THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING
DEPARTMENT OF ELECTRICAL, COMPUTER, AND SOFTWARE ENGINEERING

Name: _____

COMPSYS201 - 2019

Fundamentals of Computer Engineering

COURSEBOOK

It would appear that we have reached the limits of what it is possible to achieve with computer technology. - John Von Neumann, circa 1949

Coursebook Table of Contents

Coursebook Introduction

Lecture Material

Part I (with Oliver Sinnen)	**Slides**
- History and Trends	3-15
- Combinational Logic	
○ Design	16 - 47
○ Components	48 - 84
○ Design in VHDL	85 - 112
- Sequential Logic Design	
○ Memory elements	113 - 148
○ Counters & clock dividers	149 - 160
- Finite State Machine (FSM)	161 - 196

Part II (with Bill Collis)

- Introduction and C Review
- Macros and Bit Masking
- Polling
- Memories
- Timers and Counters
- Interrupts
- ADCs
- State Machines
- Fitness-for-Purpose

Coursebook Introduction

Computers are everywhere around us, and have come to dominate how we live our lives over the last couple of decades. Computers have enhanced the abilities of humans by allowing them to know more and know faster by mechanising any intellectual activity we can imagine. We are using computers to solve an ever-increasing range of problems, from industrial automation to computer vision to big data analysis to speech synthesis to clinical diagnosis and much more.

David Evans[1] says that there are two reasons why we should study computing:

1. Nearly all of the most exciting and important technologies, arts, and sciences of today and tomorrow are driven by computing.
2. Understanding computing illuminates deep insights and questions into the nature of our minds, our culture, and our universe.

In our context as engineers, we need to know what computers are and how they work so that we know what is and isn't possible. We are likely to use computers every day for the rest of our lives. A pilot has to know how a plane works so that they can make decisions about how to best operate it, and similarly it is critical for Electrical, Computer Systems, and Software Engineers to understand how computers work so that we can better utilise them in our projects. We also have to be able to talk to other engineers and speak a common language.

A computer is an electronic device that is capable of receiving information (data), performing computational operations on that information following a sequence of instructions, and produce a result. Another common way to describe this is **Inputs -> Processing -> Outputs**. This generally involves some form of **memory** so that the computer can remember what it is doing and store data somewhere. There are many types of computers that meet this definition, from *general purpose* computers like your laptop or desktop, to *embedded* computers like the ones hidden in your washing machine or car, to *super*computers that crunch massive amounts of data.

Computers are complex systems – they are made of many parts, and we will introduce you to some of those parts in this course. One of the key concepts of computer development is ***abstraction***, which means hiding away details. The idea is to work on a small part of the system, and make sure that small part works well, and then build on top of that with the knowledge that the small part of the system should work correctly. This allows different groups of people to work on different parts of the system, and to piece it all together afterwards and the system as a whole should work.

We can see that this is really useful when it comes to computers – remember that at the very lowest level, computers are just a bunch of transistors that can be switched on or off (and therefore equivalent to 1 or 0). If every software

[1] David Evans (University of Virginia) is the author of 'Introduction to Computing'

developer had to worry about how to configure every transistor and set them on or off, we would never be able to achieve large-scale software projects. In one sense, abstraction frees higher-level engineers to worry about higher-level problems; in another sense, having a good understanding of what is happening at the lower levels allows engineers to produce better performing, more optimised solutions that take full advantage of the resources available to them.

Modern computer systems follow a ***hierarchy*** that looks something like this:

In this course, we will start from the bottom and work our way up. How transistors work is already covered in ELECTENG101 and other courses like ELECTENG210 so we will start with logic gates and Boolean logic. At each level, we are abstracting away the levels below it, while understanding how those lower levels work. We will build our way up, and by the end of the course, we can write simple software with a good understanding of how that code becomes 1s and 0s and produces an intelligible useful result.

There are three main reference resources that are available to you as a student:

The first is the lecture slides, which are presented during lectures and available in this coursebook and on Canvas. We offer them as hardcopy so you can and should make notes during the lectures on them. We will also provide lecture recordings, but we cannot guarantee that recordings will be available for technical reasons. Additionally, they should not be treated as a substitute for attendance in class, especially because there may be some in-class demos that are not recorded.

The second is the recommended textbooks, which go in-depth into the topics taught in this course (as well as other topics not in this course). However, they often have too much detail, and use a lot of words. The recommended textbooks are mostly available from the library or short loan, so you shouldn't need to buy any of them.

The third reference is Canvas in general where you will find all other resources, including additional texts for reading, tutorial questions, lab sheets etc. These are also essential resources for this course and your learning.

Copyright Notice

Material which is clearly indicated as owned by the University of Auckland may only be used for "not-for-profit" educational purposes or private research and study in accordance with the Copyright Act 1994, provided that textual and graphical content are not altered and that the University's ownership of the material is acknowledged. The University reserves the right withdraw this permission at any time. Reposting of course materials online without permission of the copyright owner can result in copyright infringement. This can occur when students post material belonging to faculty staff onto websites which make a profit from onselling course information.

The permission to reproduce copyright protected material does not extend to any material on this site that is identified as being the copyright of a third party or individual staff members or students. Authorisation to reproduce such material must be obtained from the copyright owners concerned.

This coursebook utilises material from:
S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design* (McGraw Hill, Third Edition 2009).
F. Vahid, *Digital Design with RTL Design, VHDL, and Verilog* (John Wiley & Sons, Second Edition 2011).
COMPSYS201 Lecture Slides from previous years, Copyright Kevin Wang, Zoran Salcic and many others.
Some images from Wikimedia Commons, either in the public domain or under CC licenses.

COMPSYS201 – Part I
Fundamentals of Computer Engineering

History and Trends in Digital Systems

A/Prof Oliver Sinnen

Slides by Dr Kevin Wang

Some slides in this section are modified from:
Digital Design, with RTL Design, VHDL, and Verilog, 2nd edition, Copyright © 2010 Frank Vahid (http://www.ddvahid.com)
Fundamentals of Digital Logic with VHDL Design, 3/e, Copyright © McGraw Hill

Recommended Course Book

Digital Design

Some slides in this course have been modified from the slides to accompany the textbook *Digital Design, with RTL Design, VHDL, and Verilog*, 2nd Edition, by Frank Vahid, John Wiley and Sons Publishers, 2010.
http://www.ddvahid.com

Copyright © 2010 Frank Vahid

*Instructors of courses requiring Vahid's Digital Design textbook (published by John Wiley and Sons) have permission to modify and use these slides for customary course-related activities, subject to keeping this copyright notice in place and unmodified. These slides may be posted as unanimated pdf versions on publicly-accessible course websites.. PowerPoint source (or pdf with animations) may **not** be posted to publicly-accessible websites, but may be posted for students on internal protected sites or distributed directly to students by other electronic means. Instructors may make printouts of the slides available to students for a reasonable photocopying charge, without incurring royalties. Any other use requires explicit permission. Instructors may obtain PowerPoint source or obtain special use permissions from Wiley – see http://www.ddvahid.com for information.*

History of Computer I

- Pre-computer age: mechanical machine
  - Difference engine and analytical engine
  - Perform certain mathematical calculations/algorithms, not general purpose
  - Punch cards for controlling the machine
  - Process decimal values rather than binary values
    - Decimal is more intuitive to human than binary values

History of Computer II

- Harvard Mark I (1944)
  - Electro-mechanical, weighed almost 5 tons and had a clock of 66Hz
  - 0.3 seconds for an addition, 6 seconds for a multiplication, and 15 seconds for a division
  - Reads instructions from punched paper tapes
    - No conditional branches
  - Relatively more reliable than other electronic computers at that time
    - Only broke down once a week

ENIAC

- ENIAC (Electronic Numerical Integrator and Computer) (1946)
  - First completely electronic, general purpose calculator
  - Weighed 30 tons and consumed 200kW
  - 200 µs for addition, 6 ms for division
  - Used decimal arithmetic

Electronic Computers

- EDVAC (Electronic Discrete Variable Automatic Computer) (1949)
  - Weighed 8 tons and consumed 56kW
  - Used binary arithmetic
  - Stored program machine
- The concept of a stored program computer was introduced by John von Neumann in the late 40s
  - Instructions (and data) are stored in Memory to allow computers to perform tasks in sequence
- Dominant problem at that time was low reliability
  - MTBF (mean time between failures) often <20 mins
  - Unreliability was mainly caused by vacuum tubes and storage medium failures

Paper Memory

- Paper memory
  - Was once used as primary memory device
  - A program consists of a stack of punch cards
  - Later, punch cards and paper tapes were used as input/output storage media
  - Punch cards and paper tapes were not replaced until the 70s when electronic memory was introduced
  - Not re-writable, not random access (sequential access)

Practical Computer Systems

- Central Processing Unit (CPU), memories, input and output (I/O) devices
- Multiple buses are used to interconnect CPU and various peripherals, e.g. graphics card

Types of Modern Computer

- One categorisation of modern computers is based on the amount of resources available
- Each may have different internal architecture, but we will only be covering the fundamentals in this course

Supercomputers and Servers

- Supercomputers are the most powerful computers around, often designed for special applications, such as weather forecast and earthquake monitoring
  - Many millions of processor cores (https://www.top500.org/)
  - At UoA we have a supercomputer cluster called NeSI
- Servers nowadays typically refer to the cluster of computing resources sitting in a data centre offering so-called cloud services
  - Typically used for enterprise applications
  - Also capable of running general purpose applications

Desktop Computers

- DELL T3400 Workstation (Intel CPU + 8GB RAM + 1TB Hard Drive)

Hard Drive

CPU

(Underneath the heat sink)

Peripheral interfaces and buses

RAM

Image sources:

https://en.wikipedia.org/wiki/Xeon

https://en.wikipedia.org/wiki/Western_Digital

Single Board Computers

- Example: BeagleBone black

Image source: beagleboard.org

Microcontrollers

- Example: the Atmel Xplained Mini kit

Image source: blog.atmel.com

Embedded Systems

- The advancement of semiconductor technologies has lead to the miniaturisation of computing devices
  - Low power
  - Battery operated
  - Typically have wireless communication capability (e.g. Bluetooth)
  - Typically equipped with certain sensors (e.g. weather station, motion sensor, GPS)

Design abstraction

- Different engineers work at different abstraction levels
  - Processor designers work at Control Unit/Datapath and Processor level
  - Embedded system engineers work at OS level
  - Software engineers work at OS level or above
  - However, digital design basics are fundamental to all engineers/designers
- Different types of processor (including Virtual Machine) have different internal architectures
  - Code developed for different processors are not necessarily compatible with each other
  - Knowledge of the underlining architecture will help in developing more optimised code
- Increasing need to support cross-platform and portability
  - e.g. write code for app once, have it work on all the computers
  - e.g. Java runs on a "virtual machine"

COMPSYS201 – Part I
Fundamentals of Computer Engineering

Combinational Logic Design

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Introduction

- Boolean algebra
  - What is Boolean algebra?
  - Why do we use it?
  - How to use Boolean algebra to design/modify digital circuits
- Combinational logic circuits
- Representations of combinational logic
  - Boolean functions, truth tables, logic circuit diagrams
  - Conversion between representations
  - Standard representations
- Combinational design process: Translate from equation (or table) to circuit through well-defined steps

Modern Computer System Hierarchy

Software | Application

OS | Driver | Firmware

Processor | Memory | I/Os

Control Unit (FSM) and Datapath

Flip-flops and Processing Blocks

Logic Gates

Transistors

Boolean Algebra

- Traditional algebra
  - Variables represent real numbers
  - Operators operate on real numbers, return real numbers
- Boolean Algebra
  - Variables represent 0 or 1 only
  - Operators return 0 or 1 only
  - Basic Boolean operators
    - AND: a AND b returns 1 only when both a=1 and b=1
    - OR: a OR b returns 1 if either (or both) a=1 or b=1
    - NOT: NOT a returns the opposite of a (1 if a=0, 0 if a=1)

a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

a	NOT
0	1
1	0

Relating Boolean Algebra to Digital Design

Boolean algebra (mid-1800s) — *Boole's intent: formalise human thought*

Switches (1930s) — *For telephone switching and other electronic uses*

Shannon (1938) — *Showed application of Boolean algebra to design of switch-based circuits*

Digital design

x	F
0	1
1	0

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

- Implement Boolean operators using transistors
  - Call these implementations logic gates
  - These logic gates allow us to build digital circuits by doing math
- {AND, OR, NOT} is usually referred to as the basic gate set
  - All other gates (or Boolean functions) can be made from them

More Logic Gates

- NAND gate, anything but all 1s
  - S = (AB)′ = NOT (A AND B)

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

- NOR gate, detect all 0s
  - S = (A+B)′ = NOT (A OR B)

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

- XNOR gate, detect equality

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

- XOR gate, detect odd # of 1s
  - Useful for generating "parity" bit common for detecting data transmission errors

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

- Multiple inputs gate
  - Considered as cascaded two input gates

Functional Complete Gates

- A functionally complete gate set is a set that can be used to describe the behaviour of any other gate
  - For example, {AND, OR, NOT} collectively can build any Boolean function
- NAND and NOR gates are functionally complete gates
  - You can use either NAND or NOR gates *only* to build any Boolean function
- A simple proof: use NAND only to construct the {AND, OR, NOT} gate set (try the same approach with NOR)

NOT A

A	A	Q
0	0	1
1	1	0

NOT (A NAND B)

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

NOT(A) NAND NOT(B)

A	B	X	Y	Q
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

Combinational Circuits

- Combinational circuits
  - Outputs depend solely on the combination of the present input values into the circuit
  - In comparison to sequential circuits, which have "memory" that also impacts the outputs

(a) Combinational circuit (b) Combinational circuit (c) Sequential circuit

Design Combinational Circuits Using Gates

- Motion-in-dark example
  - Let's assign the lamp switch as F (output), motion sensor as input 'a', and light sensor as input 'b'
  - Turn on lamp (F=1) when there is motion sensed (a=1) and it is dark (i.e. no light sensed (b=0, or alternatively NOT(b)=1))
  - F = a AND NOT(b)
  - The above Boolean function represents the combinational circuit we want to design
  - Build using logic gates, AND and NOT
  - Important: output F is produced solely based on the current inputs a and b

Example: Seat Belt Warning Light

- Design a combinational circuit for a seat belt warning light with the following inputs and output
  - s=1: seat belt fastened (input)
  - k=1: key inserted (input)
  - w=1: turn on warning light (output)
- Capture Boolean equation
  - If seat belt is NOT fastened, AND key inserted, then turn on the warning light
  - w = NOT(s) AND k
- Convert equation to circuit
- Only illuminate warning light if person is seated (p=1), AND seat belt NOT fastened AND key inserted
  - w = p AND NOT(s) AND k

Boolean Algebra

- By designing combinational circuits using Boolean functions, we can use algebraic methods to manipulate circuits
- Notation: Writing a AND b, a OR b, NOT(a) is cumbersome
  - Use symbols: a * b (or just ab), a + b, and a' (or $\bar{a}$)
  - Original: w = (p AND NOT(s) AND k) OR t
  - New: w = ps'k + t
  - s' known as "complement of s" (i.e. inverted s)
  - "product" and "sum" are commonly used for AND and OR terms respectively

Boolean algebra precedence, highest precedence first:

Symbol	Name	Description
( )	Parentheses	Evaluate expressions nested in parentheses first
'	NOT	Evaluate from left to right
*	AND	Evaluate from left to right
+	OR	Evaluate from left to right

Boolean Algebra Terminology

- Example equation: F(a,b,c) = a'bc + abc' + ab + c
- Variable
  - Represents a value (0 or 1)
  - Three variables: a, b, and c
- Literal
  - Appearance of a variable, in true or complemented form
  - Nine literals: a', b, c, a, b, c', a, b, and c
- Product term
  - Product of literals
  - Four product terms: a'bc, abc', ab, c
- Sum-of-products
  - Equation written as OR of product terms only
  - Above equation is in sum-of-products form. "F = (a+b)c + d" is not
- Sum term and Product of Sums

Boolean Algebra Properties

- Commutative
  - a + b = b + a
  - a * b = b * a
- Distributive
  - a * (b + c) = a * b + a * c
    - Can write as: a(b+c) = ab + ac
  - a + (b * c) = (a + b) * (a + c)
    - (This second one is tricky!)
    - Can write as: a+(bc) = (a+b)(a+c)
- Associative
  - (a + b) + c = a + (b + c)
  - (a * b) * c = a * (b * c)
- Identity
  - 0 + a = a + 0 = a
  - 1 * a = a * 1 = a
- Complement
  - a + a' = 1
  - a * a' = 0
- Null elements
  - a + 1 = 1
  - a * 0 = 0
- Idempotent Law
  - a + a = a
  - a * a = a
- Involution Law
  - (a')' = a
- DeMorgan's Law
  - (a + b)' = a'b'
  - (ab)' = a' + b'
  - Very useful!

Complement of a Function

- Sometimes we might want to find the complement (inverse) of a function F
  - May be easier to understand, easier to design
  - Example: seat belt warning light is off if all seat belts are fastened
- De Morgan's Law: $(a + b)' = a'b'$ or $(ab)' = a' + b'$
- Note: De Morgan's Law can also be applied for more than two variables, e.g.:
  - $(a + b + c)' = a'b'c'$
  - $(abc)' = (a' + b' + c')$
- Complement of $f = w'xy + wx'y'z'$
  (De Morgan's law can be applied repeatedly)
  - $f' = (w'xy + wx'y'z')'$
    $f' = (w'xy)'(wx'y'z')'$ (by De Morgan's Law)
    $f' = (w+x'+y')(w'+x+y+z)$ (by De Morgan's Law)

Example of De Morgan's Law

De Morgan's Law: $(a + b)' = a'b'$ or $(ab)' = a' + b'$

- Behaviour description of an aircraft toilet sign
  - 3 toilets, each with sensor inputs (a, b, c), equals 1 if door is locked
  - Light "Available" output sign is on (S=1) if any toilets are available
- Boolean function
  - $S = a' + b' + c'$
- Transform
  - $(abc)' = a'+b'+c'$ (by De Morgan's Law)
  - $S = (abc)'$ (more efficient?)

- Alternative: Instead of designing an "Available" light, design an "Occupied" light
  - Opposite of "Available" function
  - $S' = (a' + b' + c')'$
  - $S' = (a')' * (b')' * (c')'$ (by De Morgan's Law)
  - $S' = a * b * c$ (by Involution Law)
  - Makes intuitive sense. Logically, occupied means all doors are locked

Representations of Combinational Logic

- A combinational circuit can be represented in different ways
  - Above shows seven representations of the same functions F(a,b), using four different methods: English, Boolean Equation, Circuit, and Truth Table

Converting among Representations

- Must be able to convert from any representation to another (six possibilities)
- Common conversions
  - Equation to circuit (we did this earlier)
  - Circuit to equation
    - Start at inputs, write expression of each gate output and combine

Truth Table Representation

- Define value of F for each and every possible combination of input values
  - 2-input function: 4 rows
  - 3-input function: 8 rows
  - 4-input function: 16 rows

a	b	F
0	0	
0	1	
1	0	
1	1	

(a)

a	b	c	F
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(b)

a	b	c	d	F
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

(c)

- Q: Use truth table to define function F(a,b,c) = 1 when "abc" is 5 or greater

a	b	c	F
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Converting among Representations

- Truth table to equation (which we can then convert to circuit)
  - OR each input term that should output 1
- Equation to truth table
  - Evaluate equation for each input combination (row)
  - Creating intermediate columns may help

Q: Convert to truth table:
F = a'b' + a'b

Inputs				Output
a	b	a' b'	a' b	F
0	0	1	0	
0	1	0	1	
1	0	0	0	
1	1	0	0	

Inputs		Outputs	Term
a	b	F	F = sum of
0	0	1	a'b'
0	1	1	a'b
1	0	0	
1	1	0	

F = a'b' + a'b

Q: Convert to equation

a	b	c	F	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	1	ab'c
1	1	0	1	abc'
1	1	1	1	abc

F = ab'c + abc' + abc

Converting from Truth Table to Equation

Example:

- Parity bit: Extra bit added to data, intended to enable detection of data transmission errors
  - e.g. errors can occur on wires due to electrical interference
- Even parity: Set a parity bit so the total number of 1s (data + parity) is even
  - e.g. if data is 001, parity bit is 1 → 0011
  - Want equation, but easiest to start from truth table for this example

a	b	c	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Convert to eqn. P = a'b'c + a'bc' + ab'c' + abc

Converting from Circuit to Truth Table

- No straight forward direct conversion
- First convert from circuit to equation, then equation to table

Inputs						Outputs
a	b	c	ab	(ab)'	c'	F
0	0	0	0	1	1	1
0	0	1	0	1	0	0
0	1	0	0	1	1	1
0	1	1	0	1	0	0
1	0	0	0	1	1	1
1	0	1	0	1	0	0
1	1	0	1	0	1	0
1	1	1	1	0	0	0

Standard Representation: Truth Table

- How can we determine if two functions are the same?
- We can always use algebraic method (i.e. Boolean algebra manipulation) to try and simplify/expand one function to the other
- For example:

  $F = A \cdot C' + B' \cdot C' + A \cdot C + B' \cdot C$

  $G = A' \cdot B' + A \cdot B + A \cdot B'$

  $F = A \cdot (C' + C) + B' \cdot (C' + C)$

  $= A + B'$

  $G = A' \cdot B' + A \cdot (B + B')$

  $= A' \cdot B' + A$ -> what if we stopped here

  $= A + B'$
- If we failed to derive the same equation for G and F, does that prove they are not equal? No!

- Solution: Convert to truth tables
- Standard representation—for a given function, only one version in standard form exists
  - Any given function can have *one and only one* truth table

Q: Determine if F1=ab+a' is same function as F2=a'b'+a'b+ab, by converting each to truth table first

F1 = ab + a'

a	b	F
0	0	
0	1	
1	0	
1	1	

F2 = a'b' + a'b + ab

a	b	F
0	0	
0	1	
1	0	
1	1	

Therefore, F1 = F2

Canonical Form – Sum of Minterms

- Truth table size grows exponentially with increasing number of inputs
  - Deriving truth table for equality comparison is not always feasible – it just gets too large!
- Use standard form of equation instead
  - Known as the canonical form
  - The canonical form is typically presented as sum of minterms
  - A minterm is a product term with every input literal appearing exactly once, in true or complemented form
  - First multiply-out the equation to sum-of-products form
  - Then expand each term until all terms are minterms

Q: Determine if F(a,b)=ab+a' is equivalent to F(a,b)=a'b'+a'b+ab, by converting the first equation to canonical form (the second already is)

F = ab+a' (already sum of products)
F = ab + a'(b+b') (expanding term)
F = ab + a'b + a'b' (Equivalent – same three terms as other equation)

Canonical Form – Sum of Minterms

- Q: Determine whether the functions G(a,b,c,d,e) = abcd + a'bcde and H(a,b,c,d,e) = abcde + abcde' + a'bcde + a'bcde(a' + c) are equivalent

G = abcd + a'bcde
G = abcd(e+e') + a'bcde
G = abcde + abcde' + a'bcde
G = a'bcde + abcde' + abcde

Equivalent

H = abcde + abcde' + a'bcde + a'bcde(a' + c)
H = abcde + abcde' + a'bcde + a'bcdea' + a'bcdec
H = abcde + abcde' + a'bcde + a'bcde + a'bcde
H = abcde + abcde' + a'bcde
H = a'bcde + abcde' + abcde

- <u>Homework</u>: prove these two functions are equivalent using truth tables

Compact Sum of Minterms Representation

- List each minterm as a number
- Number can be determined from the binary representation of the variable values
  - a'bcde corresponds to 01111, or 15
  - abcde' corresponds to 11110, or 30
  - abcde corresponds to 11111, or 31
- Thus, H = a'bcde + abcde' + abcde can be written as:
  - H = ∑m(15,30,31)
  - "H is the sum of minterms 15, 30, and 31"

Multiple-Output Circuits

- Many circuits have more than one output
- Can give each a separate circuit, or can share gates
  - Sharing gates can save precious logic resources
- Example: F = ab + c′, G = ab + bc

Option 1: Separate circuits Option 2: Shared gates

Multiple-Output Example

BCD to 7-Segment Converter

Multiple-Output Example

BCD to 7-Segment Converter

TABLE 2-4 4-bit binary number to seven-segment display truth table

w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

a = w′x′y′z′ + w′x′yz′ + w′x′yz + w′xy′z
+ w′xyz′ + w′xyz + wx′y′z′ + wx′y′z

b = w′x′y′z′ + w′x′y′z + w′x′yz′ + w′x′yz
+ w′xy′z′ + w′xyz + wx′y′z′ + wx′y′z

...

