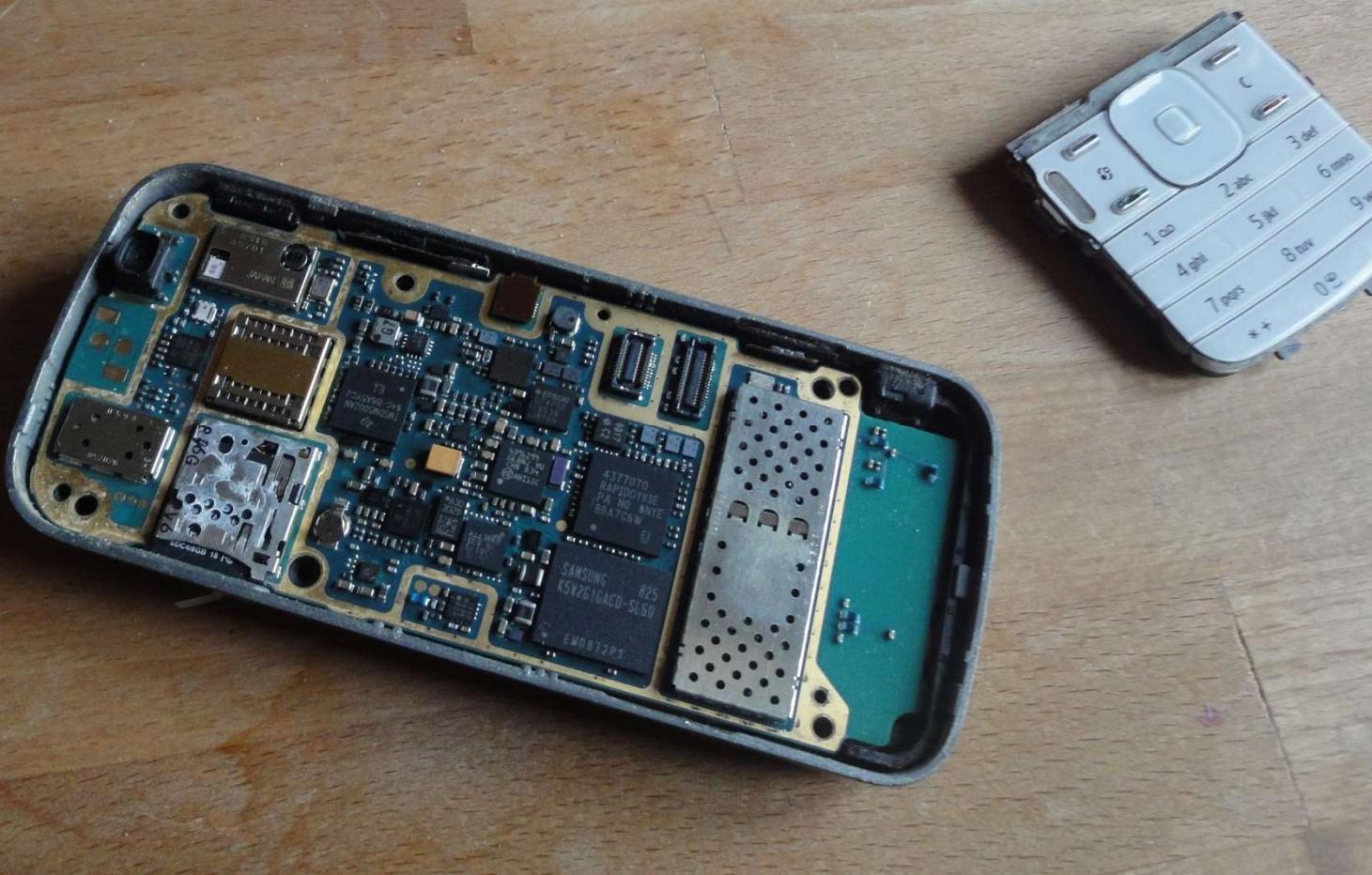




Name: _____



COMPSYS201 - 2021

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	i-iii
Lecture Material	
Part I (with Kevin Wang)	1-92
1. History and Trends in Digital Systems	1
2. Combinational Logic Design	8
3. Combinational Logic Components	24
4. Sequential Logic Design – Memory	47
5. Sequential Logic Design – Counter/Timer	60
6. Finite State Machine	66
7. Register Transfer Level Design	80
Part II (with Bill Collis)	93-180
1. Introduction	96
2. Learning outcomes and success criteria	97
3. The way of the program	98
4. Review your prior learning	98
5. Atmel/AVR range of 8-bit microcontrollers	99
6. Code Style	106
7. The interrelatedness of hardware and software in Embedded Systems	107
8. Describe the Embedded System as reactive and responsive to its environment	118
9. Memory mapped IO and Pointers	126
10. Timer/Counters	128
11. External Interrupts	144
12. Serial Data Communications	150
13. The ADC	155
14. State machine programming	169
15. Driving output devices	174
16. Fitness-for-purpose	179
17. Useful resources	180

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 Evans¹ 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 *supercomputers* that crunch massive amounts of data.

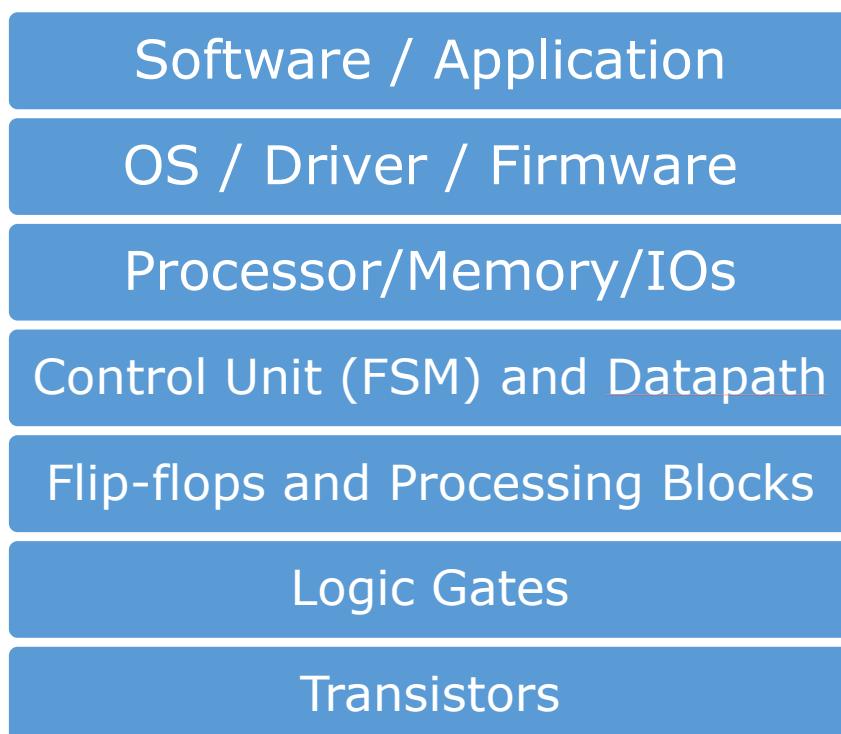
Computers are complex systems – they are made of many parts, and we will introduce you to some of those parts in this course. One of the key concepts of computer development is **abstraction**, which means hiding away details. The idea is to work on a small part of the system, and make sure that small 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

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

can be switched on or off (and therefore equivalent to 1 or 0). If every software 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 on Canvas. These are good for very brief summaries of what is being covered in this course – but they often do not go into detail, because the detail is provided verbally by the lecturer. This is generally accompanied by 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.

This coursebook is the third option, and tries to sit in the middle – it hopefully provides enough detail that you can study from it, but does not purport to be an exhaustive source of information. If you need more examples or deeper explanation, refer to one of the textbooks or have a look online.



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 Zoran Salcic.

Some images from Wikimedia Commons, either in the public domain or under CC licenses.

COMPSYS201 – Part I

Fundamentals of Computer Engineering

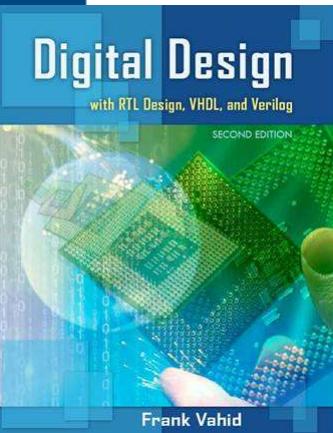
History and Trends in Digital Systems

Dr. Kevin Wang
kevin.wang@auckland.ac.nz

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



1



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>

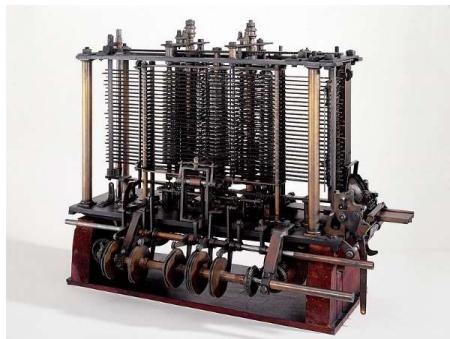
Copyright © 2010 Frank Vahid

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.

2

History of Computer I

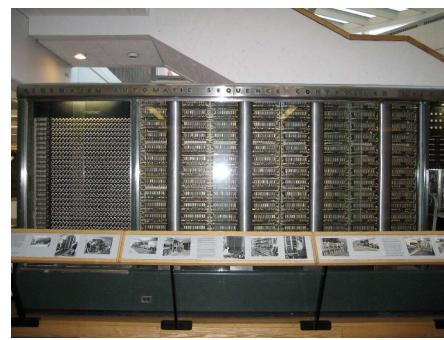
- Pre-digital age: mechanical machine
 - Difference engine and analytical engine
 - Perform certain mathematical calculations/algorithms, not general purpose
 - Punch cards for controlling the machine
 - Process decimal values rather than binary values
 - Decimal is more intuitive to human than binary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



3

History of Computer II

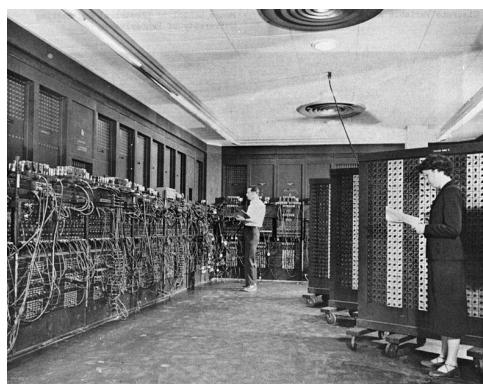
- Harvard Mark I (1944)
 - Electro-mechanical, weighed almost 5 tons and had a clock of 66Hz
 - 0.3 seconds for an addition, 6 seconds for a multiplication, and 15 seconds for a division
 - Reads instructions from punched paper tapes
 - No conditional branches
 - Relatively more reliable than other electronic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



4

ENIAC

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



5

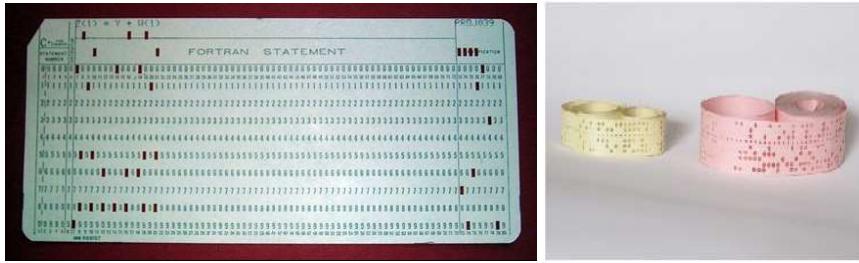
Electronic Computers

- 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

6

Paper Memory

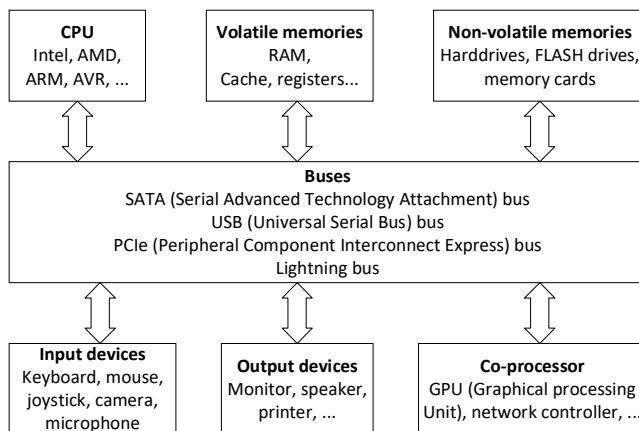
- Paper memory
 - Was once used as primary memory device
 - A program consists of a stack of punch cards
 - Later, punch cards and paper tapes were used as input/output storage media
 - Punch cards and paper tapes were not replaced until the 70s when electronic memory was introduced
 - Not re-writable,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



7

Practical Computer Systems

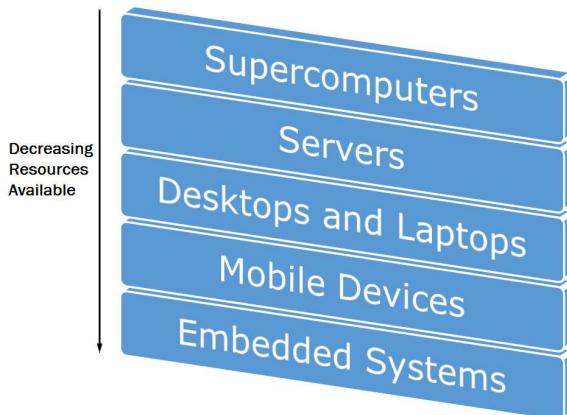
- Central Processing Unit (CPU), memories, input and output (I/O) devices
- Multiple buses are used to interconnect CPU and various peripherals, e.g. graphics card



