out (LIFO)
Like a stack of books

Push to stack (storing in the stack)
Pop stack (reading from the stack)

Stack pointer: SP
16-bits
Stored in a special purpose register

Need 16-bit because 2KB is 2^{12}.
Therefore more than 8-bits required.

- Stack pointer
  - Register that keeps track of top of stack
  - AVR: SPH:SPL
    - 2 I/O registers for 16-bit stack pointer

Assembly Programs - Stack Initialisation

- Assembly language programs must initialise the stack before using it

```
.def temp=r16
    ldi temp,low(RAMEND)
    out SPL, temp
    ldi temp,high(RAMEND)
    out SPH, temp
```

.def is an assembler directive, not an instruction. It's like a macro in C.

AVR Timers/Counters

Tuesday, 21 September 2021 12:08 PM

- Can be used to perform actions with pre-defined timing: e.g. do some action every 100ms
- Can be used to **generate PWM** (today's topic)
- Can be used to count pulses (logic transitions on a given pin)
- Separate hardware to the CPU
- Each timer/counter unit is tied to particular pins on the microcontroller. By configuring the timer/counter these pins are changed from their default GPIO operation to timer/counter operation
- Configuration includes reading/writing to I/O registers:
  - Timer/Counter register
  - Control register
  - Output compare registers
  - Interrupt registers

AVR Timer/Counter Summary

ATmega324A

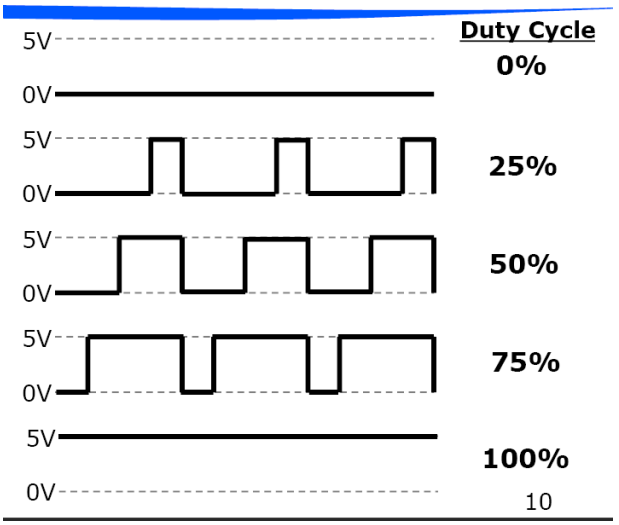
Timer/Counter 0	Timer/Counter 1	Timer/Counter 2
Pins: PB3 (OC0A) & PB4 (OC0B)	Pins: PD5 (OC1A) & PD4 (OC1B)	Pins: PD7 (OC2A) & PD6 (OC2B)
8-bit timer/counter	16-bit timer/counter	8-bit timer/counter
Supports PWM	Supports PWM	Supports PWM
Modes of operation: Normal, CTC, Fast-PWM and phase correct PWM	Modes of operation: Normal, CTC, Fast-PWM, PC-PWM and PFC-PWM	Modes of operation: Normal, CTC, Fast-PWM and phase correct PWM
Clock prescalar: No clock, F, F/8, F/64, F/256, F/1024	Clock prescalar: No clock, F, F/8, F/64, F/256, F/1024	Clock prescalar: No clock, F, F/8, F/32, F/64, F/128, F/256, F/1024
I/O Registers: TCNT0, TCCR0A, TCCR0B, OCR0A, OCR0B, TIMSK0, TIFR0	I/O Registers: TCNT1H, TCNT1L, TCCR1A, TCCR1B, TCCR1C, OCR1AH, OCR1AL, OCR1BH, OCR1BL, TIMSK1, TIFR1	I/O Registers: TCNT2, TCCR2A, TCCR2B, OCR2A, OCR2B, TIMSK2, TIFR2

7

Pulse Width Modulation (PWM)

Monday, 20 September 2021 5:33 PM

- Varying the duty cycle of a periodic pulse
- **Duty cycle** = % of time that signal is on (high)

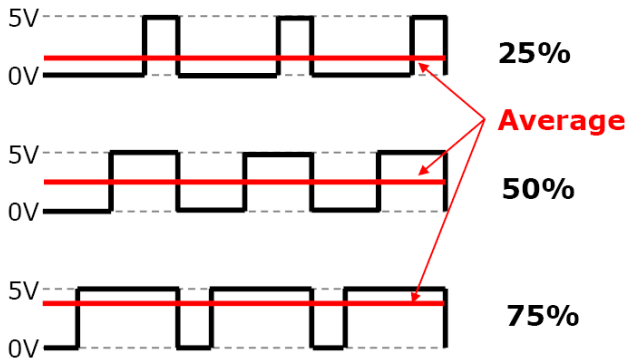


Remember

- The period for all below is the same.
- It is just the percentage of time that the signal is high that changes.

PWM - Average Value

- If periodic pulse is "fast enough" - signal looks like the average value



$$\text{Duty Cycle} = \frac{T_{ON}}{T_{ON}+T_{OFF}} \cdot 100\%$$

Average voltage formula

$$V_{avg} = \frac{1}{T}\int_0^T V(t)\,dt$$

... and

$$\text{Duty Cycle} \propto V_{avg}$$

- This means that PWM can be used to control the average voltage delivered to a load
  - e.g. controlling the speed of a DC motor

- Pulse width can be varied over time so that the "average value" also changes with time

Uses of PWM

- PWM is useful for generating signals that look analogue from a purely digital source
- Can be used for
  - Varying brightness of lights/LEDs (e.g. fading)
  - Motor control (variable speed)
  - Generating audio (with appropriate filtering)

Example: LED Brightness Control (Fading)

PWM on the AVR

Monday, 20 September 2021 11:30 PM

Fast PWM Mode

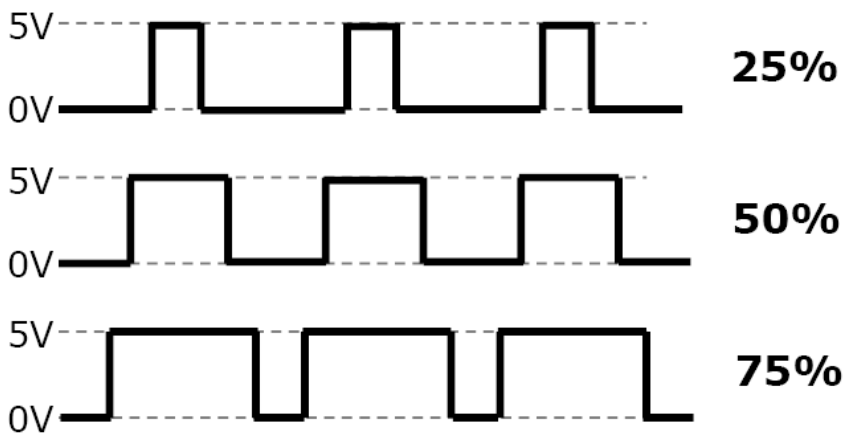
Example with Timer/Counter 0

- Two options - Waveform Generation Mode bits (recall other modes from lab 13)

Phase Correct PWM

Monday, 20 September 2021 11:31 PM

- Centres of pulses remain at the same point in the period
- Better for motor speed control
- Implemented by having timer count up and then down
- We won't use this mode however.
    - We will only use FAST PWM mode



In this Lecture

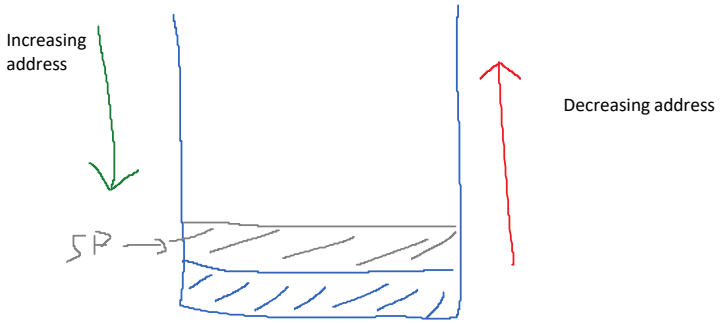
Wednesday, 22 September 2021 9:08 AM

- Stacks revisited
- Polling
- Interrupts

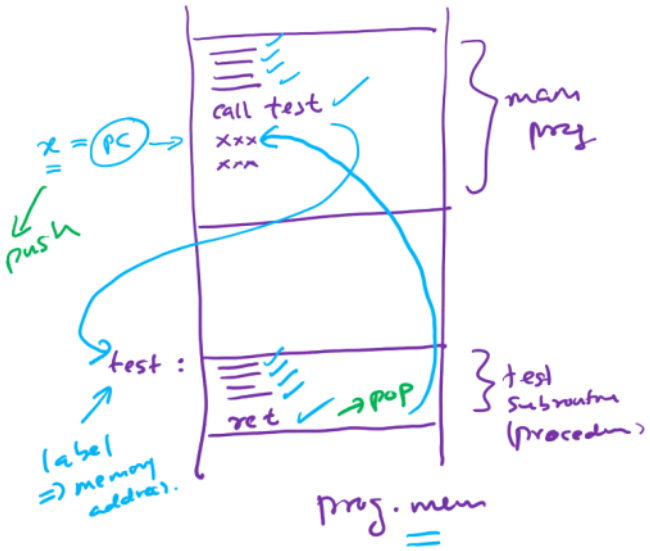
AVR Stack

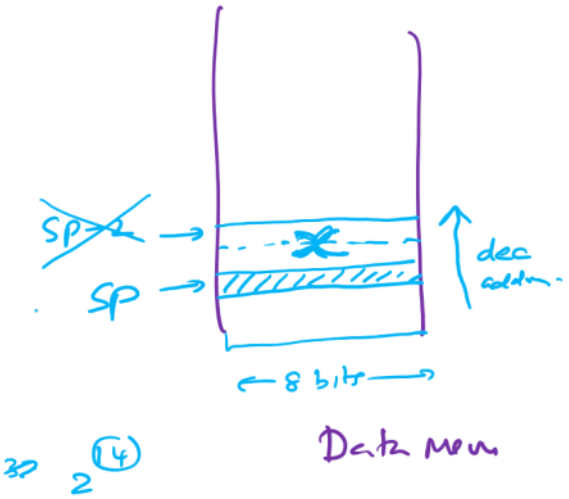
Monday, 20 September 2021 11:32 PM

- Instructions
  - `push Rr`
    - Value in register Rr place on top of stack
    - SP <-- SP - 1 (top of stack pointer adjusted)
  - `pop Rd`
    - Value on top of stack placed in register Rd
    - SP <-- SP + 1 (stack pointer adjusted)
- Procedure call: `call`, `rcall`, `icall`, `ret` instructions
  - e.g. call instruction: call label
    - Program counter value (return address) placed on top of stack
    - SP <-- SP - 2 (since 16 bits needed for return address)
    - PC <-- label value.