Combinational Logic Design Process

Steps	Description
Step 1: Capture behaviour	Create a truth table or equations, whichever is the most natural for the given problem, to describe the desired behaviour of each output of the combinational logic.
Step 2: Create Boolean functions (optional)	This step is only necessary if you captured the function using a truth table instead of equations. Create an equation for each output by ORing all the minterms equal to 1 for that output. Simplify the equations if desired.
Step 3: Implement as a gate-level logic circuit diagram	For each output, create a circuit corresponding to the output equation. Sharing gates among multiple outputs is (optionally) preferred.

Combinational Logic Design

- Example: Three 1s Pattern Detector
- Detect three consecutive 1s in an 8-bit input: abcdefgh
  - 00011101 → 1
  - 10101011 → 0
- Step 1: Capture the function
  - Truth table or equation?
    - Truth table too big: 2^8=256 rows
  - Equation: create terms for each possible case of three consecutive 1s
  - y = abc + bcd + cde + def + efg + fgh
- ~~Step 2: Create Boolean functions~~
- Step 3: Implement as a gate-level logic circuit

Combinational Logic Design

- Example: Number of 1s Counter
- Output in binary on two outputs yz the # of 1s on three inputs
  - 010 → 01 (a single 1)
  - 101 → 10 (two 1s)
  - 000 → 00 (No 1)
- Step 1: Capture the function
  - Truth table or equation?
  - Truth table is straightforward
- Step 2: Create Boolean functions
  - y = a'bc + ab'c + abc' + abc
  - z = a'b'c + a'bc' + ab'c' + abc
  - Optional: Let's simplify y:
    y = a'bc + ab'c + ab(c' + c)
    = a'bc + ab'c + ab
- Step 3: Implement as a gate-level logic circuit

Inputs			(# of 1s)	Outputs	
a	b	c		y	z
0	0	0	(0)	0	0
0	0	1	(1)	0	1
0	1	0	(1)	0	1
0	1	1	(2)	1	0
1	0	0	(1)	0	1
1	0	1	(2)	1	0
1	1	0	(2)	1	0
1	1	1	(3)	1	1

Learning Outcomes

- Combinational circuits
  - Circuit whose outputs are functions of the present inputs
- Boolean logic gates: AND, OR, NOT
  - Enables use of Boolean algebra to design circuits
  - Functionally complete gates
- Boolean algebra
- Representations of Combinational Logic
  - Boolean functions, truth tables, logic circuit diagrams
  - Conversion between representations
  - Standard representations
- Combinational design process: Translate from equation (or table) to circuit through well-defined steps

COMPSYS201 – Part I
Fundamentals of Computer Engineering

Combinational Logic Components

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Introduction

- Typical combinational logic components
  - Multiplexer
  - Decoder
  - Encoder/Priority encoder
- Arithmetic circuits
  - Adder
  - Comparator
- Negative number representation and subtractor
  - 2's complement representation
  - Subtractor
- Shifter
- Multiplier

Multiplexer (Mux)

- Multiplexer: Routes one of its N data inputs to its one output, based on binary value of select inputs
  - 4-to-1 mux → needs 2 select inputs (i.e. 2-bit) to indicate/index which one of the 4 data inputs to route to the 1 output
  - 8-to-1 mux → 3-bit select inputs, 8 data inputs, 1 data outputs
  - N select inputs (or select bits) →can index up to 2N data inputs
  - Data input/output can be of any width, but must be the same width
  - One of the most popular combinational building blocks

Mux Internal Design

- Multiplexer can be easily designed following the combinational circuit design process

2x1 mux: $f = s'w_0 + sw_1$

(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

(c) Circuit

4x1 mux:

$f = s_0's_1'w_0 + s_0's_1w_1 + s_0s_1'w_2 + s_0s_1w_3$

(a) Graphic symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

(c) Circuit

Mux Internal Design

- Large muxes can be difficult to design in the traditional way, why?
- In modern digital systems design, it is very common to design larger components using smaller components of the same type – modular design

Using 2-to-1 muxes to build a 4-to-1 mux

Using 4-to-1 muxes to build a 16-to-1 mux

N-bit Mux

- Example: selecting two 4-bit inputs, A (a3, a2, a1, a0), and B (b3, b2, b1, b0)
  - A 4-bit 2x1 mux can select between two 4-bit inputs A or B
  - Internally, 4 separate 1-bit 2x1 muxes share a common select input s0
  - N-bit Mux means the width of data inputs and output are N-bit
  - Regardless of the width of data I/O, two data inputs -> 2x1 mux

Buses

- N-bit bus: N wires to carry N-bits of data
  - Circuit drawings can become cluttered
- Convention for drawing buses
  - Single bold line with small angled line across
  - The number represents the width of the bus

Decoders

- A decoder takes a coded input data (typically smaller in size) and maps it to an output with a different coding
- A binary decoder maps a binary encoded input into the so-called "one-hot" encoded output
- In one-hot encoding, each output bit has its corresponding input binary value
  - 2-bit binary inputs represent up to 4 binary values, which requires 4 output bits in one-hot encoding ->2-to-4 binary decoder
  - 3-bit binary inputs represent up to 8 binary values, which requires 8 output bits in one-hot encoding -> 3-to-8 binary decoder
- An enable signal (EN) means the output is only valid when the enable is on
  - The X notation is to indicate "don't care" (i.e. it doesn't matter what the values of w0 and w1 are, because if En is 0 then the output is invalid)

En	w_0	w_1	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	X	X	0	0	0	0

Decoders

- Decoders can be designed using the combinational circuit design process, or modular design approach using smaller decoder components

A 2-to-4 binary decoder circuit

A 3-to-8 decoder using two 2-to-4 decoders

Encoders

- An encoder is the opposite of a decoder – maps a coded input data (typically larger in size) to an output with a different coding
- A binary encoder converts a one-hot encoded input into a binary encoded output
- One-hot encoding assumes one and only one bit is set
  - What happens if there are multiple bits set? Conflicting outputs
  - Can this happen in practice? Yes
  - If it does happen, how to resolve this? Use a priority encoder

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Priority Encoders

- To prevent the problem where multiple input bits are set in an encoder, a special type of encoder called a priority encoder is designed
- The encoder outputs (via truth table) indicate the active input that has the highest priority
- When an input with a higher priority is asserted, the other inputs with the lower priorities are ignored

w_3	w_2	w_1	w_0	y_1	y_0	valid
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Assume w_0 has the lowest priority and w_3 the highest

Intuitive Method to Design an Adder: Imitate Adding by Hand

- Intuitive adder design: mimic how people do addition by hand
- One column at a time
  - Compute sum
  - Send carry to next column

Intuitive Method to Design an Adder: Imitate Adding by Hand

- Create component for each column
  - Adds that column's bits, generates sum and carry bits

Half-Adder

- Half-adder - adds 2 bits, generates sum and carry
- Design using combinational design process

Step 1: Capture the function

Inputs		Outputs	
a	b	co	s
0	0		
0	1		
1	0		
1	1		

Step 2: Create equations

co = ab

s = a'b + ab' (same as s = a xor b)

Step 3: Implement as circuit

Full-Adder

- Full-adder: adds 3 bits, produces sum & carry
- Design using combinational design process

Step 1: Capture the function

Inputs			Outputs	
a	b	ci	co	s
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Step 2: Create equations

co = a'bc + ab'c + abc' + abc

co = a'bc +abc +ab'c +abc +abc' +abc

co = (a'+a)bc + (b'+b)ac + (c'+c)ab

co = bc + ac + ab

s = a'b'c + a'bc' + ab'c' + abc

s = a'(b'c + bc') + a(b'c' + bc)

s = a'(b xor c)' + a(b xnor c)

s = a xor b xor c

Step 3: Implement as circuit

Carry-Ripple Adder

- By cascading half-adder and full-adders, we can build an adder that adds like we would by hand
- Called a carry-ripple adder
  - 4-bit adder: adds two 4-bit numbers, generates 5-bit output
  - 5-bit output can be a 4-bit "sum" plus 1-bit "carry out"
  - Can easily build any an adder of any size

(a)

(b)

Carry-Ripple Adder's Behaviour

Assume all inputs initially 0

0111+0001
(answer should be 01000)

Output after 2ns (1 FA delay)

Wrong answer? – No, it just needs more time for carry to ripple through the chain of full adders (this is why it is called carry ripple adder)

Carry-Ripple Adder's Behaviour

0111+0001
(answer should be 01000)

Outputs after 4ns (2 FA delays)

Outputs after 6ns (3 FA delays)

Output after 8ns (4 FA delays)

Correct answer appears after 4 FA delays

Cascading Adders

- Using full-adder instead of half-adder for first bit, we can include a "carry in" bit in the addition
  - Smaller adders with carry in can be connected to form bigger adders

Adder Example: DIP-Switch-Based Adding Calculator

- Example: create a calculator that adds two 8-bit binary numbers, specified using DIP switches
  - DIP switch: Dual-inline package switch, move each switch up or down to represent 0 or 1 (or vice versa)

Equality Comparator

- N-bit equality comparator: Outputs 1 if two N-bit binary numbers are equal
  - Two binary numbers are equal if and only if every corresponding bit pair is equal
- Example: 4-bit equality comparator with inputs A and B
  - a3 = b3, a2 = b2, a1 = b1, a0 = b0
  - Two bits are equal if both 1, or both 0
  - eq = (a3b3+a3'b3') * (a2b2+a2'b2') * (a1b1+a1'b1') * (a0b0+a0'b0')
  - Note that function inside parentheses is XNOR
  - eq = (a3 xnor b3) * (a2 xnor b2) * (a1 xnor b1) * (a0 xnor b0)

0110 = 0111 ?

Representing Negative Numbers

- In fact, before designing subtractor, we should discuss negative numbers
- How to represent negative numbers in binary
- Signed-magnitude
  - Use leftmost bit for sign bit
  - So -5 would be:
    - 1101 using four bits or 10000101 using eight bits
    - What is 0000 and 1000?
- Two's complement
  - A complement with respect to 2^N for an N-bit number
    - e.g. 001 + 111 = 1000, for a 3-bit number 001
    - Its complement with respect to 2^3 (i.e. 1000) is 111
  - This also leads to the leftmost bit representing the sign bit
  - It avoids multiple forms of 0
  - It allows us to perform subtraction using addition

Ten's Complement

- Before introducing two's complement, let's consider ten's complement
  - Ten's complement is the number that, when added, results in 10
  - i.e. x + y = 10, so if x is known, then y = 10 - x
- Nice feature of ten's complement
  - Instead of subtracting a number, adding its complement results in an answer that is exactly 10 too much
  - So just drop the MSB 1 – results in subtracting using addition only

Adding the complement results in an answer that is exactly 10 too much – dropping the tens column gives the right answer.

Two's Complement is Easy to Compute: Just Invert Bits and Add 1

- Hold on!
  - Sure, adding the ten's complement achieves subtraction using addition only
  - But don't we have to perform *subtraction* to determine the complement in the first place? e.g. we only know that the complement of 4 is 6 by subtracting 10-4=6 in the first place.
- True, but in binary, it turns out that the two's complement can be computed easily
  - Two's complement of 011 is 101, because 011 + 101 is 1000
  - Easier method: Invert all the bits, and then add 1
  - The complement of 011 is 100+1 = 101. It works!

Q: What is the two's complement of 0101? A: 1010+1=1011 (check: 0101+1011=10000)

Q: What is the two's complement of 0011? A: 1100+1=1101

Two's Complement

- How two's complement represents negative numbers
  - Suppose we have a 4-bit number
  - Positive numbers are 0 to 7: 0000 to 0111, exactly the same as unsigned binary
  - Negative numbers
    - -1: Take two's complement of 1
      0001 → 1110+1 = 1111
    - -2: 0010 → 1101+1 = 1110 …
    - -8: 1000 → 0111+1 = 1000
    - So -1 to -8: 1111 to 1000
  - Leftmost bit still indicates the sign of number, known as sign bit. 1 means negative
- Value range of signed and unsigned N-bit numbers
  - Unsigned: 0 to $(2^N)-1$
    - e.g. Unsigned 8-bit: 0 to 255
  - Signed (2's complement): -2^{N-1} to $(2^{N-1})-1$
    - e.g. Signed 8-bit: -128 to 127

Two's Complement

- While it is very efficient for computation, the mapping between decimal values and two's complement binary numbers is not intuitive
  - e.g. 1000 is -8
- How to find the decimal value of a binary number in two's complement representation
  - Sign bit value indicates the sign
  - Positive numbers in two's complement are exactly the same as unsigned binary representation
    - e.g. 0001 is 1, 0010 is 2 and so on
  - Magnitude can be found by taking two's complement again
    - e.g. 1110 → sign bit is 1, so it is a negative number
      → = -(0001+1)
      → = -0010 = -2
    - e.g. 1000 → sign bit is 1, so it is a negative number
      → = -(0111+1)
      → = -1000 = -8

Adder/Subtractor with Two's Complement

- With two's complement, a subtractor can be built using an adder, by inverting B's bits, and setting carry in to 1
  - A – B = A + (-B)
  - = A + (two's complement of B)
  - = A + invert_bits(B) + 1

- Adder/subtractor: control input (sub) determines whether add or subtract
  - Use 2x1 mux – sub input decides to pass either B or B'
  - Alternatively, use XOR gates – if sub input is 0, B's bits pass through; if sub input is 1, XOR inverts B's bits

Adder/Subtractor Example: Calculator

- Without two's complement, we need to use a separate adder and subtractor
- Improved by using adder/subtractor, and two's complement numbers

Overflow

- Depending on the type of representation, a different range of numbers can be represented
- In arithmetic computation, sometimes the result can't be represented with given number of bits
  - Also happens to unsigned representation (i.e. carry)!
  - In two's complement arithmetic, this is called overflow, i.e. the result is either too large a magnitude of positive or negative
  - e.g. 4-bit two's complement addition of 0111+0001 (7+1=8). But 4-bit two's complement can't represent number >7
    - 0111+0001 = 1000 is WRONG answer, 1000 in two's complement is -8, not +8
  - Adder/subtractor should indicate when overflow has occurred, so result can be discarded

Detecting Overflow: Method 1

- For two's complement numbers, overflow occurs when the two numbers' sign bits are the same but differ from the result's sign bit
  - If the two input numbers' sign bits are initially different, overflow is impossible
  - Adding positive and negative can't exceed largest magnitude positive or negative
- Simple overflow detection circuit for 4-bit adder
  - overflow = a3'b3's3 + a3b3s3'
  - Include "overflow" output bit on adder/subtractor

sign bits

```
 0 1 1 1      1 1 1 1      1 0 0 0
+0 0 0 1     +1 0 0 0     +0 1 1 1
---------    ---------    ---------
1 0 0 0      0 1 1 1      1 1 1 1
overflow     overflow     no overflow
(a)          (b)          (c)
```

If the numbers' sign bits have the same value, which differs from the result's sign bit, overflow has occurred.

Detecting Overflow: Method 2

- Even simpler method: Detect difference between carry-in to sign bit and carry-out from sign bit
- Yields simpler circuit: overflow = c3 xor c4

```
  1 1 1        0 0 0        0 0 0
  0 1 1 1      1 1 1 1      1 0 0 0
+ 0 0 0 1    + 1 0 0 0    + 0 1 1 1
---------    ---------    ---------
0 1 0 0 0    1 0 1 1 1    0 1 1 1 1
overflow     overflow     no overflow
(a)          (b)          (c)
```

If the carry into the sign bit column differs from the carry out of that column, overflow has occurred.

Shifters

- Shifting (e.g. left shifting 0011 yields 0110) is useful for:
  - Bit manipulation
  - Multiply/divide by 2
    - Shift left once is same as multiplying by 2
      e.g. 0011 (3) becomes 0110 (6)
    - Why? Essentially appending a 0
    - Multiplying a decimal number by 10 is the same as appending a 0, i.e. by shifting left 55 becomes 550
    - Shift right once is the same as dividing by 2

Shifter Example: Temperature Averager

- Four input temperatures
- Want to output the average of those temperatures
- Add
  - Use three adders
- Then divide by four
  - Same as shift right by 2

Multiplier – Array Style

- Can build multiplier that mimics multiplication by hand
  - Notice that multiplying multiplicand by 1 is same as ANDing with 1

```
    0110   (the top number is called the multiplicand)
    0011   (the bottom number is called the multiplier)
    ----   (each row below is called a partial product)
    0110   (because the rightmost bit of the multiplier is 1, and 0110*1=0110)
   0110    (because the second bit of the multiplier is 1, and 0110*1=0110)
  0000     (because the third bit of the multiplier is 0, and 0110*0=0000)
 +0000     (because the leftmost bit of the multiplier is 0, and 0110*0=0000)
 --------
 00010010  (the product is the sum of all the partial products: 18, which is 6*3)
```

- Generalised representation of multiplication by hand

```
                   a3   a2   a1   a0
                 x b3   b2   b1   b0
-----------------------------------------
                 b0a3 b0a2 b0a1 b0a0   (pp1)
            b1a3 b1a2 b1a1 b1a0   0    (pp2)
       b2a3 b2a2 b2a1 b2a0   0    0    (pp3)
+ b3a3 b3a2 b3a1 b3a0   0    0    0    (pp4)
-----------------------------------------
p7 p6   p5   p4   p3   p2   p1   p0
```


Multiplier – Array Style

- Multiplier design – array of AND gates

Arithmetic Logic Unit

- One of the most common combinational logic components in modern computing device that can do
  - Arithmetic (add, subtract etc.)
  - Logic operations (AND, OR, NOT etc.)
  - Shift operations

  - The available operations depend on individual ALU design
  - For example:

S_1	S_0	Operation
0	0	F = A + B
0	1	F = A – B
1	0	F = A AND B
1	1	F = A OR B

Learning Outcomes

- Mux, decoder, encoder
- Adder
  - Half/full adder
  - Carry-ripple adder
  - Incrementer
- Equality comparator
- 2's complement signed representation for negative values
  - Subtraction using 2's complement
  - Overflow for signed arithmetic in 2's complement
- Multiplier
- Arithmetic Logic Unit (ALU)

COMPSYS201 – Part I
Fundamentals of Computer Engineering

Combinational Logic Design in VHDL

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

VHDL basics

- HDL is short for Hardware Description Language
- VHDL = VHSIC Hardware Description Language
- VHSIC = Very High Speed Integrated Circuit
- HDLs are formal languages for describing digital circuits
  - Specification (formal description-documentation)
  - Simulation (description of functionality that can be checked)
  - Synthesis (production of real circuit in target technology)
- VHDL is NOT case sensitive
  - databus, Databus, DataBus, DATABUS are ALL the same
- VHDL is strongly typed
  - Types must be defined by the developer. If you don't define the types or try to mix types, then the compiler will complain

VHDL basics - Naming and Labelling

General rules according to VHDL-87

- All names should start with an alphabet character (a-z or A-Z)
- Use only alphabet characters (a-z or A-Z) digits (0-9) and underscore (_)
- Do not use any punctuation or reserved characters within a name (!, ?, ., &, +, -, etc.)
- Do not use two or more consecutive underscore characters (__) within a name (e.g. Sel__A is invalid)
- All names and labels in a given ENTITY and ARCHITECTURE must be unique

VHDL basics - Comments

- Comments in VHDL are indicated with a double dash "--"
  - Comment indicator can be placed anywhere in the line
  - Any text that follows in the same line is treated as a comment
  - Carriage return terminates a comment
  - No method for commenting a block extending over multiple lines

Examples:

```
-- main subcircuit
Data_in <= Data_bus;    -- reading data from the input FIFO
```

- Rules for good practice:
  - Explain the function of a module to other designers
  - Explanatory, not just restatement of code
  - Locate close to code described

VHDL Designs

- A VHDL design consists of several separate design units, the two main design units you should know are:
  - ENTITY - describes a component's interface to the external world and represents the most basic building block in a design
  - ARCHITECTURE - describes a component's behaviour. A single ENTITY can have multiple ARCHITECTUREs

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY nand_gate is
    PORT(
        a    : IN STD_LOGIC;
        b    : IN STD_LOGIC;
        z    : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model of nand_gate
is
BEGIN
    z <= a NAND b;
END model;
```


Entity & Architecture Declaration

- Entity Declaration describes the interface of the component, i.e. input and output ports

- Architecture describes an implementation in a design ENTITY
- Architecture example for NAND gate:

```
ARCHITECTURE model of nand_gate is
BEGIN
    z <= a NAND b;
END model;
```

Entity & Architecture Declaration

- We can also illustrate the entity-architecture relationship as in the diagram below
- In VHDL the architecture can be expressed in a more algorithmic way

VHDL Basics - Objects and Data types

- Three types of objects:
  - Signals (model wires)
  - Variables (model memory elements)
  - Constants (symbolic names for clarity)
- Objects are differentiated by their names (or identifiers)
- Each object must be declared and have a data type (i.e. strongly typed)
- VHDL has the following predefined data types
  - bit ('0' or '1') & bit_vector
  - std_logic & std_logic_vector
  - integer
- Users can define their own data types
- Std_logic is the most common and can take the following values:

Value	Meaning
'X'	Unknown
'0'	Forced (strongly driven) 0
'1'	Forced (strongly driven) 1
'Z'	High Impedance
'W'	Weak (weakly driven) 0, modelling undriven
'L'	Weak (weakly driven) 0, modelling a pull-down
'H'	Weak (weakly driven) 1, modelling a pull-up
'-'	Don't care

VHDL Basics - Operators

Operator category	Operator symbol	Operation performed
Logical	AND OR NAND NOR XOR XNOR NOT	AND OR Not AND Not OR XOR Not XOR NOT
Relational	= /= >>

VHDL Example - Multiplexers

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (  w0, w1, s     : IN      STD_LOGIC ;
            f             : OUT     STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    WITH s SELECT
        f <=w0 WHEN '0',
            w1 WHEN OTHERS ;
END Behavior ;
```

VHDL code for a 2-to-1 multiplexer, using <u>WITH/SELECT</u> statement (or selected signal assignment)

VHDL Example - Multiplexers

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (   w0, w1, s     : IN      STD_LOGIC ;
             f             : OUT     STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    f <= w0 WHEN s = '0' ELSE w1 ;
END Behavior ;
```

VHDL code for a 2-to-1 multiplexer, using <u>WHEN/ELSE</u> statement (or conditional signal assignment)

VHDL Example - Multiplexers

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT (  w0, w1, w2, w3   : IN     STD_LOGIC ;
            s                : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            f                : OUT    STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Behavior OF mux4to1 IS
BEGIN
    WITH s SELECT
        f <= w0 WHEN "00",
             w1 WHEN "01",
             w2 WHEN "10",
             w3 WHEN OTHERS ;
END Behavior ;
```