8

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



9

Supercomputers and Servers

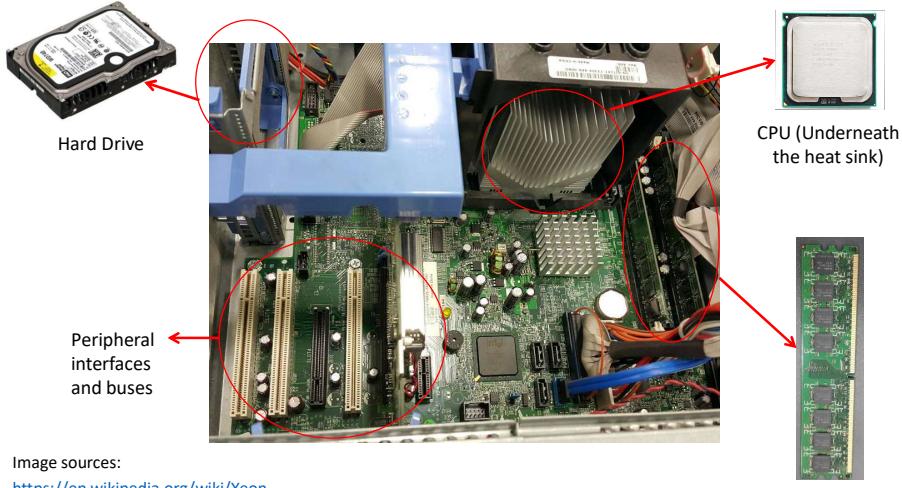
- Supercomputers are the most powerful computers around, often designed for special applications, such as weather forecast and earthquake monitoring
 - Many millions of processor cores (<https://www.top500.org/>)
 - At UoA we have a supercomputer cluster called NeSI
- Servers nowadays typically refer 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



10

Desktop Computers

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

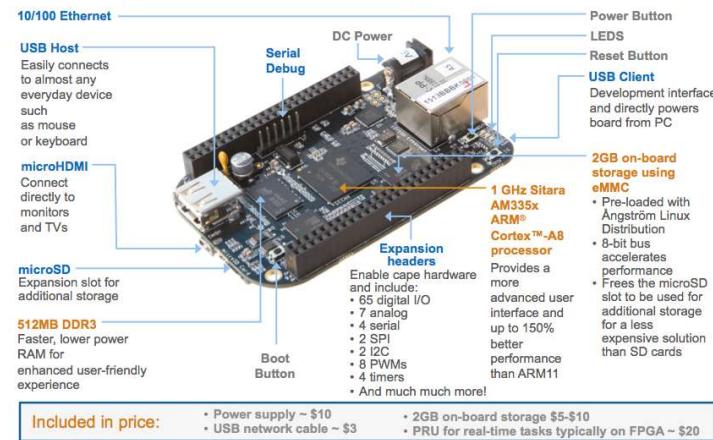


11

Single Board Computers

- Example: BeagleBone black

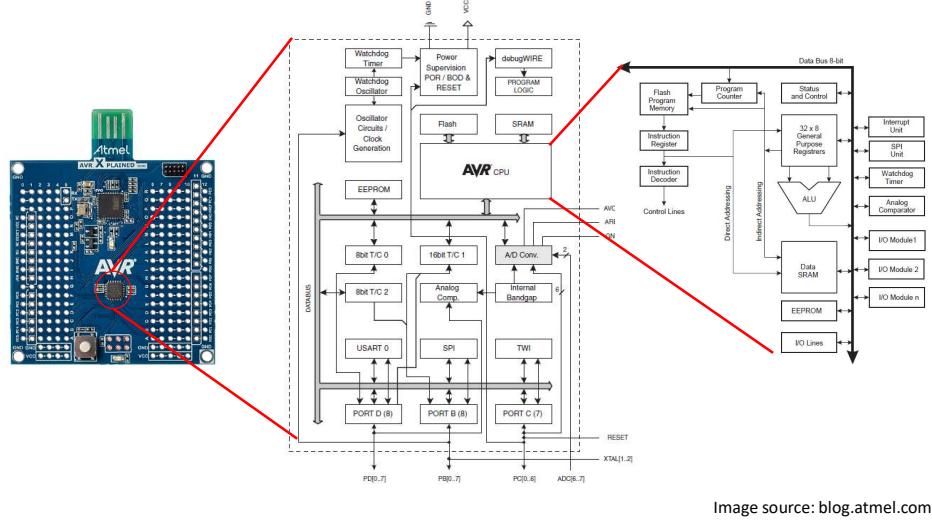
BeagleBone Black 1 GHz performance ready to use for \$45



12

Microcontrollers

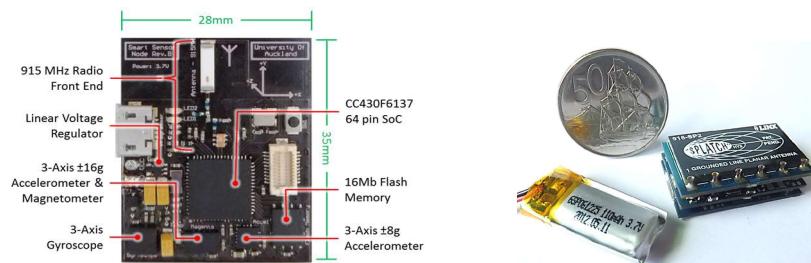
- Example: the Atmel Xplained Mini kit



13

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)



14

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”



15

COMPSYS201 – Part I Fundamentals of Computer Engineering

Combinational Logic Design

Dr. Kevin Wang
kevin.wang@auckland.ac.nz

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



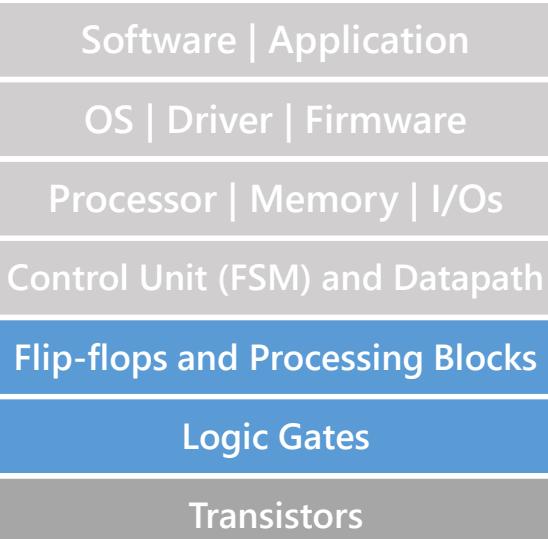
16

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

17

Modern Computer System Hierarchy



18

Boolean Algebra

- Traditional algebra
 - Variables represent real numbers
 - Operators operate on real numbers, return real numbers
- Boolean Algebra
 - Variables represent binary numbers (i.e. 0 or 1 only)
 - Operators return binary numbers
 - 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

19

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

Symbol

NOT

OR

AND

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 as logic gates

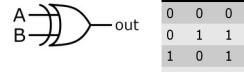
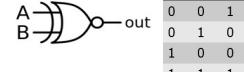
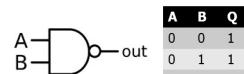
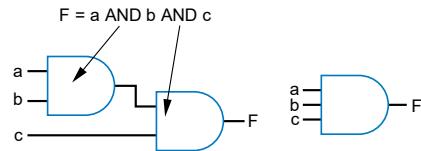
• These logic gates allow us to build digital circuits by doing math
(e.g. Boolean algebra)

- {AND, OR, NOT} is usually referred to as the basic gate set
 - All other gates (or Boolean functions) can be made from them

20

More Logic Gates

- NAND gate, detect anything but all 1s
 - $S = (AB)' = \text{NOT } (A \text{ AND } B)$
- NOR gate, detect all 0s
 - $S = (A+B)' = \text{NOT } (A \text{ OR } B)$
- XNOR gate, detect equality
- XOR gate, detect odd # of 1s
 - Useful for generating "parity" bit common for detecting data transmission errors
- Multiple inputs gate
 - Considered as cascaded two input gates



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

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

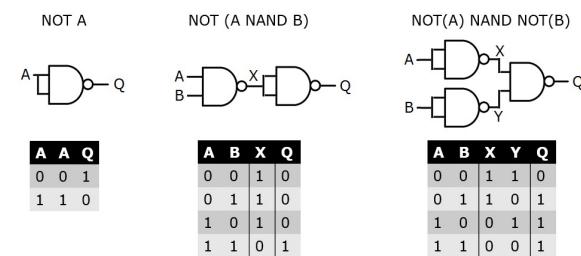
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

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

21

Functional Complete Gates

- A functionally complete gate set is a set of gates 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)



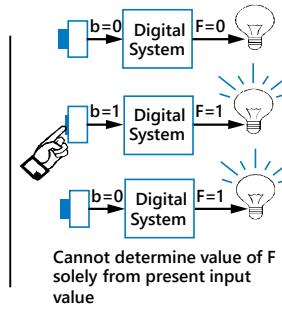
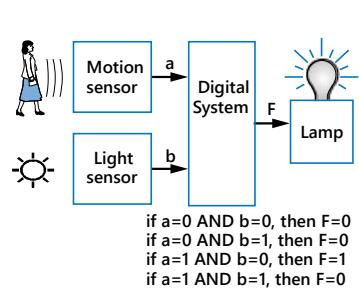
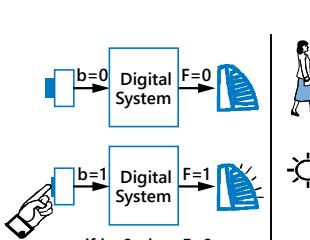
22

Combinational Circuits

- Combinational circuits

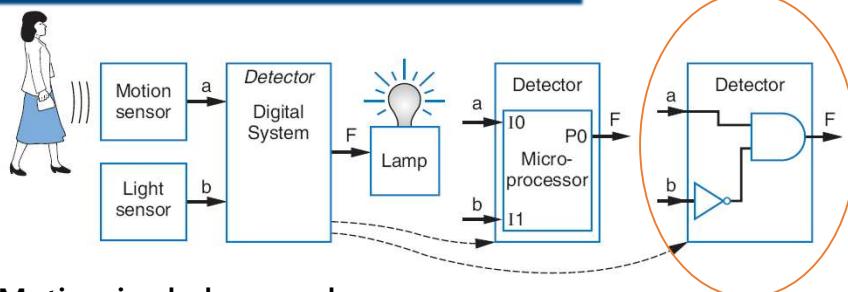
- Outputs depend solely on the combination of the present input values into the circuit

- In comparison to sequential circuits, which have "memory" that also impacts the outputs



23

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 $\text{NOT}(b)=1$))

- $F = a \text{ AND } \text{NOT}(b)$

- The above Boolean function represents the combinational circuit we want to design

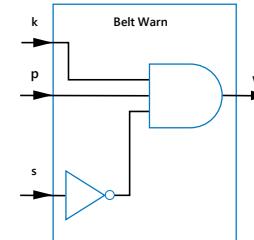
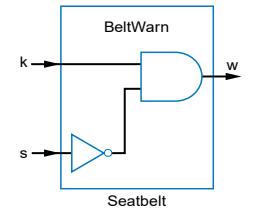
- Build using logic gates, AND and NOT

- Important: output F is produced solely based on the current inputs a and b

24

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
 - $w = \text{NOT}(s) \text{ AND } k$
- Convert equation to circuit
- Only illuminate warning light if person is seated ($p=1$), AND seat belt NOT fastened AND key inserted
 - $w = p \text{ AND } \text{NOT}(s) \text{ AND } k$



25

Boolean Algebra

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

26

Boolean Algebra Terminology

- Example equation: $F(a,b,c) = a'bc + abc' + ab + c$
- Variable
 - Represents a value (i.e. in true or complemented form, 0 or 1)
 - Three variables: a, b, and c
- Product term
 - Product of variables, for example, $a'bc$, abc' , ab , c
- Minterm
 - A product term of each and every input variable appears only once (either in true or complemented form)
- Sum of products
 - Equation written as OR of product terms only, as the example equation, " $F = (a+b)c + d$ " is not
- Sum term and Product of Sums
 - Sum of variables, e.g. $(a+b)$
 - Product of sums, e.g. $(a+b)(a+c')$

27

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!

Note: No need to memorise,
but need to know how to
apply these properties

28

Complement of a Function

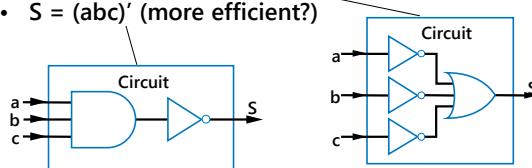
- Sometimes we might want to find the complement (inverse) of a function F
 - May be easier to understand and/or easier to design
 - Example: seat belt warning light is off if all seat belts are fastened
- De Morgan's Law: $(a + b)' = a'b'$ or $(ab)' = a' + b'$
- De Morgan's Law can also be applied for 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)

29

Example of De Morgan's Law

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

- Behaviour description of an aircraft toilet available 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



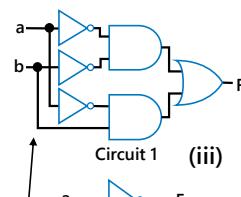
30

Representations of Combinational Logic

English 1: F outputs 1 when a is 0 and b is 0, or when a is 0 and b is 1

English 2: F outputs 1 when a is 0, regardless of b's value

(i)



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

Truth table

(iv)

Equation 1: $F(a,b) = a'b' + a'b$
Equation 2: $F(a,b) = a'$

(ii)

The function F

Circuit 1 (iii)

Circuit 2

- 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

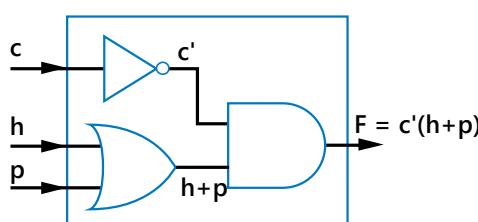
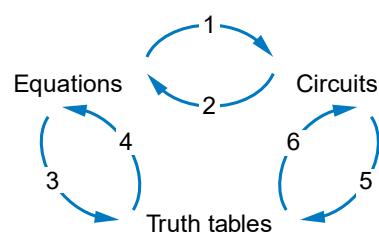
31

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



32

Truth Table Representation

- Exhaustively define value of F for each and every possible combination of input variable values

- 2-input function: 4 rows
- 3-input function: 8 rows
- 4-input function: 16 rows
- 5-input function: 32 rows
- n-input function: 2^n rows
- Each combination is a minterm!