Function Call Example





4

I/O

Monday, 20 September 2021 11:40 PM

CPU Interacting with I/O

- CPU operates faster speed than I/O and I/O can be sometimes asynchronous
- CPU needs to know
  - When data is available from input devices
  - When output devices are ready to accept data
- Two options
  - **Polling** driven I/O
    - CPU is always listening to I/O
  - **Interrupt** driven I/O
    - CPU does its own thing and listens to I/O when attention is called

Polling the Input(s)

- Processor periodically asks devices and waits for response
- Inefficient (wasting time asking and waiting!)
  - But may be OK if the processor has nothing else to do

Interrupts

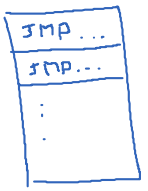
- Device(s) tells CPU when it is ready
- Examples
  - Device ready to accept more output
    - Serial transmit buffer is empty
  - Device has input ready
    - Key pressed on keyboard
  - Regular interrupt
    - Timer generates interrupt every e.g. 20ms

- More efficient for CPU
- Changes the flow of control

- CPU receiving interrupt causes it to execute software to handle the interrupt
  - Software routine is called
    - **Interrupt handler**, or
    - **Interrupt service routine** (ISR)
  - When the handler completes, control must be returned where we left off. → Stack
- Issues
  a. How does the CPU know where the ISR is?
  b. All registers (including the status register) must have their values preserved - How?
  What happens if another interrupt arrives whilst we're servicing an interrupt?

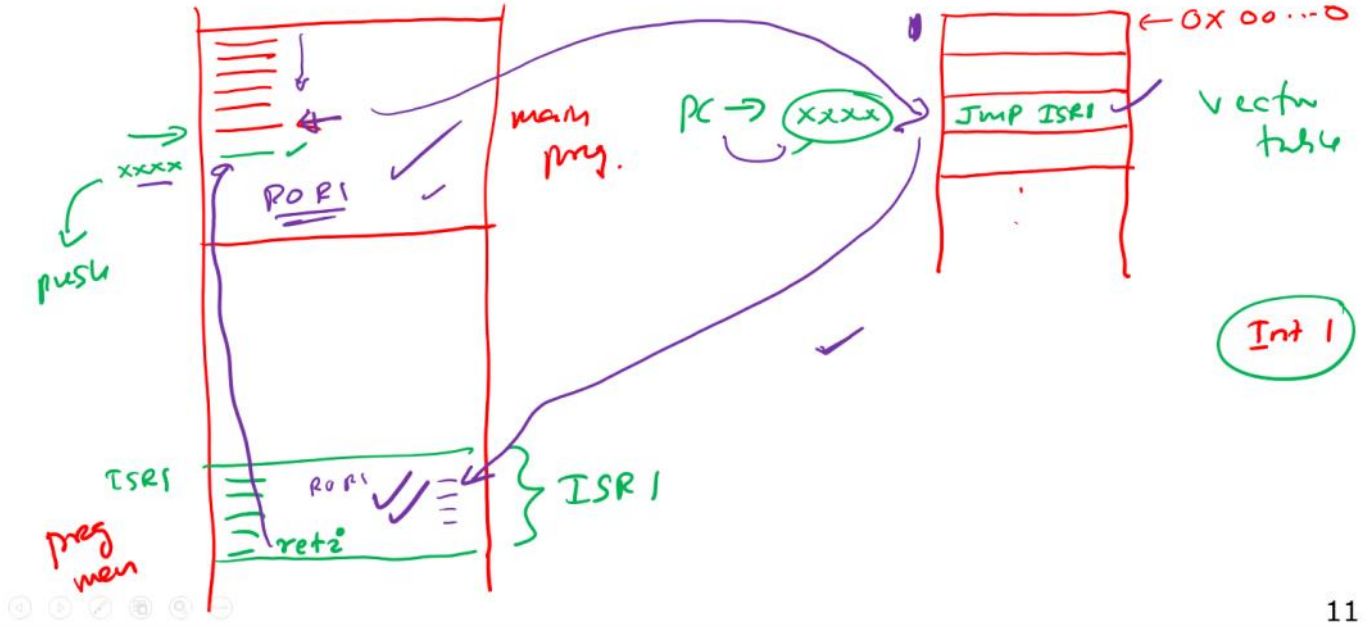
Finding the Interrupt Handler

- Based on interrupt source, determine address of ISR for that interrupt
  - Look in **interrupt vector table** (often at beginning of memory).
    Table contains:
    - Table of addresses, or
    - Table of jump instructions (AVR does it this way)



- Jump to address

- Note: vector sometimes refers to the interrupt number (index into the vector table) and sometimes the address of the interrupt handler itself.


xxxx
push
R0 R1
main prog.
PC → xxxx
JMP ISR1
← 0x 00...0
vector table
Int 1
ISR1
R0 R1
reti
ISR 1
prog mem

AVR Interrupt

Monday, 20 September 2021 11:45 PM

- For theATmega324A (from datasheet, pages 69-70)

Vector	*Program Addr*	*Source*	*Interrupt Definition*
1	0x0000	RESET	RESET, Watchdog Reset, etc.
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
	...		
11	0x0014	TIMER2_COMPB	Timer/Counter2 Compare Match B
12	0x0016	TIMER2_OVF	Timer/Counter2 Overflow
	...		
20	0x0026	SPI_STC	SPI Serial Transfer Complete
21	0x0028	USART0_RX	USART0 Receive Complete
22	0x002A	USART0_UDRE	USART0 Data Register Empty
23	0x002C	USART0_TX	USART0 Transmit Complete
24	0x002E	ANALOG_COMP	Analog Comparator
	...		

A Template for AVR Interrupt Service Routine (ISR)

-
```
isr_label:
    ; Save any registers used
    push r0
    in r0, SREG
    push r0

    ; … perform operation
    ; … (usually input/output)

    ; Restore registers
    pop r0
    out SREG, r0
    pop r0
    ; return from interrupt
    reti
```

AVR Interrupts in C

- `#include <avr/interrupt.h>`
- Interrupt handler can be written as a C function, using a special ISR macro, e.g.
```
ISR (INT0_vect) {
    /* code here */
}
```
- This is the handler for the `INT0` external interrupt
  - Interrupt vector table setup and register saving/restoring happen automatically
  - Use source name from datasheet followed by `_vect`
- See AVR LibC manual - `avr/interrupt.h` documentation

Setting up Interrupts

Besides the handler, also need to:

- Set up conditions for the interrupt
- Enable that particular interrupt
  - An I/O register bit for each interrupt controls whether that interrupt is enabled or not
- Clear the specific interrupt flag (to ensure interrupt doesn't trigger immediately)

    - Usually done by writing **1** to some I/O register bit
- Turn on interrupt globally
    - `sei()` macro (same as `sei` assembly language instruction)

Volatile Variables in C

- Example
  `volatile uint8_t count;`
- Where a variable can be changed outside the normal flow of a program (e.g. in an interrupt handler) it should be declared as volatile
    - Prevents code optimiser from assuming variable value doesn't change and optimising code away

Nested Interrupts

- Chance that another device interrupts during ISR execution
    - Could simply disable subsequent interrupts during the ISR
        - Usually, the global interrupt enable flag is one of the ones restored by the return interrupt instruction
    - Better to prioritise interrupts, and allow higher priority interrupts to interrupt lower priority ones (and not vice versa, or at the same level)
        - Many CPUs support this (not AVR)

AVR Interrupt Priority

- Priorities only matter when two (or more) interrupts are waiting to be serviced
    - Highest priority (lowest vector) will be serviced first
- Interrupts are disabled during ISR
    - Re-enabled by RETI instruction
- Possible to enable interrupts during ISR
    - Priorities not relevant here
    - Any incoming (or pending) interrupt may be serviced - even if lower priority

Preserving Register Values

Monday, 20 September 2021 11:46 PM

- If an ISR uses registers
  - Save copies of those registers before using them
  - Restore values afterwards (before returning from interrupt)
- In particular, status register must be saved
  - Many CPUs do this automatically
  - AVR does not - must do so manually (if status register would be changed by handler)
- Called **transparency**
  - When ISR finishes, computer is in the same state as it was before the interrupt.

Interrupt Summary - Sequence of Actions

Hardware

1. Device asserts interrupt line
2. CPU determines source of interrupt
3. Push PC (return address) onto stack
   a. Some CPUs push status register as well, but not AVR
4. Determine Address of ISR (e.g. from interrupt vector table)
5. CPU puts address of ISR into PC
6. CPU disables interrupt

Software

7. Save all registers to be used
8. Determine source of interrupt (if this is not done by hardware)
9. Handle the interrupt by executing the ISR code (e.g. Do required input or output)
10. Restore Registers
11. Execute return from interrupt instruction

Back to Hardware

12. Interrupts are re-enabled
13. CPU pops PC off the stack and restores it
14. Execution continues form where it was "interrupted"

Instructions which change the Status Register

How do you know which instructions change the status register?

See the instruction set reference

Traps

Monday, 20 September 2021 11:36 PM

- Traps are software interrupts, i.e. caused by events in software rather than in hardware
  - Exceptional events
- Examples
  - Overflow
  - Divide by zero
  - Undefined opcode
- Traps save continually checking for errors
- Trap handlers (service routines) don't always return to the original program
- AVR doesn't provide traps
  - But is possible to generate hardware interrupts from software.