VHDL code for a 4-to-1 mux, using <u>WHEN/ELSE</u> statement

VHDL Example - Multiplexers

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
PACKAGE mux4to1_package IS
    COMPONENT mux4to1
        PORT (  w0, w1, w2, w3   : IN     STD_LOGIC ;
                s                : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                f                : OUT    STD_LOGIC ) ;
    END COMPONENT ;
END mux4to1_package ;
```

VHDL code for a 4-to-1 multiplexer component package

VHDL Example - Multiplexers

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY work ;
USE work.mux4to1_package.all ;

ENTITY mux16to1 IS
    PORT (  w       : IN    STD_LOGIC_VECTOR(0 TO 15) ;
            s       : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            f       : OUT   STD_LOGIC ) ;
END mux16to1 ;

ARCHITECTURE Structure OF mux16to1 IS
    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
BEGIN
    Mux1: mux4to1 PORT MAP ( w(0), w(1), w(2), w(3), s(1 DOWNTO 0), m(0) ) ;
    Mux2: mux4to1 PORT MAP ( w(4), w(5), w(6), w(7), s(1 DOWNTO 0), m(1) ) ;
    Mux3: mux4to1 PORT MAP ( w(8), w(9), w(10), w(11), s(1 DOWNTO 0), m(2) ) ;
    Mux4: mux4to1 PORT MAP ( w(12), w(13), w(14), w(15), s(1 DOWNTO 0), m(3) ) ;
    Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
END Structure ;
```

Modular code for building a 16-to-1 mux using 4-to-1 muxes

VHDL Example - Decoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT (  w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            En  : IN    STD_LOGIC ;
            y   : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <= "1000" WHEN "100",
             "0100" WHEN "101",
             "0010" WHEN "110",
             "0001" WHEN "111",
             "0000" WHEN OTHERS ;
END Behavior ;
```

VHDL code for a 2-to-4 binary decoder

VHDL Example – Priority Encoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT (  w    : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            y    : OUT    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            z    : OUT    STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    y <=   "11" WHEN w(3) = '1' ELSE
           "10" WHEN w(2) = '1' ELSE
           "01" WHEN w(1) = '1' ELSE
           "00" ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

VHDL code for a priority encoder

VHDL Basics – Process Statement

- The process statement in VHDL
  - Is typically used to describe algorithmic processes in a sequential fashion, which is more intuitive to humans
  - Can be used to describe both combinational and sequential logic
- The execution of a process statement is determined by its sensitivity list
  - The process statement is only triggered if an event (i.e. change in value) in one of the signals in the sensitivity list is detected
    - "Change in value" depends on the potential values an object can take
  - In reality, the synthesiser tool will try and map such behaviour into a hardware component
- More discussion regarding process will come during the sequential circuit design section

VHDL Basics – Flow Control Statements

- There are four common flow control statements in VHDL:
- Concurrent signal assignment
  - Can only be used outside a *process* statement
  - WHEN/ELSE statement, also referred to as the conditional signal assignment
  - WITH/SELECT statement, also referred to as the selected signal assignment
- Sequential signal assignment
  - Can only be used within a *process* statement
  - IF/ELSIF/ELSE statement. A similar version of WHEN/ELSE but a sequential statement
  - CASE/WHEN statement. A similar version of WITH/SELECT but a sequential statement

VHDL Example – Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (  w0, w1, s  : IN     STD_LOGIC ;
            f          : OUT    STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        CASE s IS
            WHEN '0' =>
                f <= w0 ;
            WHEN OTHERS =>
                f <= w1 ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

VHDL code for a 2-to-1 Mux, using CASE/WHEN statement

VHDL Example – Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (  w0, w1, s  : IN     STD_LOGIC ;
            f          : OUT    STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        IF s = '0' THEN
            f <= w0 ;
        ELSE
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

VHDL code for a 2-to-1 Mux, using IF/ELSE statement

VHDL Example – Binary Decoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY dec2to4 IS
    PORT (  w    : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            En   : IN     STD_LOGIC ;
            y    : OUT    STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
    PROCESS ( w, En )
    BEGIN
        IF En = '1' THEN
            CASE w IS
                WHEN "00" =>          y <= "1000" ;
                WHEN "01" =>          y <= "0100" ;
                WHEN "10" =>          y <= "0010" ;
                WHEN OTHERS =>        y <= "0001" ;
            END CASE ;
        ELSE
            y <= "0000" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

VHDL code for a 2-to-4 binary decoder, using IF/ELSE and CASE/WHEN statements

VHDL Example – 7-Segment Decoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY seg7 IS
    PORT ( bcd    : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           leds   : OUT    STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;
ARCHITECTURE Behavior OF seg7 IS
BEGIN
    PROCESS ( bcd )
    BEGIN
        CASE bcd IS                          --      abcdefg
          WHEN "0000"        => leds <=   "1111110" ;
          WHEN "0001"        => leds <=   "0110000" ;
          WHEN "0010"        => leds <=   "1101101" ;
          WHEN "0011"        => leds <=   "1111001" ;
          WHEN "0100"        => leds <=   "0110011" ;
          WHEN "0101"        => leds <=   "1011011" ;
          WHEN "0110"        => leds <=   "1011111" ;
          WHEN "0111"        => leds <=   "1110000" ;
          WHEN "1000"        => leds <=   "1111111" ;
          WHEN "1001"        => leds <=   "1110011" ;
          WHEN OTHERS        => leds <=   "-------" ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```


VHDL Example – Priority Encoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY priority IS
    PORT (  w    : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            y    : OUT    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            z    : OUT    STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        IF w(3) = '1' THEN
            y <= "11" ;
        ELSIF w(2) = '1' THEN
            y <= "10" ;
        ELSIF w(1) = '1' THEN
            y <= "01" ;
        ELSE
            y <= "00" ;
        END IF ;
    END PROCESS ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

VHDL code for a priority encoder, using IF/ELSE statement

VHDL Example – Full Adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT (  Cin, x, y    :  IN     STD_LOGIC ;
            s, Cout      :  OUT    STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;
```

VHDL code for a 1-bit full adder (structural style)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT (  Cin, x, y   : IN     STD_LOGIC ;
                s, Cout     : OUT    STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

VHDL code for a 1-bit full adder package declaration

VHDL Example – 4-bit Carry-Ripple Adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
    PORT (  Cin     : IN     STD_LOGIC ;
            X, Y    : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            S       : OUT    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            Cout    : OUT    STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

VHDL code for a 4-bit carry-ripple adder

VHDL Example – 16-bit Adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
        PORT (  X, Y        : IN        STD_LOGIC_VECTOR(15 DOWNTO 0) ;
                S           : OUT       STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
        S <= X + Y ;
END Behavior ;
```

VHDL code for a 16-bit adder (behavioural style)

VHDL Example – ALU

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY alu IS
        PORT(
                A, B        : IN        STD_LOGIC_VECTOR(15 DOWNTO 0);
                S           : IN        STD_LOGIC_VECTOR(1 DOWNTO 0);
                F           : OUT       STD_LOGIC_VECTOR(15 DOWNTO 0));
END alu;

ARCHITECTURE behavior OF alu IS
BEGIN
        F <=    A + B when S = “00” else
                A – B when S = “01” else
                A AND B WHEN S = “10” else
                A OR B;
END behavior;
```

VHDL code for a 16-bit ALU

COMPSYS201 – Part I
Fundamentals of Computer Engineering

Sequential Logic Design – Memory elements

Introduction

- Recap on bit storage:
  - Latches and
  - Flip-flop
- Timing characteristics of flip-flop
- Combine flip-flops to build multi-bit storage – register
- Combining multiplexer with register to form different variant of registers
  - Shift register
  - Multifunction register
- Designing memory components using VHDL

Sequential Circuits

- Sequential circuit
  - Output depends not just on present inputs (as in combinational circuit), but also on past sequence of inputs
  - The past sequence of inputs will lead to the current system "state", which are stored in "memory bits"
  - Simple example: a circuit that counts up in binary, an alarm system

Must know sequence of past inputs to know output

Cannot determine the value of F solely from present input value

SR-Latch

- If S=1, R=0, then output Qa is set
  - Sometimes Qa is just referred to as Q, the non-inverted output
- If S=0 and R=1, then output Qb is set
  - Sometimes Qb is just referred to as Q', the inverted output
- If S=R=0, the circuit maintains its state
  - i.e. Qa and Qb will maintain their values
- S=R=1 is a forbidden state
  - Both outputs will be set to 0
  - What happens if the inputs go from S=R=1 to S=R=0?
    - Outputs go into oscillation state, or sometimes called metastable state

(a) Circuit

S	R	Q_a	Q_b	
0	0	0/1	1/0	(no change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Characteristic table

SR-Latch Timing Diagram

(a) Circuit

(c) Timing diagram

S	R	Q_a	Q_b	
0	0	0/1	1/0	(no change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Characteristic table

Level Sensitive SR-Latch

- To improve the control on loading time, a control signal "Clk" is introduced with two AND gates
  - If Clk=1, S and R inputs only propagate through to change S′ and R′
  - If Clk=0, S=R=S′=R′=0, which maintains the latch state
  - Inputs are sensitive to the level of control signal Clk, hence "level sensitive"
  - If we consider Clk as a periodic signal, it becomes a clock signal
  - S=R=1 is still forbidden
    - In fact this situation is made a little worse if S=R=1 and signal Clk goes from 1 to 0

(a) Circuit

Clk	S	R	$Q(t+1)$
0	x	x	$Q(t)$ (no change)
1	0	0	$Q(t)$ (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table

SR-Latch Timing Diagram

(a) Circuit

Clk	S	R	Q(t + 1)
0	x	x	Q(t) (no change)
1	0	0	Q(t) (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table

(c) Timing diagram

Level Sensitive D-Latch

- SR latch requires careful design to ensure SR=11 never occurs
  - Metastability should be avoided at all times
- D latch relieves designers from that burden
  - Inserted inverter ensures R is always the opposite of S
  - Still sensitive to the level of control signal C

D latch symbol

Problem with Level-Sensitive D Latch

- D latch still has a problem (as does SR latch)
  - When C=1, how many latches will a signal travel through?
  - Depends on how long C=1 is for and the internal propagation delay of each latch
    - Clk_A – signal may travel through many latches
    - Clk_B – signal may travel through fewer latches
    - The problem is, you have no control of how many…

Problem with Level-Sensitive D Latch

The signal can propagate from D1 and Q2 which is not desired behaviour

The signal is unable to travel from D1 to Q1 in time

D Flip-Flop

- For latches, we faced the issue of having no precise timing control on when we will load the memory
- Flip-flop: Bit storage/memory that stores on an active clock edge only (i.e. either rising or falling edge)
- One design – master-servant (or master-slave)
  - Clk = 0 – master enabled, loads D, appears at Qm. Servant disabled
  - Clk = 1 – master disabled, Qm stays same. Servant latch enabled, loads Qm, appears at Qs
  - Thus, value at D "appears" like it is only stored when Clk changes from 0 to 1 when we observe the output at Qs

D Flip-Flop

- Flip-flop solves the problem of not knowing how many latches a signal travels through when C=1
  - Signal travels through exactly one flip-flop, regardless of Clk signal width
  - Why? All four flip-flops are loaded simultaneously ONLY on the rising edge of Clk– then all four flip-flops stop loading also at the same time, until the next rising edge. Doesn't matter how long Clk is 1.

Symbol for rising-edge triggered D flip-flop

Internal design: Just invert servant clock rather than master

Symbol for falling edge-triggered D flip-flop, with an inversion bubble in front of edge-triggered clock input

D Latch vs. D Flip-Flop

- Latch is level-sensitive
  - Stores D when C=1 or C=0 (depending on the design)
- Flip-flop is edge triggered
  - Stores D when C changes 0 to 1 (rising edge) or 1 to 0 (falling edge)
- Comparing behaviour of latch and flip-flop:

Latch follows D while Clk is 1

Flip-flop only loads D during Clk rising edges

Clock Signal

- Flip-flop Clk inputs typically connect to the same clock signal (i.e. the inputs are *synchronised*)
  - Generated by an oscillator component that outputs a periodic signal
  - The example below shows a clock signal with "Period" = 20 ns, "Frequency" = 1/20 ns = 50 MHz
  - "Cycle" is the duration of 1 period (20 ns); example below shows 3.5 cycles
  - "Duty cycle" is the ratio of ON time for a signal. A 50% duty cycle clock signal mean the clock is ON for 50% of its period

Period/Freq shortcut:

Freq.	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	**1 ns**
100 MHz	10 ns
10 MHz	100 ns

Flip-flop Timing Parameters

- t_{su} (setup time) – the minimum time that the input D signal must be stable prior to the active edge of the Clock signal
- t_h (hold time) – the minimum time that the input D signal must remain stable after the active edge of the Clock signal
- t_{cQ} – the propagation delay for input signal D to appear at the output Q, usually given as a range
- Violating setup/hold time can lead to
  - Metastable state: Oscillating state other than stable 1 or 0
  - Eventually settles to either 1 or 0, but we don't know which

Service Call Button Using D Flip-Flop

- D flip-flop will store one bit
- Inputs are Call, Cancel, and present value of D flip-flop, Q
- Truth table shown below

Call	Cancel	Q	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Preserve value: if Q=0, make D=0; if Q=1, make D=1

Cancel -- make D=0

Call -- make D=1

Let's give priority to Call -- make D=1

Circuit derived from truth table, using the combinational logic design process

Basic Register

- While flip-flop is good, data is usually multiple bits
  - e.g. storing a 16-bit binary number or an 8-bit image pixel
- Register: multiple flip-flops sharing a common clock signal, i.e. all flip-flops are "synchronised"
  - From this point, we'll use registers for storage
  - But now you know what's inside a register

Example Using Registers: Temperature Display

- Temperature history display
  - Sensor outputs temperature as a 5-bit binary number
  - Clock signal C pulses every hour
  - Record temperature on each pulse, display last three recorded values

Note that registers are only loaded on rising clock edges (internally, just multiple flip-flops)

Basic Register with Load control

- N-bit register: Stores N bits, N is the width of the register
  - Storing data into register is called Loading
  - Opposite of storing is Reading (does not alter contents)
- Basic register: Loads every cycle (every active clock edge)
  - Can we design a register that only loads when we ask it to?
  - Internally, we can add a 2-to-1 Mux in front of each flip-flop to
    - Either accept external inputs (I3…I0, when load=1)
    - Or to maintain the current outputs (Q3…Q0, when load=0) through individual feedback loop
  - Note that we can omit the clock line – it is implied

Shift Register

- Shift right register
  - Move each bit one position right
  - Rightmost bit is "dropped"
  - Assume 0 shifted into leftmost bit

- Don't confuse with a shifter
  - A shifter is a combinational logic component that shifts its current 'inputs'. It has no memory capability
  - Shift register shifts individual flip-flop content to the subsequent stage flip-flop and remembers that

Q: Do four right shifts on 1001, showing value after each shift

A:

1001 (original)
0100
0010
0001
0000

- Simplest internal design: Connect one flip-flop output to next flip-flop's input

Shift Register

- Similar to the basic register, we would like to control when the shift register does shift. To allow register to either shift or retain, again use 2x1 muxes
  - shr: "0" means retain, "1" means shift right
  - shr_in: value to shift in

Multifunction Registers

- Registers may have multiple functions
  - Load, shift, clear (load all 0s), set (load all 1s)
  - And retain present value, of course
- We can easily design a multifunction register using muxes
  - Just connect each mux input to achieve desired functions

Function

s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	(unused - let's load 0s)

Register Design Example

Desired register operations

- Load, shift left, synchronous clear, synchronous set
- Unique Mux control inputs for each operation

s2	s1	s0	Operation
0	0	0	Maintain present value
0	0	1	Parallel load
0	1	0	Shift left
0	1	1	Synchronous clear
1	0	0	Synchronous set
1	0	1	Maintain present value
1	1	0	Maintain present value
1	1	1	Maintain present value

Step 1: Determine mux size

5 operations: above, plus maintain present value (don't forget this one!) → Use 8x1 mux

Step 2: Create register operation table

Step 3: Connect mux inputs for one flip-flop, then repeat for others

Step 4: Map control lines (optional)

s2 = clr'*set
s1 = clr'*set'*ld'*shl + clr
s0 = clr'*set'*ld + clr

Inputs				Outputs			
clr	set	ld	shl	s2	s1	s0	Operation
0	0	0	0	0	0	0	Maintain present value
0	0	0	1	0	1	0	Shift left
0	0	1	X	0	0	1	Parallel load
0	1	X	X	1	0	0	Set to all 1s
1	X	X	X	0	1	1	Clear to all 0s

Multifunction Registers with Separate Control Inputs

- Sometimes, instead of having unique Mux control inputs for each operation (difficult to remember), users may prefer separate control inputs for each function

ld	shr	shl	Operation
0	0	0	Maintain present value
0	0	1	Shift left
0	1	0	Shift right
0	1	1	Shift right – shr has priority over shl
1	0	0	Parallel load
1	0	1	Parallel load – ld has priority
1	1	0	Parallel load – ld has priority
1	1	1	Parallel load – ld has priority

s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	Shift left

Truth table for combinational circuit

Inputs			Outputs		Note
ld	shr	shl	s1	s0	Operation
0	0	0	0	0	Maintain value
0	0	1	1	1	Shift left
0	1	0	1	0	Shift right
0	1	1	1	0	Shift right
1	0	0	0	1	Parallel load
1	0	1	0	1	Parallel load
1	1	0	0	1	Parallel load
1	1	1	0	1	Parallel load

s1 = ld'*shr'*shl + ld'*shr*shl' + ld'*shr*shl

s0 = ld'*shr'*shl + ld

Register Operation Table

- Register operations are typically shown using a compact version of table (like a truth table)
  - X means don't care (i.e. does not matter whether value is 0 or 1)
  - Put highest priority control input on left to make reduced table simple -> How to decide priority?
  - Does the reduced table look familiar? ->Priority encoder

Inputs			Outputs		Note
ld	shr	shl	s1	s0	Operation
0	0	0	0	0	Maintain value
0	0	1	1	1	Shift left
0	1	0	1	0	Shift right
0	1	1	1	0	Shift right
1	0	0	0	1	Parallel load
1	0	1	0	1	Parallel load
1	1	0	0	1	Parallel load
1	1	1	0	1	Parallel load

ld	shr	shl	Operation
0	0	0	Maintain value
0	0	1	Shift left
0	1	X	Shift right
1	X	X	Parallel load

Register Design Example

Step 4: Map control lines

s2 = clr'*set
s1 = clr'*set'*ld'*shl + clr
s0 = clr'*set'*ld + clr

Inputs				Outputs			
clr	set	ld	shl	s2	s1	s0	Operation
0	0	0	0	0	0	0	Maintain present value
0	0	0	1	0	1	0	Shift left
0	0	1	X	0	0	1	Parallel load
0	1	X	X	1	0	0	Set to all 1s
1	X	X	X	0	1	1	Clear to all 0s

VHDL Basics – Process Statement

- The process statement in VHDL
  - Is typically used to describe algorithmic processes in a sequential fashion, which is more intuitive to human
  - Can be used to describe both combinational and sequential logic
- The execution of process statement is determined by its sensitivity list
  - The process statement is only triggered if an event (i.e. change in value) in one of the signals in the sensitivity list is detected
    - "Change in value" depends on the potential values an object can take
  - In reality, the synthesiser tool will try and map such behaviour into a hardware component
- Another typical use of the Process statement is to design memory elements of a sequential circuit
  - Different to combinational logic, sequential circuit Processes are usually sensitive to clock and/or reset signals ONLY
  - Memory elements MUST be implemented using Process statements

VHDL Basics – Signal v.s. Variable Assignment

- In VHDL, the first object we learned was *signal*, which models a physical wire (or connection)
- Another object type called *variable* is used within the Process statement, which in most cases model memory elements
- The distinct difference between signals and variables is during assignment
  - Signal assignment uses <= whereas variable assignment uses :=
  - Signal assignments within a Process are all updated concurrently at the same time at the end of the process
    - All assignments take the original values when entering the process
    - Order of assignments does not matter
  - Variable assignments within a Process are updated immediately after the statement
    - Subsequent assignments will take newly assigned variable value
    - Order of assignments does matter

Signal v.s. Variable Assignment Examples

```
PROCESS(clk)
VARIABLE F1, F2: STD_LOGIC;
BEGIN
    IF (clk'event and clk='1')
    THEN
        F1 := A OR B;
        F2 := C AND F1;
        Z <= A XOR F2;
    END IF;
END PROCESS;
END ARCHITECTURE behaviour;
```

```
PROCESS(clk)
SIGNAL F1, F2: STD_LOGIC;
BEGIN
    IF (clk'event and clk='1')
    THEN
        F1 <= A OR B;
        F2 <= C AND F1;
        Z <= A XOR F2;
    END IF;
END PROCESS;
END ARCHITECTURE behaviour;
```


Latch and Flip-Flop in VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
PORT (
   D, Clk: IN STD_LOGIC ;
   Q:      OUT STD_LOGIC) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
   PROCESS ( D, Clk )
   BEGIN
      IF Clk = '1' THEN
         Q <= D ;
      END IF ;
   END PROCESS ;
END Behavior ;
```

VHDL code for a D latch

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
PORT(
   D, Clock: IN STD_LOGIC ;
   Q:        OUT STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS ( Clock )
   BEGIN
      IF Clock'EVENT AND Clock = '1' THEN
         Q <= D ;
      END IF ;
   END PROCESS ;
END Behavior ;
```

VHDL code for a D flip-flop

Basic Register in VHDL


```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY reg8 IS
PORT(
   D:          IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- input
   reset, clk: IN STD_LOGIC; -- aync. clr. and clock
   Q:          OUT STD_LOGIC_VECTOR(7 DOWNTO 0) -- output
); END reg8;

ARCHITECTURE Behavior OF reg8 IS
BEGIN
   process(reset, clk)
   begin
      if reset = '0' then
         Q <= "00000000";
      elsif clk'event and clk='1' then
         Q <= D;
      end if;
   end process;
END Behavior;
```

Basic Register with Load control in VHDL


```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY reg8 IS
PORT(
   D:          IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- input
   load:       IN STD_LOGIC; -- load/enable
   reset, clk: IN STD_LOGIC; -- aync. clr. and clock
   Q:          OUT STD_LOGIC_VECTOR(7 DOWNTO 0) -- output
); END reg8;

ARCHITECTURE Behavior OF reg8 IS
BEGIN
   process(reset, clk)
   begin
      if reset = '0' then
         Q <= "00000000";
      elsif clk'event and clk='1' then
         if load = '1' then
            Q <= D;
         end if;
      end if;
   end process;
END Behavior;
```

Shift Register in VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity shift_reg is
port(
   clk:          in std_logic;
   load, ser_in: in std_logic; --load, serial_in
   Q:            out std_logic_vector(3 downto 0)
); end shift_reg;

architecture Behavior of shift_reg is
begin
   process
      variable temp: std_logic_vector(3 downto 0);
   begin
      if clk'event and clk='1' then
         if load='1' then
            temp := ser_in & temp(3 DOWNTO 1);
         end if;
      end if;
      Q <= temp;
   end process;
end Behavior;
```


Multifunction Register in VHDL

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

entity uni_reg_case IS
PORT(
   D             :in std_logic_vector(7 downto 0);
   ser_in, clk   :in std_logic;
   mux_sel       :in std_logic_vector(1 downto 0);
   Q             :out std_logic_vector(7 downto 0)
); end uni_reg_case;

architecture Behavior of uni_reg_case is
   variable tmp:std_logic_vector(7 downto 0);
begin
   process(clk)
   begin
      if (clk'event and clk = '1') then
         case mux_sel is
            when "00" => tmp := "00000000";
            when "01" => tmp := D;
            when "10" => tmp:=tmp(6 downto 0) & ser_in;
            when "11" => tmp:=ser_in & tmp(6 downto 0);
            when others => tmp := tmp;
         end case;
      end if;
   end process;
   Q <= tmp;
end Behavior;
```