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

a	b	F
0	0	
0	1	
1	0	
1	1	

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	

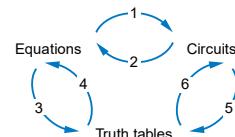
a	b	c	d	F
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

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	

33

Converting among Representations

- Truth table to equation (which we can then convert to circuit)
 - OR each product term that should output 1 (sum of products)



Inputs	Outputs	Term
a 0 0	F 1	$F = \text{sum of}$
a 0 1	F 1	$a'b'$
a 1 0	F 0	$a'b$
a 1 1	F 0	

$$F = a'b' + a'b$$

Q: Convert to equation

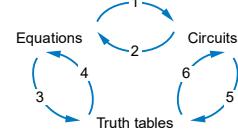
a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$F =$$

34

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	

Converting from Truth Table to Equation



Example:

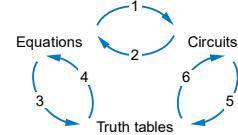
- Parity bit: Extra bit added to data, intended to enable detection of data transmission errors
 - e.g. errors can occur due to interference
- Even parity: Set a parity bit so the total number of 1s (data + parity) is even
 - e.g. if data is 001, parity bit is 1 → 0011
 - Want equation, but easier to 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

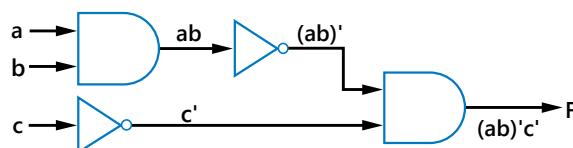
Convert to eqn. $P = a'b'c + a'bc' + ab'c' + abc$

35

Converting from Circuit to Truth Table



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



Inputs			ab	$(ab)'$	c'	Outputs
a	b	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

36

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 $F_1 = ab + a'$ is same function as $F_2 = a'b' + a'b + ab$, 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,

37

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 each and every input variable 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)

$$\begin{aligned} F &= ab+a' \text{ (already sum of products, but not sum of minterms)} \\ F &= \\ F &= \end{aligned}$$

38

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

$$H = abcde + abcde' + a'bcde + a'bcde(a' + c)$$

$$H = abcde + abcde' + a'bcde + a'bcdea' + a'bcdec$$

$$H = abcde + abcde' + a'bcde + a'bcde + a'bcde$$

$$H = abcde + abcde' + a'bcde$$

$$H = a'bcde + abcde' + abcde$$

- Homework: prove these two functions are equivalent using truth tables

39

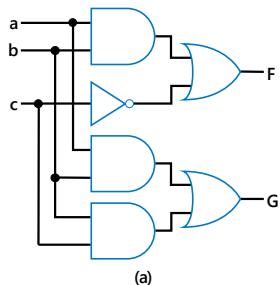
Compact Sum of Minterms Representation

- List each minterm as a number
 - Note: Each minterm corresponds to a row in truth table, representing a unique combination of input variable values
- 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:
 - $H = \sum m(15, 30, 31)$
 - $\sum m$ represents sum of minterms
 - "H is the sum of minterms 15, 30, and 31"

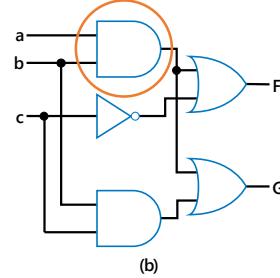
40

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

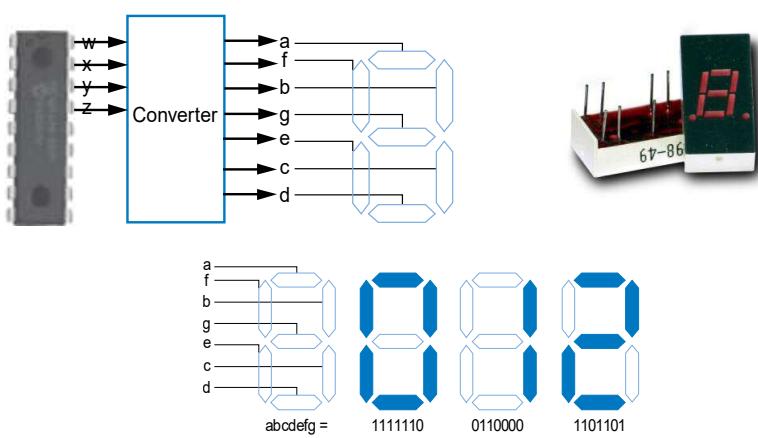


Option 2: Shared gates

41

Multiple-Output Example

BCD (Binary Coded Decimal) to 7-Segment Converter



42

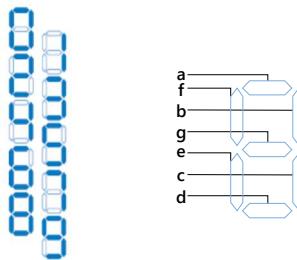
21

Multiple-Output Example

BCD to 7-Segment Converter

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

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



$$a = w'x'y'z' + w'x'yz' + w'x'yz + w'xy'z + w'xyz' + w'xyz + wx'y'z' + wx'y'z$$

$$b = w'x'y'z' + w'x'y'z + w'x'yz' + w'x'yz + w'xy'z' + w'xy'z + wx'y'z' + wx'y'z$$

...

43

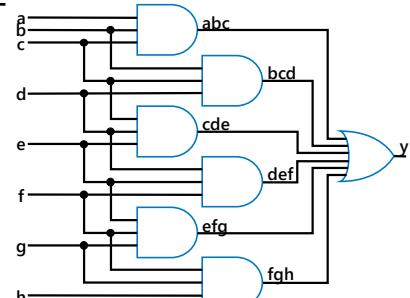
Combinational Logic Design Process

Steps	Description
Step 1: Capture behaviour	Create a truth table or Boolean equation, 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 equations (optional)	This step is only necessary if you captured the behaviour using a truth table instead of an equation. Create an equation for each output by ORing all the minterms equal to 1 for that output. Simplify the equation 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.

44

Combinational Logic Design

- Example: Three 1s Pattern Detector
- Detect three consecutive 1s in an 8-bit input: abcdefgh
 - $00011101 \rightarrow 1$
 - $10101011 \rightarrow 0$
- 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



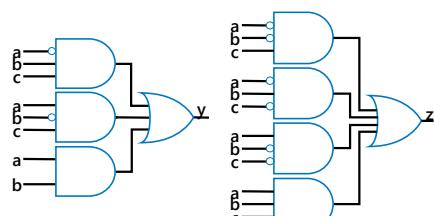
45

Combinational Logic Design

- Example: Number of 1s Counter
- Output in binary on two outputs yz the # of 1s on a 3-bit input
 - $010 \rightarrow 01$ (a single 1)
 - $101 \rightarrow 10$ (two 1s)
 - $000 \rightarrow 00$ (No 1)
 - Why 2-bit output?
- Step 1: Capture the function
 - Truth table or equation?
 - Truth table is straightforward
- Step 2: Create Boolean functions
 - $y = a'b'c + ab'c' + abc'$
 - $z = a'b'c + a'b'c' + ab'c' + abc$
 - Optional: Let's simplify y:

$$y = a'b'c + ab'c + ab(c' + c) \\ = a'b'c + ab'c + ab$$
- 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



46

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 circuit design process: Translate from equation (or table) to circuit through well-defined steps

47

COMPSYS201 – Part I Fundamentals of Computer Engineering

Combinational Logic Components

Dr. Kevin Wang
kevin.wang@auckland.ac.nz

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



48

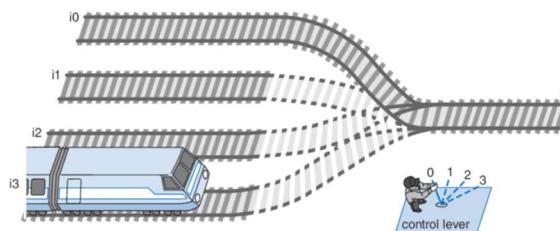
Introduction

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

49

Multiplexer (Mux)

- Multiplexer: Routes one of its N data inputs to its one data output, based on binary value of select inputs
 - 4-to-1 mux \rightarrow needs 2-bit select inputs to index (or uniquely identify) which one of the 4 data inputs to route to the 1 output
 - 8-to-1 mux \rightarrow 3-bit select inputs, 8 data inputs, 1 data output
 - k select inputs (or select bits) \rightarrow can index up to 2^k data inputs
 - Data input/output can be of any width, but must be the same width
 - For example, 8-bit input must be routed to 8-bit output
 - One of the most popular combinational building blocks

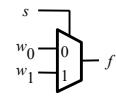


50

Mux Internal Design

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

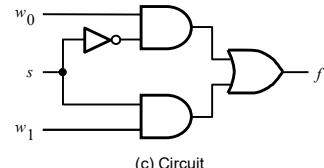
2x1 mux: $f = s'w_0 + sw_1$



(a) Graphical symbol

s	w_0	f
0	0	0
1	1	1

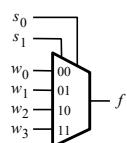
(b) Truth table



(c) Circuit

4x1 mux:

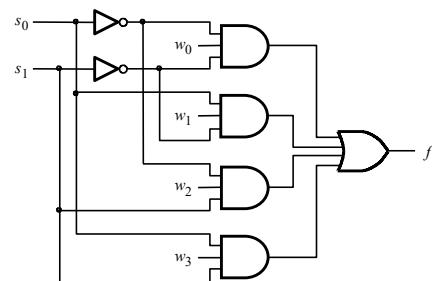
$$f = s_0's_1'w_0 + s_0's_1w_1 + s_0s_1'w_2 + s_0s_1w_3$$



(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

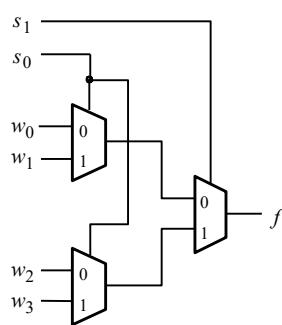


(c) Circuit

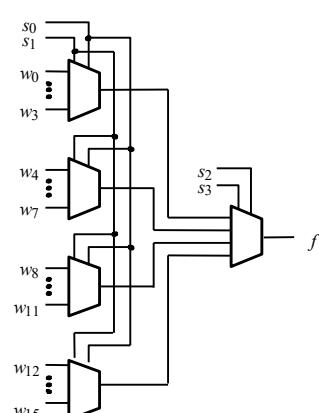
51

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



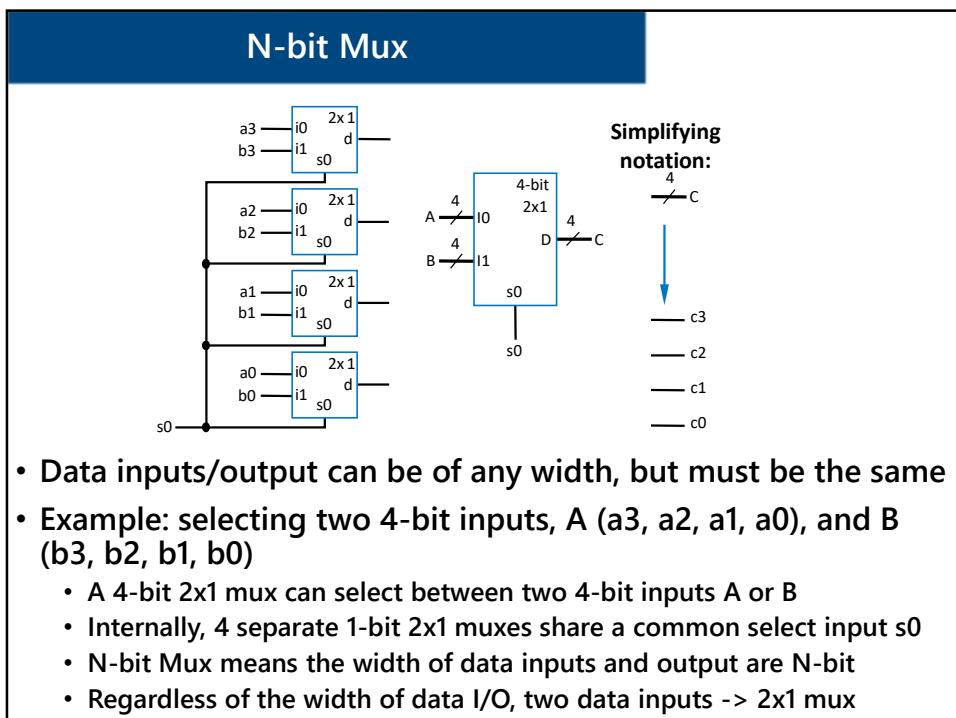
Using three 2-to-1 muxes to build a 4-to-1 mux



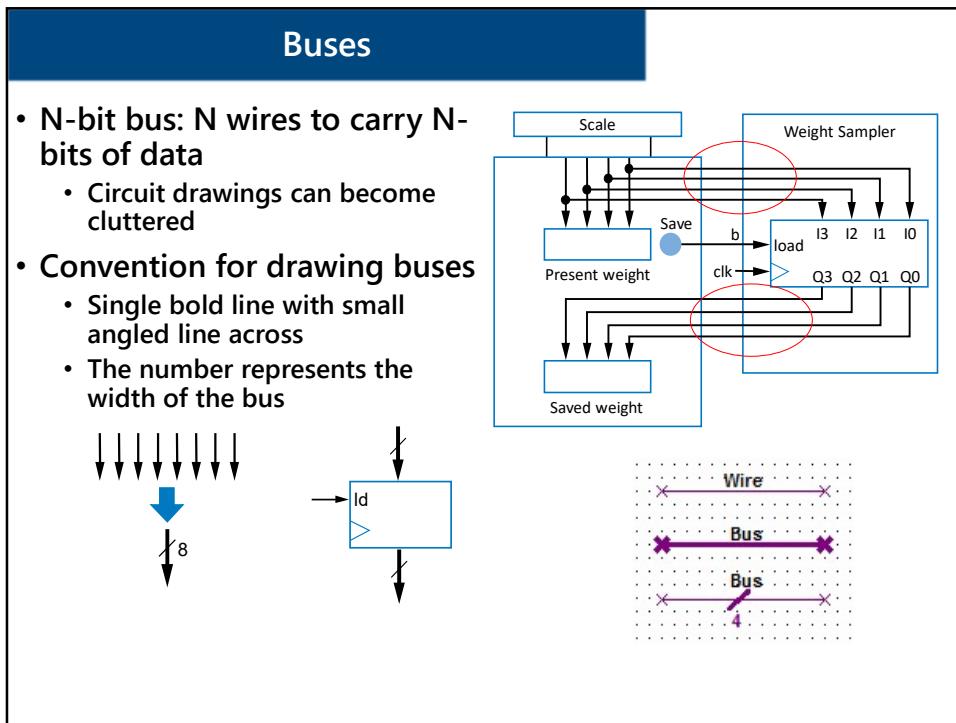
Using five 4-to-1 muxes to build a 16-to-1 mux