Serial Communications

Wednesday, 6 October 2021 9:07 AM

- Communications can be serial or parallel
- AVRs and many other devices support serial communication
  - Communicate one bit at a time
- Two types of serial communication:
  - Synchronous Transfer e.g.
    - SPI - Serial Peripheral Interface
  - Asynchronous Transfer, e.g.
    - UART - Universal Asynchronous Receiver and Transmitter
    - AVR Atmega324A has USART - Universal Synchronous/Asynchronous Receiver and Transmitter
      - ATmega324A (IN students) - USART0 and USART1
    - USB

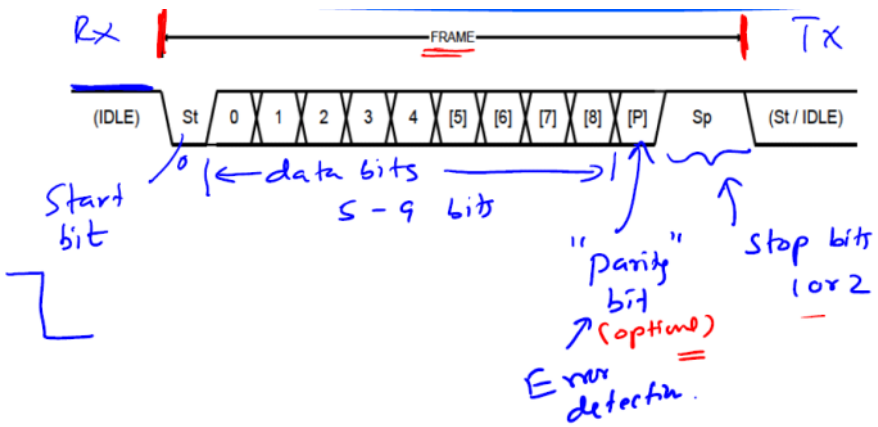
Synchronous vs Asynchronous

- Synchronous
  - Clock signal is transmitted with the data
- Asynchronous
  - Clock signal must be recovered from data stream
    - Data stream must be encoded to allow this to happen, e.g.
      - Start and stop bits
      - 4B/5B
      - 8B/10B
      - Bit stuffing (used in USB low speed and full-speed)
      - Etc

AVR USART

- USART = **U**niversal **S**ynchronous and **A**synchronous **R**eceiver/**T**ransmitter
- Key points
  - Serial communication (we communicate **frames**, 1 bit at a time)
  - Asynchronous
    - Receiver and Transmitter can have different clock rates
    - Clock not transmitted with data
    - Uses start and stop bits
- Often see term UART used
- Datasheet pages for UART
  - ATmega324A - pages 175 to 201

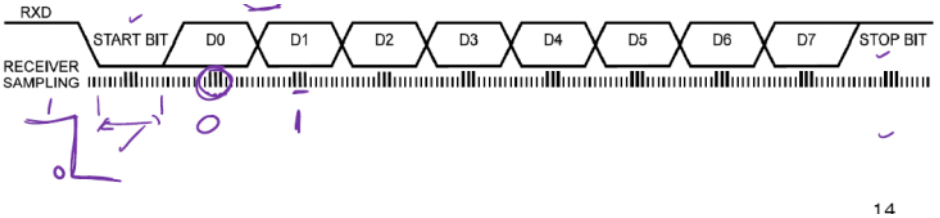
UART Frames



- Parity bit is used for error detection
- Even parity - the number of 1's (in the data bits) including the parity bit must be even
- Odd parity - the number of 1's (in the data bits) including the parity bit must be odd
- Parity bit is created at the Tx (transmitter) and parity check is performed at the Rx (receiver)

Recovering the Data

- Receiver must know the baud rate
  - But clocks don't have to be synchronised
- Start bit (1->0 transition) triggers sampling
- Multiple samples per bit
- AVR uses 16 (or 8 on double speed mode)



Baud rate

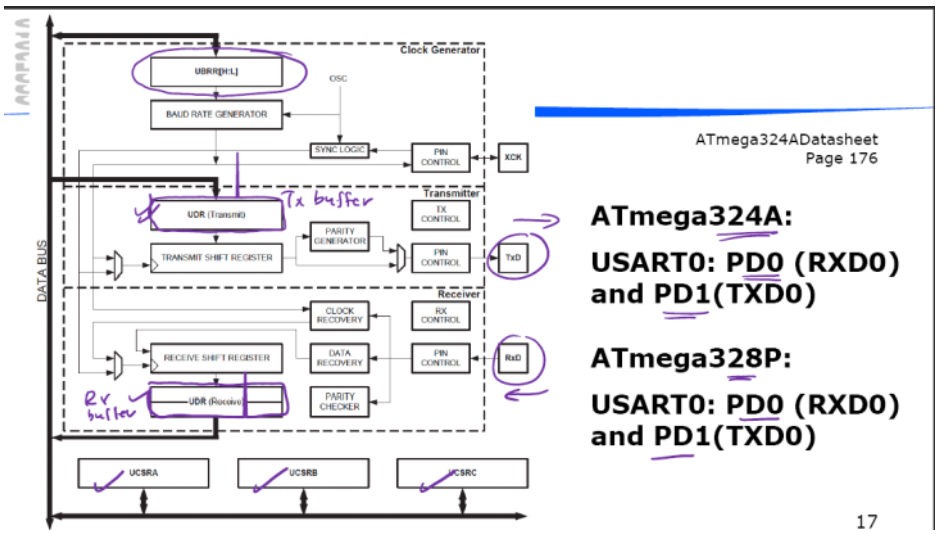
- Baud rate = symbols per second
- This is different from bit rate
  - e.g. frames of 8-bits, no parity, one start bit, one stop bit
    - 10 symbols per 8 bits
    - Therefore, 9600 baud = 7680 bits per second = 960 bytes per second
- There are a number of industry standard baud rates:
  - 2400
  - 4800
  - 9600
  - 14400
  - 19200
  - 28800
  - 38400
  - 57600
  - 115200

AVR Baud Rate

- Baud rate register is set based on
  - Device clock speed (8MHz for ATmega324A)
  - Number of clock cycles between samples - 16 samples per symbol
  - Can calculate - or see datasheet tables
    - Pages 198 to 201 for ATmega324A

$$UBBRn = \frac{f_{osc}}{16BAUD} - 1$$

We will mainly use the Asynchronous Normal Mode of the USART.



Interacting with AVR UART0 - Key I/O Registers

- USART Data Register UDR0
  - Actually two registers - one for reading, one for writing
  - Data written to this register is transmitted
  - Data arriving over the serial port can be read from this register

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDRn (Read)
	TXB[7:0]								UDRn (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- USART Control and Status Registers (UCSR0A, UCSR0B, UCSR0C)

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

[From page 193-196 of ATmega324A datasheet]

- USART Baud rate registers (UBRR0H, UBRR0L)
- Either compute the Baud rate register value using the formula given or use the common example values given in the datasheet

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	UBRR[11:8]				UBRRnH
	UBRR[7:0]								UBRRnL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Serial Interrupts

- 3 interrupt sources associated with each AVR serial port
  - Receive complete
    - Frame (usually character) received
  - Transmit complete
    - Frame sent and no data waiting to be sent
  - Data Register empty
    - Ready to accept new data for transmission

Serial Communications on the PC

- IO Board provides a USB-serial device
- On PC we use a terminal program
  - Putty
- Need to specify
  - COM port (which PC serial port you're communicating over)
    - May need to use device manager to determine this
  - Baud rate, start/stop/parity bits etc

RS232

- Serial Communication Standard
  - Standardised over 40 years ago, still in use
  - Typically uses a 9-pin D-sub connector
- Voltage levels
  - Logical one = Negative voltage (-3 to -15V)
  - Logical zero = Positive voltage (+3 to +15V)
- Full duplex (can transmit and receive simultaneously)
- Various subsets of signals possible
  - Need at least
    - Transmit Data (TxD)
    - Receive Data (RxD)
    - Ground
- Cab turn out 0-5V serial into RS232 using a converter chip (e.g. MAX-232)

Analog to Digital Conversion (ADC)

- Voltage between 0 and some maximum voltage (Vref) converted to digital value
- ADC has some <u>resolution</u>
  - e.g. 10 bits (range 0 to 1023)

ADC on the ATmega324A

- Port A pins on ATmega324A can be used as analog inputs
- One input can be converted at a time (using an analog multiplexer)
- Can choose Vref - we'll pick V_{cc} = 5V
- ADC resolution = 10 bits (0 to 1023)
- Conversion takes time
  - Up to 3200 clock cycles (0.4ms at 8MHz)
  - Can trade-off accuracy for speed

Bus Masters and Slaves

Wednesday, 6 October 2021 9:09 AM

- Master
  - Device which initiates (and controls) the transfer of data
- Slave
  - Waits for requests
  - Responds to master
- Some devices can be both masters and slaves (at different times)
  - Memory can never be a bus master

I^2C, TWI

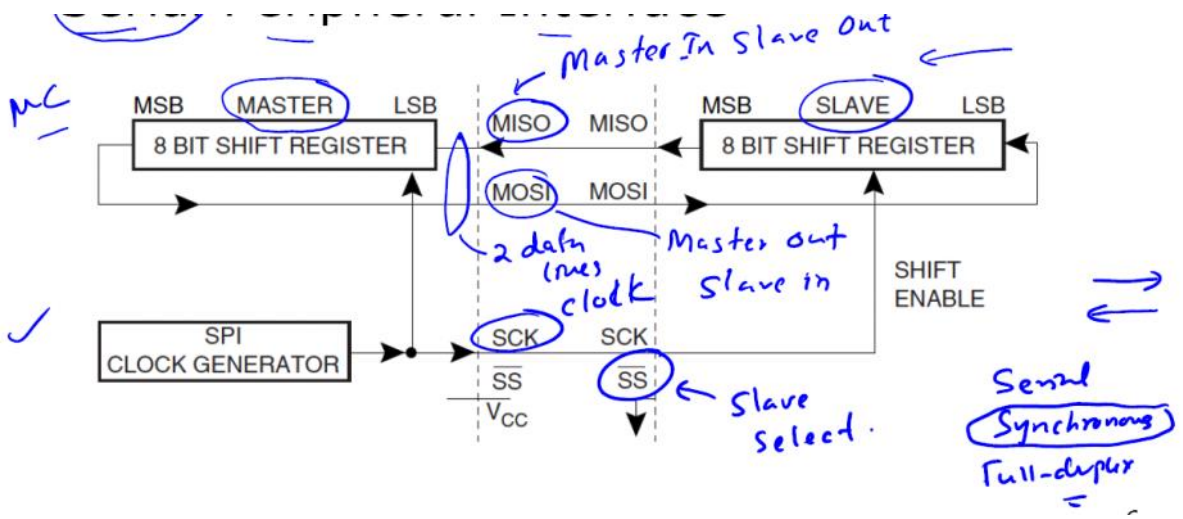
- I^2C = Inter-Integrated Circuit
  - Created by Philips Semiconductor (now NXP)
- TWI = two wire interface
  - Name used by Atmel. But these are the same thing.
  - It is a serial synchronous communication