Multifunction Register in VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity uni_reg is
port(
   D:       in std_logic_vector(7 downto 0);
   ser_in:  in std_logic;
   sl, sr, load, reset, clk: in std_logic;
   Q:        out std_logic_vector(7 downto 0)
); end uni_reg;

architecture Behavior of uni_reg is
   variable tmp: std_logic_vector(7 downto 0);
begin
   process(clk)
   begin
      if (clk'event and clk='1') then
         if reset='1' then
            tmp:="00000000";
         elsif load='1' then
            tmp:=D;
         elsif sl='1' then
            tmp:=tmp(6 downto 0) & ser_in;
         elsif sr='1' then
            tmp:=ser_in & tmp(6 downto 0);
         end if;
      end if;
   end process;
   Q<=tmp;
end Behavior;
```

VHDL code for a Multifunction Register with individual control lines

Memory Element Summary

- Flip-flops and formation of basic registers
- Timing characteristics and behaviour of flip-flops and registers
- Multiplexer and priority encoder and their use in forming different variants of registers
- Hierarchical composition of registers and multiplexers
- Designing memory components using VHDL

COMPSYS201 – Part I
Fundamentals of Computer Engineering

Sequential Logic Design – Counters and Timers

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Introduction

- Design counters using
  - Registers
  - Incrementer/decrementer
  - Mutiplexer
- Clock dividers
- Designing counters/timers/clock dividers in VHDL

Binary Counter

- N-bit binary up-counter can be designed by
  - An N-bit register, an incrementer, and an N-input AND gate to detect terminal count
  - Terminal (last) count, t_c equals 1 in the period the count value (i.e. the register content) is just before rollover
- Behaviour:
  - N-bit register that increments (add 1) its current value on each clock cycle
  - 0000, 0001, 0010, 0011, ..., 1110, 1111, 0000
  - Count "rolls over" from 1111 to 0000
  - In this case t_c=1 when the counter reaches 1111

Counter Example: Turnstile with Display

- Count people passing through a turnstile
  - Assume turnstile generates 1 pulse on P input per passing person
  - Connect the count output to a display
  - A reset button that can reset count
  - How many bits should we use for the counter?
- Simple solution using an up-counter

Up/Down Binary Counter

- Counter can count either up or down
  - Includes both incrementer and decrementer
  - Use *dir* input to select, via 2x1 mux: *dir*=0 means up
  - Likewise, *dir* selects appropriate terminal count value (all 1s or all 0s)

Counter with Parallel Load

- Up-counter that can be loaded with external value
  - Why? Sometimes we don't want to start counting from 0
  - Design using 2x1 mux. *ld* input selects the external value (*ld*=1) or the incremented counter value (*ld*=0) to load the internal register
  - What happens when both *ld*=1 and *cnt*=1? *ld* has priority over *cnt*

Counter with Parallel Load

- Example: a pulse generator that generates a pulse every 20 clock cycles
  - Use 5-bit up counter with parallel load
  - Set parallel load input to 12 (01100)
  - Use terminal count to trigger reload
    - When count reaches 31, next cycle loads 12
  - Counting from 12 (inclusive) to 31 gives 20 counts
  - What if we want to start counting from 0 but roll over at a different value other than 2^N-1 for an N-bit counter?
    - Trigger *clr* at a specific value (19)
  - What if we want to use a down counter?
    - Load 19 and trigger *ld* when reachin 0 (i.e. t_c=1)

Common Counter Use: Clock Divider

- Freq. of a physical oscillator is fixed
- If we have a 512 MHz oscillator, but want a 2 MHz clock
  - Need to divide the oscillator freq. by 256
  - Design using an 8-bit binary up-counter, and use tc output as pulse
    - Counts from 0 to 255 (256 counts), so t_c pulses every 256 cycles
    - Issue: the duty cycle of this clock is not 50%
    - Solution: use a D flip-flop. The output pulse (t_c) from the counter generates a rising edge for triggering the D flip-flop to toggle

Example: 1 Hz Pulse Generator from 60 Hz Clock

- Clock Division by something other than a power of 2
- Example: U.S. electricity standard uses 60 Hz signal
  - Device may convert that to 1 Hz signal to count time (seconds)
- Use a 6-bit up-counter
  - Up counter: Detected count of 59 clears counter to 0; t_c thus pulses every 60 cycles
    - Note: Detect 59, not 60 (because the first number, 0, is included)
  - Down counter approach also possible: when count reaches 0, load 59

Counter with Parallel Load in VHDL

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity counter is
port(
   clk:           in std_logic;
   ld, cnt, rst: in std_logic;
   counter_init: in std_logic_vector (3 downto 0);
   counter_out:  out std_logic_vector (3 downto 0)
); end counter;
```

```
architecture behaviour of counter is
begin
   process (clk)
      variable count_temp: std_logic_vector(3 downto 0);
   begin
      if (rising_edge(clk)) then
         if (rst='1') then
            count_temp:="0000";
         elsif (ld='1') then
            count_temp:=counter_init;
         elsif (cnt='1') then
            count_temp:=counter_temp+1;
            if (count_temp="1001") then
               count_temp:="0000";
           end if;
         else
            count_temp:=count_temp;
         end if;
      end if;
      counter_out<=count_temp;
   end process;
end behaviour;
```

Clock Divider in VHDL

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity clk_div is
port(
   clk:       in std_logic;
   clk_1hz:   out std_logic;
   clk_100hz: out std_logic
); end clk_div;
```

```
architecture behaviour of clk_div is
   signal clk100hz_count:   integer :=0;
   signal clk1hz_count:     integer :=0;
   signal clk1hz_temp:      std_logic;
   signal clk100hz_temp:    std_logic;
begin
   process (clk)
   begin
      if (rising_edge(clk)) then
         if (clk100hz_count>50) then
            clk100hz_count <= 0;
            clk1hz_count   <= clk1hz_count+1;
            clk100hz_temp  <= NOT clk100hz_temp;
         else
            clk100hz_count <= clk100hz_count+1;
         end if;

         if (clk1hz_count>100) then
            clk1hz_count <= 0;
            clk1hz_temp  <= NOT clk1hz_temp;
         end if;
      end if;
   end process;

   clk_1hz <= clk1hz_temp;
   clk_100hz <= clk100hz_temp;
end behaviour;
```


Learning Outcomes

- Design counters using
  - Registers
  - Incrementer/decrementer
  - Mutiplexer
- Clock dividers
- Design counters/timers/clock dividers using VHDL

COMPSYS201 – Part I
Fundamentals of Computer Engineering

Finite State Machine (FSM)

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Introduction

- So far we have introduced some simple sequential circuit components
  - Memory components
  - Counters, clock dividers
  - The design approach is largely intuitive based on integrating smaller components together
  - For larger and more complex circuits, we need a formal approach of designing a sequential circuit
- We will introduce the concept and design method of the Finite State Machine (FSM), which allows us to
  - Capture the behaviour of a sequential circuit
  - Design more sophisticated controllers for digital systems
- Design FSMs using VHDL

Mathematical Formalisms

- Two formalisms for capturing behaviours
  - Boolean equations for combinational circuit design
  - Finite State Machines (FSMs) for sequential circuit design
- Not required, but tremendously beneficial
  - Structured methodology
  - Correct-by-design circuits
  - Automated design, verification, and many more other advantages
- The mathematical model of a finite state machine is described by six elements:
  - The input alphabet (the set of input signals, Σ)
  - The output alphabet (the set of output signals, Γ)
  - The finite set of states (the possible combinations of state variables, S)
  - The initial state (where a state machine starts, S0)
  - The state transition function (how we move between states, δ)
  - The output function (the output values for each state, ω)
  - An FSM description is usually referred to as a sextuple $\{\Sigma, \Gamma, S, S_0, \delta, \omega\}$

General Form of a Sequential Circuit

- Sequential circuits: are essentially combinational circuits (eg: adder, mux) and memory elements (eg: register)
  - The memory elements store the "state" of the circuit
  - The combinational logic can be further separated into
    - Next state combinational logic, which implements the state transition function, determining the next state based on the current state and current input
    - Output combinational logic, which implements the output function that determines, determining the output based on the current state and optionally the current inputs

General Form of a Sequential Circuit

- Based on this general form, we can capture sequential circuit behaviour using two types of FSM models
  - Moore Model: Outputs depend on current state only
    - A FSM that fits the Moore Model definition is a Moore Machine
  - Mealy Model: Outputs depend on current state and inputs
    - A FSM that fits the Mealy Model definition is a Mealy Machine
  - Every Moore machine has a corresponding Mealy machine

Moore machine

Mealy machine

Graphical Representation of FSM

- A FSM is commonly represented in graphical form, which is referred to as the state transition diagram
- Let's take a look of a hypothetical example with:
  - Three states {SA, SB, SC}, a single bit input, A, and a single bit output F
  - Arrows indicate transitions between states
    - Conditions that lead to transitions are above/below the arrows
    - Each arrow implicitly takes one clock cycle, when the input conditions are satisfied the transition happens at the active clock edge
    - With the exception of the top left arrow, indicating SA is the initial state
  - 'X' represents "don't care", which means the value can be either 1 or 0
  - Output F, in this case only depends on the current state, so this diagram represents a Moore type FSM

Tabular Representation of FSM

- The same FSM can also be represented in a tabular form, called a state transition table (or simply state table)
  - Contains the same information as the state transition diagram and they are interchangeable
- In the state table, we can again observe that output F is only dependent on the current state

Current State	Input(s) A	Next State	Output(s) F
SA	X	SB	0
SB	0	SA	0
	1	SC	
SC	0	SA	1
	1	SB	

Mealy FSM

- For each Moore FSM, there should be a corresponding Mealy FSM
  - Sometimes one is more intuitive than the other
  - Outputs of a Mealy FSM depend on BOTH the current inputs and current states. Therefore the output is now associated on the transition arrows
  - The previous Moore FSM is shown here in its Mealy equivalent
  - It is obvious in the state transition table that output can be different even within the same state, caused by different inputs

Current State	Input(s) A	Next State	Output(s) F
SA	X	SB	0
SB	0	SA	1
	1	SC	0
SC	0	SA	1
	1	SB	0

Designing a Sequential Circuit using FSM

- The system description is usually given in plain English
- The following steps aim to develop a FSM based on a given description:
  - Step 1: List the states and the most obvious transitions
  - Step 2: Add input(s) and output(s) for those transitions and states
  - Step 3: Create/label more transitions for uncovered cases
  - Step 4: Check each state for full coverage of transtitions
  - Step 1 to 4 might take a couple of iterations. Once you become more familiar with the design process, these steps can be simplified

Moore FSM example

Example: Design an electronic lock that needs to open if a particular input pattern is detected. The lock has a single-bit input I, and assumes that every clock cycle a new bit arrives. Let's say that the unlock code is "1001". If we receive these bits in order, then we need to make the output F = 1 (unlock), otherwise, it should stay locked with F = 0 (lock)

- Obviously this is a sequential circuit, because we need to remember what was received previously, i.e. output is NOT based only on current input
- We know input is I and output is F and for Moore FSM output is associated with states
- Each transition is caused by a certain input condition to move from the current state to the next state at the active clock edge

Moore FSM example

- Step 1: List the states and the most obvious transitions
- Step 2: Add input(s) and output(s) for those transitions and states
- Since we are detecting the pattern "1001", four transitions are needed for these four consecutive input bits, so we need five states
  - The following state transition diagram only shows the case with correct pattern, which is incomplete

Moore FSM example

- Each state should have its unique meaning
  - State SA is the initial state, means nothing has been received, output is 0
  - State SB represents the case when a single '1' was received in the previous clock cycle, output is 0
  - State SC represents when the pattern "10" was received in the previous two consecutive clock cycles, output is 0
  - State SD represents when the pattern "100" was received in the previous three consecutive clock cycles, output is 0
  - State SE is when the correct unlock pattern was received in the previous four consecutive clock cycles, output is 1

Moore FSM example

- Now let's analyse the behaviour and add more transition arrows
- Step 3: Create and label more transitions for uncovered cases
  - Self loops: the transition goes back to the same state. There are two cases:
    - In the initial state if we keep receiving bit 0, then it is like nothing has been received.
    - In state SB, if '1' is received again, the unique meaning of SB still holds, i.e. a single '1' was received in the previous clock cycle
  - In state SC, if a '1' was received, it breaks the correct pattern. However, we did receive a single '1' in the previous clock cycle, so we transit back to state SB
  - In state SD, if a '0' was received, it breaks the correct pattern and it does not match with any other states' patterns, so it transits back to state SA

Moore FSM example

- For each state, there should be the right amount of transition arrows according to the number of input bits, for example
  - 1 bit input, will have at most two transition arrows (i.e. I=0 and I=1)
  - 2 bits input, will have at most four transition arrows (i.e. XY=00, 01, 10, 11)
  - So for each state, we need to analyse and make sure that all cases are covered
- Step 4: Refine the FSM to make sure all cases are covered
- In the case of a lock, we probably want the lock to reset after it is opened, i.e. considering the next bit as the beginning of a new sequence
  - So state SE will transit back to state SA if a '0' is received and to state SB if a '1' is received
  - Input sequence "1001001" will only generate F=1 once. NO sequence overlapping allowed

Alternative Moore FSM Example

- Step 4: Refine the FSM to make sure all cases are covered
- Alternatively, rather than reset the lock to start a new sequence, what if we want to continuously check for the correct pattern (overlapping)?
  - State SE will transit to different states according to the input value
  - If I=0, then state SE has received a pattern "10" in the previous two consecutive clock cycles, so it will transit to state SC
  - If I=1, then state SE has received a '1' in the previous clock cycle, so it will transit to state SB
  - Considering a input sequence "1001001", it will generate F=1 twice, because we allow sequence to overlap

Mealy FSM example

- Step 1: List the states and the most obvious transitions
- Step 2: Add input(s) and output(s) for those transitions and states
- In Mealy FSM, output is generated based on the current input and current state
  - When in state SD, i.e. we have already received "100" in the previous 3 consecutive clock cycles
  - If we receive another '1' in state SD, the correct pattern has been received, output F=1 can be generated
  - So we don't need one extra state SE for generate output F=1
  - For the same system, Mealy machine requires fewer states, operates faster and requires less resources compared to Moore machine

Mealy FSM example

- Step 3: Create and label more transitions for uncovered cases
- Step 4: Refine the FSM to make sure all cases are covered
- If we would like to detect non-overlapping sequence, i.e. once a pattern is detected, we restart to detect a new sequence
  - So in state SD, it does not matter whether the input I=0 or 1, we will transit back to SA to start a new sequence
  - However, output F is different according to input value

Alternative Mealy FSM Example

- Step 3: Create and label more transitions for uncovered cases
- Step 4: Refine the FSM to make sure all cases are covered
- If we allow overlapping sequence
  - Then in state SD, we will transit to SB if input I=1, because the received '1' is considered as the last bit of one sequence and the first bit of the following sequence
  - We will transit back to SA to start a new sequence if I=0

Common Mistakes when Capturing FSMs

1. Non-exclusive transitions: More than one valid transition for the same input combination
2. Incomplete transitions: Did not cover all the possible input cases
3. Stuck states: A state that has no way to leave

Sequential Circuit Design

- Now that we know how to model sequential circuit behaviour using FSM, we can design the sequential circuit (or sometimes referred to as a controller) using the following steps:
  1. Create the FSM (by drawing the state transition diagram) based on a given system description
  2. Set up the architecture:
     a. Determine the number of inputs and outputs required
     b. Encode the states with a unique value to each state (the values can be in different coding scheme, typically binary)
     c. Determine the required width of the state register based on the number of states and state encoding
  3. Convert the FSM diagram into a state table including state values
  4. Derive the Boolean equations, and implement combinational logic

Sequential Circuit Design: Electronic Lock Example

- Let's take the previous electronic lock example. We already have an FSM that captures the behaviour of the lock
- Step 1: Create the FSM (by drawing the state transition diagram)

- Step 2: Set up the architecture
  - We a single bit input {I} and a single bit output {F}
  - We have 4 states, so we need two bits state variables (assuming binary encoded states), which will be stored in a 2-bit register

Sequential Circuit Design: Electronic Lock Example

- Step 2: Set up the architecture
- We have 4 states, so we need two bits state variables (assuming binary encoded states)
  - The state will be stored in a 2-bit register
- Step 3: Convert the FSM diagram into a state table including encoded state values

Current State	Current Input	Next State	Output
S_1S_0	I	nS_1nS_0	F
00 (SA)	0	00 (SA)	0
	1	01 (SB)	0
01 (SB)	0	10 (SC)	0
	1	01 (SB)	0
10 (SC)	0	11 (SD)	0
	1	01 (SB)	0
11 (SD)	0	00 (SA)	0
	1	00 (SA)	1

Sequential Circuit Design: Electronic Lock Example

- Step 4: Derive the Boolean equations, and implement the combinational logic

$nS_1 = S_1'S_0A'+S_1S_0'A'$

$nS_0 = S_1'S_0'A+S_1'S_0A+S_1S_0'A'+S_1S_0'A$

$= S_1'A+S_1S_0'$

$F = S_1S_0A$ (Mealy)

Current State	Current Input	Next State	Output
S1S0	A	nS1nS0	F
00 (SA)	0	00 (SA)	0
	1	01 (SB)	0
01 (SB)	0	10 (SC)	0
	1	01 (SB)	0
10 (SC)	0	11 (SD)	0
	1	01 (SB)	0
11 (SD)	0	00 (SA)	0
	1	00 (SA)	1

Simplifying Notations

- Sometimes FSMs and logic circuit diagrams can become too complex and there are some common simplification that we can apply
- FSMs
  - Assume unassigned outputs are implicitly assigned with 0
- Sequential circuits
  - Assume unconnected clock inputs are connected to the same clock signal

State Minimisation

- So far we have been working with very simple FSMs
  - Few states and transitions
  - Relatively easy to derive the state transition diagram without much redundancy
- When an FSM becomes too complex, it is good to make sure there are no redundant states to ensure minimal resource usage and optimal performance
- If we have two states in an FSM that are equivalent to one another, then we have redundancy
- For two states to be equivalent, the following two conditions need to be satisfied:
  - Outputs MUST be the same for both states, *and*
  - Both states must transition to equivalent states for all input combinations

State Minimisation Example

- Let's have look at a relatively complex FSM in its state transition diagram form
- It is rather difficult to tell whether any states are redundant (i.e. satisfying the equivalent state conditions)

State Minimisation Example

- There are many approaches to ensure optimal FSM design, we are going to use a simple approach called row matching
- Let's convert the state transition diagram into a state table
  - With the state table, it is much easier to compare the output(s) and next state
- We can easily spot that the rows of current state D matches with rows of current state E, i.e.
  - They have identical outputs
  - They transition to equivalent next states in every possible input combination

Current State	Input(s)	Next State	Output(s)
A	L' R'	A	X
	L || R	B	
B	L' R'	D	Y
	L || R	B	
C	L' R	C	X, Y
	L' R'	D	
	L	B	
D	R	C	X, Z
	R'	E	
E	R	C	X, Z
	R'	E	

State Minimisation Example

- Since state E is equivalent to state D, we only need to keep one
- The row matching approach is iterative, so we need to look at the state table again
  - This time, no rows have identical outputs. We can conclude there is no more equivalent state
- In this example, we reduced from 5 to 4 states, so we will only need 2 bit state variables, rather than 3 bits

Current State	Input(s)	Next State	Output(s)
A	L' R'	A	X
	L || R	B	
B	L' R'	D	Y
	L || R	B	
C	L' R	C	X, Y
	L' R'	D	
	L	B	
D	R	C	X, Z
	R'	D	

Moore FSM in VHDL – 1 Process Style

Moore FSM in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY simple IS
PORT (
   Clock, Resetn, w: IN STD_LOGIC;
   z:                OUT STD_LOGIC);
END simple;

ARCHITECTURE Behavior OF simple IS
   TYPE State_type IS (A, B, C);
   SIGNAL y : State_type;
```

```
BEGIN
   PROCESS ( Resetn, Clock )
   BEGIN
      IF Resetn = '0' THEN
         y <= A ;
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         CASE y IS
         WHEN A =>
            IF w = '0' THEN
               y <= A;
            ELSE
               y <= B;
            END IF;
         WHEN B =>
            IF w = '0' THEN
               y <= A;
            ELSE
               y <= C;
            END IF;
         WHEN C =>
            IF w = '0' THEN
               y <= A;
            ELSE
               y <= C;
            END IF;
         END CASE;
      END IF;
   END PROCESS;

   z <= '1' WHEN y = C ELSE '0';
END Behavior;
```

Moore FSM – 2 Process Style

Moore FSM – 2 Process Style

```
ARCHITECTURE Behavior OF simple IS
   TYPE State_type IS (A, B, C);
   SIGNAL y_present, y_next: State_type;
BEGIN
   PROCESS ( w, y_present )
   BEGIN
      CASE y_present IS
      WHEN A =>
         IF w = '0' THEN
            y_next <= A;
         ELSE
            y_next <= B;
         END IF;
      WHEN B =>
         IF w = '0' THEN
            y_next <= A;
         ELSE
            y_next <= C;
         END IF;
      WHEN C =>
         IF w = '0' THEN
            y_next <= A;
         ELSE
            y_next <= C;
         END IF;
      END CASE;
   END PROCESS;
```

```
   PROCESS (Clock, Resetn)
   BEGIN
      IF Resetn = '0' THEN
         y_present <= A;
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         y_present <= y_next;
      END IF;
   END PROCESS;

   z <= '1' WHEN y_present = C ELSE '0';
END Behavior;
```

Mealy FSM in VHDL

Mealy FSM in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Mealy IS
PORT (
   clock, resetn, w: IN STD_LOGIC;
   z:                OUT STD_LOGIC);
END Mealy;
```

```
ARCHITECTURE Behavior OF Mealy IS
   TYPE State_type IS (A, B);
   SIGNAL y : State_type;
BEGIN
   PROCESS(resetn, clock)
   BEGIN
      IF resetn='0' THEN
         y <= A;
      ELSIF (clock'EVENT AND clock='1') THEN
         CASE y IS
            WHEN A =>
               IF w='0' THEN
                  y <= A;
               ELSE
                  y <= B;
               END IF;
            WHEN B =>
               IF w='0' THEN
                  y <= A;
               ELSE
                  y <= B;
               END IF;
         END CASE;
      END IF;
   END PROCESS;