52

26



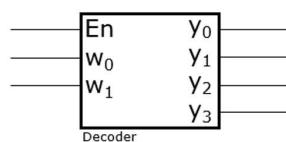
53



54

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 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 can 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)

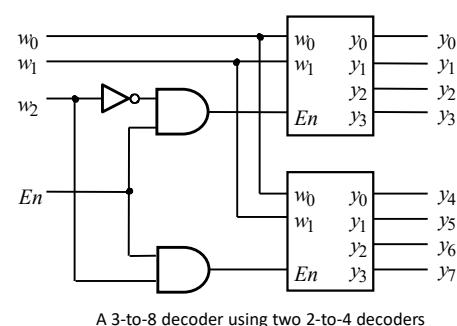
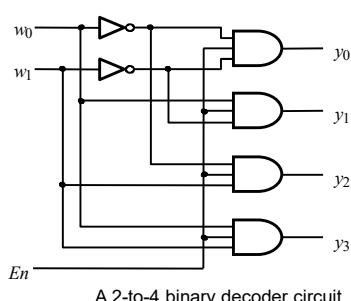


En	w ₀	w ₁	y ₀	y ₁	y ₂	y ₃
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	X	X	0	0	0	0

55

Decoders

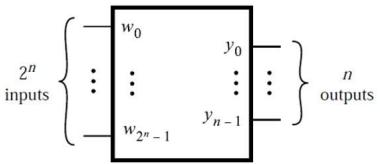
- Decoders can be designed using the combinational circuit design process, or modular design approach using smaller decoder components



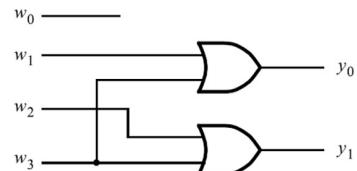
56

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?
 - Can this happen in practice?
 - If it does happen, how to resolve this?



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



57

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 indicate the active input that has the highest priority
 - Starting from the left most column assigning inputs from higher to lower priority
- When an input with a higher priority is asserted, the other inputs with the lower priorities are ignored (i.e. assigned with "don't care" value)

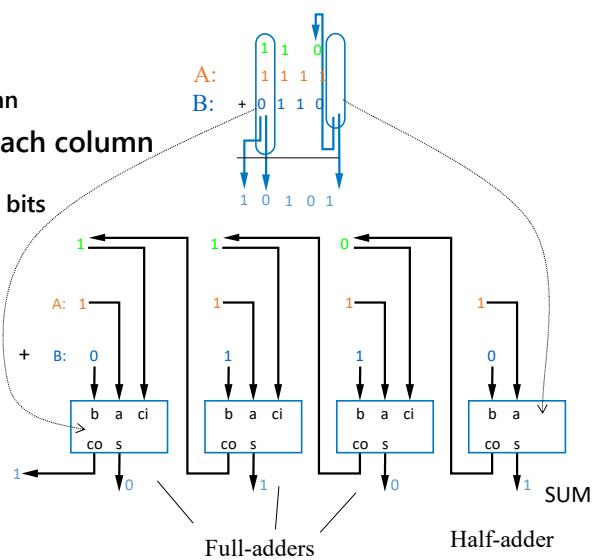
w_3	w_2	w_1	w_0	y_1	y_0	valid
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Assume w_0 has the lowest priority and w_3 the highest

58

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



59

Half-Adder

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

Step 1: Capture the function

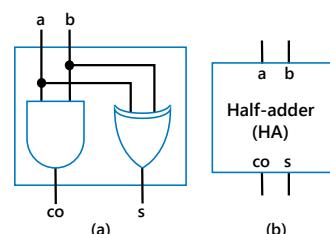
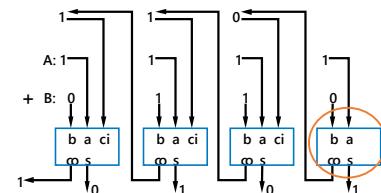
Inputs		Outputs
a	b	co s
0	0	0 0
0	1	0 1
1	0	1 0
1	1	1 1

Step 2: Create equations

$$co = ab$$

$$s = a'b + ab' \text{ (same as } s = a \oplus b)$$

Step 3: Implement as circuit



60

Full-Adder

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

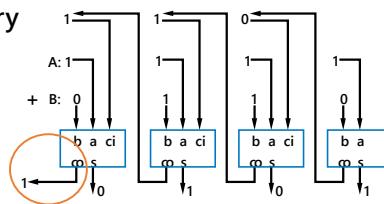
Step 1: Capture the function

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		

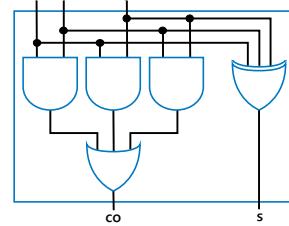
Step 2: Create equations

$$\begin{aligned} co &= a'b'c + ab'c' + abc' + abc \\ co &= a'b'c + abc + ab'c + abc' + abc \\ co &= (a'+a)bc + (b'+b)ac + (c'+c)ab \\ co &= bc + ac + ab \end{aligned}$$

$$\begin{aligned} s &= a'b'c + a'bc' + ab'c' + abc \\ s &= a'(b'c + bc') + a(b'c' + bc) \\ s &= a'(b \oplus c)' + a(b \oplus c) \\ s &= a \oplus b \oplus c \end{aligned}$$



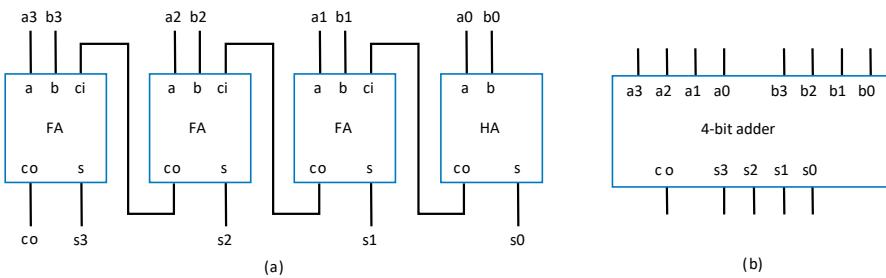
Step 3: Implement as circuit



61

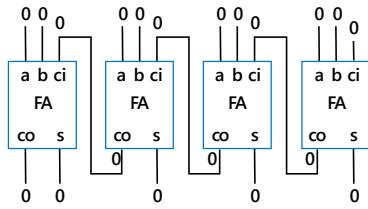
Carry-Ripple Adder

- 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
 - Any drawback?

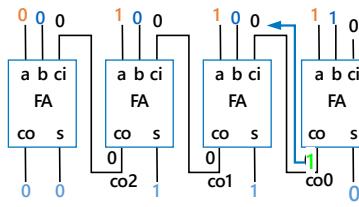


62

Carry-Ripple Adder's Behaviour



Assume all inputs initially 0



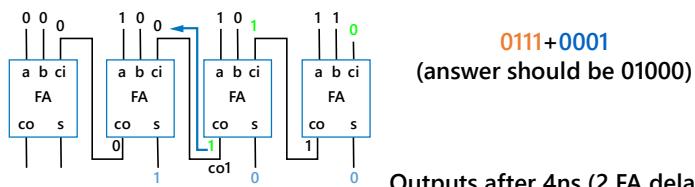
$0111 + 0001$
(answer should be 01000)

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)

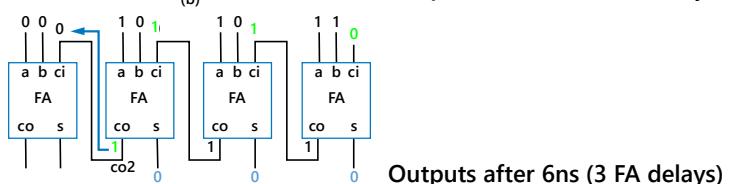
63

Carry-Ripple Adder's Behaviour

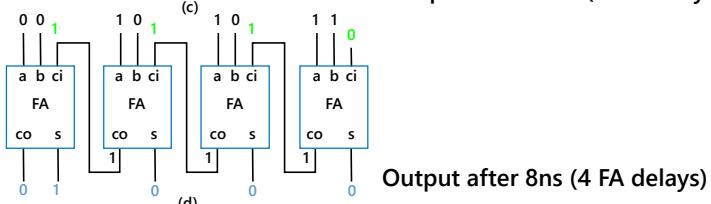


$0111 + 0001$
(answer should be 01000)

Outputs after 4ns (2 FA delays)



Outputs after 6ns (3 FA delays)



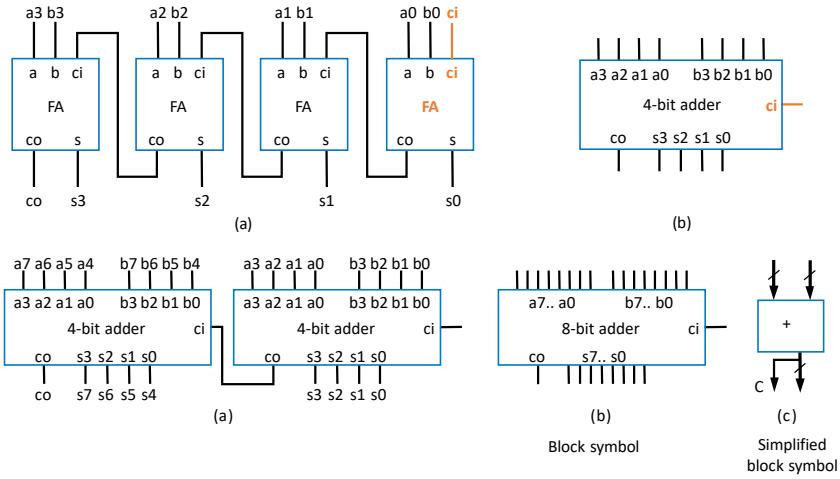
Output after 8ns (4 FA delays)

Correct answer appears after 4 FA delays

64

Modular Design - Cascading Adders

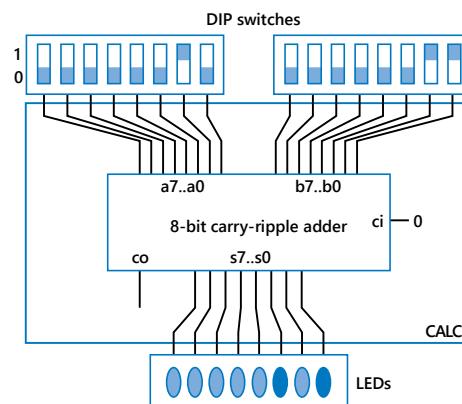
- Using full-adder instead of half-adder for first bit, we can include a “carry in” bit in the addition
 - Smaller adders with carry in can be connected to form bigger adders



65

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)
 - Combinational circuit?

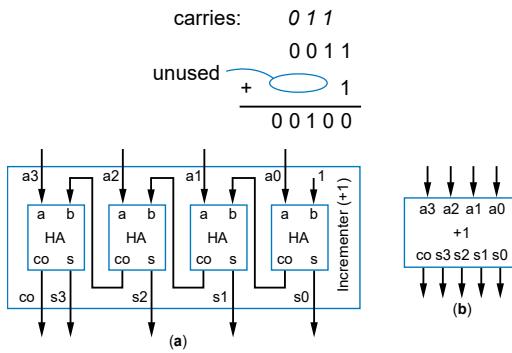


66

Incremente

- Adds 1 to input A

Inputs				Outputs				
a3	a2	a1	a0	c0	s3	s2	s1	s0
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0	0
0	1	0	0	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	1	0	0	0	1	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	1	1
1	0	1	0	0	1	1	0	0
1	0	1	1	0	0	1	1	0
1	1	0	0	0	1	1	0	1
1	1	0	1	0	0	1	1	0
1	1	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0

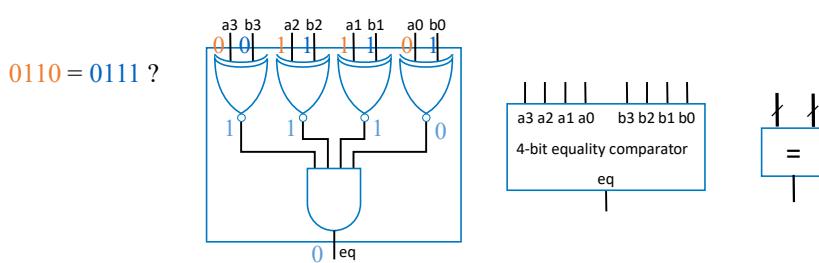


- Very common operation, for example, in loops
- Could design using combinational design process, but easier to design using the carry-ripple adder concept with half-adders only

67

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
 - Very common in conditional operations
- Example: 4-bit equality comparator with inputs A and B
 - $a_3 = b_3, a_2 = b_2, a_1 = b_1, a_0 = b_0$
 - $eq = (a_3b_3 + a_3'b_3') * (a_2b_2 + a_2'b_2') * (a_1b_1 + a_1'b_1') * (a_0b_0 + a_0'b_0')$
 - Note that function inside parentheses is XNOR
 - $eq = (a_3 \text{ xnor } b_3) * (a_2 \text{ xnor } b_2) * (a_1 \text{ xnor } b_1) * (a_0 \text{ xnor } b_0)$



68

Magnitude Comparator

- N-bit magnitude comparator:
Two N-bit binary inputs A and B, three 1-bit outputs for $A > B$, $A = B$, or $A < B$
- How to design? Consider comparing two decimal values
 - How to determine $25 > 15$ or $25 > 23$?
 - Compare each pair of digits, starting from the left most (or the most significant digit)
- It is the same for binary numbers
 - Start from comparing the most significant bit pair
 - If not equal, the number with the bigger bit (i.e. bit=1) has the bigger value
 - If equal, compare the next bit pair
 - Repeat step 2 and 3 until you reach a conclusion, or find two numbers are equal