TWI on AVR

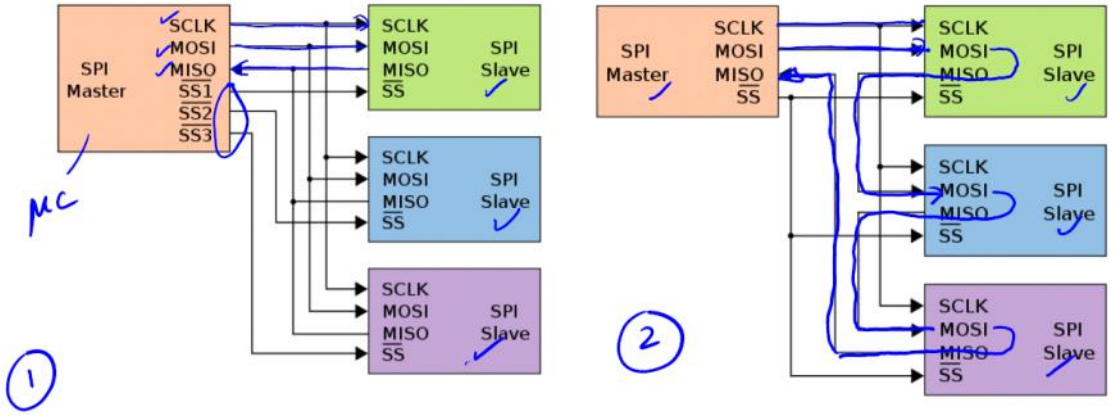
- Atmega324A has one TWI port
  - Alternate use for pins
    - C0 (SCL)
    - C1 (SDA)
- See datasheet pages 211 to 239 for details
  - Includes full description of TWI

SPI

- Serial Peripheral Interface



SPI - Other Configurations



SPI on the AVR

- Atmega324A has one SPI port
  - Shares pins with upper half of port B
  - Hardware takes care of generating signals
  - Can be master or slave
  - See datasheet pages 166 to 174
  - USARTs can also be used as SPI masters
- For Master operation, can choose clock rate divider:
  - 2 (4MHz @ 8MHz clock)
  - 4 (2MHz)
  - 8 (1MHz)
  - 16 (500kHz)
  - 32 (250kHz)
  - 64 (125kHz)
  - 128(62.5kHz)
- For slave operation, fastest clock is $\frac{f_{osc}}{4}$ (2MHz in our case)

Bit Bashing

- Sometimes microcontroller doesn't have support for a bus standard
  - Or enough interfaces of a particular standard
- Can simulation I/O standards using general purpose ports

AVR I/O Coding

Volatile Variables

- Example: `volatile int running;`
- Tells computer that variable can change outside of standard flow of control
  - e.g. in interrupt handler or a thread
- Prevents compiler applying optimisations
- Example:

```
running = 0;
while(!running) {
    ; /* Do nothing */
}
```

Setting I/O Register Bits

- When setting/clearing specific bits in I/O registers, be explicit about the bits

Example (all have the same effect)

- Poor
  - `TCCR1B = 10;`
- Better:
  - `TCCR1B = 0x0A;`
- Best:
  - `TCCR1B = (1 << WGM12) | (1 << CS11);`
- Makes code easier to understand

- Where do names come from?
  - See datasheet (note errors - see IO register summary)
- e.g. for TCCR0B (page 112)

- Bit names are defined as macros, e.g.
  `#define WGM02`
- Can use name names in assembly language also, e.g.
  `ldi r16, (1 << WGM02) | (1 << CS01)`

Clearing/Toggling I/O register bits

- Same rules apply
- To clear a bit, e.g.
  `PINC &= ~(1 << 5)`
- To toggle a bit, e.g.
  ```
  /* Enable Output Compare A Interrupt for timer 2 */
  TIMSK2 ^= (1 << OCIE2A)
  ```

Assembly Language Format

Thursday, 14 October 2021 1:01 PM

Formula: `ldi r20, i ; register r20 = i`

- Statements have 4 parts (called fields):
  - A field for label
  - A field for operation (opcode)
  - A field for operand(s)
  - A field for comments
- Labels needed so statements can be jumped to, and so that data can be referenced.
- A label usually starts in column 1
- Some assemblers require a semicolon to indicate end of a statement, others don't
- Opcode has symbolic abbreviation for an instruction opcode
  - e.g. LDS for Load Direct from Data Space
- Operand field specifies addresses and registers used as operands of the instruction
- Register names vary from machine to machine
  - Pentium has eax, ebx, esi, edi, etc
  - Sparc has o0 … o7, i0 … i7, l0 … l7, g0 … g7
  - AVR has simple r0-r31
- Comments field is space for programmer to put helpful explanations
  - Only for human consumption
  - Vital for assembly language programs

Other Stuff

- Need to provide other information to the assembler, e.g.
  - Where in memory the program should be stored etc.
  - Nicer names to use for registers (e.g. temp, SPH)
- A command (directive) to the assembler itself!
  - Example `.byte`
- Precede with "."
- Called "**Pseudo-instruction**" or "**directive**"
  - These are commands for the assembler.
    - e.g. .def temp = r16

Memory Segments

Tuesday, 19 October 2021 10:13 AM

- Different types of memory are known as **segments** to the assembler
- Assembler directives enable code/data to be placed into different segments
- AVR has
  - Data segment (RAM)
    - Can't place values here, just reserve space (for variables)
  - Code segment (Flash)
    - Can place program code or constant data here
  - EEPROM Segment
    - Can place constants here

- Data memory (
  - .dseg
- Program Memory (32KB)
  - .cseg
- EEPROM
  - .eseg

Pseudo Instructions

From the Atmel Studio Help

These are for the Microchip Studio Assembler AVRASM2

- `.byte`. Reserve space; only allowed in dseg
  - Data memory only
- Segment directives `.cseg` and `.dseg` allow the text and data segments to be built up in pieces:

```
                    .dseg
amount: .byte 2
                    .cseg
formula: ...
                    .dseg
count:  .byte 2
```

define byte

• .def temp = r16
• .equ
• .org
ldi temp, 10

Directive	Description
BYTE	Reserve byte(s) to a variable.
CSEG	Code Segment
CSEGSIZE	Program memory size
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DSEG	Data Segment
DW	Define Constant word(s)
ENDM, ENDMACRO	EndMacro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn Macro expansion in list file on
MACRO	Begin Macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression
ELSE,ELIF	Conditional assembly
ENDIF	Conditional assembly

`.db`: Initialise constant in code or EEPROM segment
`.dw`: As above but defines a 16-bit word
`.def`: Make a definition (for registers only)
`.device`: Specify the exact processor that this program is designed for
    .device Atmega324A
`.include`: Include a file
`.exist`: Stop processing this file
`.equ`: Equate (not changeable)
`.set`: Equate (but changeable)
    `.equ sreg = 0x3f;` status register
    `.set io_offset = 0x23;` for now
`.org`: Set location counter - i.e. address (in any segment)

- The above all apply to the assembler built into the Atmel Studio tool
- Other assemblers (even targeting the AVR) use different syntax.

Assembly Process

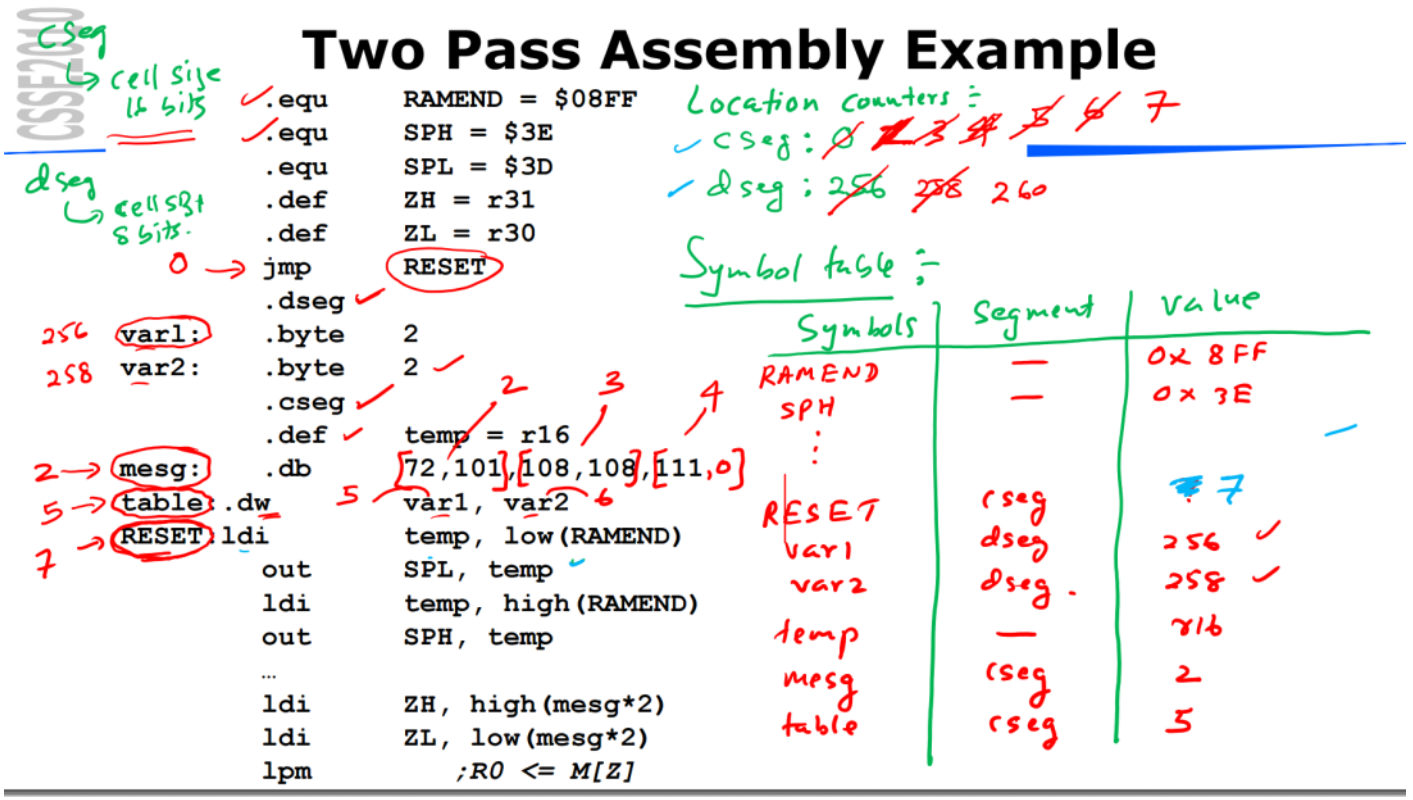
Tuesday, 19 October 2021 10:32 AM

- Assemble language program consists of one line statements
  - We can not always just read each statement and generate machine code
  - For example, first statement is

    `jmp RESET`
  - We don't know the address of the RESET label, so we can't generate the instruction
    - We know the opcode but not the operand
  - This is called **forward reference problem**

Two Pass Assembly Process

- We need to process the file twice
- Pass One
  - Define all the symbols (labels etc.)
- Pass Two
  - Values of symbols known
  - Can **assemble** each instruction
    - i.e. produce the actual machine code