   WITH y SELECT z <= w WHEN B,
                  z <= '0' WHEN others;
END Behavior;
```

Learning Outcomes

- Defined FSM model to capture sequential behavior
  - Using mathematical models – Boolean equations for combinational circuit, and FSMs for sequential circuits – is important
  - Moore and Mealy FSM
- Extend FSM to standard sequential circuit (or controller) design
- State minimisation
- Design FSMs in VHDL

Concluding Remarks – Digital Design Flow

Part-2 Microcontroller (uC) based embedded systems (ES)

Contents

2.1. Introduction ... 3
2.2. Learning outcomes: ... 4
2.3. Atmel and the Xplained mini ... 5
2.4. C review ... 7
2.5. LO: Understand interrelatedness of hardware and software in ES's ... 8
2.6. LO Understand the ES as an automaton ... 9
2.7. Bit masking theory ... 10
2.7.1. Macros ... 10
2.7.2. Hardware theory - how a uC pin can either be an i/p or an o/p ... 11
2.8. Describe the ES as reactive and responsive to its environment ... 13
2.9. Die Program (dice - plural, die - singular) ... 15
2.10. Polling ... 16
2.10.1. Hardware topic: Switch issues ... 17
2.10.2. A 'not-so-simple' debounce problem ... 19
2.11. AVR memory types ... 20
2.11.1. What type of number are you? ... 20
2.12. External Interrupts ... 23
2.12.1. Interrupt complications ... 27
2.13. Timer/Counters ... 29
2.13.1. AVR Timers ... 30
2.13.2. Timer are complex (allow multidimensional problems to be solved) ... 30
2.13.3. Timer Registers ... 31
2.13.4. Timer example – 'heart beat' led flash ... 33
2.13.5. The 'about' 1 second timer ... 34
2.13.6. The 'exact' 1 second timer ... 35
2.13.7. The 'exact' 1 second flash direct hardware output timer ... 36
2.13.8. Make a short duration pulse at a regular time interval ... 37
2.13.1. Use ICP1 to measure pulse width ... 39
2.14. ES development overview ... 47
2.14.1. Writing bug free code (hopefully) ... 48
2.14.2. LO: Understand the importance of transparent software practices ... 49
2.14.3. How we get from C to hex in ATMEL Studio 7. ... 51
2.14.3.1. The processor CPU and ALU ... 52
2.14.4. What about when the ES doesn't work quite right ... 53

2.14.5. Pointers and memory mapped IO 54
2.14.6. Volatile 55
2.15. ADC 56
2.15.1. The voltage divider 56
2.15.2. ES analog sensors 59
2.15.3. The full conversion process 60
2.15.4. Quantization error 62
2.15.5. AVR ADC inputs 63
2.15.6. ADC Channel select 64
2.15.7. Sampling rate 66
2.15.8. ADC modes 67
2.15.9. ADC resolution 68
2.15.10. ADC Example 1 69
2.16. State machine programming 72
2.16.1. Coding state machines in C 74
2.16.2. Breath tester state machine example 75
2.16.3. Electronic safe state machine 77
2.17. Serial Data Communications 79
2.17.1. Uses for serial and parallel data communications 79
2.17.2. Serial communication protocols 80
2.17.3. Protocol parameters 80
2.17.4. Communication channel types or modes 80
2.17.5. Transmission timing 81
2.17.6. Information transfer 81
2.17.7. ATMega328 internal communication hardware datasheet section 20 82
2.17.8. Serial communications example code 83
2.18. Fitness-for-purpose 85
2.19. Driving output devices 86
2.20. Useful resources 89

2.1.Introduction

Embedded systems are at the centre of the exponential increase of smart devices and IoT systems in the world around us. ES's are increasingly becoming a feature of this period of history such that it is now known as 'Industry 4.0' or IoT - the internet of things. Common ES's have at their heart microcontrollers (other digital controllers include FPGAs, ASICs, and microprocessors). ES's are ubiquitous (everywhere - but so much so that we are generally unaware of them) they automate, balance, maximise, minimise, regulate and control our everyday lives better than we can ourselves! A washing machine measures the dirtiness and weight of clothes and determines for us the amount of water and the length of the washing cycle, the output emissions of a car are monitored and the ES controls the ratio of fuel, air and exhaust gas injected into the engine making decisions on our behalf about fuel consumption, efficiency and pollution. Electricity controllers link to payment systems and switch our power on and off as we pay (or not) our power bills. Very soon our toilet roll holders will tweet us when the roll has run out!

www.designtaxi.com

This part of the course is an introduction to the programming of uC's and how as part of ES's they sense and control the environment. Because ES's come in so many varied forms, with so many varied features and have so many varied roles, finding a single definition is not easy. Instead a focus on the core characteristics of ES is needed: their automatic and reactive operation in real-time, the close interrelationship between hardware and software and the unique tools that engineers rely on to develop them. ES's are vital to all three specialisations within the department because you will undoubtedly come across the ubiquitous ES at several different stages in your careers and you must be aware of the important constraints that they have and their 'fit' within the purposes of the systems they are embedded within.

www.asbbank.co.nz

www.microchip.com

2.2.Learning outcomes:

Student learning goal: develop a viable mental model of a microcontroller based Embedded System
To achieve this goal you will need to begin to:

LO 1: understand the interrelatedness of hardware and software in ES's

- explain how program commands work that link hardware and software
- with reference to a datasheet, write code that manipulates/checks single bits in registers

LO 2: understand the ES as an automaton

- describe the common C code variants used to make an ES into an automaton
- discuss ES's with regard to unattended consistent and reliable operation
- explain how an ES can operate with the illusion of concurrency

LO 3: understand the ES as reactive and responsive to its environment

- explain polling and its limitations
- explain contact bounce issues with physical switches and software de-bounce code
- explain how internal uC timers are used to make an ES responsive and reactive
- setup a uC timer in software to make an uC responsive to an environmental event
- explain how uC external interrupts are used to make an ES reactive
- setup external uC interrupts in software
- describe the significant characteristics of an internal ADC
- describe the use of a sensor in a voltage divider circuit
- identify issues relating to dynamic range and quantization with analog sensors
- explain issues of an ES's responsiveness with regard to polling, blocking and interrupts
- describe a software state machine in terms of states, actions, conditions and transitions
- turn a state machine model into C code, using while / switch software pattern
- describe the serial protocol and its parameters used within the uC USART
- setup the uC USART registers

LO 4: understand the importance of transparent software practices for an ES

- apply a formal approach to writing program code through use of a template, macros and naming conventions for variables and functions that makes code transparent
- write comments in code that explain function rather than textual features

LO 5: understand how 'fitness-for-purpose' guides uC choice for an ES

- list specifications of an uC that would inform choosing a uC for an ES
- describe the different memories of the microcontroller and their uses
- explain uses of different variable types and their limitations
- describe implications of over/under flow of variables
- interface an LED to a uC, taking into account the DC characteristics of each

LO 6: understand the various roles of an IDE

- describe the pre-processor, compiler and program uploading processes
- compile and upload compiled code to a microcontroller and test it

LO 7: develop as an engineering student by working with complex (multidimensional) tasks

2.3.Atmel and the Xplained mini

Atmel (now owned by Microchip) is one of many brands of microcontrollers. Other brands include: Texas, Cypress, Renasas, ST, TI, National, NXP, Freescale…

- The Atmel brand includes several different ranges of uC's - AVR8, UC3(32 bit) and ARM
- The 8 bit AVR's includes several sub types: ATtiny, ATmega, ATxmega, AT90
- ATMEL uCs represent about _____ of the world market share of 8 bit uC's – in excess of _____ per year are sold at an average cost of around _____ each
- Each sub type has several variants e.g. ATtiny45, ATmega2560, _____
- When learning about uC's you will often start with a development board or kit.
- The Atmel Xplained Mini development board has an ATmega328P – the board costs < _____
- But you can get an equivalent Arduino Uno Board from aliexpress.com for about _____

www.atmel.com

www.wikimedia.org

www.zeptobars.com

www.arduino.cc

Figure 4-1. Block Diagram

ATMega328 Datasheet

The ATmega328P Xplained Mini (development board)

www.atmel.com

List specifications of a uC that would inform choosing a uC for an ES.

PORTS – collections of I/O (input/output) pins – places we can connect:

I/P devices: _____

O/P devices: _____

ATMega328P has 3 ports: PORT___ PORT___ PORT___

Ports each have _____ and each pin on a port is named _____ Because of the number of physical pins on the ATmega328P package and the way the Xplained mini board is made we do not have access to all 8 pins on each port

- e.g. PORT___ has: ____ ____ ____ ____ ____ ____ ____ ____
- e.g. PORT___ has: ____ ____ ____ ____ ____ ____ ____ ____
- e.g. PORT___ has: ____ ____ ____ ____ ____ ____ ____ ____

We can connect either input or output devices to a pin to sense or control the real world. An initial aspect of using a pin is to set up which direction it will be – input – to read an external input device or as an output to control an external output device.

ATmega328P data: http://www.microchip.com/wwwproducts/en/ATmega328P

All datasheet references will be to the complete datasheet, we will use version 11/2016

What is the difference between an FPGA and a uC (microcontroller)?

What is the difference between VHDL and C?

An FPGA is a collection of logic blocks (circuits) within a single package. VHDL is not written into an FPGA, VHDL is the language used to configure the arrangement of signals between the logic blocks and which and where I/O pins connect to the logic blocks; in this way VHDL tells an FPGA what it will _____**.** The name Field Programmable Gate Array describes its functionality quite well it is array of gates (logic blocks) which is programmable (changeable) in the field (away from the factory). Importantly it is possible to program the arrangement of logic within the FPGA so that different functions can operate concurrently and so an FPGA is able to do multiple tasks independently and in _____.

A uC is a single computer condensed into one chip (integrated circuit), it is a highly complex logic circuit whose hardware is unchangeable. The programs for a uC are sequences of instructions written in C (or other programming languages); these are compiled into machine code (and stored in .hex files). Machine code is written into the microcontroller's Flash (program memory). In this way programs tell the fixed hardware of a uC what it will _____. Importantly the different parts of a C program running in a uC operate _____**.**

2.4.C review

LO: Develop as an engineering student by working with complex (multidimensional) tasks

Take time to review the information below from the course notes from ENGGEN 131

page 5: the difference between programs that are compiled and interpreted
page 6: code blocks, array indexes and logical operators, the last paragraph
page 9: the diagram – we will compare this to the process used in ES development
page 10 preprocessor (#include, #define), compiler
page:11 the main() function
page 18: overflow
page 20: the 'warning'
page 21: Global and local variables
page 24: unary operators
page 25: special assignment operators and operator precedence
page 31: basics of arrays
page 32: Boolean expressions
page 33/34: 'if' selection statement
page 36: loops,
while(){}
for(){}
do{} while()
page 52: functions
page 53: void
page 54: return
page 55: calling a function
page 58: function prototypes
page 60: variable scope
page 63: parameters as local variables
page 69: pointers and parameter passing

2.5. LO: Understand interrelatedness of hardware and software in ES's

A core learning objective is to work fluidly between the dimensions of hardware and software.

Describe how program commands work that link hardware and software

```
/***** Project Header *****/
// Project Name: Cylon 8
/***** Hardware defines *****/
//make sure this matches your oscillator
#define F_CPU 16000000//crystal
/***** Includes *****/
#include <avr/io.h> //_____________________________
#include <util/delay.h>
/***** User macros *****/
#define TIME_DELAY 100 //___________________________
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRD = 0xff; //setup all 8 pins as outputs for the 8 LEDs
    /***** Loop code *****/
    while (1) {
        PORTD = 0b00000001;      //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00000011;      //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00000111;      //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00001110;      //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00011100;      //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00111000;      //
        _delay_ms(TIME_DELAY);
        PORTD = 0b01110000;      //
        _delay_ms(TIME_DELAY);
        PORTD = 0b11100000;      //
```

www.byyourcommand.net

```
    } //end while(1)
} //end of main
```

```
_delay_ms(TIME_DELAY);
```

If these lines of code were removed what would the Cylon look like?

Software that controls hardware

```
DDRD = 0xff;
```

make all 8 pins of portD outputs

```
PORTD = 0b00000001;
```

force bit 0 high ***AND*** all other bits low of register PORTD.

```
_delay_ms(TIME_DELAY);
```

uC's carry out instructions at rates much faster than environments can recognise or respond to, programs can make use of simple delays to make the ES meet the demands of the environment. This code blocks or pauses the execution of the program briefly so that the people have time to see the LED pattern.

2.6.LO Understand the ES as an automaton

A core ES understanding is seeing the ES as an automaton - an automatic machine that carries out instructions without human intervention. An ES begins running on power up, follows a set of predetermined instructions without human intervention and will do so until the power is removed. In contrast the software run on a PC is transactional in nature, it requires a human to decide which path the software will take.

The way an ES becomes an automaton is via this simple program operation

Explain the common C code used to make an ES into an automaton

```
int main(void) { //c programs begin executing at the main function

    while (1) {
        …
        …
        …
    }

}                //C programs finish at end of the main function
```

2.7. Bit masking theory

Binary numbers consist of groups of single bits (e.g. bytes = 8 bits). Usually single bits of a number need to be manipulated; this is done using understandings of **Boolean algebra and Boolean identities**

1. X **or** 1 = 1
2. X **or** 0 = X
3. X **and** 0 =0
4. X **and** 1 = X

Code to set 1 bit of a number or register

Code to clear 1 bit of a number or register

Code to check if 1 bit of number (or register) is set

Code to check if 1 bit of number (or register) is cleared

2.7.1. Macros

While the statements above work perfectly, for code transparency reasons we must not go 'littering' our code with statements like those above, as it makes code totally unreadable and almost impossible to debug. Learn early on to write macros at the top of your program that keep hardware specific detail in one place.

2.7.2. Hardware theory - how a uC pin can either be an i/p or an o/p

Describe how program commands work that link hardware and software

Each pin has three single bit registers or latches associated with it,

- PORTx.n e.g.
- PINx.n e.g.
- DDRx.n e.g.

tri-state buffer

C	In	Out
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

Hi-Z (like a tap)

How a µC pin works as an INPUT

Figure 18-2. General Digital I/O(1)

ATmega328 datasheet

The initial power-up state of all pins is as inputs- the DDR (data direction register) output is 0. This means the **tri-state buffer** has a high Z output (it's like a tap is off) and signals from the PORT register cannot go through to the pin. The signal (1 or 0) from the physical pin comes through to the PIN register. When the uC encounters a command to read the pin e.g if (SW_1_IS_LOW) {} or more precisely if (~PINB & (1<<PB3)) {}, a 1 is put on RPx allowing the tri-state buffer output to reflect the value of the hardware pin onto the databus.

This diagram from the datasheet represents
the logic for one pin of one port.
How many times is this block repeated in the ATmega328P?

How a µC pin works as an output

Figure 18-2. General Digital I/O(1)

ATmega328 datasheet

To make the pin an output requires the DDR to output a 1, this opens the tri-state buffer 'tap' allowing the value stored in the PORT register to go through to the physical pin. This is the effect of the command DDRA |= (1<<A5). Or we could make all 8 bits of a port outputs all at once by writing DDRA=0xFF . At any stage in our program we can drive the physical pin high or low using the commands PORTA |= (1<<A5) or PORTA &= ~(1<<A5). The 1 or 0 is stored in PORTA.5 and because the tri-state buffer is not hi-Z (tap is open) it goes through to the pin.

A special effect of this hardware is that the PIN register always reflects the state of the hardware pin even when the port is setup as an output. So you can check the state of an output device at any stage in your program by reading the PIN register - can you see how this could be used in the quiz game controller?

<u>Confusing terminology alert - pin & port</u>: When we say 'port' we are generally referring to a collection of 8 physical I/O pins not a PORT register. When we say 'D3' we are referring to a single physical pin on the uC. When we talk about reading or writing a pin, we mean a single physical pin not the PIN register which is for input only. When we say 'port D4' we are referring to the physical pin, and may be reading or writing it. Try to refer to the register explicitly 'PORTA register', 'PINA register', but sometimes you just have to recognise the context the term is being used in to know if it is the input register, the output register or the physical hardware pins being referred to.

2.8.Describe the ES as reactive and responsive to its environment

```
/***** Hardware macros *****/
//Hardware macros for outputs
1.   #define SET_RED_LED   PORTD |= (1<<PD6)
2.   #define CLR_RED_LED   PORTD &= ~(1<<PD6)
3.   #define SET_ORN_LED   PORTB |= (1<<PB0)
4.   #define CLR_ORN_LED   PORTB &= ~(1<<PB0)
5.   #define SET_GRN_LED   PORTB |= (1<<PB2)
6.   #define CLR_GRN_LED   PORTB &= ~(1<<PB2)
7.   #define SET_DONT_CROSS_LED   PORTB |= (1<<PB3)
8.   #define CLR_DONT_CROSS_LED   PORTB &= ~(1<<PB3)
9.   #define SET_CROSS_LED   PORTB |= (1<<PB4)
10.  #define CLR_CROSS_LED   PORTB &= ~(1<<PB4)
11.  //Hardware macros for inputs
12.  #define PB_SW_1_IS_LOW   ~PIND & (1<<PD2)
13.  /***** User macros *****/
14.  #define ORANGE_DELAY 3000
15.  #define RED_DELAY 500
16.  #define PED_DELAY 8500
17.  /***** Main function *****/
18.  int main(void) {
19.      // Initially all pins are automatically configured as inputs
20.      // make pins B4,B3,B2,B0,D6 outputs
21.      DDRB |= (1 << PB4);  //B4 set as output
22.      DDRB |= (1 << PB3);  //B3 set as output
23.      DDRB |= (1 << PB2);  //B2 set as output
24.      DDRB |= (1 << PB0);  //B0 set as output
25.      DDRB |= (1 << PB4);  //D6 set as output
26.      // Activate internal pull-up resistor
27.      DDRD |= (1 << PD2);
28.      /***** Run once code goes here *****/
29.      //setup state of all outputs
30.      //do not assume they are off
31.      SET_GRN_LED;
32.      CLR_ORN_LED;
33.      CLR_RED_LED;
34.      SET_DONT_CROSS_LED;
35.      CLR_CROSS_LED;
36.      /***** Loop code *****/
37.      while (1) {
38.          if(PB_SW_1_IS_LOW){
39.              SET_ORN_LED;
40.              CLR_GRN_LED;
41.              _delay_ms(ORANGE_DELAY);
42.              SET_RED_LED;
43.              CLR_ORN_LED;
44.              _delay_ms(RED_DELAY);
45.              SET_CROSS_LED;
46.              CLR_DONT_CROSS_LED;
47.              _delay_ms(PED_DELAY);
48.              CLR_CROSS_LED;
49.              SET_DONT_CROSS_LED;
50.              SET_GRN_LED;
51.              CLR_RED_LED;
52.          }
53.      } //end while(1)
```

Which line(s) of code is/are part of making the ES responsive to its environment?

The pullup resistor is an essential part of an input circuit. When the switch is:

- **not pressed**,
- **pressed**,

Without a pullup resistor when the switch is open the input pin would be _____

The uC can then be programmed to react differently to whether the switch is open or closed

Checking to see if a pin is low

```
#define SWITCH_IS_LOW          ~PIND & (1<<PD2)

if ( SWITCH_IS_LOW ){

}
```

OR checking to see if a pin is high

```
#define SWITCH_IS_HIGH         PIND & (1<<D2)

if ( SWITCH_IS_HIGH ){

}
```

Draw a pulldown circuit

2.9.Die Program (dice - plural, die - singular)

Apply a formal approach to writing program code

```
/***** Prototypes for Functions *****/
void one();
void two();
void three();
void four();
void five();
void six();
/***** Declare & initialise global variables *****/
uint8_t count=0;
/***** Main function *****/
int main(void) {
    /***** Loop code *****/
    while (1) {
        count++;
       if (TACT_SW_1_IS_LOW) {
            switch (count) {
                case 1:
                    one(); //function call
                    break;
                case 2:
                    two();
                    break;
                …
                …
            }
            //delay a bit
        }
    } //end while(1)
} //end of main
/***** Functions *****/
void one(){//turn on centre LED4
    PORTB=0b
    PORTD=0b
}
void two(){
    PORTB=0b
    PORTD=0b
}
void three(){
    PORTB=0b
    PORTD=0b
}
void four(){
    PORTB=0b
    PORTD=0b
}
void five(){
    PORTB=0b
    PORTD=0b
}
void six(){
    PORTB=0b
    PORTD=0b
}
```

Making code transparent with functions

Functions have a single purpose

Functions have descriptive names

Functions fit in 1 page

Use function prototypes

The use of one(), two() etc. as function names makes sense in this trivial program, however if the code become more complex, these might need more explanatory names. What would you name them to make them as fully transparent as possible?

2.10. Polling

Describe how an ES operates with the illusion of concurrency

Polling refers to the concept of regularly checking something (e.g. voters before an election). The uC often has to read check multiple things such as inputs. This means that the program in a uC has to **poll** or regularly check each pin. ES Polling must be at a rate that is fast enough not to miss any input event. When polling inputs, the rate of polling relates to how fast the environment changes input states. If we poll faster than the environment can change,then no environmental change will ever be missed and we will have the illusion of concurrency. In this example polling is demonstrated by sequentially checking four contestants switches. The illusion of concurency is demonstrated by the rate of polling being so high that it no switch press would every be missed .

1www.instructables.com

How does a quiz game controller function?
Describe its function here. This is its algorithm, it can be drawn in diagram form or written as plain language or written in pseudo-code.

2.10.1. Hardware topic: Switch issues

Explain contact bounce of physical switches and simple software de-bounce code

- Anatomy of a cheap push-button switch: When the button is pressed the whole metal plunger moves down closing the circuit between the two metal contacts. It can bounce a lot when this happens.

- The tact switch is much better (but still not perfect), when the button (2) is pressed the metal dome(3) flexes in a snap action touching the contacts(5); this is a rapid and firmer motion so there are few bounces.

http://www.omron.com/ecb/products/sw/special/switch/basic02-03.html

What happens when two solid objects come into contact at high speed?

We get what is called contact bounce

https://www.youtube.com/watch?v=4sA1c1WErUk

Don't try this at home!
https://www.youtube.com/watch?v=lzsTO3YMYAU

How do you think a uC responds to switch contacts bouncing?

De-bouncing switches

De-bouncing is the process where we solve contact bounce issues using either hardware or software

Hardware solutions

The Xplained mini board uses a simple 0.1uF capacitor across the switch

This circuit is an

A very reliable hardware solution

Reference: ganssle.com

Debounce software solutions

Often a short delay using _delay_ms(…) will be sufficient, sometimes more complex timing methods are used. We will look at using simple delays to solve issues in specific contexts.

2.10.2. A 'not-so-simple' debounce problem

Contact bounce can cause a switch to be detected multiple times. There is a potential issue from contact bounce in this program.

1. Initially understand the program. When the switch is pressed we increase the count and then wait for the switch to be released before continuing. It is important to understand why this is done. We only want one switch press to be detected, if we did not wait the loop would run so fast that count would increase rapidly even if we pressed and release the switch as fast as we could. 2. Now recognise the second complication of contact bounce in this program. A single press of the button can be interpreted as several presses and releases. Also when you release the switch it can be interpreted as several presses and releases as well

```
//no debounce
while (1) {
    if (SW_UP_IS_LOW) {
        count++;
        while (SW_UP_IS_HIGH){
        }
    }
    if (SW_CLEAR_IS_LOW) {
       count = 0;
    }
    display(count);
 } //end while(1)
```