Example: compare A (a₃, a₂, a₁, a₀) and B (b₃, b₂, b₁, b₀)

- First compare a₃ and b₃. If equal, compare a₂ and b₂, and so on
- Stop if comparison not equal (the two bits are 0 and 1, or 1 and 0)—whichever of A or B has the 1 is therefore greater. If we never see an unequal bit pair, then A=B

$$A=1011 \quad B=1001$$

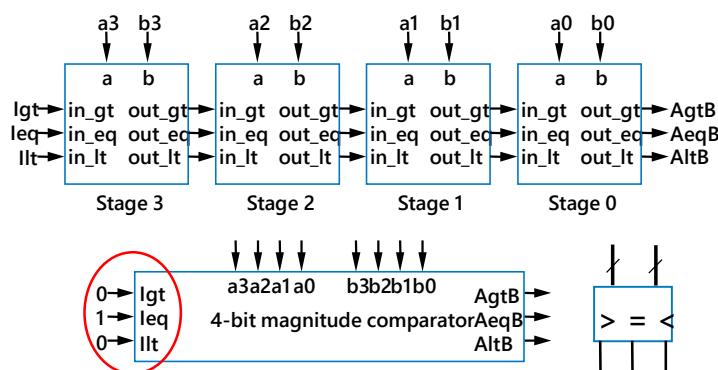
1011	1001	Equal
1011	1001	Equal
1011	1001	Not equal

So A > B

69

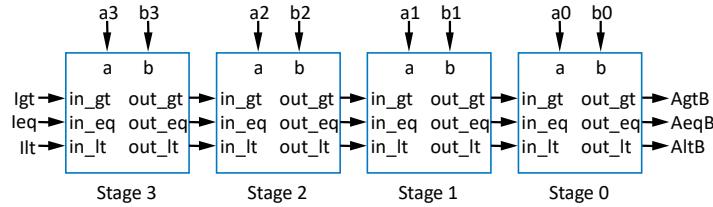
Magnitude Comparator

- Similar to carry-ripple adder, we do comparison step by step
 - Start from left, compare each bit pair, pass results to the right
 - Each bit pair is called a stage
 - First stage MUST have default input values (lgt=1 and llt=llt=0) to trigger the comparison of the two data inputs (e.g. a₃ and b₃ in the example below)



70

Magnitude Comparator



- For each stage, we need to determine 3 outputs:

- $out_{gt} = in_{gt} + (in_{eq} * a * b')$
 - $A > B$, if in_{gt} is already determined in a higher stage, or
 - $(in_{eq} * a * b')$, if higher stages all equal and in this stage $a=1$ and $b=0$
- $out_{lt} = in_{lt} + (in_{eq} * a' * b)$
 - $A < B$, if in_{lt} is already determined in higher stage, or
 - $(in_{eq} * a' * b)$, if higher stages all equal and in this stage $a=0$ and $b=1$
- $out_{eq} = in_{eq} * (a \text{ XNOR } b)$
 - $A=B$ (so far) is already determined in higher stages and in this stage $a=b$ too

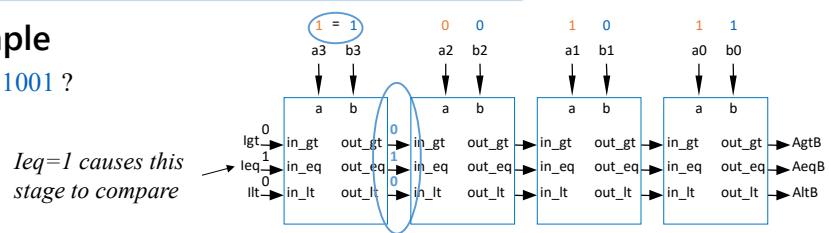
71

Magnitude Comparator

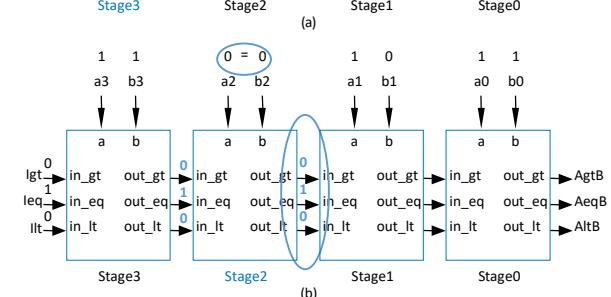
Example

$1011 = 1001$?

$Ieq=1$ causes this stage to compare



- Final answer appears on the right
- Takes time for answer to "ripple" from left to right
- Thus, this is also in "carry-ripple style", like the carry-ripple adder
- Try completing the example on the right



72

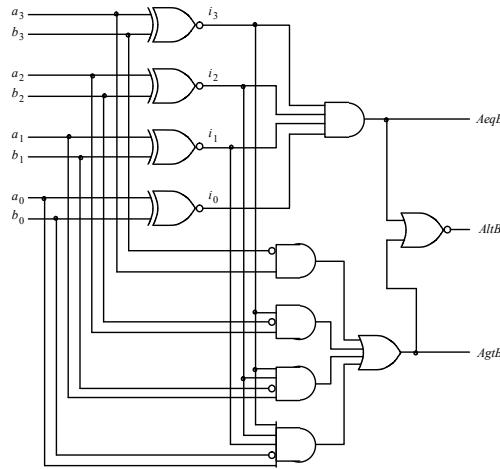
Alternative 4-bit Comparator

- There are alternative ways to design a 4-bit magnitude comparator (e.g. with less resources or faster performance)
- This is not in carry-ripple style

$$AeqB = i_3 i_2 i_1 i_0 \quad i_k = \overline{a_k \oplus b_k}$$

$$AgtB = a_3 \overline{b_3} + i_3 a_2 \overline{b_2} + i_3 i_2 a_1 \overline{b_1} + i_3 i_2 i_1 a_0 \overline{b_0}$$

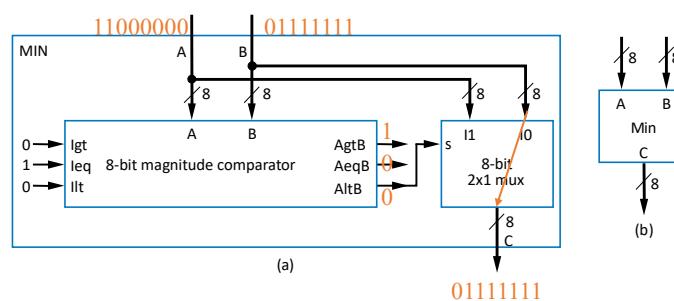
$$AltB = \overline{AeqB + AgtB}$$



73

Magnitude Comparator Example: Minimum of Two Numbers

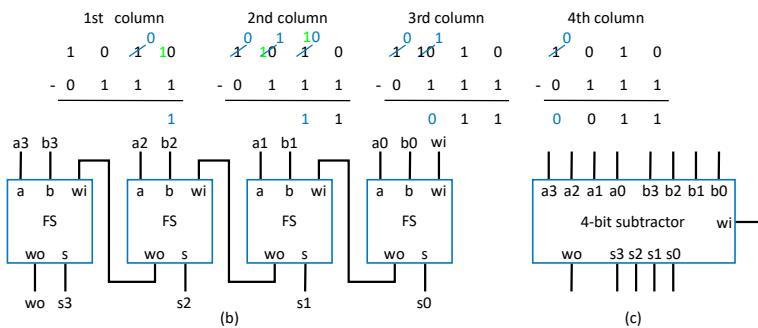
- Design a combinational component that computes the minimum of two 8-bit numbers
 - Solution: Use an 8-bit magnitude comparator and an 8-bit 2x1 mux
 - If $A < B$, $C=A$, else, $C=B$
 - If $A < B$, pass A through mux, else, pass B



74

Subtractor and Signed Numbers

- Can build subtractor as we built the carry-ripple adder
 - Mimic subtraction by hand
 - Compute the borrows from columns on left
 - Use full-subtractor component:
 - wi is borrow to column on right, wo borrow from column on left
 - Possible, but borrow is more difficult to design than carry
 - There is an easier way to design subtractor, but first we need to understand signed numbers representations



75

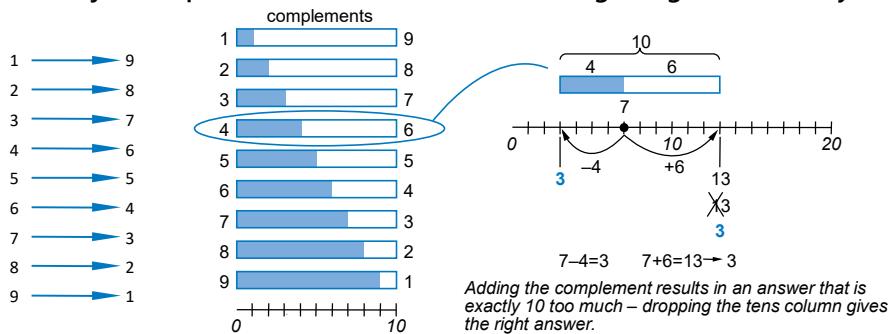
Representing Negative Numbers

- So far, we have only considered unsigned binary numbers
- How to represent negative numbers in binary?
- Signed-magnitude
 - Use the leftmost bit as a sign bit
 - So -5 would be:
 - 1101 using four bits or 10000101 using eight bits, where the left most bit is the sign (1 means negative) and the other bits are the magnitude ($101_2 = 5_{10}$)
 - This representation is not very efficient and non-intuitive to carry out arithmetic operations
- Two's complement
 - A complement with respect to 2^N for an N-bit binary number
 - $001 + 111 = 1000$, for a 3-bit binary 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

76

Ten's Complement

- Before introducing two's complement, let's consider ten's complement
 - A complementation for a decimal number is 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



77

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 (that can only take the value of 0 or 1), it turns out that the two's complement can be computed easily
 - Two's complement of 011 is 101, because $011 + 101 = 1000$
 - 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: $1100 + 1 = 1101$

78

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
 $0001 \rightarrow 1110+1 = 1111$
 - -2: 0010 $\rightarrow 1101+1 = 1110 \dots$
 - -8: 1000 $\rightarrow 0111+1 = 1000$
 - So -1 to -8: 1111 to 1000
 - Leftmost bit still indicates the sign, known as sign bit. 1 means negative
- Value range of signed and unsigned N-bit binary 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

79

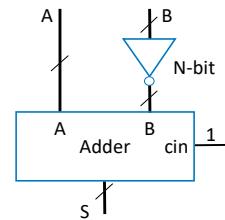
Two's Complement

- 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 for negative numbers can be found by taking two's complement again
 - e.g. 1110 \rightarrow sign bit is 1, so it is a negative number
 $\rightarrow = -(0001+1)$
 $\rightarrow = -0010 = -2$
 - e.g. 1000 \rightarrow sign bit is 1, so it is a negative number
 $\rightarrow = -(0111+1)$
 $\rightarrow = -1000 = -8$

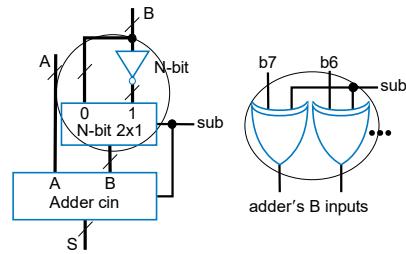
80

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
 - $A - B = A + (-B)$
 - $= A + (\text{two's complement of } B)$
 - $= A + \text{invert_bits}(B) + 1$



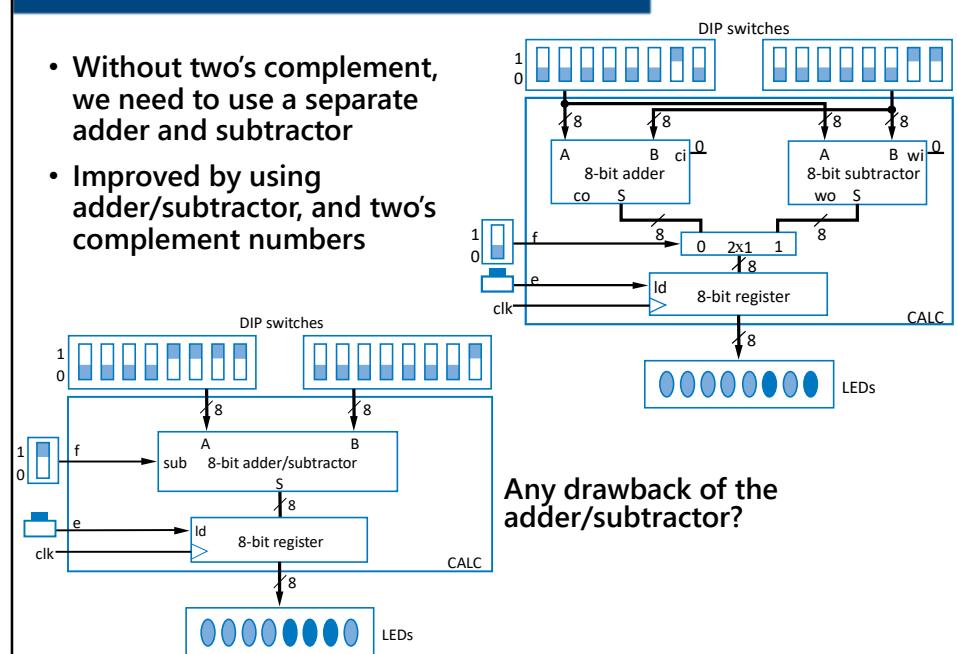
- Adder/subtractor: control input (sub) determines whether add or subtract
 - Use 2x1 mux – sub input decides to pass either B or B'
 - Alternatively, use XOR gates – if sub input is 0, B's bits pass through; if sub input is 1, XOR inverts B's bits



81

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



82

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
 - In unsigned binary arithmetic, this is represented by carry
 - In two's complement arithmetic, this is called overflow, i.e. the number is either too large or too small
 - For example, 4-bit two's complement addition of 0111+0001 (7+1=8)
 - But a 4-bit two's complement number can only represent -8 to 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