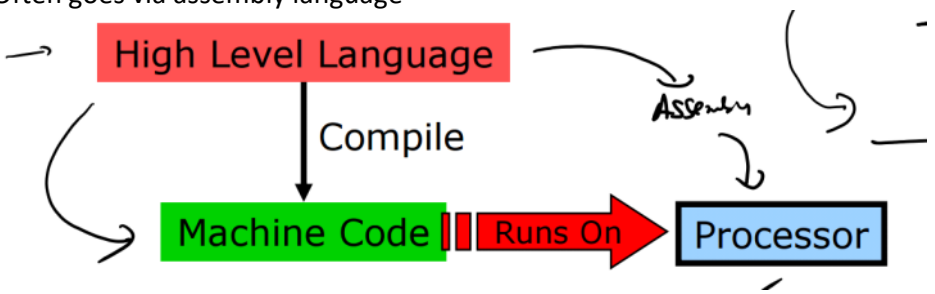


Compilation and Interpretation

Thursday, 14 October 2021 1:01 PM

Compilation

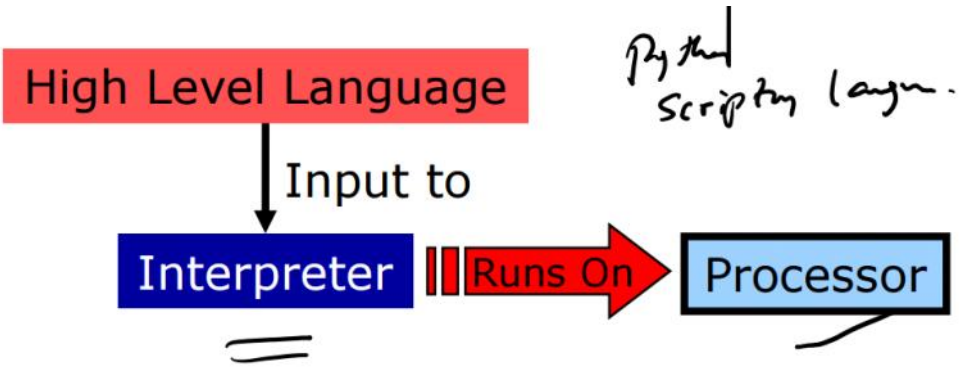
- Programs must be converted to machine code to run
  - This process is called compilation
    - Often goes via assembly language



  - Not all programs are compiled

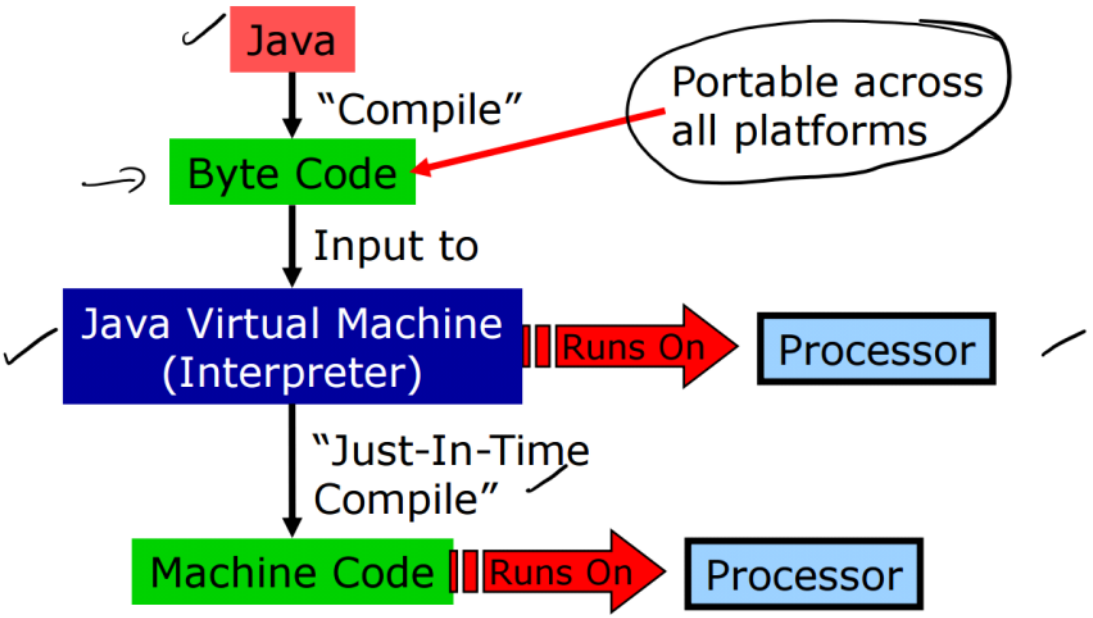
Interpretation

Some languages are interpreted



Java

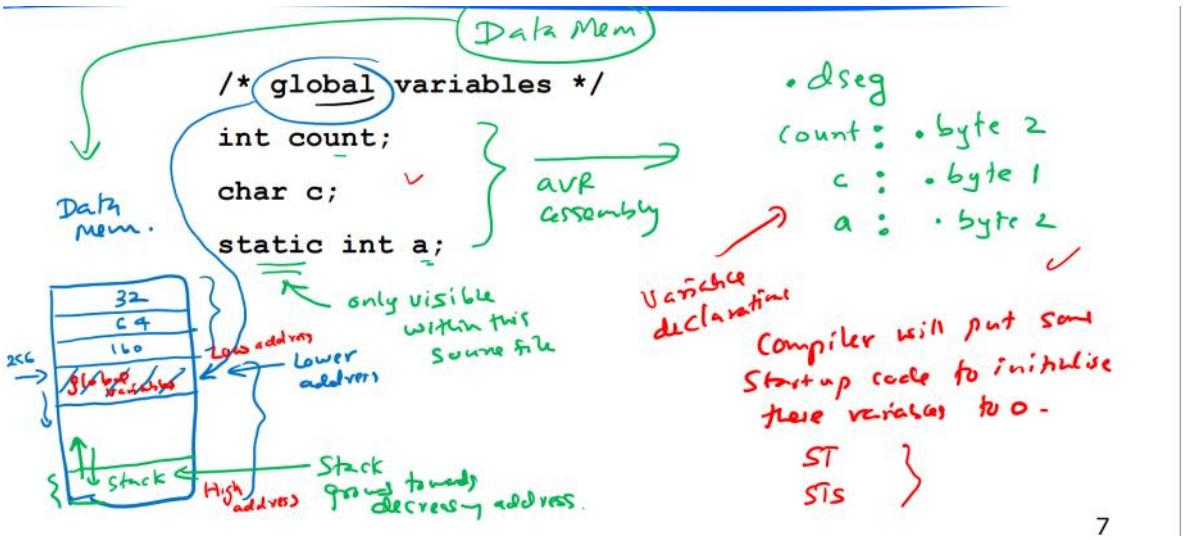
Works a bit differently

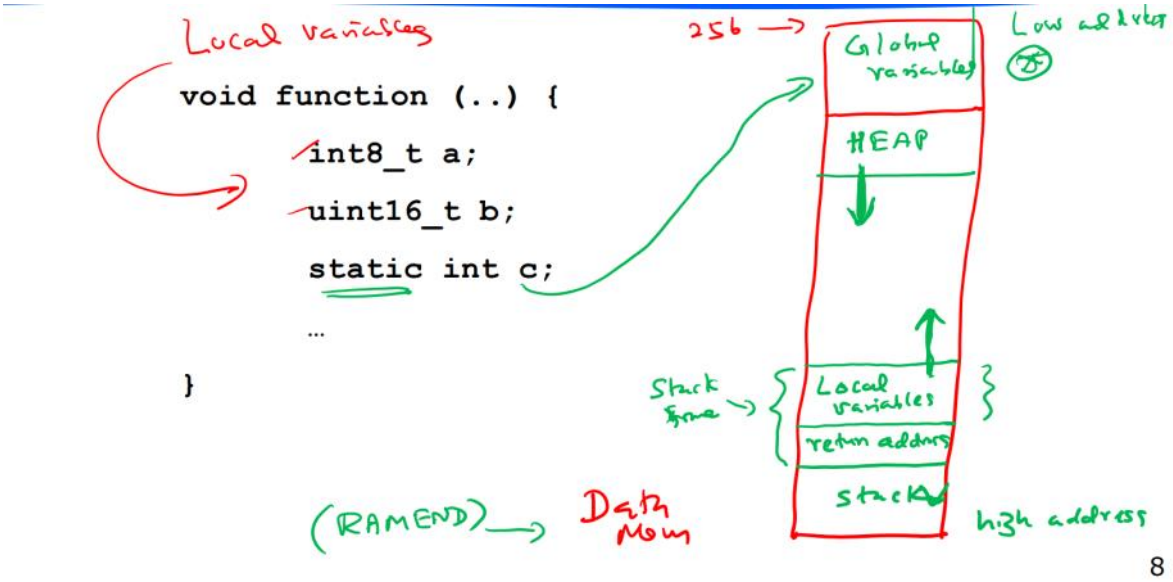


Compiling (Simplified View)

- Mapping variables to memory
- Translating C code to assembler instructions
- From assembler instructions to machine code (you already know this procedure)

How do variables in C map to memory?





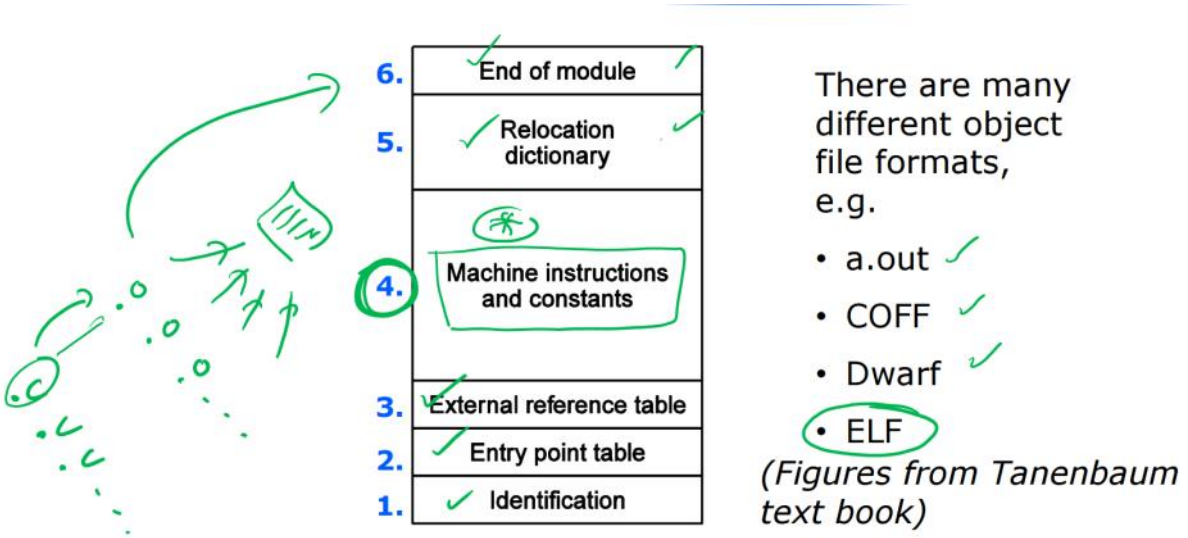
Compiling C Code

- Similar problem to the assembly process
- Compiler often generates **object files** (.o) - as an intermediate step
  - (Assemblers can generate object files also)
- Generate all necessary object files first (from macros, libraries, .h files)
- Link them all together and work out the final details such as memory locations at the end