```
//with debounce
#define DEBOUNCE_DELAY 10
while (1) {
    if (SW_UP_IS_LOW) {
    }
   if (SW_CLEAR_IS_LOW) {
       count = 0;
   }
   display(count);
 } //end while(1)
```

How long should the delay be?

2.11. AVR memory types

Describe the different memory types in the microcontroller and their uses

Knowing some details (initially size and type) of the 3 different types of memory in the uC helps us understand what some of the crucial constraints and limitations of ES's will be.

- FLASH –storing programs
  - the ATmega328P has a _____ of flash-_____
- SRAM – stores temporary data
  - the ATmega328P has _____ of SRAM(also called IRAM) -_____
- EEPROM – stores long term data
  - the ATmega 328P has a _____ EEPROM - _____

2.11.1. What type of number are you?

www.gansle.com

When writing programs for a uC, computer languages offer the ability to constrain a number to a defined range so that we can make the most of the limited SRAM. If data varies over a small range we may be able to use just 1 byte of the RAM. Data that may vary over a larger range or data that requires more accuracy will require more than 1 byte.

Describe the different memories of the microcontroller and their uses

Memory types in C include int, unsigned int, short, long, float, etc. We do not often use these names when discussing uC programming; we use names for data types that clearly express the limitations of each type. For example in a PC the int and unsigned int types are 4 bytes in size, whereas with programs written for an AVR, they are 2 bytes. The limits of the types are:

- uint8_t unsigned 8 bit range
- uint16_t unsigned 16 bit
- uint32_t unsigned 32 bit
- uint64_t [(1)] unsigned 64 bit
- int8_t signed 8 bit
- int16_t signed 16 bit
- int32_t signed 32 bit
- int64_ t [(1)] signed 64 bit ()
- float/double [(2)] 32 bit - 1.2E-38 to 3.4E+38

[(1)] if you are using numbers this big then you probably shouldn't be using an 8 bit uC! Using an 8 bit uC for calculations or comparisons using multi-byte variables require many repetitive

operations e.g. adding two 16 bit numbers requires two operations, first adding the lower byte of each number then adding the upper byte of each number.

(2) in programming AVR's a double and a floats are both the same - 4 bytes – not like in C on a PC where a double is 8 and a float is 4. Using doubles requires the use of a math library, as the AVR does not have the hardware to do floating point calculations, using the functions in a math library takes up a lot of your flash memory. Without a floating point math processor, division is often done by repetitive subtractions – trying to do this with floating point is ugly and will really slow down the overall processing with your ES , if you need decimals it may be better to use a fixed rather than floating point number system.

Data errors

Some data type errors can create BIG problems. In 1996 the Ariane rocket exploded 40 seconds after launch at a loss of $500M. Specifically a 64 bit floating point number relating to the velocity of the rocket was converted to a 16 bit signed integer. The number was larger than 32,767, and thus the conversion failed. Software designers were certain that the variables didn't need protection from overflow! https://hownot2code.com/2016/09/02/a-space-error-370-million-for-an-integer-overflow/

By Phrd - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=6376804

In 1991 28 soldiers were killed and 100 more injured when a patriot missile failed to track an incoming Scud missile. It was discovered that a 24 bit variable was used to store real time in 1/10ths of a second losing some 0.000000095 Sec of accuracy, so the patriot missile system failed to detect the Scud approaching and never launched.

Essential ES understanding: it is important to understand why it important to know the limitations of your data types.

Guidelines for type selection:

1. use the smallest type possible
2. use unsigned types where possible
3. know the boundary cases (limits) for your data

What sort of things could go wrong in a car engine control unit if an overflow/underflow occurred?

Here is a question from a survey on ES development from www.gansle.com. Should an engineer know the answer to a question like this?

*** 20. If the product resulting from your current project malfunctioned, what is the worst possible outcome?**

- ◯ Death of Multiple People
- ◯ Death of One Person
- ◯ Serious Injury of One or More People
- ◯ Minor Injury to One or More People
- ◯ Product Recall by Company
- ◯ Diminished Sales and/or Brand Reputation
- ◯ Customers Return Products
- ◯ Customers are Annoyed
- ◯ I don't know.

This question reveals the crucial importance of non-technical (social) factors to what engineers do. If there are any doubts about the validity of asking these questions then reading about the Therac-25 should make this quite clear.

http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html

Explain over/under flow of variables

Would you use an 8 bit type to store speed for a speed camera?

> The fastest known speeding ticket was issued in May 2003 in Texas. The motorist was operating a Koenigseggs CCR, a Swedish sports car and was allegedly going **242 mph** (**389 km/hr**) in a 75 mph zone. The driver was arrested and his sport car was towed.

if you declared a variable `uint8_t km_per_hour = 0;`
and then detected the speed of the car going 389km/hr.
What value would be stored in the variable?

2.12. External Interrupts

LO: describe how uC external interrupts are used to make an ES reactive

Timers are great – however uC's have another type of interrupt called an external interrupt, this allow the ES to react to untimed but urgent things that occur around in its environment, without having to poll for them.

uC's are sequential devices so they can only do one thing at a time; if the uC is busy doing one task it might miss an input of short duration, or if the number of inputs to check is large then even if it checks them all, it may be not be often enough, and miss one, or more. Importantly if the input is mission critical then it must be consistently monitored so that it is never missed.

Also delays like _delay_ms() are sometimes essential in ES software because we need to slow outputs down so that slow environments are given the time to react to the system, or so that a UI component can be recognised by a user (e.g. a flashing LED). While the uC is waiting however it cannot do anything else, hence the name busy wait for _delay_ms() commands.

In response to this manufacturers now implement special hardware features in uC's called external interrupts. There are several different interrupt facilities in the AVR. By connecting an input to an external interrupt pin we don't have to poll the pin but let the interrupt do it for us. A hardware interrupt is effectively ....

External interrupts INT0 and INT1

LED_1 does a double flash every 2 seconds. The switches are connected to D.2 and D.3, these are also the interrupt inputs INT0 and INT1. When the interrupt is triggered the uC completes execution of its current instruction then carries out the instructions in the ISR (Interrupt Service Routine), when those instructions are finished it carries on execution at the next instruction in the main (or delay) function Because of interrupt features we can give the ..

It doesn't matter where the code is it will jump to the ISR, carry it out, then return

External interrupts setup

The External Interrupts are triggered by the INT0 and INT1 pins. The External Interrupts options are:

- falling (negative) edge/potential on the pin
- rising (positive) edge
- low level- will repeatedly happen while the pin is low

The set up for INT0 or INT1 requires bits in 3 registers to be setup and appropriate ISR (Interrupt Service Routines) to be written

17.2.1 EICRA – External Interrupt Control Register A

The External Interrupt Control Register A contains control bits for interrupt sense control.

Bit	7	6	5	4	3	2	1	0	
(0x69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

write the code to setup INT1 as rising edge

EICRA _____

EIMSK – External Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

EIMSK _____

SREG – AVR Status Register

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
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	

The AVR Status register is manipulated using special GCC macros sei(); and cli(); to set/clear the Global Interrupt Enable in SREG.

The default setting is that bit I is 0 in SREG, so are any interrupts enabled on powerup?

Setting up External Interrupts INT0 & INT1

```
/***** Includes *****/
```

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
/***** Hardware macros *****/
//Hardware macros for outputs
#define SET_LED_2   PORTB |= (1<<PB0)
#define CLR_LED_2   PORTB &= ~(1<<PB0)
#define TOGGLE_LED_1   PORTB ^= (1<<PB4)
//Hardware macros for inputs
#define SW_1_IS_LOW   ~PIND & (1<<PD2)
#define SW_1_IS_HIGH   PIND & (1<<PD2)
#define SW_2_IS_LOW   ~PIND & (1<<PD3)
#define SW_2_IS_HIGH   PIND & (1<<PD3)
/***** Interrupt Service Routines *****/
ISR(INT0_vect){
    SET_LED_2;
}
ISR(INT1_vect){
    CLR_LED_2;
}
/***** Main program *****/
int main(void) {
    // Initially make all micro pins outputs
    DDRB = 0xff; //set as outputs
    DDRC = 0xff; //set as outputs
    DDRD = 0xff; //set as outputs
    // make these pins inputs
    DDRD &= ~ (1 << PD2);
    DDRD &= ~ (1 << PD3);
    PORTD |= (1 << PD2); //activate internal pullup resistor
    PORTD |= (1 << PD3); //activate internal pullup resistor
    //Interrupt setups





    /***** Loop code *****/
    while (1) {
        TOGGLE_LED_1;
        _delay_ms(100);
    } //end while(1)
} //end of main
```

2.12.1. Interrupt complications

Discuss issues of an ES's responsiveness with regard to polling, blocking and interrupts

Lengthy interrupts

Once an interrupt is triggered and the ISR begins to execute, the global interrupt bit in the AVR's SREG is disabled, this means that if the same interrupt occurs during this period of time the AVR will not sense it and it will not be detected (other, different types of interrupts however can be queued).

So how long should an ISR be?

Short interrupt code

What about when ISR's are very short – only taking a few cycles of the clock – if we connect mechanical switches to an external interrupt pin – we might find that one bouncing contact input causes many interrupts

So how short should an ISR be?

Too many interrupts

When your interrupt is executing your main program is not,
Excessive interrupts can…

Interrupts and variables

When variables are shared between the main function and interrupt functions they must be declared as volatile so that the compiler knows not to optimise them away

Atomicity

The AVR memory is 8 bits wide, so when 16 bit variables are manipulated they get changed 1 byte at a time. However should an interrupt happen at the point in time when only 1 half of the variable has changed and then the interrupt routine changes the variable the value will get corrupted? This issue relates to what is called atomicity. While in C a command may appear as a single operation, in the machine code multiple operations may be taking place. Interrupts should not be able to occur during non-atomic operations

```
/***** Declare & initialise global variables *****/
volatile uint16_t time_delay = 500;

void my_function(){

   _________ //disable interrupts to protect critical section

   uint16_t count = time_delay; //actually involves multiple operations

   ___________
}
ISR(INT0_vect){
    time_delay++; //ISR changes value in variable
}
```

Why wouldn't atomicity be an issue when developing applications for a phone, tablet or PC?

2.13. Timer/Counters

LO: understand the ES as reactive and responsive to its environment

The uC's ability to react to the real world with any sort of timed accuracy is limited by its sequential nature. It can be impossible to calculate accurate timing within even a simple while(1){} loop because a program can react to different inputs, take different paths and therefore different time for each run through the loop.

To assist ES engineers accurately control timing, uC manufacturers have included a number of hardware counter/timers into their devices as standard features. This means that while the main software process is doing its regular jobs, it can be interrupted on a fixed timing schedule to do a critical task or to control an output or check an input.

Some events in the real world need to be accurately controlled within tight timing constraints- sometimes in the order of 10^{-6} or 10^{-9} seconds. An example is a petrol engine and controlling the timing of the explosion of fuel in the cylinder increases fuel efficiency, develops the most power and reduces emissions.

A microcontroller is required to control the timing of output devices such as this not only with great accuracy but reliably (without ever missing an event).

See the animation at https://en.wikipedia.org/wiki/Petrol_engine

Some events in the real world occur at varying intervals and the timing of these events must be measured sometimes with 10^{-6} or 10^{-9} second's accuracy.

For example in vehicles the speed of each wheel needs to be measured accurately for the ABS (anti-lock braking system) to work. A microcontroller measures the speed of each wheel then determines whether one has a different speed to the others to see if the car is at risk of skidding; the brake for that wheel is then released slightly so that it keeps traction with the road. Polling (regular checking) of the input sensors for each wheel may not be good enough in this situation; the ES may need to react more quickly than the polling in a while loop allows. We can use the internal timers of the uC to interrupt normal program flow to check the wheel speed, meaning the uC will never miss an event.

http://simnascar.com/tag/abs/

2.13.1. AVR Timers

The AVR uC's have one or more internal hardware binary timer/counters that count at a fixed rate, when a counter gets to the top of its count and overflows an interrupt is triggered. Then it starts counting again.

Watch out for the use of the word: timer – counter – divider – prescaler in this section. These devices are all physical binary counters based upon flip flops, but we use the words differently to describe their different function.

2.13.2. Timer are complex (allow multidimensional problems to be solved)

- AVR's generally have
  - two 8 bit timer/counters
  - one 16 bit timer/counter
- clocking of the timer can be configured to be from:
  - the CPU clock source
  - the CPU clock source divided down (via a pre-scaler/divider)
  - Timer0 and Timer1 can be clocked via a related pin
  - Timer2 can be clocked from an external crystal
- counting is generally from 0 upwards
- timing completion/time out can be configured to occur
  - when the timer has overflowed
  - at a pre-set value,
- when the timer completes its count the timer can be configured to
  - modify a related output pin (set/clr/toggle)
  - interrupt the program and run the code in a function (called an ISR - Interrupt Service Routine)
- timers can be configured to start
  - when the input clock is configured
  - automatically after they overflow
- timers can be stopped
  - by disabling the input clock
- PWM is a further special mode for the timer

Each timer has a few related pins associated with it

Timer0
T0
OC0A
OC0B
Timer1
T1
ICP1
OC1A
OC1B
Timer 2
OC2A
OC2B
TOSC1
TOSC2

Figure 5-2. 28-pin MLF Top View

2.13.3. Timer Registers

Each timer in the ATmega328P has a set of registers to control them- these are described in the ATmega328 datasheet (sections 19,20,21,22)

- the timer/counter register itself
  - 8 bit - counts form 0 to
  - 16 bit - counts from 0 to
- a number of Timer Counter Control Registers
- a number of Output Compare Registers
- an interrupt mask register
- a flag register

Timer 0 Registers (8Bit)

- TCNT0 T0 (Timer 0) – the actual 8 bit binary counter
- TCCR0A T0 counter control register A – setup timer features
- TCCR0B T0 counter control register B– setup timer features
- TIMSK0 T0 interrupt mask register –enable T0 specific interrupts
- TIFR0 T0 interrupt flag register – we can check/clear interrupt conditions
- OCR0A T0 output compare register A the value that TCNT0 will be compared to
- OCR0B T0 output compare register B the value that TCNT0 will be compared to

Timer 1 (16 bit)

- TCNT1H/TCNT1L Timer 1 - High and low bytes of TCNT1
- TCCR1A
- TCCR1B
- TCCR1C
- TIMSK1
- TIFR1
- ICR1H/ICR1L
- OCR1AH/OCR1AL
- OCR1BH/OCR1BL

Timer 2 (8bit)

- TCNT
- TCCR2A
- TCCR2B
- TIMSK2
- TIFR2
- OCR2A
- OCR2B
- ASSR

The number of registers involved with each timer indicates that they have many options making them initially complex. A number of examples will help you become familiar with the datasheet, the control bits in the various control registers and what they do.

Setting up Timer1 registers (datasheet 20.14.2)

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

20-7 Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}/1$ (No prescaling)
0	1	0	$clk_{I/O}/8$ (From prescaler)
0	1	1	$clk_{I/O}/64$ (From prescaler)
1	0	0	$clk_{I/O}/256$ (From prescaler)
1	0	1	$clk_{I/O}/1024$ (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Write code to setup the following options: assume that the timer is being reconfigured during program use. This means that we must both clear (make 0) some bits and set (make 1) some bits.

Timer 1 no prescale:

Timer 1 prescaler =1024:

Timer1 stopped:

if we were working with Timer0 rather than Timer1 then we would change?

2.13.4. Timer example – 'heart beat' led flash

Describe how internal uC timers are used to make an ES responsive

How to accurately control the timing of an external device without involving counting in the programs main loop.

Often users need to feel confidence that a piece of equipment is functioning. A steady lit LED is used to indicate that power is on; a flashing LED makes users believe that the equipment is actually working (like its alive, so we use the term heart beat).

Goal: Configure a timer to toggle an LED on an output every second

Timer Counter 1, (16 bit) overflow (the simplest operation)

The timer register (TCNT1) counts up from 0 to 65535 then overflows back to 0 (total count = 65536). In TIMSK1 bit TOIE1 is set so that on overflow (TCNT1 goes back to 0) an interrupt occurs and the code in an ISR (interrupt service routine) is run.

The rate of the counting is dependent upon F_CPU, the clock frequency.

At 16MHz a count of 1 takes _____ Sec = _____ mS = _____ uS = _____ nS

65,536 counts take _____ mS. in terms of some real world applications this is too fast.

A slower rate can be produced using a prescaler (counter/divider) for dividing the crystal frequency down before it goes to the main timer1 counter.

16Bit timer counter

Clock	No prescale	/8	/64	/256	/1024
TCNT0 input	16Mhz				
Time period of 1 count	0.0625 uS 62.5 nS				
Overflow (65536 counts)	4,096 mS				

2.13.5. The 'about' 1 second timer

```
/***** Project Header *****/
// Project Name: Timer1-FFFFOverflow

/***** Hardware defines *****/
#define F_CPU 16000000//crystal

/***** Includes *****/
#include <avr/io.h>
#include <avr/interrupt.h>
/***** Hardware macros *****/
#define TOGGLE_LED_1   PORTD ^= (1<<PD3) //could be any pin
/***** ISRs *****/
ISR(TIMER1_OVF_vect) { //called when the timer overflows
    TOGGLE_LED_1;
}
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRD |= (1 << PD3);            //setup led pin as output

    /***** Timer Config *****/
    TCCR1B |= (1 << CS12);         // Prescaler /256
    TIMSK1 |= (1 << TOIE1);        // enable T1 overflow interrupt
    sei();                         // enable global interrupts

    /***** Loop code *****/
    while (1) {
        //nothing is needed here
        // however whatever is here will be interrupted every second
    } //end while(1)
} //end of main
```

What would have to change to make this an LED on A.2?

If we used the 8 bit timer / counter then the prescaler outputs and 8 bit overflow would be

Clock	No prescale	/8	/64	/256	/1024
Counter input	16Mhz				
Time period of 1 count	0.0625mS				
length of 256 counts (include overflow to 0)					

Can you think of a way to use an 8 bit timer to get a delay of about 1 second?

2.13.6. The 'exact' 1 second timer

To get a more accurate value for a 1 second timer we can configure OCR1A so that the counter no longer counts to 0xFFFF (65535) then overflows but to some lesser value _____and then overflows

This value is setup in register _____

```
/***** Project Header *****/
// Project Name: Timer1-1Sec OCR1A Interrupt
/***** Hardware defines *****/
#define F_CPU 16000000//crystal

/***** Includes *****/
#include <avr/io.h>
#include <avr/interrupt.h>

/***** Hardware macros *****/
#define TOGGLE_LED_1   PORTD ^= (1<<PD3) //can be any pin on the uC

/***** ISRs *****/
ISR(TIMER1_COMPA_vect) { // ISR called every time T1 overflows OCR1A value
    TOGGLE_LED_1;             //TCNT1 automatically clears to 0 at OCR1A
}

/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRD |= (1 << PD3);             //set led pin as output

    /***** Timer Config *****/
    TCCR1B |= (1 << WGM12);    // CTC clear timer on compare match mode
    TCCR1B |= (1 << CS12);     // Prescaler
    OCR1A   = 62499;          // the value to count up to
    TIMSK1 |= (1 << OCIE1A);  // enable CTC interrupt
    sei();                    // enable global interrupts

    /***** Loop code *****/
    while (1) {
        //nothing is needed here for the timer
        // however code that is here will be interrupted every second
    } //end while(1)
} //end of main
```

2.13.7. The 'exact' 1 second flash direct hardware output timer

A further option is to not use an ISR but to connect an LED to one of Timer1's related pins (OC1A or OC1B) This means once the timer is setup there is no ongoing software processing at all (i.e. no ISR) it is all hardware.

```
/***** Hardware defines *****/
#define F_CPU 16000000//crystal

/***** Includes *****/
#include <avr/io.h>

/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRB |= (1 << PB1);             //setup OC1A pin (B1) as output

    /***** Timer Config *****/
    TCCR1B |= (1 << WGM12);      // CTC mode
    TCCR1A |= (1 << COM1A0);     // toggle OC1A (B1 )output on match
    TCCR1B |= (1 << CS12);       // Prescaler /256
    OCR1A = 62499;

    /***** Loop code *****/
    while (1) {
    } //end while(1)
} //end of main
```

This option is completely autonomous, after it is setup it keeps going, no loop code, no interrupts

Can you develop a mathematical model for calculating timer counts?

if there is a formula for this then there are calculators to help you figure it out as well, HOWEVER…

What would OCR1A need to become if we wanted 500mS on and 500mS off, and which prescale value should we use?

- prescaler 1 not possible as the max count is 65535 and we need to count to 16,000,000
- prescaler 8 not possible as the max count is 65535 and we need to count to 2,000,000
- prescaler 64 not possible as the max count is 65535 and we need to count to 250,000
- prescaler 256 OCR1 = _____
- prescaler 1024 OCR1A = _____

2.13.8. Make a short duration pulse at a regular time interval

OCR1A and OCR1B registers can be configured to control their respective pins OC1A and OC1B to control two different output devices, or the two interrupts COMPA_vect and COMPB_vect can be configured to provide two different interrupt events. One example that combines the two timer Interrupts would be to make an LED flash briefly once per second

```
/***** Includes *****/
#include <avr/io.h>
#include <avr/interrupt.h>
/***** Hardware macros *****/
#define SET_LED_1   PORTD |= (1<<PD3) //could be any pin
#define CLR_LED_1   PORTD &= ~(1<<PD3) //could be any pin

/***** ISRs *****/
ISR(TIMER1_COMPA_vect) { //called every time TCNT1 reaches TCCR1A
    CLR_LED_1;
}
ISR(TIMER1_COMPB_vect) { //called every time TCNT1 reaches TCCR1B
    SET_LED_1;
}
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRD |= (1 << PD3);             //set led pin as output
    /***** Timer Config *****/
    TCCR1B |= (1 << WGM12);         // CTC mode
    TCCR1B |= ((1 << CS10) | (1 << CS12)); //Prescaler /1024
    OCR1A   = 19531;             // off time
    OCR1B   = 18531;                // on time
    TIMSK1 |= (1 << OCIE1A);        // enable CTC interrupt match A
    TIMSK1 |= (1 << OCIE1B);        // enable CTC interrupt match B
    sei();                          // enable global interrupts
    /***** Loop code *****/
    while (1) {
    } //end while(1)
} //end of main
```

This sort of timer could be used in an electric fence controller.

http://www.zarebasystems.com/blog/farm-equipment/why-electric-fence-system/

What is the on and off time for the LED in the previous code?

What should OCR1A and OCR1B be set to for an electric fence pulse of 35mS every 0.75 seconds?

Setup the timer for an electric fence pulse of 35mS every 2.7 seconds

2.13.1. **Use ICP1 to measure pulse width**

There are two main methods for using the timer related input pins these are: counting the number of pulses that occur on a related input pin (T0,T1) and counting the length of a pulse using ICP1. The lab requires you to use the second option.

ICP1 is PIN B.) of the uC allows an external event on the pin to cause the uC to capture the value in TCNT1 when the input event occurred.

The timer is free-running (always counting) so the positive edge of ICP1 could be at any count of the timer. In the diagram above if the two counts were 15345 and 32071 in the above diagram what would the total pulse width be? The clock is 16MHz and the prescaler is set to 1024.

In the lab you will be displaying the time in hundredths of seconds (0.1S to 25.5S) using flashing LEDs

e.g. 13.7 seconds would be 1 flash of the first LED, 3 flashes of the second LED, and 7 flashes of the third LED.

An algorithm is needed for the process, and there are two complications that have be taken into account:

1. The ICP1 can be setup as either positive or negative edge triggered - but not both at once. So once the positive edge has been detected the pin must be reconfigured to sense a negative edge detect. And after the negative edge has been detected the pin must be reconfigured to sense a positive edge detect.
2. The overflow may or may not occur during the pulse measurement, this must be accounted for in your algorithm. Your logic must work whether there is an overflow or not

situation	formula
	Pulse duration =
	Pulse duration =

Write 1 algorithm that works for both situations.

Lab setup - Timer1 prescale = 1024

- TCCR1B |= (1<<CS12) | (1<<CS10);

Positive edge detect with noise canceller

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ICNC1: Input Capture Noise Canceler**

Setting this bit (to one) activates the Input Capture Noise Canceler. When the noise canceler is activated, the input from the Input Capture pin (ICP1) is filtered. The filter function requires four successive equal valued samples of the ICP1 pin for changing its output. The Input Capture is therefore delayed by four Oscillator cycles when the noise canceler is enabled.

- **Bit 6 – ICES1: Input Capture Edge Select**

This bit selects which edge on the Input Capture pin (ICP1) that is used to trigger a capture event. When the ICES1 bit is written to zero, a falling (negative) edge is used as trigger, and when the ICES1 bit is written to one, a rising (positive) edge will trigger the capture.

When a capture is triggered according to the ICES1 setting, the counter value is copied into the Input Capture Register (ICR1). The event will also set the Input Capture Flag (ICF1), and this can be used to cause an Input Capture Interrupt, if this interrupt is enabled.