83

Detecting Overflow: Method 1

- For two's complement addition, 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 different, overflow is impossible, i.e. the result of addition will always be within the representable range if two operands are of different signs
 - Adding positive and negative can't exceed largest magnitude positive or negative
- Simple overflow detection circuit for 4-bit adder
 - $\text{overflow} = a_3'b_3's_3 + a_3b_3s_3'$
 - Include "overflow" output bit on adder/subtractor

sign bits

$\begin{array}{r} 0 & 1 & 1 & 1 \\ + 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \end{array}$ <p>overflow (a)</p>	$\begin{array}{r} 1 & 1 & 1 & 1 \\ + 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 \end{array}$ <p>overflow (b)</p>	$\begin{array}{r} 1 & 0 & 0 & 0 \\ + 0 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \end{array}$ <p>no overflow (c)</p>
---	---	--

84

Detecting Overflow: Method 2

- Even simpler method: Detect difference between carry-in to sign bit and carry-out from sign bit
 - Yields simpler circuit: $\text{overflow} = c_3 \text{ xor } c_4$

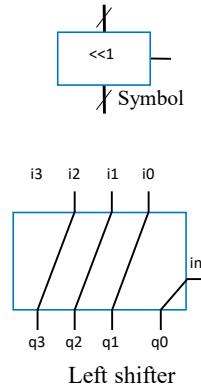
$ \begin{array}{cccc} 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{array} $	$ \begin{array}{ccccc} 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array} $	$ \begin{array}{ccccc} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array} $
$+ 0 \quad 0 \quad 0 \quad 1$ <hr/> (a)	$+ 1 \quad 0 \quad 0 \quad 0$ <hr/> (b)	$+ 0 \quad 1 \quad 1 \quad 1 \quad 1$ <hr/> (c)

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

85

Shifters

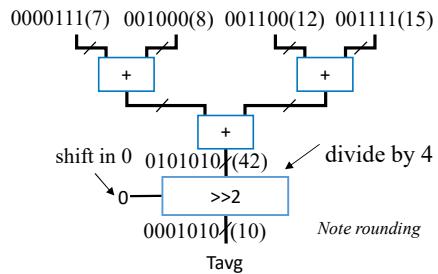
- Shifting (e.g. left shifting 0011 yields 0110) is useful for:
 - Bit manipulation
 - Shift left once is same as multiplying by 2
 - e.g. 0011 (3) becomes 0110 (6)
 - Why? Essentially appending a 0
 - Multiplying a decimal number by 10 is the same as appending a 0, i.e. by shifting left 55 becomes 550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



86

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



87

Strength Reduction

- What is strength reduction?
 - Multiplication/division are costly, which are referred to as "strong" instructions/operations
 - Strength reduction is a transformation to convert and replace strong instructions with weaker ones
 - e.g. $2*C$ can be reduced to a left shift of 1-bit
- Many multiplications are not by power of 2 constants
 - e.g. by 5 or 10
 - We can use a multiplier component, but shifters are small and fast
- Replace multiplication with shifts and adds
 - Operator strength reduction (multiply is "stronger" than shift/add)
 - e.g. $5*C \rightarrow 4*C + C$ ($4*C$ same as $C<<2$)

88

Strength Reduction

- Replacing division with shifts and adds is slightly harder
 - Approximate division as a fraction
 - e.g. $C/5 = 0.20*C$, approx. equals $(102/512)*C = 0.199*C$
 - The fraction must have its denominator as a power of 2 value, which can be reduced to right shift
 - Based on the picked denominator, find out the numerator value
 - Then break down the numerator into sum of power of 2 values
 - $102 = 62+32+4+2$
 - $(102/512)*C = C*(64+32+4+2)/512 = (C*64 + C*32 + C*4 + C*2)/512$
 - $C/5 = ((C<<6) + (C<<5) + (C<<2) + (C<<1)) >> 9$
- Strength reduction v.s. multiplier
 - A strength reduced circuit can only perform a specific calculation, multiplier and divider is more flexible (can work with different numbers)
 - A strength reduced circuit is more resource efficient

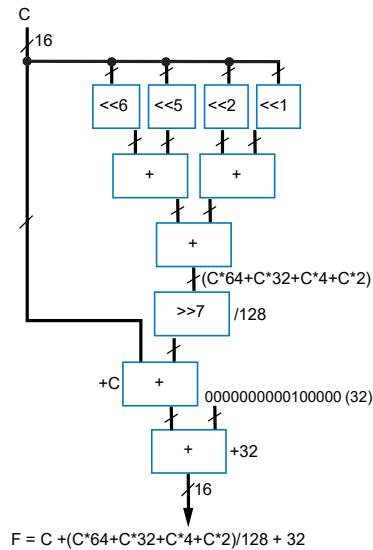
89

Strength Reduction Example

- Celsius to Fahrenheit Converter
 - $F = C*9/5 + 32$
 - $F = C + C*4/5 + 32$
 - $F = C + 4*C*1/5 + 32$
 - $F = C + 4*(C*64+C*32+C*4+C*2)/512 + 32$
($1/5$ is the same as the previous example)
 - $F = C + (C*64+C*32+C*4+C*2)/128 + 32$

Note:

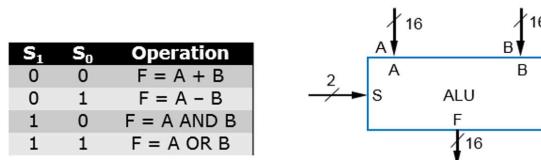
- Uses wider buses to avoid losing bits during shifts
- Determine biggest possible internal values, set width accordingly
- Do divisions as late as possible to reduce rounding errors



90

Arithmetic Logic Unit

- One of the most common combinational logic components in modern computing device that can do
 - Arithmetic operations (add, subtract etc.)
 - Logic operations (AND, OR, NOT etc.)
 - Shift operations
 - The available operations depend on individual ALU design
 - For example:



91

Learning Outcomes

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

92

COMPSYS201 – Part I

Fundamentals of Computer Engineering

Sequential Logic Design – Memory elements

Dr. Kevin Wang
kevin.wang@auckland.ac.nz

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



93

Introduction

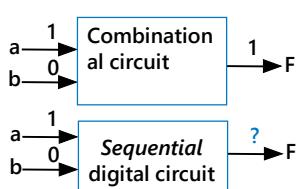
- Bit storage:
 - Latches and
 - Flip-flop
- Timing characteristics of flip-flop
- Combine flip-flops to build multi-bit storage – register
- Combining multiplexer with register to form different variant of registers
 - Shift register
 - Multifunction register

94

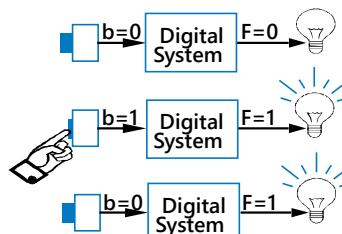
Sequential Circuits

- Sequential circuit

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



Must know sequence of past inputs to know output

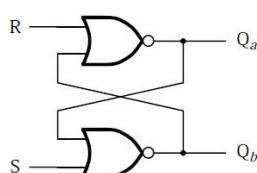


Cannot determine the value of F solely from present input value

95

Basic SR-Latch

- If $S=1, R=0$, then output Q_a is set
 - Sometimes Q_a is just referred to as Q , the non-inverted output
- If $S=0$ and $R=1$, then output Q_b is set
 - Sometimes Q_b is just referred to as Q' , the inverted output
- If $S=R=0$, the circuit maintains its state
 - i.e. Q_a and Q_b will maintain their values
- $S=R=1$ is a forbidden state, which can not be maintained
 - Both outputs will be forced to 0
 - What happens if the inputs go from $S=R=1$ to $S=R=0$?

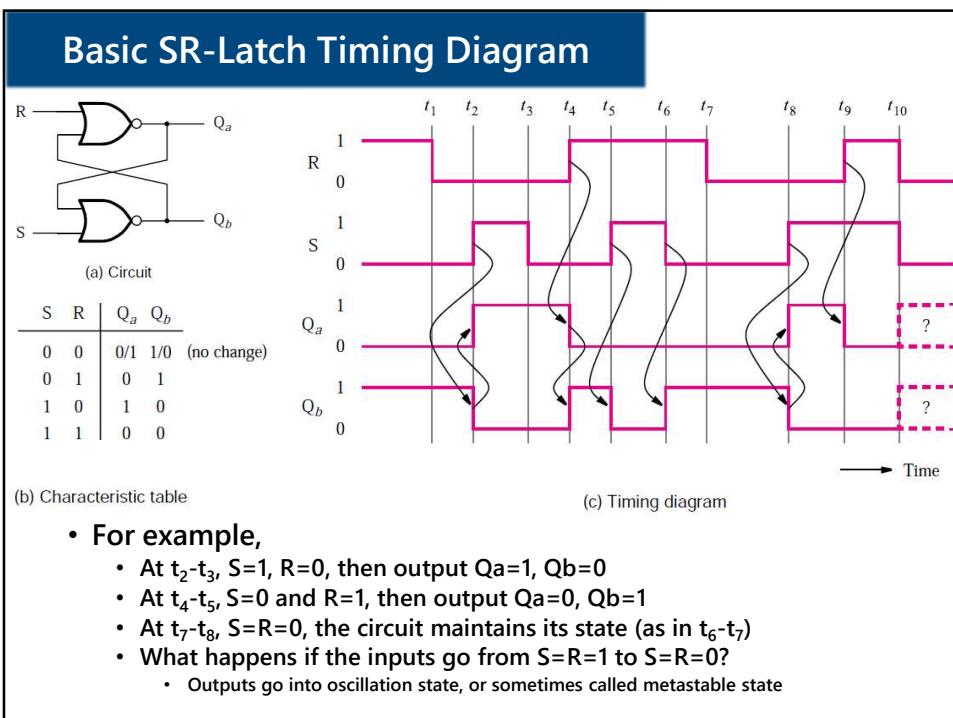


(a) Circuit

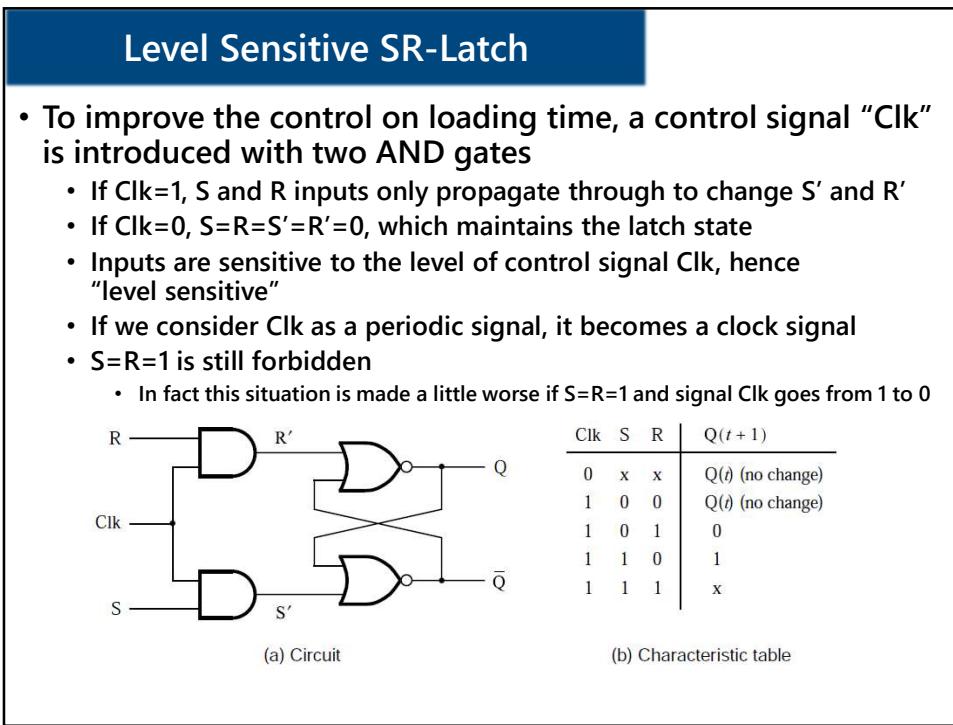
S	R	Q_a	Q_b
0	0	0/1	1/0
0	1	0	1
1	0	1	0
1	1	0	0

(b) Characteristic table

96

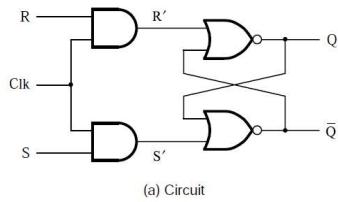


97



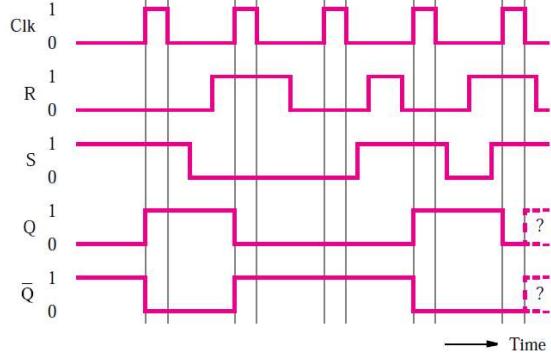
98

Level Sensitive SR-Latch Timing Diagram



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

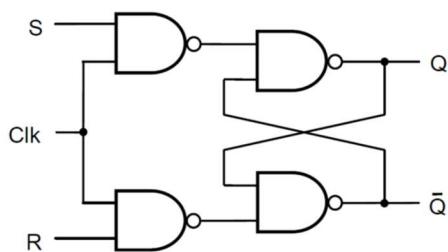
(b) Characteristic table



Note: With sequential circuits and memory elements, there is a notion of time, which is shown in the characteristic table

99

Level Sensitive SR-Latch with NAND gate

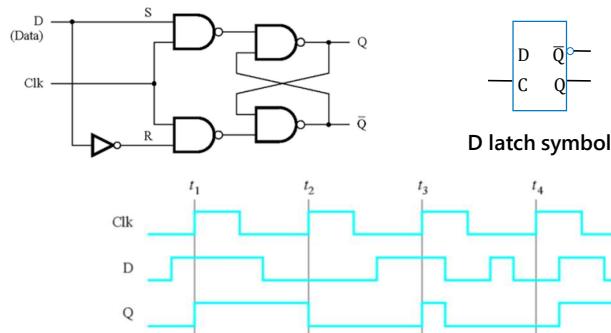


Clk	S	R	$Q(t+1)$
0	x	x	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

100

Level Sensitive D-Latch

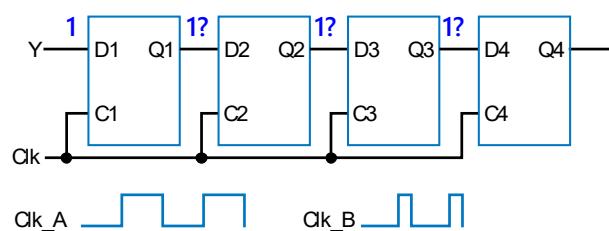
- SR latch requires careful design to ensure SR=11 never occurs
 - Metastability should be avoided at all times
- D latch relieves designers from that burden
 - Inserted inverter ensures R is always the opposite of S
 - Still sensitive to the level of control signal C (When C=1, Q follows the present D value; when C=0, Q remains at the last seen D value when C=1)