Object Files

- Often useful to be able to compile/assemble modules separately and then link them together
  - Don't need to recompile/reassemble every file if only one of them is changed
- What does this mean
  - We don't know the location of the code (i.e. which addresses it will end up at)
  - When do we know the location, have to "fix" the code
  - To represent this we need what are called **relocatable object modules**
- We'll use terms "object module" and "object file" interchangeably

Object File Structure



1. Identification Section

Contains
- Name of module
- Location of the other sections in the file
- Assembly date

2. Entry Point Table

- Contains
  - List of symbols defined in the module that other modules may reference
  - Values of symbols (addresses)
    - Procedure entry points
    - Variables
    - Addresses are relative to beginning of this object module
- Example
  - File may define a procedure called **abs_value** at address 10

3. External Reference Table

- Contains
  - List of all symbols that are used in the module but not defined in the module (i.e. they're defined externally)
    - Procedure names
    - Variables
  - List of instructions which use those symbols

4. Machine Instructions and Constants

- Contains
  - Assembled code and constants
    - Code segment
    - Data segment (initialised and uninitialised data)
  - Note, that address referenced in code are not correct - they need **relocation**

5. Relocation Dictionary

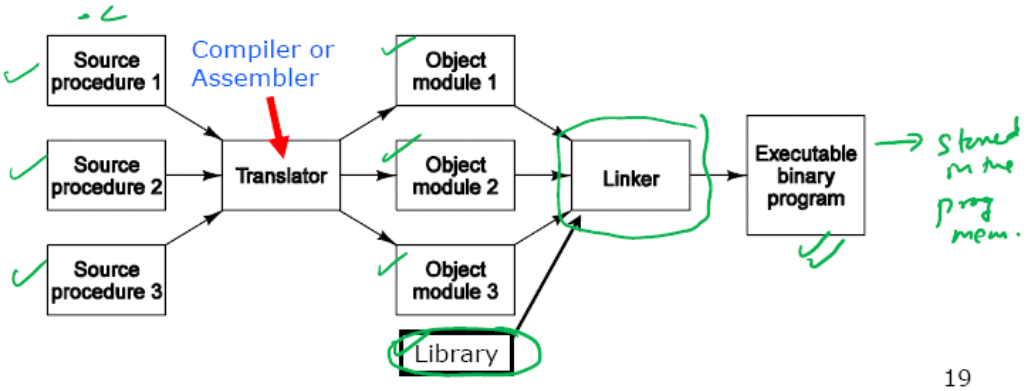
- Contains
  - List of addresses of code/data in section 4 that need to be relocated
- A **relocation constant** will be added to instructions/data at these addresses
  - Different constant for code and data

6. End of Module

- Contains
  - (Maybe) checksum - to catch errors
  - Address at which to start execution

Linker

- **Linker** combines object modules into an executable binary program (machine code)



Libraries

- Library
  - Collection of object files with additional index
  - Provides collection of useful functions/procedures
- Don't reinvent the wheel - use library code where possible
- Example: Standard C library
  - Standard input/output
  - String handling
  - Date and time functions
  - General utilities

Header Files vs Library

- Header (.h) files are **not** the library
- Header (.h) files:
  - Define pre-processor macros that are useful
  - Declare the functions that are in the library (function prototypes)
  - Declare variables that are available externally (extern)
- The library is the collection of object files that implements the functions

e.g. `#include <math.h>`

Linking

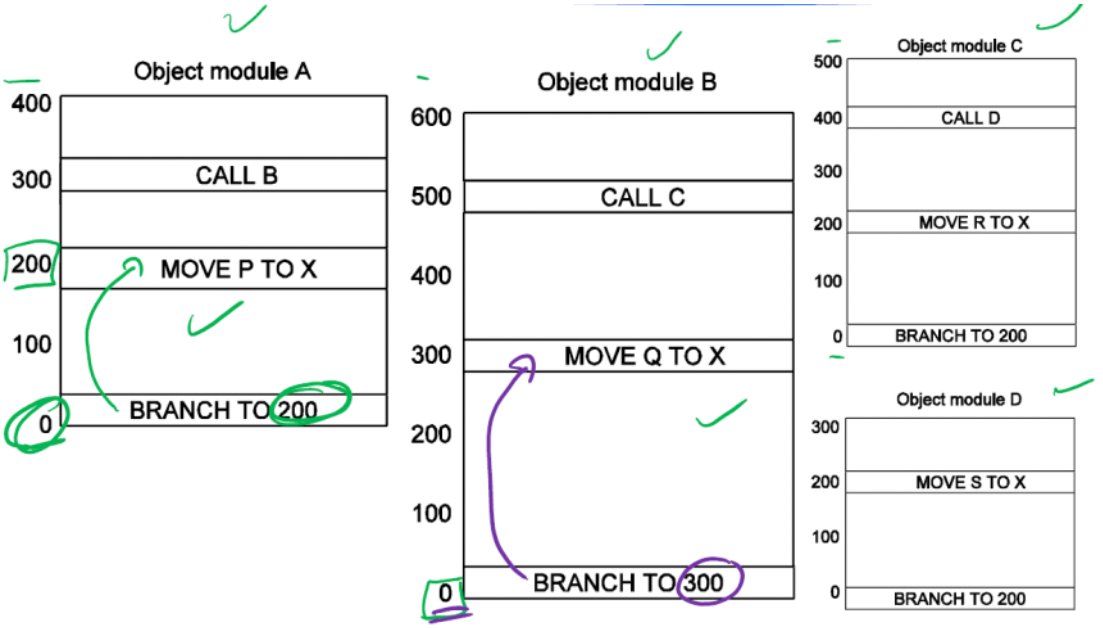
- Similar problem to assembler
  - **External reference problem** - symbols used before value known
- Linker therefore needs two passes as well
- Addresses need relocating or resolving when modules joined together
  - **Relocating** = add some constant value
  - **Resolving** = finding value of "external" address

Dynamic Linking

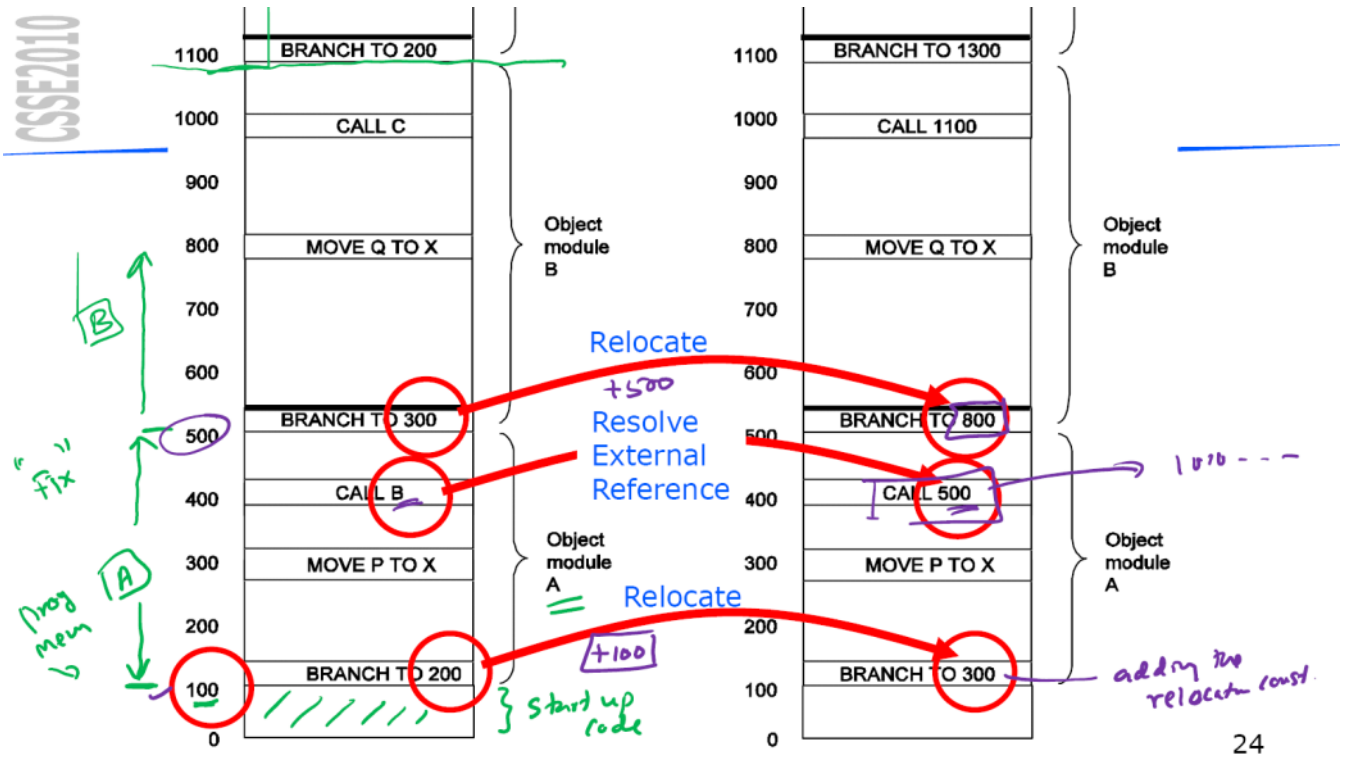
- So far, all object codes (library & main programs) are compiled into binary
- Alternatively, linking can happen at run-time
  - Link when the procedure is first called
- Example
  - DLLs on Windows
- Advantages
  - Saves space
    - Libraries aren't linked into binary executable

- Library can be updated independently of the programs that use it
  - Can be a disadvantage though - "DLL hell" on earlier versions of Windows

Example



23



24

Computer Memory

Tuesday, 19 October 2021 10:39 AM

Computers need memory to
- Store temporary results
- Store programs
- Remember settings when power is off

Many different types of memory
- Random access
- Static vs. dynamic
- Volatile vs non-volatile
- Read-only
- Primary vs secondary