When the ICR1 is used as TOP value (see description of the WGM13:0 bits located in the TCCR1A and the TCCR1B Register), the ICP1 is disconnected and consequently the Input Capture function is disabled.

TIMSK1 – Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 5 – ICIE1: Timer/Counter1, Input Capture Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Input Capture interrupt is enabled. The corresponding Interrupt Vector (see "Interrupts" on page 57) is executed when the ICF1 Flag, located in TIFR1, is set.

- **Bit 0 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Overflow interrupt is enabled. The corresponding Interrupt Vector (See "Interrupts" on page 57) is executed when the TOV1 Flag, located in TIFR1, is set.

- TCCR1B |= _____ //positive edge capture
- TCCR1B | = _____ //noise canceller
- TIMSK1 |= _____ //enable ICP interrupt
- TIMSK1 |= _____ //enable overflow interrupt
- sei();//enable global interrupts

```
#define ICP_ON_POS_EDGE _______________________ //detect ICP positive edge

volatile uint16_t t1_count = 0;
volatile uint16_t t2_count = 0;
volatile uint16_t overflow_count = 0;
volatile uint32_t total_count = 0;
volatile uint8_t pulse_width = 0;


ISR(TIMER1_CAPT_vect)
{
    if ( ICP_ON_POS_EDGE ){         //pulse start, positive edge









    }
    else {                          //pulse finish, negative edge









    }
}

ISR(TIMER1_OVF_vect) {
    overflow_count++; //increment overflow count
}
```

Timer resolution, range, step size and accuracy

Timers are either 8-bit **resolution** or 16-bit **resolution** in the AVR, so they have a range of discrete values either from 0 to 255 or 0 to 65535. The process of quantization involves changing continuous values (e.g. time) to discrete values (values in a timer) so there are going to be some inaccuracies involved, the larger the step-size the more inaccurate the time period will be. For a clock frequency of 16MHz

Prescaler value of	Frequency input to counter	**Step-size **** time for a count of 1	Period for a count of 2	**Range (8 bit timer)** Timer starts at 0, counts to 255 and overflows back to 0 = count of 256	**Range (16 bit)** Timer starts at 0, counts to 65535 and overflows back to 0 = count of 65536
1	16MHz	62.5nS	125nS		4mS
8	2MHz	500nS	1uS		32.768mS
64		4uS	8uS		262mS
256		16uS	32uS		1.04S
1024		64uS	128uS		4.194304S

** The best accuracy we can get using that prescaler with the 16MHz clock

How much accuracy is enough accuracy? This depends entirely upon the situation.

1. Does an electric fence controller that pulses once every 3 seconds need better than _____uS accuracy?

2. What timing accuracy does ABS need in your car?

3. What timing accuracy does a microwave timer need?

Can you see why engineers often work in teams?

Who would an engineer team up with to design answers to 1/2/3?

Timer Accuracy calculations

The discrete (non-continuous) nature of timers means that we may not always get the exact timing value we desire, so we must often do accuracy calculations for different frequencies, timers and prescale values. In the table below we can work out:

- an exact value for the number of counts
- then quantize (choose the nearest whole number for OCR)
- decide which timer we can use knowing that
  - Timer0 & Timer2 can count from 0 to 255
  - Timer 1 can count from 0 to 65535
- work out the error for each option to determine if we can get the accuracy we require.

Set up a timer to provide a timing period of 0.419mS**(+/-0.5%)** = 416.905uS to 421.095uS

	OCR value = F_CPU * delay period /prescaler -1				
Freq	16000000	16000000	16000000	16000000	16000000
prescaler	1024	256	64	8	1
Step-size (sec)	0.000064	0.000016	0.000004	0.0000005	62.5E-09
desired period (Sec)	**0.000419**	**0.000419**	**0.000419**	**0.000419**	**0.000419**
Number of counts	5.546875	25.1875	103.75	837	6703
discrete OCR value (floor)	5	25	103		
discrete OCR value (ceiling)	6	26	104		
	delay period = (OCR Value + 1) * prescaler / F_CPU				
Chosen OCR value	5	6	25	104		
actual period		448uS	416uS	420uS		
error		29.0E-6	3.0E-6	1.0E-6		
error%		6.9%	0.7%	0.2%	0%	0%

0.419mS(+/-0.5%)

T0/T2 use OCR0A/OCR2A =

T1 use OCR1A =

Timers and PWM (pulse width modulation)

Timers can be used to turn outputs on and off with different ratios of on to off time, this is called PWM. There are many uses for PWM, some of the common ones include:

- Fan/motor speed
- LED / light brightness
- Servo motors

The key concept is to generate a fixed frequency signal but vary the ratio of the on/off time

If we connected a motor to a uC pin, which of the following three pulses would make the motor go slowest?

LO: describe how internal uC timers are used to make an ES responsive

Timer uses	How timer(s) could be used
Anemometer	Count number of revolutions per second or per 10seconds, Or measure the period between pulses
time-lapse camera shutter controller	Press a switch to turn on an output, start a timer, When timer timeouts, turn off output and stop timer
Beeper	Make an output of a certain frequency and a certain duration
engine rpm	
fridge door	When a switch is opened start a timer When the switch is closed stop and reset the timer After the timeout make a beeper sound until switch is closed
Siren	Varying the frequency of a tone
scanning 7 segment displays	Very common- drive 1 digit at a time, but fast enough so that users will not see the flickering.
Real time clock	Timer2 has a special mode that divides down a 32,768Hz crystal to get a 1 second interrupt If a crystal already exists on the board, then configure an timer to for a 1 second or some fraction of 1 second interrupt Then make a make a clock in software (remember to account for different days in months and leap years)
De-bouncing switches	To avoid blocking code – the use of _delay_ms() use a timer instead.
ultrasonic object ranging	Send a short duration pulse of ultrasonic frequency, start a timer. Stop the timer when the pulse reflection is detected and work out the length of time
servo motors	Pulse width modulation
Motor speed control	PWM, make sure the frequency matches the characteristics of the inductor in the motor, so that the has time to respond to the changing between 1 and Ramp up a motors from stopped to full speed slowly
delayed turn on /off of an output	Lock lecture theatre doors so that late students cannot come in.
Dead-man switch	Press start switch, output goes on, Press the switch every so often or the output goes off As used in …
Man-down switch	Mount a switch to the side of a two-way radio, if the radio lies on its side for more than ½ second send out an emergency signal As used in …
Mix colours of RGB LED into any colour	RGB LEDs have three separate LEDs in one package, to make other colours vary the brightness of each LED individually using PWM.

How is a uC accuracy controlled?

- Internal RC oscillator?

- External crystal?

2.14. ES development overview

So far you have used a simulator to develop your ideas around simple AVR uC based ES's, you can see an overview YouTube video about this here at HelloWorldSimulation.XplainItToMe.com

Here we will look at how to get code into a real uC development board an overview of this process can be found in a YouTube video here at HelloWorldProgram.XplainItToMe.com

When developing ES software a number of processes are encountered

1. IDE – Atmel Studio 7
2. Writing code
3. Syntax errors
4. Compiling
5. uploading

2.14.1. **Writing bug free code (hopefully)**

It is cheaper and easier to prevent bugs from creeping into your programs than it is to find and kill them once you realise they are there. 56% of the coding work done is rewriting existing code (www.gansle.com), so the clearer and more readable your code is, the easier it will be to modify - even if you are the only person who will ever read it.

http://img.timeinc.net/time/daily/2010/1010/360_stein_1018.jpg

Become a _____

- think about how you will document your code before you begin writing it
- think about code reliability and portability (re-use) and not about your convenience or trying to show off by writing esoteric code
- stick to one naming convention: for variables and functions. In embedded C use lowercase with underscore separators, for constants uppercase with underscore separators
  void read_temperature()
  uint8_t greenhouse_temperature = 0;
  #define TEMPERATURE_COEFFICICENT 324
- Use descriptive variable and function names:
- Think about the correct type of variable to use:
  - the temperature outside **uint8_t temp** (is this ok?)
  - the temperature in a cool store
  - the relative humidity in a greenhouse
  - the floor a lift is on
  - whether the car door is locked or unlocked
- Put only one task into a function
- Keep functions to 1 viewable page of code

- Always use {} with if statements- even though not entirely necessary
- Indent your code – 4 spaces
- Comment blocks and lines of code with their purpose and function (draw diagrams)
  - avoid stating the obvious about syntax unless it is complex or abstract
- Use local variables whenever possible. If a variable is used only in a function, then it should be declared inside the function

2.14.2. LO: Understand the importance of transparent software practices

Apply a ~~spaghetti~~ formal approach to writing program code

```
/***** Project Header *****/
// Project Name: Cylon 14
// Author: Bill
// Date: 21/02/2014  11:51
// Version: 1
// This code is a …

/***** Hardware defines *****/
//make sure this matches your oscillator setting
#define F_CPU 16000000//crystal
/***** Includes *****/
#include <avr/io.h>
#include <util/delay.h>
/***** User macros *****/
#define TIME_DELAY 100 //
/***** Main function *****/
int main(void) {
    /***** IO Hardware Config *****/
    DDRB = 0xff; //setup all pins as outputs
    DDRD = 0xff; //setup all pins as outputs
    /***** Loop code *****/
    while (1) {
        PORTD = 0b00000001; //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00000011; //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00000111; //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00001110; //
        _delay_ms(TIME_DELAY);
        PORTD = 0b00011100;
        _delay_ms(TIME_DELAY);
        PORTD = 0b00111000;
        _delay_ms(TIME_DELAY);
        PORTD = 0b01110000;
        _delay_ms(TIME_DELAY);
        PORTD = 0b11100000;
        …

    } //end while(1)
} //end of main
```

www.freefoodphotos.com

If someone opened up your code, what would it look like inside?

Key structural elements in the code

- **Template**
  - Structures your code
  - into different sections
- **Title block**
  - Name
  - Date
  - Author
  - Version
- **Macros**
  - #define TIME_DELAY 100
  - no magic numbers
  - CAPITALS
- **Comments**
  - describe function not syntax
- **Layout**
  - indentation
    -
  - syntax highlighting
    -
- **Why are these points so critical?**
  - engineers are not haphazard in what they do, professionalism in documentation is essential , most organisations will have formal methods of structuring their planning and documentation.

At the core of professionalism in programming is **transparency**. Code must be *readable, logical, understandable and maintainable*. In programs indentation and macros are used to structure code to make it transparent!! There is no place for obfuscation in programming, do not be an obfuscator!

In this code some of the lines use macros, some don't, which line of code is wrong?

```
//pedestrian crossing program
/***** Hardware macros *****/
//Hardware macros for outputs
#define SET_RED_LED   PORTD |= (1<<PD6)
#define CLR_RED_LED   PORTD &= ~(1<<PD6)
#define SET_ORN_LED   PORTB |= (1<<PB0)
#define CLR_ORN_LED   PORTB &= ~(1<<PB0)
…
…
1.  while (1) {
2.  if(PB_SW_1_IS_LOW){
3.  SET_ORN_LED;
4.  PORTB &= ~(1<<PB2);
5.  _delay_ms(3000);
6.  SET_RED_LED;
7.  PORTB &= ~(1<<PB0);
8.  _delay_ms(500);
9.  PORTB |= (1<<PB3);;
10. CLR_DONT_CROSS_LED;
11. _delay_ms(8500);
12. PORTB &= ~(1<<PB4);
13. PORTB |= (1<<PB3);
14. PORTB |= (1<<PB2);
15. CLR_RED_LED;
16. }
17. }
```

2.14.3. **How we get from C to hex in ATMEL Studio 7.**

Describe the functions of an IDE - the process of compilation

2.14.3.1. The processor CPU and ALU

Microprocessors and microcontrollers contain a CPU (central processing unit) which has an ALU (arithmetic logic unit). Microcontrollers also contain memory: SRAM, FLASH and EEPROM. In a microprocessor based system, memory is separate e.g. a hard drive. The CPU control section sequentially fetches instructions from the program memory ('flash') and decodes them. Take the simple program to add 2 numbers in the box on the right. The machine code stored in Flash (starting at address 0x0090) is: 83, e6, 80, 93, 03, … The machine code can be viewed in the .hex file – however a more intelligible view of flash is found in the .lss file.

```
#include <avr/io.h>
volatile uint8_t a = 0;
volatile uint8_t b = 0;
volatile uint16_t c = 0;
int main(void)
{
   a = 99;
   b = 4;
   c = a + b;
   while (-1)  {   }
}
```

You need to know a few things: C programs are not stored in the uC – they are compiled into machine code and that is stored in the Flash. The human readable version of machine code is called assembler code or assembly language (assembler commands: ldi, sts, lds, add, adc are described in the datasheet). Instructions either transfer data between the general purpose registers and SRAM/EEPROM/hardware registers or carry out simple operations on data in the general purpose registers using the ALU. Instructions such as +, -, << are single ALU operations and limited in that most of them only work on registers so take only 1 clock cycle. Thus the AVR is a RISC- reduced instruction set computer as opposed to a Complex (CISC) where a single instruction may be more powerful containing multiple read, write and ALU operations. The AVR ALU is a simple logic device with no multiplication or division built into hardware. There is a contiguous address space for the general purpose registers, the hardware registers and the SRAM; whereas EEPOM and FLASH are separate address spaces. We refer to the IO as being memory-mapped IO – because it's addressed like it was SRAM. The AVR is a Harvard Architecture; if program and data memory were contiguous then it would be a Von-Neuman architecture.

2.14.4. **What about when the ES doesn't work quite right**

These are called semantic or logic errors

We need to check the systems correct operation and identify/debug any errors

What could we add to an ES to help assist debugging?

2.14.5. **Pointers and memory mapped IO**

Describe the different memories of the uC and their uses

In ENGGEN131 pointers were introduced. Think about what pointers are for, to store the address of a variable. In ES one powerful use for pointers is to refer to the address of hardware registers. We can do this because IO in a uC is memory-mapped – appears to be just like memory such as SRAM or FLASH. When you think about this, it makes sense because IO and memory are just locations with addresses .

In this 3 digit counter the 3 displays are connected to 3 different ports, but wired alike.

in C we use knowledge of integer arithmetic and a very helpful function called modulus (%) to help us separate a number into individual digits.

In **integer arithmetic**

9/3 =3 9 % 3 = 0

10/3 = 3 10 % 3 = 1

11/3 =3 11 % 3 = 2

12/3 = 4 12 % 3 = 0

modulus = A-B*(A/B)

no rounding

e.g. if count = 246

hundreds = count / 100 =

tens = count % 100

tens = tens /10

units = …

We **could** write three different display functions

```
void disp(int16_t count){
    //split the count into 3 digits
    disp_hundreds (hundreds);
    disp_tens (tens);
    disp_units (units);
}
void disp_hundreds(uint8_t hundreds){
    if(number == 0){
        PORTA = 0b11000000; //PORT A
    }
    if(number == 1...
    if(number == 2...
}
void disp_tens(uint8_t tens){
    if(number == 0){
        PORTC = 0b11000000; //PORT C
    }
    if(number == 1...
    if(number == 2...
}
void disp_units(uint8_t units){
    if(number == 0){
        PORTB = 0b11000000; //PORT B
    }
    if(number == 1...
    if(number == 2...
} (but this is WASTEFUL of our memory)
```

An efficient use of flash memory is to write one function not three. To do this the common and unique aspects of the functions must be identified. In this case the unique aspects are the 3 addresses of the port registers (A,B,C). The rest of the functions are identical. We can then write one function and pass it a port address. We can do this because the IO is memory-mapped in the AVR.

```
void disp(int16_t count){
    volatile uint8_t* p_portaddress =  & PORTC; //declare a pointer
    //split the count into the required 3 digits
    p_portaddress = &PORTA;// pointer contains the address of PORTA
    display_digit(p_portaddress, hundreds);





}

void display_digit(volatile uint8_t *port, uint8_t number){
    if(number == 0){
        *port=0b11000000; //store the value in the correct register
    }
    if(number == 1){…….}
```

Memory	Name	contents	
0x103			
0x102			
0x101			
0x100			
…			
0x29	PIND		
0x28	PORTC		
0x27	DDRC		
0x27	PINC		

2.14.6. **Volatile**

Compilers convert high level languages to machine code, but they also are written to perform optimization during this process. Optimizations they perform include the removal of useless or unreachable code. When the same variable appears in different scopes it can appear to the compiler as unreachable or different variables, it will then remove them. The word volatile in English means unpredictable and when used as a keyword instructs the compiler not to optimize code related to the variable as it may change in ways outside the compiler's understanding. Variables associated with pointers may change during runtime so can appear as useless or unreachable so must be declared as volatile.

2.15. ADC

The real world is not digital, it is analog. So for an ES to be reactive to real world events it must be able to convert information from the analog world to the digital one.

2.15.1. The voltage divider

At the core of converting energy in the real world into the electrical world is this fundamental electrical circuit, the …

Working in context with voltage dividers is at the core of using ADCs and sensor subsystem design, and this involves understanding **ratios** (equivalent fractions)

 "I understand they all have the same value, but I have to tell you, the ones on the right feel like more bang for your buck."	The voltage divider is a

This ratio can be expressed in 4 different ways depending upon what we want to use it for

Situation	Solve for the unknown
	Vout =
we know the upper and lower resistor values and what we want the output to be so therefore	Vin =
	R1 =
	R2 =

ADC conversion process

Describe the significant characteristics of the internal ADC

Inside the uC is an analog to digital converter it turns a voltage into a 10 bit binary number and stores in TWO registers- your code can then read the registers and perform calculations on the value

This is called quantization, the process of mapping a continuous signal to a digital one. Quantization replaces each real number with an approximation from a finite set of discrete values

The Atmel AVR range has a **single** internal ADC with _____

resolution is the number of bits in the digital output code of the ADC

The AVR ADC has _____ discrete values or _____ steps: 0 to _____

When Vin = 0 the adc output value is _____

When Vin = Vref the adc output value is _____

The relationship between input voltage and output number is a linear process and can be expressed as a _____

What is the simple ratio that expresses the relationship between input voltage, ADC resolution, ADC output value, reference voltage?

The ATmega328P has 3 different reference options:

- 1.1V (accurate internal reference)
- AVCC (use the power supply as a reference)
- AREF (connect your own reference voltage - cannot exceed VCC)

Vref = 1.1V, Vin = 0.67V	ADC value =
Vin = 2.2V, Vref = 5.03V	ADC value =
ADC value = 754, Vref =5.07V	Vin =
ADC value = 432, Vref =1.1	Vin =

ADC reference select

Figure 28-1. Analog to Digital Converter Block Schematic Operation

Table 28-3. ADC Voltage Reference Selection

REFS[1:0]	Voltage Reference Selection
00	AREF, Internal V_{ref} turned off
01	AV_{CC} with external capacitor at AREF pin
10	Reserved
11	Internal 1.1V Voltage Reference with external capacitor at AREF pin

What would be the command(s) to select the internal Vref?

If you needed to change Vref during the program,

what would be the command(s) to select AVcc as the reference?

2.15.2. **ES analog sensors**

Describe the use of a sensor in a voltage divider circuit

In an ES voltage divider circuit there are not two fixed value resistors but one component which has variable resistance and one which is fixed. Either can be the upper or lower resistor in the circuit.

Light measurement	Temperature measurement	Displacement (potentiometer)
		(linear or lagarithmioc over 270 degrees range)
As the light level goes up the output voltage goes _____	As the temperature goes up the output voltage goes _____	As the displacement of the object changes the output voltage changes

When selecting a value for the fixed resistor in a voltage divider start by choosing a value in the middle of the range you are wanting the device to be most sensitive in; e.g. in a thermistor circuit if the range of measurements is -10 to +40 then find out what the resistance is at 15 degrees. This is where the measurements will be most sensitive to change in temperature.

2.15.3. **The full conversion process**

ORP12 Datasheet, ww.alldatasheet.com

R1 = LDR in sunlight (400lux) Vin = 5V, R2 = 10k, Vref =1.1V

Which way around should our Vdiv circuit be?

The value converted by the ADC will be:

Solve problems of dynamic range and quantization when employing analog sensors

Mostly we need to solve the voltage divider so that we have a useful range of values
In the last exercise we converted lux to a number in the ADC using a fixed value of R of 10K
Complete the table and plot the ADC voltage on the graph (scale RHS

LUX	R LDR (from graph)	VDiv out (R=10K, Vin=5V)	ADC (ref=1.1V)
10	90k	4.5V	1023
20	5k	1.7V	1023
30	3k5	1.3V	1023
40	2k5	1V	930
50	2k	0.83V	775
100	1k02	0.46V	430
400	300	0.1456	135
700	190	0.083V	86
1000	140	0.069V	64

How will we convert the non-linear exponential ADC readings of Lux to actual Lux values in a variable, that we could display? The AVR has no floating point processor so we could use the math library to add floating point capability
however that might use up a lot of flash
Is there a way we could do it with integer arithmetic?

Another Situation	Unknown R value
Car accelerator potentiometer Vin = 12V, Vref = 1.1V, RPot = 47k R fixed = ??	

ADC LSB voltage or step size (sometimes called voltage resolution)

This is the minimum change in voltage required to change the ADC output by 1

ADC value = 75, Vref =1.1V	Vin =	LSB voltage
ADC value = 76, Vref =1.1V	Vin =	

ADC value = 75, Vref =5V	Vin =	LSB voltage
ADC value = 76, Vref =5V	Vin =	

Which step size above is 'better'?

'better' relates to context and the system being fit-for-purpose. In an abstract or de-contextualised example like this there is no 'better' or 'best'.
We need to know the situation that an ADC will be used in to make a decision about what are the 'best' or minimum attributes for it.

2.15.4. **Quantization error**

In the above examples input voltage levels that are in between the two values e.g. 0.080677V or 0.08106V cannot be accurately represented. In an ADC with a 5V reference an output value of 0 means the input voltage was somewhere between 0V and 4.89mV

ADC conversion is an approximation or rounding process. This means there is error in the quantization process and this quantization error is the difference between the actual sampled value and the quantized value.

The quantization error is what relationship to the two values?

To reduce the error

1. we can use an ADC with higher resolution (number of levels/bits)

2. we can reduce Vref

but this reduces the range of the input voltage we can measure as well

ADC converters become more expensive and complex as features such as resolution (number of bits) are increased. And circuits become more susceptible to noise with lower input voltage ranges, so changing these things is not easy. Atmel engineers and product people must have decided on a feature set at some stage that was suitable for a wide range of users and uses.

2.15.5. **AVR ADC inputs**

The AVR ADC has 8 channels (inputs) available for measurements external to the uC

ADC channels
ADC0 (PC0)
ADC1 (PC1)
ADC2 (PC2)
ADC3 (PC3)
ADC4 (PC4)
ADC5 (PC5)

ADC6 (not an I/O pin)
ADC7 (not an I/O pin)

2.15.6. **ADC Channel select**

The AVR's have _____ ADC pins but only ____ ADC converter. A _____ is used to select which channel a conversion will be carried out on. Channel selection is made via the ADMUX register

Although there are 8 external ADC channels on the ATmega328P (M1A package) the MUX is capable of 16 selections. Three of the other 8 selections are internal:

Figure 28-1. Analog to Digital Converter Block Schematic Operation

The selection of which channel is to be read from is made in the ADMUX register

ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0
(0x7C)	**REFS1**	**REFS0**	**ADLAR**	**–**	**MUX3**	**MUX2**	**MUX1**	**MUX0**
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Write the command to select channel 5 for an ADC conversion.

ADMUX | = 0x05 might or might not work. Why?

Write commands to setup the ADC to read from the internal temperature sensor

2.15.7. **Sampling rate**

The process of making a conversion inside an AVR is via the method called successive approximation, it is reasonably complex involving 10 comparisons so it cannot happen in a single clock cycle of the main CPU clock (F_CPU). To give ADC circuits the time needed for conversions they are clocked at rates lower than F_CPU and the actual conversion takes 13 cycles of this lower clock speed. This clock speed is 50kHz to 200kH. The clock rate is provided by the ADC prescaler which is setup using three bits in ADCSRA.