101

Problem with Level-Sensitive D Latch

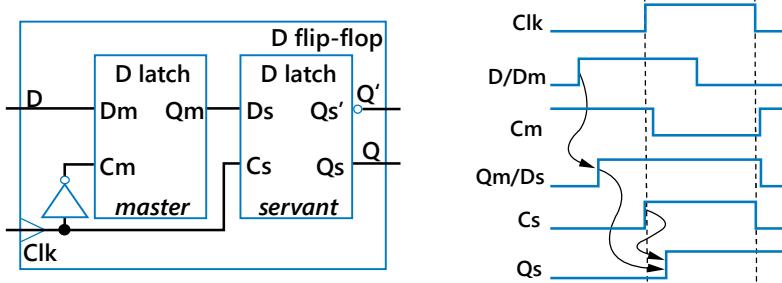
- D latch still has a problem (as does SR latch)
 - When C=1, how many latches will a signal travel through when connected in a cascaded fashion?
 - Depends on how long C=1 is for and the internal propagation delay of each latch
 - Clk_A – signal may travel through many latches
 - Clk_B – signal may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



102

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)
 - $\text{Clk} = 0$ – master enabled, loads D, appears at Q_m . Servant disabled
 - $\text{Clk} = 1$ – master disabled, Q_m stays same. Servant latch enabled, loads D_s (or Q_m), appears at Q_s
 - Thus, value at D “appears” like it is only stored when Clk changes from 0 to 1 when we observe just the output at Q_s



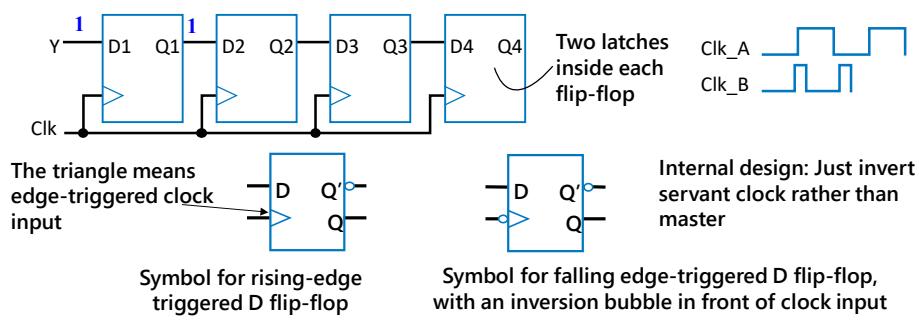
103

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 in one clock cycle (or period), regardless of Clk signal width
 - Why? See the example below:

All four flip-flops are loaded simultaneously at the same rising edge of Clk – then all four flip-flops stop loading also at the same time (synchronously). Doesn't matter how long Clk is 1

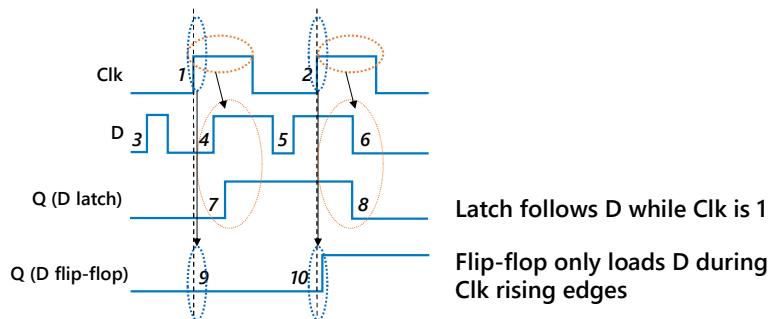
The current value Y will be loaded at D1 at one rising edge, but will not appear at Q1 fast enough to be loaded by D2 at the same rising edge



104

D Latch vs. D Flip-Flop

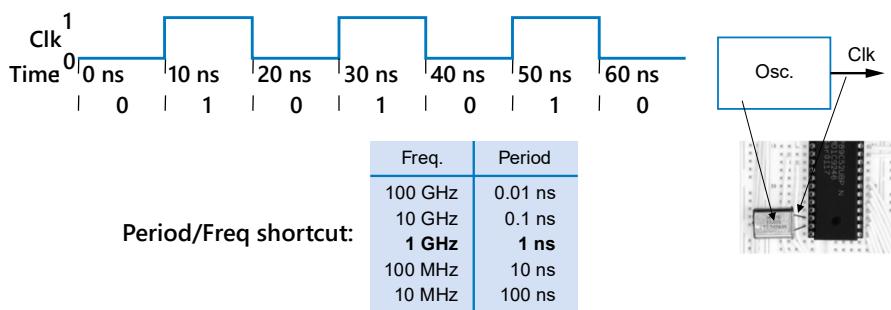
- Latch is level-sensitive
 - Stores D when C=1 or C=0 (depending on the design)
- Flip-flop is edge triggered
 - Stores D when C changes 0 to 1 (rising edge) or 1 to 0 (falling edge)
- Comparing behaviour of latch and flip-flop:



105

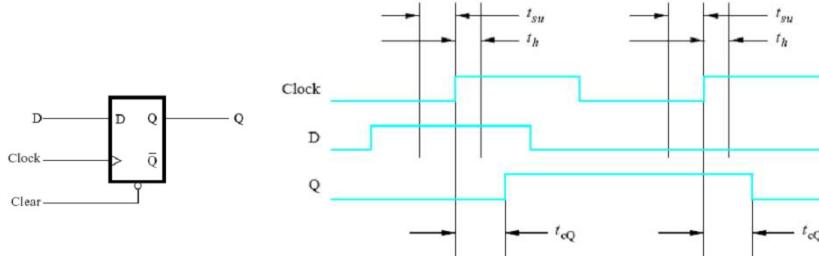
Clock Signal

- Flip-flops' Clk inputs typically connect to the same clock signal (i.e. the flip-flops 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



106

Flip-flop Timing Parameters

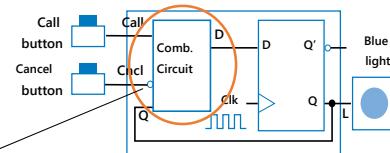


- t_{su} (setup time) – the minimum time that the input D signal must be stable prior to the active edge of the Clock signal
- t_h (hold time) – the minimum time that the input D signal must remain stable after the active edge of the Clock signal
- t_{cQ} – the propagation delay for input signal D to 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

107

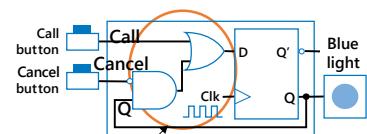
Service Call Button Using D Flip-Flop

- 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

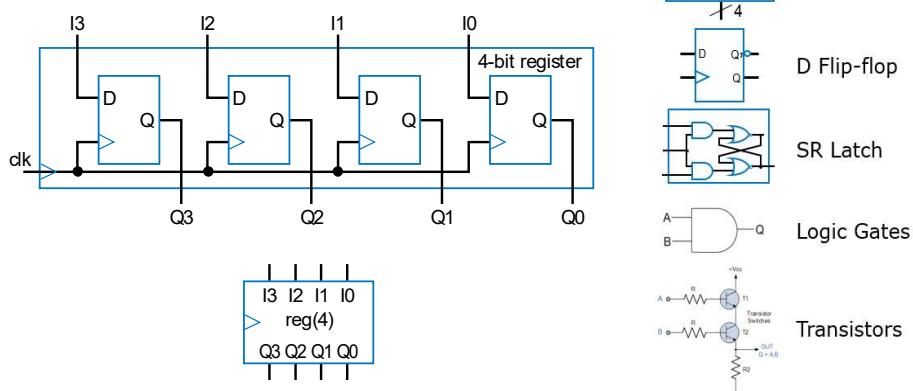


Circuit derived from truth table, using the combinational logic design process

108

Basic Register

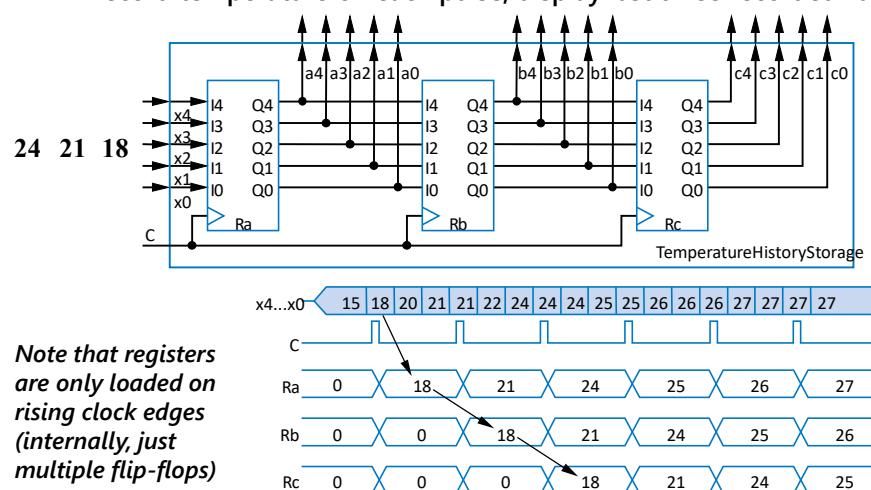
- While flip-flop is good, data is usually multiple bits
 - e.g. storing a 16-bit binary number or an 8-bit image pixel
- Register: multiple flip-flops sharing a common clock signal, i.e. all flip-flops are "synchronised"
 - From this point, we'll use registers for storage
 - But now you know what's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



109

Example Using Registers: Temperature Display

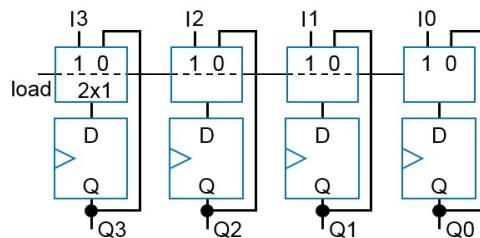
- Temperature history display
 - Sensor outputs temperature as a 5-bit binary number
 - Clock signal C pulses every hour
 - Record temperature on each pulse, display last three recorded values



110

Basic Register with Load control

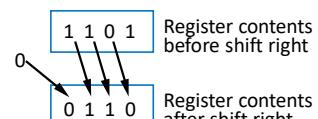
- N-bit register: Stores N bits, N is the width of the register
 - Storing data into register is called Loading
 - Opposite of storing is Reading (does not alter contents)
- Basic register: Loads every cycle (every active clock edge)
 - Can we design a register that only loads when we ask it to?
 - Internally, 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



111

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

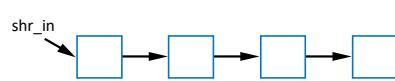


Q: Do four right shifts on 1001, showing value after each shift

A:

1001 (original)
0100
0010
0001
0000

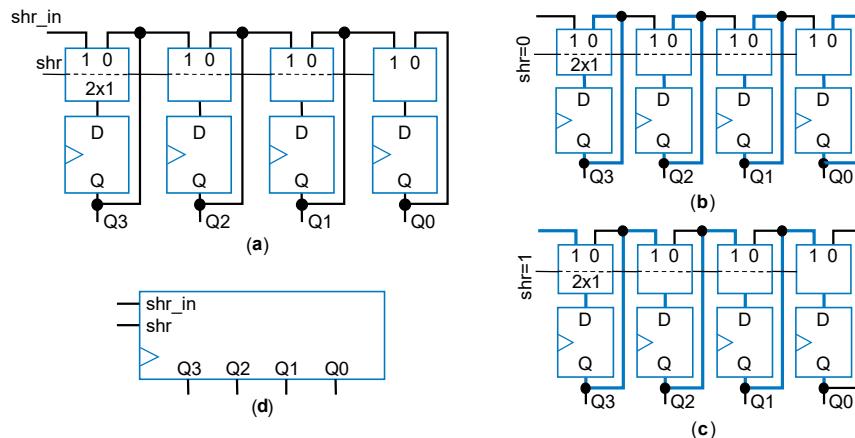
- Simplest internal design: Connect one flip-flop output to next flip-flop’s input



112

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: control input, "0" means retain, "1" means shift right
 - shr_in: data input, value to shift in



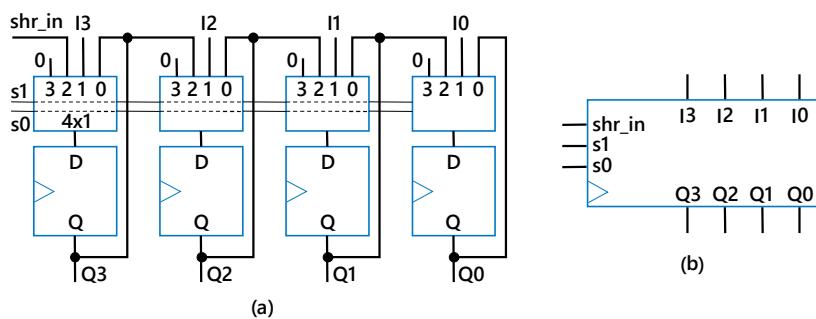
113

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
 - Input orders for all muxes need to be the same

Function

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



114

Register Design Example

Desired register operations

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

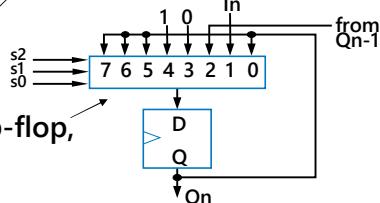
Step 1: Determine mux size

5 operations → Use 8x1 mux (why?)