Memory Types

Static memory
- Flip flops or latches used to store bits
- SRAM

Dynamic memory
- Each cell is a switch (transistor) plus capacitor

These are both types of RAM = Random access memory
- RAM originally meant could access any cell
- Now means volatile read/write memory
- Static memory = SRAM
- Dynamic memory = DRAM

Volatility

- **Volatile** memory
  - Contents "forgotten" when power is switched off
  - Both SRAM and DRAM are volatile
    - DRAM forgets contents within milliseconds if not refreshed, i.e. contents written)
- **Non-volatile** memory
  - Contents remembered when power is switched off
  - Many types
    - ROM
    - EPROM
    - EEPROM
    - Flash

Non-Volatile Memory Chips

- **ROM** = Read only memory
  - Programmed by memory manufacturer - contents hardwired in silicon
- **EPROM** = Erasable Programmable ROM
  - Can erase, e.g. with ultraviolet light
- **EEPROM** = Electrically Erasable PROM
  - Can erase by applying a certain voltage

- **Flash** = New EEPROM
  - (Used to be different technology to EEPROM but converging)
  - Main difference - Flash memory written/erased in blocks, EEPROM written/erased one cell at a time
    - ATMega324A supports two flash blocks - Application section + Boot loader section
    - Other AVR devices have one, other have more

Memory Capacities

Monday, 25 October 2021 6:34 AM

Megabyte

- A megabyte is one million bits
- Conventionally
  - 1MB = 2^{10}KB= 1024kB = 2^{20}bytes = 1,048,576 B
  - Sometimes, 1MB = 10^6B = 1,000,000 B
  - Or could be in between 1,000 kB = 1,024,000B
- Conventionally
  - 1GB = 2^{10}MB = 2^{20}KB = 2^{30}B = 1,073,741,824 B
  - Sometimes 1GB = 10^9B = 1,000,000,000 B
  - This is a 7.4% difference
  - Or could be in between: e.g. $1000 \times 2^{20} = 1{,}048{,}576{,}000$ B

When to use powers of 2? When to use powers of 10?

- For main memory capacity, always use powers of 2
- For hard disks, manufactures base sizes on powers of 10
  - Makes hard drives seem bigger
  - Operating Systems (e.g. Windows) base sizes on powers of 2
- For data speeds (transfer rates), usually based on powers of 10
  - 1 Gbps = 10^9 bits per second
- Occasionally a mixture
  - 1.44MB floppy disk stores $1.44 \times 1000 \times 1024$ bytes

Kibibytes, Mebibytes

- Kilo-, mega-, giga prefixes mean powers of 10 in SI units
- To avoid confusion, standardised terms have been created
  - 1 kibibyte = 1KiB = 1024 bytes
  - 1 mebibyte = 1MiB = 1024 KiB
  - 1 gibibyte = 1GiB = 1024 MiB
- Unfortunately, very few people use them as they are not known by most people

Memory

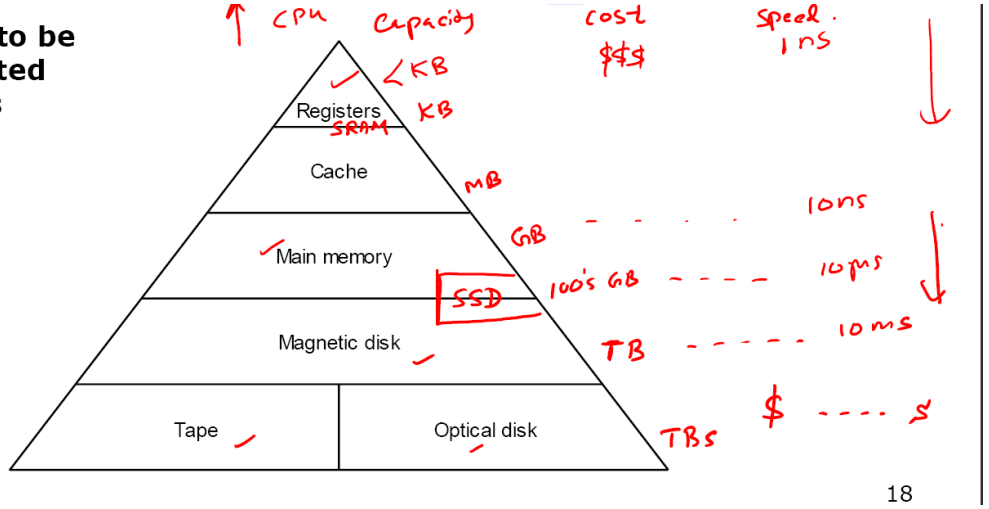
Monday, 25 October 2021 6:41 AM

Types of DRAM

- Bits stored as charge on capacitor, not in flip-flops
- Cells are smaller, so can pack more bits on a chip than for static memory
- Slower than static memory
- Many different types of DRAM over the ages
  - Fast page mode (FPM DRAM)
  - Extended Data Out (EDO DRAM)
  - Synchronous DRAM (SDRAM)
  - Rambus DRAM (RDRAM)
  - Double Data Rate Synchronous Dynamic Random Access Memory (DDR DRAM) 1, 2, 3 and 4 (and 5 haha)

Memory Hierarchy

Figure to be annotated in class



Cache Memory

- CPUs are faster than main memory
  - Very fast memory is expensive
- Performance suffers if CPU Is always waiting for memory
- Cache
  - Small amount of very fast memory combined with larger slow memory
  - Most commonly used memory words kept in cache
- CPU looks in cache before main memory
- Average access time greatly reduced if many words are in the cache

Cache Levels

Often multiple levels of cache

- Level 1 (L1)
  - Inside CPU itself
  - Runs at the same speed as CPU
- Level 2 (L2)
  - May be on separate chip (possibly inside the same module) or on the same chip as CPU
  - May be slightly slower than CPU
- Level 3 (L3)
  - Some machines have a third level

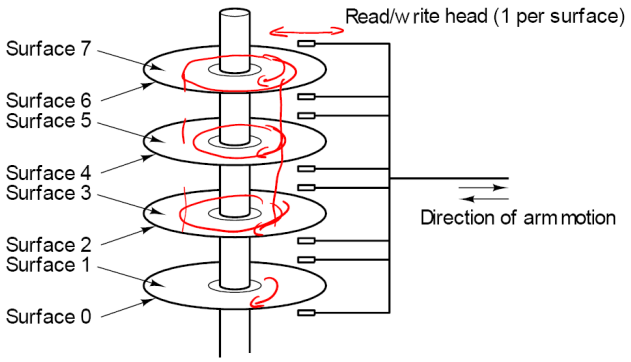
- The levels are in increasing capacity. e.g.

- L1 has a very small capacity
- L2 has a larger capacity
- L3 has the largest capacity (16 MB for Ryzen 5 1600X)

Cache for Hard Disks

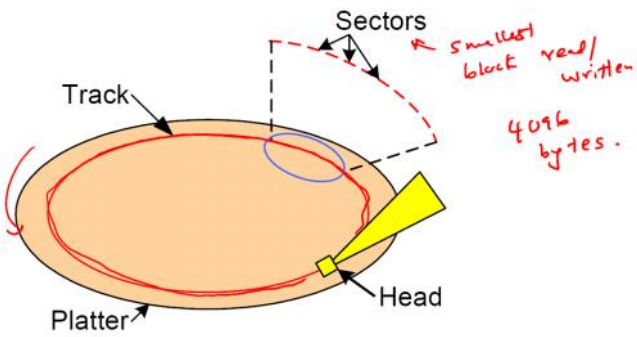
- Increasingly common
- Small fast solid state drives cache content from larger slower magnetic disks

Magnetic Disks



- Rotating platters with magnetised coating
  - Stack of platters
- Data stored magnetically in circular tracks
- Read/write heads float above platter surfaces
- Usually use both sides of a platter

Sectors and Tracks



Cylinders

- Set of tracks at a given radial position
- Number of tracks in cylinder = 2 × number of platters
  - Assuming both sides of the platter are used
- Heads move together, i.e. to a certain cylinder

Sectors

- Tracks divide into fixed-length sectors
  - Smallest data unit, i.e. must read/write a whole sector at a time
- Sector consists of
  - Preable
    - Allows head to synchronise data
  - Data (often 4096 bytes)
  - Error correction codes (ECCs)
  - Inter-sector gab
- About 85% of the disk capacity is usable by the operating system
  - About 95% of this usable for user data
  - 5% file system overhead

Access Time

- Access time = seek time + rotational latency + transfer time

- Seek time
  - Time to move heads to the correct cylinder

- Average seek time = $\frac{1}{3}$ ×edge to edge seek time

- Rotational latency
  - Time to wait for the sector to arrive under the head
- Transfer time
  - Time for sector to pass under the read/write head
  - Negligible compared with above

Calculating Access Time

- Example:
  What's the *average* access time for a hard disk which rotates at 7200rpm and has an edge-to-edge seek time of 10ms? (Assume that seek time is proportional to the number of tracks to seek.)

120 rps

Avg access time = Avg seek time + Avg rotational latency

$= \frac{1}{3}$ [edge to edge seek time] $+ \frac{1}{2}$ [rotation time]

$= \frac{1}{3}[10\text{ ms}] + \frac{1/120}{2}\text{ s} \Rightarrow \underline{\underline{7.5\text{ ms}}}$

Virtual I/O - Files

Tuesday, 19 October 2021 10:39 AM

- User/programmer do not want to worry about the unit numbers, heads, cylinders, sectors, timeout errors, retries, etc.
- Common I/O abstraction is the file
  - To the Operating System, a file is just a sequence of bytes
    - Any structure is up to the application program

File System

- Computers are easier to use (and program) when files have names
- File system
  - Associates files (collections of sectors) with names and directories
- Some sectors used for index information (e.g. File Allocation Table)
- Other sectors store the files themselves
- Many different file systems
  - FAT
  - FAT32
  - NTFS
  - exFAT
  - EXT2
  - EXT3
  - EXT4
  - APFS
  - HFS+
  - UFS
  - ZFS

Disk Formatting

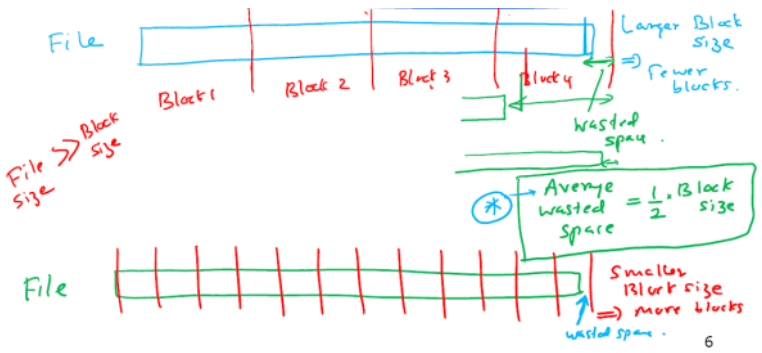
- Low level format
  - Reinitialises all sectors
    - Preambles, data (empty), ECCs
  - No file system
- High level format
  - Creates empty file system
  - Only writes index/table sectors