Figure 28-1. Analog to Digital Converter Block Schematic Operation

Table 24-5. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

If F_CPU = 10MHz, then what would be all the possible values of the ADC prescaler and which would meet the specification of 50kHz - 200kHz?

- 10,000,000 / 2 = 5,000,000
- 10,000,000 / 4 =
- 10,000,000
- 10,000,000
- 10,000,000
- 10,000,000
- 10,000,000

2.15.8. **ADC modes**

There are two modes for ADC conversions: single conversion and free running. In addition an ADC interrupt can be setup so that an ISR is called when a conversion is completed, this makes 4 options.

Single conversion mode Non-interrupt	Single conversion mode With interrupt	Free running mode Non-interrupt	Free running mode With interrupt

ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0
(0x7A)	**ADEN**	**ADSC**	**ADATE**	**ADIF**	**ADIE**	**ADPS2**	**ADPS1**	**ADPS0**
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

Datasheet	notes
Bit 7 – ADEN: ADC Enable Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress will terminate this conversion.	
Bit 6 – ADSC: ADC Start Conversion In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC. ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.	
Bit 5 – ADATE: ADC Auto Trigger Enable When this bit is written to one, Auto Triggering of the ADC is enabled. The ADC will start a conversion on a positive edge of the selected trigger signal. The trigger source is selected by setting the ADC Trigger Select bits, ADTS in ADCSRB.	
Bit 4 – ADIF: ADC Interrupt Flag This bit is set when an ADC conversion completes and the Data Registers are updated. The ADC Conversion Complete Interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a Read-Modify-Write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.	
Bit 3 – ADIE: ADC Interrupt Enable When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.	

Table 26-6. ADC Auto Trigger Source Selections

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match A
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

(in register ADCSRB)

code to trigger ADC on Timer/Counter0 overflow _____

2.15.9. **ADC resolution**

Do we actually need all 10 bits of resolution?

If we can use 8 bit resolution then we can set ADLAR (ADC left adjust) and just read ADCH

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
(0x79)	–	–	–	–	–	–	ADC9	ADC8	ADCH
(0x78)	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

When an ADC conversion is complete, the result is found in these two registers.

ADLAR is 0 by default

unit16_t adc_result = ADCW; // 1023 = 0b00000011 11111111 ADCW get both bytes together

ADMUX |= (1<<ADLAR); //SET ADC to left adjust conversion into ADCH

uint8_t adc_result = ADCH; //1023 = 0b11111111 11000000 which we read as 0b11111111(255)

2.15.10. ADC Example 1

Single conversion mode Non-interrupt	Free running mode Non-interrupt	Single conversion mode With interrupt	Free running mode With interrupt
Setup prescaler Setup channel ADC Enable Start Conversion Loop while ADSC is high When low read ADC value Change channel if reqd	Setup prescaler Setup channel ADC Enable Start Conversion Monitor ADIF When low read ADCW Clear ADIF (write 1 to it)	Setup prescaler Setup channel ADC Enable Write ISR{ - change channel if req'd - process ADC value} Enable ADIE Setup ADTS Start Conversion	Setup prescaler Setup channel ADC Enable Start Conversion
requires polling and is blocking	requires polling and is blocking	No polling or blocking	No polling or blocking

We will choose one of these options for our example

Corrective gait device: a person has an issue with their gait (manner of walking), we want to give them immediate feedback while they are walking to help them recognise and correct their gait.

epainassist.com

How many sensors would be required in the shoe and where would they be?

Sensor - FSR Force Sensitive resistor

13mm diameter FSR
Interlink Electronics
($10 each for quantities of 1)

V+ = 5V, Vref = 2.56V, R= 10K

Force(gm)	FSR(R)	Vout	ADC out
0	>100k		
50			
100			
500			
1000			

FSR test setup

```
#define F_CPU 8000000//crystal
/***** Includes *****/
#include <avr/io.h>
//Hardware macros
#define SET_RED_LED   PORTB |= (1<<PB3)
#define SET_ORN_LED   PORTB |= (1<<PB2)
#define SET_GRN_LED   PORTB |= (1<<PB1)
#define SET_YEL_LED   PORTB |= (1<<PB0)
#define FSR_1 2     //macro to refer to ADC channel

void all_leds_off(){
    PORTB &= 0xF0;
}
void init_ADC() {
    ADCSRA |= ( (1 << ADPS2) | (1 << ADPS2) ); //8MHZ prescaler to 64
    ADMUX |= ( (1<< REFS2) | (1 << REFS1) );  //Vref 2.56V internal nocap
    ADCSRA |= (1 << ADEN); //Power on the ADC
    ADCSRA |= (1 << ADSC); //Start initial conversion
    ADMUX |= (1 <<ADLAR); //left adjust ADC reading into ADCH
    //ADATE is not set so system will be in single conversion mode
}
uint8_t read_adc(uint8_t channel) {
    ADMUX &= 0xF0; //Clear previously read channel
    ADMUX |= channel; //Set to new channel
    ADCSRA |= (1 << ADSC); //Starts a new conversion
    while (ADCSRA & (1 << ADSC)); //BLOCK until the conversion is done
    return ADCH;
}
int main(void) {
    init_ADC(); //setup the ADC to work
    DDRB = 0xff; //set all pins initially as outputs
    DDRB &= ~(1 << PB4); //adc 2 is PB4 on an ATtiny85 make an input
    uint8_t fsr_1_val = 0; //declare and initialise value for adc readings
    while (1) {
        fsr_1_val = read_adc(FSR_1);
        fsr_1_val = fsr_1_val >> 6; //reduce range from 8 bits to 2
        all_leds_off();
        switch (fsr_1_val){
            case 0:  //FSR reading of
                SET_RED_LED;
                break;
            case 1:  //FSR reading of
                SET_ORN_LED;
                break;
            case 2:  //FSR reading of
                SET_GRN_LED;
                break;
            case 3:  //FSR reading of
                SET_YEL_LED;
                break;
        }
    } //end while(1)
} //end of main
```

2.16. State machine programming

Often students learn programming using flowcharts as planning tools; however flowcharts are unable to express the complex set of behaviours and reactions required in embedded systems.

Think of the truck in the graph below as having 4 states.

- stationary
- accelerating
- constant speed
- decelerating

A graph of velocity against time can be used to understand the four states and to consider the inputs and actions that are required to manage the system

vehicle pulls away from the traffic lights a set of inputs must and outputs conditions exist and a complex set of behavioural choices are called for.

A first state diagram could look like this

What happens if the truck has not reached constant speed and the driver sees a red light ahead?
The state machine code needs to be able to transition from any state to any other state, at the moment it cannot. However this is easily extendable to add transitions and actions

The power of a state machine diagram can be seen in the situation where the truck is able to react to changes in the environment and **transition** from one state to any other state depending upon **conditions** such as inputs or calculations within the system.

The keywords are:

- state
- transition
- condition
- action

Mealy and Moore are two types of state machines, what is the difference?

What is this type?

A large number of state machine examples can be found on the internet, including:

- Pacemaker
- Stopwatch
- Calculator
- Greenhouse
- Traffic lights

2.16.1. Coding state machines in C

Coding state machines into programming languages is all about clearly identifying and laying out the code to reduce the spaghetti, making it easier to identify sources of bugs and to add new states, transitions and actions.

One common method of coding smaller state machines is using the switch, this can also be coded using an if-else or while loop pattern. A more comprehensive pattern uses a separate function for each state and calling the appropriate function using a function pointer.

```
#define EW 0
#define NS 1
#define PED 2

uint8_t state = NS;
```

```
while (1){
   switch (state){
      case EW:
         //...
         if (PED_SW){
            state = PED;
         }
      break;
      case NS:
         //...
         if (PED_SW){
            state = PED;
         }
      break;
      case PED:
        //...
      break;
   }
}
```

```
while (1){
    if (state == EW){
       //...
       if (PED_SW){
           state = PED;
       }
    }
    else if (state = NS){
        //...
    }
    else if (state == PED){
        //...
    }
}
```

```
while (1){
    while (state == EW){
       //...
       if (PED_SW){
           state = PED;
       }
    }
    while (state = NS){
        //...
    }
    while (state == PED){
        //...
    }
}
```

2.16.2. **Breath tester state machine example**

This device is used to detect excess alcohol intake by having someone blow through a straw onto an analog alcohol sensor. The device however can give unreliable readings if the person does not blow long enough or hard enough into the device. An addition to the device was designed to capture the length of the blow and amount of air blown. Inside the plastic block is a small ball which rotates around inside. An Infrared LED and an infrared sensor are mounted opposite each other, and when the pea rotates around inside the chamber the light path between the LED and sensor is broken. Speed can be calculated using a timer.

The mechanism is very similar to a 'pea' or referee's whistle

www.madehow.com

Here is an initial state machine for the system

This has a number of problems; can you identify some of them?

2.16.3. **Electronic safe state machine**

Operating the safe:

- Enter the correct 4 digits in the keypad
- If the code is correct the door unlocks
- If the door is unlocked and remains closed for 20 seconds it automatically locks
- If the door is open and then closed it locks
- If the door is open and the A key is pressed a new code can be entered
    - While entering a new code if C is pressed the new code is cancelled
    - After entering a new 4 digit code if D is oppressed the new code is saved
    - if the door is closed before the new code is saved or cancelled the new code is ignored

```
/***** Globals *****/
uint8_t code1 = 1;
uint8_t code2 = 2;
uint8_t code3 = 3;
uint8_t code4 = 4;
uint8_t state=0;

/***** Functions *****/
void unlock;
   CLR_LOCK;
}
void lock(){
   SET_LOCK;
}
void get_key(){
   keypad_val = read_adc(KEYPAD);
}
```

Does this code express every possible combination of the logic for exiting ST_1_CORRECT?

```
while (state == ST_1_CORRECT) {
        get_key();
        if (keypad_val != code1 && keypad_val != code2 && keypad_val != UNPRESSED) {
           state = ST_LOCKED;
        }
        if (keypad_val == code2) {
           state = ST_2_CORRECT;
           wait_for_unpressed();
        }
    }
```

2.17. Serial Data Communications

- describe the different communication protocols used in the Atmel AVR
- explain the uC USART serial data communication protocol and its parameters
- setup the uC USART registers

2.17.1. Uses for serial and parallel data communications

Some devices are more complex than a single transition on a uC's binary I/O pin or an analog input pin can deal with. In these cases data is transferred using either serial or parallel communication protocols.

Serial data TO a graphics LCD.	Parallel data TO / FROM a graphics LCD.	Serial data FROM a SHT11 humidity sensor
Serial data FROM an accelerometer	Serial data FROM a DS18B20 temperature sensor	Serial data FROM a magnetic card reader
Serial data TO / FROM a Bluetooth transceiver	Serial data TO / FROM a WiFi transceiver	Serial data FROM a GPS receiver
Serial data TO / FROM an external SRAM	Serial data FROM an RFID reader	Serial data between uC's

These complex devices require a great deal of information to be sent and received. They use either serial or parallel communications to achieve this. Most of them use serial communications, (serial is preferred because of the fewer number of connections required). The difference between serial and parallel communications can be viewed here https://www.youtube.com/watch?v=PJ_bS7meE7s

Data communications needs to be completely reliable – i.e. the receiver must get the exact same information that was transmitted. To achieve the required reliability with such complex information, highly detailed protocols have been developed with very specific parameters.

- A protocol describes the rules and procedures for communication.
- A parameter is a numerical or other measureable aspect that clearly defines all rules and elements of the protocol, this includes acceptable tolerances (boundary conditions)

2.17.2. Serial communication protocols

There are several different common serial communication protocols. The AVR has several different serial communication protocols built in through special serial hardware circuits:

- USART - Universal Synchronous Asynchronous Receiver Transmitter
- TWI - two wire interface – compatible with Philips I2C or I^2C- Inter-Integrated Circuit
- SPI – serial peripheral interface

2.17.3. Protocol parameters

The parameters defined within a protocol might include such things as:

- Number of signal and control lines (wires)- including ground
- Signal level e.g. active high/low, 5V/3V3, -12V/+12V, and with % or values of tolerances
- Signal Timing /data rates– with tolerances around such things as shape (clock skew) and frequency
- Signal sequencing (MSB or LSB first)
- Codes that have specific meanings.

2.17.4. Communication channel types or modes

An important aspect of sending and receiving information reliably is that there must be an agreement between the transmitter and receiver as to which direction data is moving and when.

These are the different options that describe the direction in which data is transferred

- Simplex - data goes in only one direction TO or FROM the uC
- Half-duplex - data goes both ways but not simultaneously - only one way at a time
- Duplex – data goes both ways simultaneously
- This helpful animation shows the differences between them
    - http://www.iec-usa.com/Browse05/DTHFDUP.html

The AVR serial communications registers are:

2.17.5. **Transmission timing**

The next stage in setting up a reliable transfer of information is to define the data transmission or bit rate. The bit rate is the number of bits (1's or 0's) sent per second.

Number of bits per second = 1 / t

Bit rate is very important; the receiver must align with the transmitter settings. The receiver must know the timing of a single bit so that it can determine exactly when to decode the data as 1 or 0. Here is an example of a data stream

1. If the transmitter timing of the above signal was t1 then the data would be _____
2. If the transmitter timing of the above signal was t2 then the data would be _____

The receiver needs to be able to interpret this signal correctly; the two different ways of achieving this are synchronous and asynchronous methods:

- Synchronous:
  - A clock signal is sent along with the data on another wire
- Asynchronous
  - The receiver is setup with the timing in advance - a number of well-known bit rates exist, two are: 9600bps, 115200bps,
  - We could send an initial sequence of 10101010 so that the receiver could get a lock first

2.17.6. **Information transfer**

Information is more than single 1's or 0's. It requires groups of 1's and 0's (codes), which have meaning. If we wanted to represent numbers 0 to 9 we would need 4 bits, to send letters of the alphabet a 7 bit code has been created called ASCII (American Standard Code for Information Interchange), the letter 'A' is 0x41 = _____ This has been further extended into 8-bit and 16-bit codes called Unicode.

In asynchronous data transfer groups of bits or codes are sent with extra bits as part of a _____ to signal the beginning and end of each frame. The parameters for the frame below are 8 data bits, with 1 start bit (0) and 1 stop bit (1).

The receiver knows that the first frame has begun when the _____ is detected on the data line. The parameters for the data rate and the frame must be set up the same in the transmitter and receiver so the receiver knows how long a bit takes and can accurately tell when to measure the next bit in the sequence.

Error checking

Sometimes electrical noise in the environment can interfere with data transmission. In this case a particular bit can become corrupted. One very simple method of checking for errors involves sending an extra 'parity' bit inside each frame.

In a system with a parameter setting for even parity the total number of 1's must be even, if the parameter setting is for odd parity then the total number of 1's must be odd.

	Received data frame	Parity	Is the data correct?
A	**0101010101 1**	even	
B	**0101111101 1**	odd	
C	**0111110101 1**	odd	
D	**0101101101 1**	even	
E	**0110010101 1**	odd	

2.17.7. ATMega328 internal communication hardware datasheet section 20

USART options from the datasheet

- **Full Duplex Operation (Independent Serial Receive and Transmit Registers)**
- **Asynchronous or Synchronous Operation**
- **Master or Slave Clocked Synchronous Operation**
- **High Resolution Baud Rate Generator**
- **Supports Serial Frames with 5, 6, 7, 8, or 9 Data Bits and 1 or 2 Stop Bits**
- **Odd or Even Parity Generation and Parity Check Supported by Hardware**
- **Data OverRun Detection**
- **Framing Error Detection**
- **Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter**
- **Three Separate Interrupts on TX Complete, TX Data Register Empty and RX Complete**
- **Multi-processor Communication Mode**
- **Double Speed Asynchronous Communication Mode**

2.17.8. Serial communications example code

Setup an AVR running at 8Mhz using 8 data bits, 1 start bit, 1 stop bit, 9600 baud

```
#define FOSC 8000000 // Clock Speed
#define BAUD 9600
#define UBRR_PARAM FOSC/16/BAUD-1 //Asynchronous normal mode
//*************************************************************
//usart_init: initializes the USART system
//*************************************************************
void usart_init() {
    UCSRB = 0x08;      //enable transmitter No magic numbers!!
    UCSRB = (1 << RXEN) | (1 << TXEN); // Enable Tx and Rx
    UCSRC = 0x86;      //async, no parity, 1 stop bit, 8 data bits
    UCSRC = (1 << UCSZ00) | (1 << UCSZ01)
    UBRRH = 0x00;
    UBRRL = 0x40;
    UBRR = UBRR_PARAM;
}
//*************************************************************
//usart_transmit: transmits single byte of data
//*************************************************************
void usart_transmit(unsigned char data) {
    while((UCSRA & 0x20)==0x00)  //wait for UDRE flag
    {
    ; //DO NOTHING
    ]
    UDR = data;
}
```

Receiving serial data using the USART

What might be wrong with developing this sort of code?

```
void main (void) {
    while(1){
        while((UCSRA & 0x80)==0x00)
        { ; }//wait for RXC flag
        data = UDR;
        … //the rest of your code
    }
}
```

It is better to use interrupts …

```
//************************************************************* //usart
ISR  //*************************************************************
ISR(USART_RXC_vect)
{
    rxdata = UDR;
    sei():
}
```

set up the serial receive interrupt (add to usart_init function this)

```
void usart_init(void)
{
    UCSRB |= (1 << RXCIE);
}
```

Debugging serial communications problems

The Saleae Logic analyser

www.saleae.com www.aliexpress.com

Instead of looking at communication waveforms on an oscilloscope and reading the 1's and 0's yourself, devices such as these can interpret the serial data stream for you.

extremely useful reference: **http://www.fourwalledcubicle.com/AVRArticles.php**

2.18. Fitness-for-purpose

LO: understand 'fitness-for-purpose' in guiding microcontroller choice for an ES

Understanding the nature of an ES (the interrelatedness of hardware and software, the ES as an unattended automaton, the ES as reactive and responsive to the environment) allows the enginer a mindset of what will make the system fit-for-purpse. When choosing a microcontroller for an ES this mindset focuses us on the task the system is fulfilling and makes it possible to select an approproate device with the required feature set that is cost-effective. avoiding *'using a sledgehammer to crack a nut'*

The criteria for measuring fitness for purpose include:

- Device speed
- Power supply - battery or existing power supplies
- power consumption – being able to put a uC into some form of redued power or sleep mode is essential in reducing power
- Package (DIP, QFP) – size and assembly considerations
- Cost per unit and quantity needed
- Bus size – 8/16/32/64
    - The size of calculationsare floating point calcs required
- Memory
    - SRAM – how much data / how many and what calculations need to be processed
    - Flash – the size of the program – will extra features be added later
    - EEPROM – what non-volatile data needs saving
- Number of I/O pins
    - How many devices/peripherals – how many ADC channels
- DC characteristics -
- number and trigger type of external Interrupts
- number, size and capability of timers
- Tools and libraries available for development
    - Having to write libraries of functions for peripheral devices is not easy and can be avoided by choosing a uC/manufacturer who provides this support – the 8 bit AVRs are well supported – as are PICs and some others
    - Debug features are important for developers in cross compilation environments
- Supplier capability and longevity
    - Choosing a manufacturer that is committed to supporting their products in the long term, where new products are backward compatible with old ones
- Software security features
    - Can someone read your code out of the uC and steal your intellectual property
- Expansion capability,
    - Having special hardware interfaces built in like : UART, CAN, I2C, 1-wire,… built in
- Other features
    - Low voltage (brown-out) detection is common
    - watch dog detector

In your learning to date you should understand some of these and be able to explain them

2.19. Driving output devices

LO: list specifications of an uC that would inform choosing a uC for an ES

LEDs (Light Emitting Diodes)

Will it work?

Will it work?

Will it work?

How can we know?

we need to know the characteristics/specifications of the devices

before we can engineer a solution

LED Specifications:

Kingbright, super bright Red BL-BD-BF43R7M 4000mcd

Vf typical: 2.0V at 20mA

this means it will draw 20mA when connected to 2.0 V,

When thinking about connecting an LED to a uC we need to know about both the uC and the specific board we are using.

A few ATmega328P Specifications from the datasheet (Page 2)
Operating Voltage:
Speed Grade:

- 0 - 4MHz@
- 0 - 10MHz@
- 0 - 20MHz @

Could I design a circuit that works at?

- 4V and 16MHz? 7V and 12MHz? 1.8V and 4MHz?

Could I over-clock an AVR?
<u>Should I design a circuit that uses an over-clocked AVR?</u>

ATmega328P DC characteristics for -40 to +85 degC datasheet section 32.1

Absolute Maximum Ratings

- DC Current per I/O Pin
- DC Current Vcc and GND pins

Although the absolute maximum specs are 40mAthe datasheet gives a graph of output voltage for different frequencies and output currents, that only goes up to 20mA. So connecting devices that would require more than 20mA would require some experimental investigation.

ATmega328P: I/O Pin Output Voltage vs. Source Current (V_{CC} = 5V)

Xplained Mini 328P development board specification

- Vcc =
- Crystal frequency =

This means that an LED cannot be connected directly between an I/O pin and ground

What is the output voltage of the pin on the Xplained mini board going to be if we connect an LED that draws 20mA to it?

How many LEDs could we connect and turn on at one time?

Look thoroughly through the datasheet to work out the current drawn by all the subsystems within the uC that you will be using and then use the maximum rating of 200mA that can be drawn through the power supply pins of the uC.

Driving devices with higher current / voltage requirements

Electromechanical devices such as solenoids and motors are commonly controlled by uC's

e.g. 12V 300mA

Solenoids become useful when they are connected to mechanisms such as those found in fuel control units of cars, production lines in factories and _____

cvel.clemson.edu

Solutions include using a logic level MosFET

2.20. Useful resources

'Atmel AVR microcontroller primer programming and interfacing' - Barrett & Pack
An excellent resource on how to setup AVR timers, ADC, Serial communications

'A Baker's Dozen' Bonnie Baker
An excellent book written by an IC designer

'Trouble shooting Analog Electronics' Bob Pease
This is an excellent book written by a IC designer from National Semiconductor Corporation

'Practical AVR Microcontrollers'
Trevennor has some interesting projects and excellent coverage of AVRDude

You can download the above books from the UoA library

TinyAVR microcontroller projects for the evil genius
Dhananjay V Gadre Nehul Malhotra
Has some interesting and novel project ideas

AVR Freaks – forum and tutorials
http://www.avrfreaks.net/forum/newbie-start-here
http://www.avrfreaks.net/forums/tutorials

De-bouncing switches
http://www.ganssle.com/debouncing-pt2.htm

Embedded C resources
volatile, const, static
http://barrgroup.com/Embedded-Systems/How-To/Efficient-C-Code

AVR Tutorials
extremeelectronics.co.in/category/avr-tutorials/

Interrupts
www.avrfreaks.net/forum/tutsoft-traps-when-using-interrupts

Timers
http://www.atmel.com/Images/Atmel-2505-Setup-and-Use-of-AVR-Timers_ApplicationNote_AVR130.pdf
http://www.evilmadscientist.com/2007/quick-and-dirty-d-to-a-on-the-avr-a-timer-tutorial/
http://www.fourwalledcubicle.com/AVRArticles.php

State machine programming
https://barrgroup.com/Embedded-Systems/How-To/State-Machines-Event-Driven-Systems
http://codeandlife.com/2013/10/06/tutorial-state-machines-with-c-callbacks/

CISC/RISC
https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccis

Thanks for coming to the uC section of CS201, I have enjoyed teaching it and I am sure you will find this knowledge useful in your future as engineers.