- Deleting a file doesn't just mean it's gone!
  - OS may just delete the index entry, not overwrite the sectors where actual data is stored

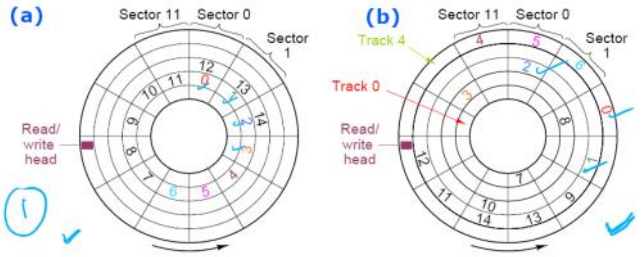
Files

- OS allows file names rather than cylinder and sector numbers
- File storage is allocated in blocks or allocation units
- Block could be as small as 1 sector or could be several consecutive sectors
  - Block sizes typically 512 bytes to 32KB
- OS maintains map from file name to a set of blocks
  - To help users, a hierarchy of directories (folders) can be used to organise files
- Larger block sizes reduce the size of this map, but increases wastes space at EOF

Block Size and Wasted Space



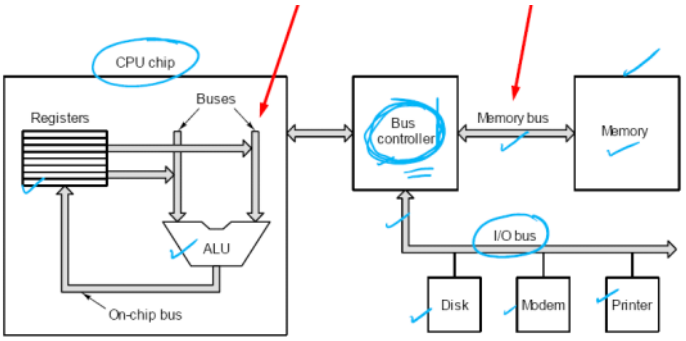
Disk Allocation



- Two possible allocations of 15 sectors assuming 1 block per sector
- However, (a) is ideal because it is not fragmented like (b)
- Fragmented allocation means slow access because of the additional seek time and rotational latency

Buses are Everywhere

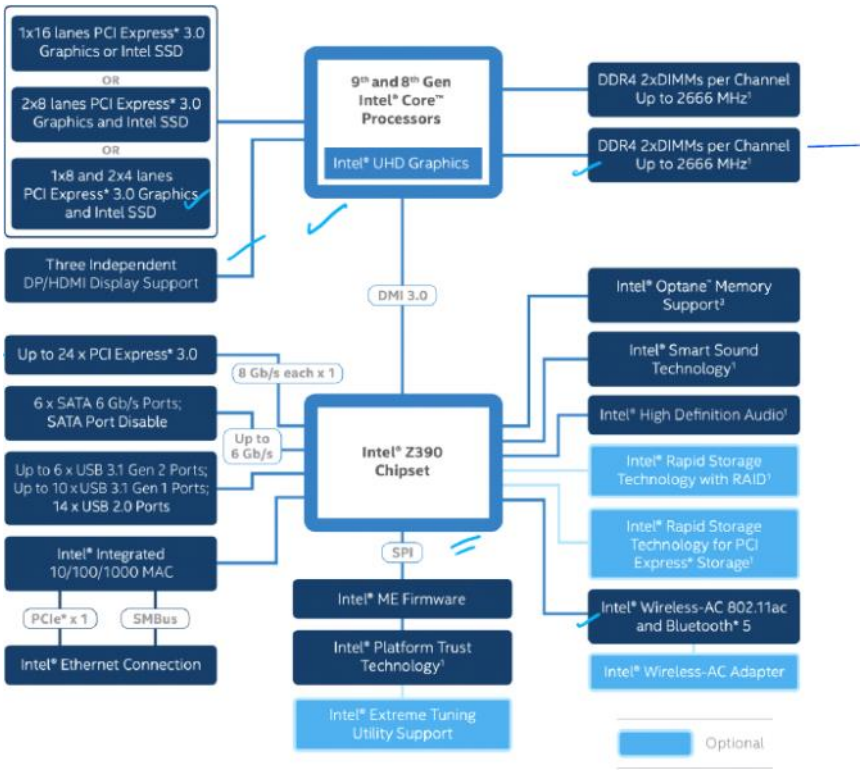
- Buses are both internal and external to the CPU



Bus Controller

- In the PC world, a Bus Controller is called a chipset
- Particular chipsets support particular bus standards
- Memory
  - DDR4, DDR3, DDR2, …
- I/O
  - PCI Express, USB, SATA, PCI, NVMe

Intel Z390 Chipset



Peripheral Buses (or I/O Buses)

- Peripheral = external input/output device
- Peripheral bus = input/output bus
- Examples
  - PCI Express
  - PCI
  - USB
  - SATA
  - SCSI
- And some outdated ones
  - ISA
  - EISA
  - Omnibus
  - Firewire
  - Unibus
  - Multibus
  - MCA
  - Nubus
  - AGP

PCI Bus

- Peripheral Component Interconnect bus
- Patented by Intel but in public domain
  - PCI 1
    - 32 bits wide, 33MHz -> 132 MB/sec maximum transfer rate
  - PCI 2.2
    - 64 bits wide, 66MHz -> 528 MB/sec maximum transfer rate
  - 32 bit cards have 120 pins; 64 bit cards have an extra 64-pin connector
- Parallel bus
- 5V and 3.3V options

PCI Express (PCIe) Bus

- High-speed, serial bus standard
- X1 version = 2 Gb/second
- X16 version = 32 Gb/second = 4GB/second

- PCI Express 2.0 (2007)
  - Doubles the data rates above)
- PCI Express 3.0 (2010)
  - 8GT/second (GigaTransfers - using 128b/130b)

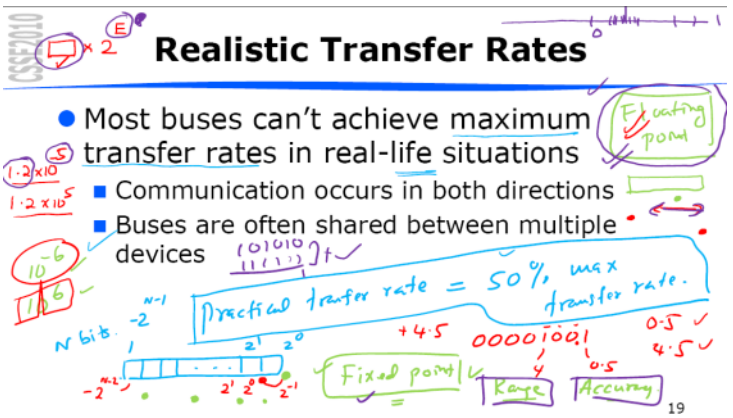
- PCI Express 4.0 (2018)
  - Double PCIe 3.0
- PCI Express 5.0 (expected 2019)
  - Double PCIe 4.0

Universal Serial Bus (USB)

- Originally only 4 wires (power, ground, 2 data)
- Devices can be powered from cable
- Up to 127 devices
- Hot-pluggable
- USB1.0 (1996, updated to v1.1 in 1998)
  - 1.5 Mb/s (low speed)
  - 12 Mb/s (full speed)
- USB2.0 (2000)
  - 480 Mb/s (high speed)
- USB3.0 (2008, 9 wires, different connector)
  - 5Gb/s (super-speed) (full duplex)
- USB3.1 (2013)
  - 10Gb/s
- USB3.2 (2017)
  - 10 Gb/s for USB-A connector
  - 20 Gb/s for USB-C connector

Realistic Transfer Rates

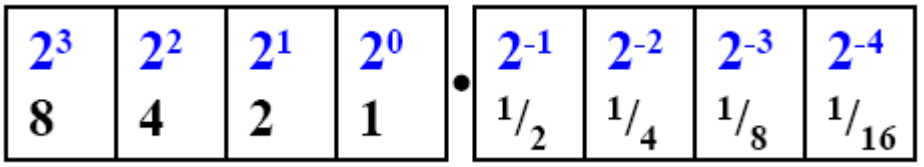
- Most buses can't achieve maximum transfer rates in real-life situations
  - Communication occurs in both directions
  - Buses are often shared between multiple devices



Floating Point Values

Tuesday, 26 October 2021 12:05 AM

Fractions - Fixed-Point

- How do we normally represent numbers?
  - We can arbitrarily specify the location of a binary point (fixed point notation), e.g.
  - 

However, not all numbers can be represented this way. What about $4 + \frac{1}{3}$

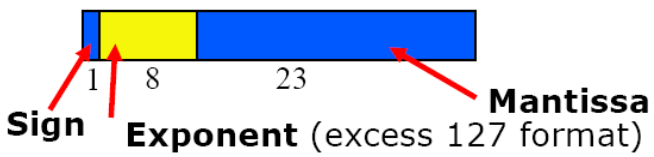
Real Numbers

Examples:

- Mass of an electron
- Mass of the sun
- Scientific or Floating Point Notation
  - Number = $x \times 10^y$
    - x fraction (or mantissa)
    - y exponent
- Same applies to binary
  - $2.0 \times 10^{33} = 1.1000101 \ldots \times 2^{1101110}$

IEEE Standard Floating Point

Single Precision (32 bits)



Double Precision (64 bits)

Exponent is excess 1023 format

- Mantissa always normalised (begins with 1)
- (so it doesn't actually need to be stored)

Floating Point Example

- [ ] Complete

Finite Precision

- Not every number can be represented precisely.
  - e.g. irrational numbers
    - 10/3
    - So (10.0/3.0) * 3.0 may not equal 10.0

IEEE Floating Point Number Characteristics

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	–126 to +127	–1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

IEEE Numerical Types

Normalized	±	0 < Exp < Max	Any bit pattern
Denormalized	±	0	Any nonzero bit pattern
Zero	±	0	0
Infinity	±	1 1 1 ... 1	0
Not a number	±	1 1 1 ... 1	Any nonzero bit pattern

Sign bit

Not a Number (NaNs)

- Used to represent the result of functions that have no solution
  - e.g. infinity divided by infinity, or

    - taking the square root of a negative number
- When NaN results from a computation, it propagates as a NaN through all subsequent operations.