Step 2: Create register operation table

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



Step 4: Map control lines (optional)

$$s2 = \text{clr}' * \text{set}$$

$$s1 = \text{clr}' * \text{set}' * \text{ld}' * \text{shl} + \text{clr}$$

$$s0 = \text{clr}' * \text{set}' * \text{ld} + \text{clr}$$

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

115

Multifunction Registers with Separate Control Inputs

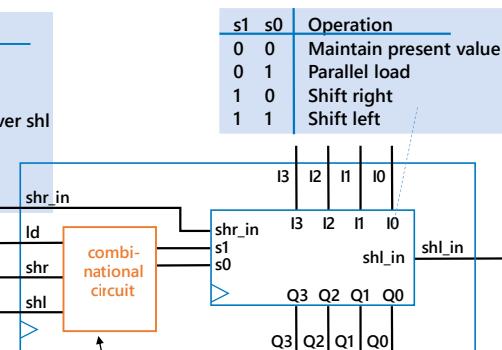
- Sometimes, instead of having Mux control inputs for each operation (difficult to remember), users may prefer separate control inputs for each function

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 Operation	
	ld	shr	shl	s1	s0
0	0	0	0	0	0
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	1	0	0
1	0	0	0	0	1
1	0	1	0	0	1
1	1	0	0	1	0
1	1	1	1	0	1



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

$$s0 = ld * shr' * shl + ld$$

116

Register Operation Table

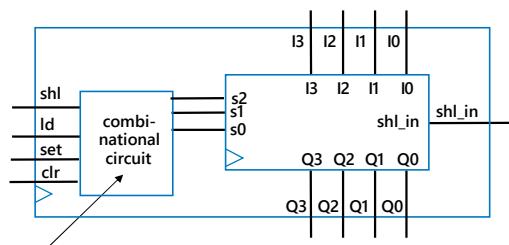
- Register operations are typically shown using a compact version of table (like a truth table)
 - X means don't care (i.e. does not matter whether value is 0 or 1)
 - Put highest priority control input on left to make reduced table simple -> How to decide priority?
 - Does the reduced table look familiar?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

Inputs			Outputs		Note
Id	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

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

117

Register Design Example



Step 4: Map control lines

$$\begin{aligned}s2 &= \text{clr}' * \text{set} \\s1 &= \text{clr}' * \text{set}' * \text{Id}' * \text{shl} + \text{clr} \\s0 &= \text{clr}' * \text{set}' * \text{Id} + \text{clr}\end{aligned}$$

Inputs			Outputs			Operation	
clr	set	Id	shl	s2	s1	s0	
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

118

Memory Element Summary

- Latches and flip-flops
- Flip-flops and formation of basic registers
- Timing characteristics and behaviour of flip-flops and registers
- Shift registers
- Multiplexer and priority encoder and their use in forming different variants of registers

119

COMPSYS201 – Part I Fundamentals of Computer Engineering

Sequential Logic Design – Counters and Timers

Dr. Kevin Wang
kevin.wang@auckland.ac.nz

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



120

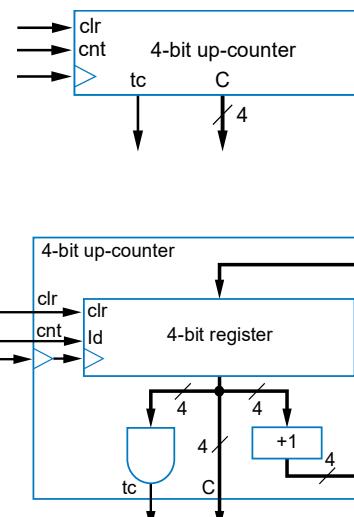
Introduction

- Design counters using
 - Registers
 - Incrementer/decrementer
 - Multiplexer
- Timers
- Clock dividers

121

Binary Counter

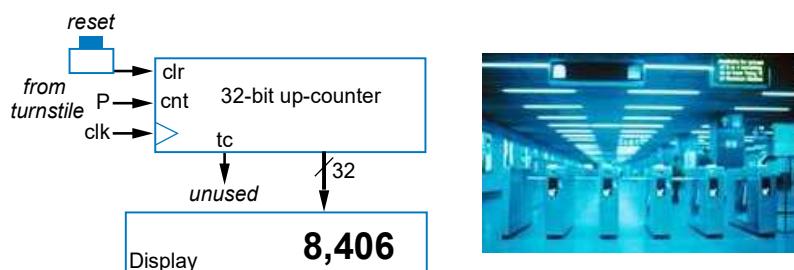
- N-bit binary up-counter can be designed by
 - An N-bit register, an incrementer, and an N-input AND gate to detect terminal count
 - Terminal (last) count, t_c equals 1 in the period the count value (i.e. the register content) is just before rollover
- Behaviour:
 - N-bit register that increments (add 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



122

Counter Example: Turnstile with Display

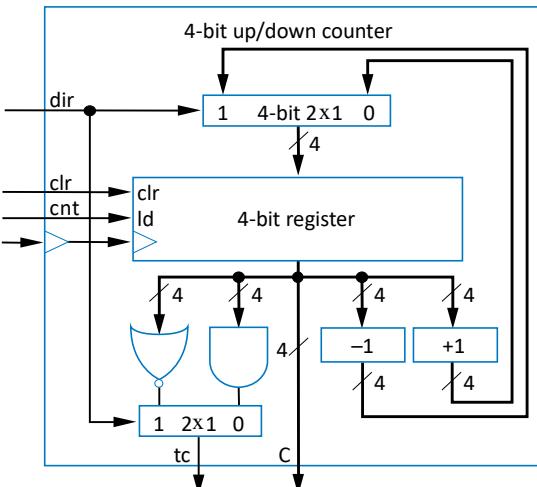
- Count people passing through a turnstile
 - Assume turnstile generates 1 pulse on P input per passing person
 - Connect the count output to a display
 - A reset button that can reset count
 - How many bits should we use for the counter?
 - Simple solution using an up-counter



123

Up/Down Binary Counter

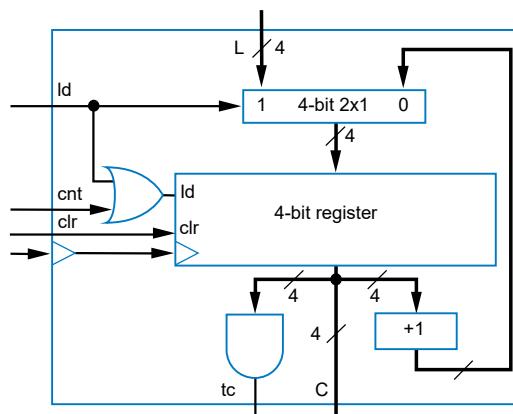
- Counter can count either up or down
 - Includes both incrementer and decrementer
 - Use *dir* input to select, via 2x1 mux: *dir*=0 means up
 - Likewise, *dir* selects appropriate terminal count value (all 1s or all 0s)



124

Counter with Parallel Load

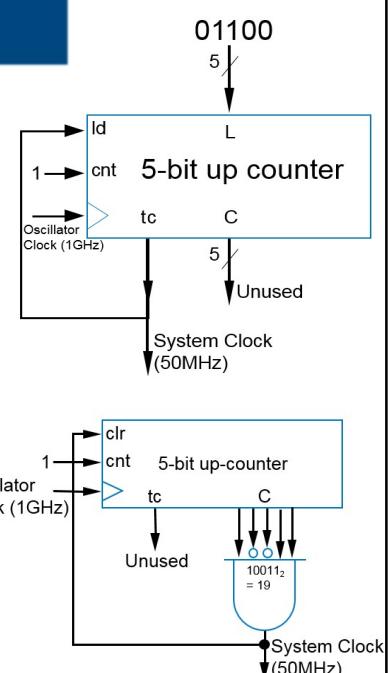
- Up-counter that can be loaded with external value
 - Why? Sometimes we don't want to start counting from 0
 - Design using 2x1 mux. ld input selects the external value ($ld=1$) or the incremented counter value ($ld=0$) to load the internal register
 - What happens when both $ld=1$ and $cnt=1$?



125

Counter with Parallel Load

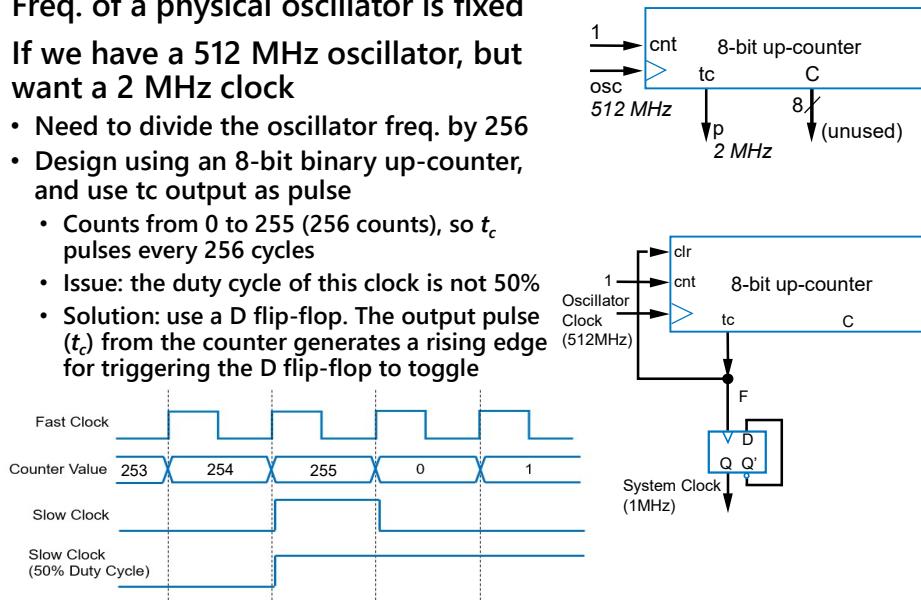
- Example: a pulse generator that generates a pulse every 20 clock cycles
 - Use 5-bit up counter with parallel load
 - Set parallel load input to 12 (01100)
 - Use terminal count to trigger reload
 - When count reaches 31, next cycle loads 12
 - Counting from 12 (inclusive) to 31 gives 20 counts
 - 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 reaching 0 (i.e. $t_c=1$)



126

Common Counter Use: Clock Divider

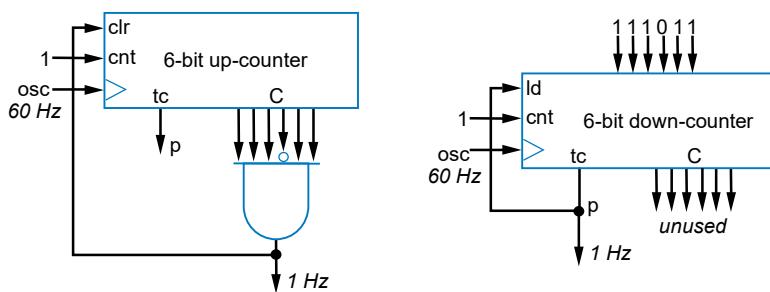
- Freq. of a physical oscillator is fixed
- If we have a 512 MHz oscillator, but want a 2 MHz clock
 - Need to divide the oscillator freq. by 256
 - Design using an 8-bit binary up-counter, and use tc output as pulse
 - Counts from 0 to 255 (256 counts), so t_c pulses every 256 cycles
 - Issue: the duty 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



127

Example: 1 Hz Pulse Generator from 60 Hz Clock

- Clock Division by something other than a power of 2
- Example: An 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



128

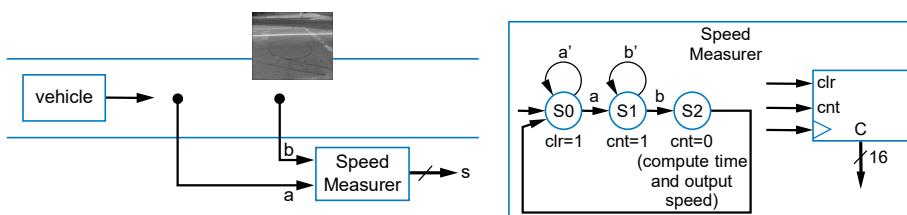
Common Use of Counter: Timer

- Counter is essentially used to count events
 - Timer considers each active clock edge as a timing event (indicating a fixed amount of time has passed, which is one clock cycle)
- The basic relationship between the number of counts, counter clock period and desired time period is as follows
$$\text{count} = \text{desired time period} / \text{counter clock period}$$

129

Example: Event Timer using an Up-Counter

- Example: Highway speed measurement system
 - Initially timer (or counter) value is cleared to 0
 - Two sensors "a" and "b" in road
 - Detect when "a" becomes 1, set $\text{cnt}=1$. Set $\text{cnt}=0$ when "b" is 1
 - If clock is 1 kHz (i.e. period is 1 ms), then time is $\text{Count} * 0.001\text{s}$
 - If a and b is separated by 0.01 km, then vehicle speed in km/h is:
$$0.01 \text{ km} / (\text{time} * (1 \text{ hour} / 3600 \text{ seconds}))$$
 - e.g. if C is 500, then speed is $0.01 / ((500 * 0.001) / 3600) = 72 \text{ km/h}$



130

Learning Outcomes

- Design counters using
 - Registers
 - Incrementer/decrementer
 - Multiplexer
- Clock dividers
- Timers

131

COMPSYS201 – Part I Fundamentals of Computer Engineering

Finite State Machine (FSM)

Dr. Kevin Wang
kevin.wang@auckland.ac.nz

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



132

Introduction

- So far we have introduced some simple sequential circuit components
 - Memory components
 - Counters, timers, 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 any sequential circuit
 - Design more sophisticated sequential circuit for digital systems

133

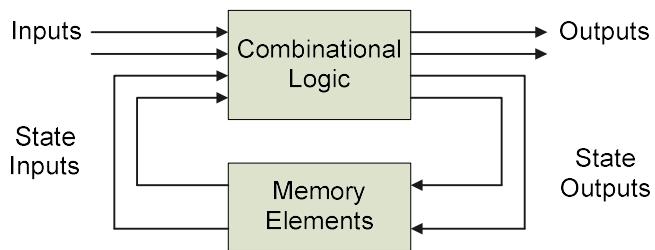
Mathematical Formalisms

- 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, S_0)
 - The state transition function (how we move between states, δ)
 - The output function (the output values for each state, ω)

134

General Form of a Sequential Circuit

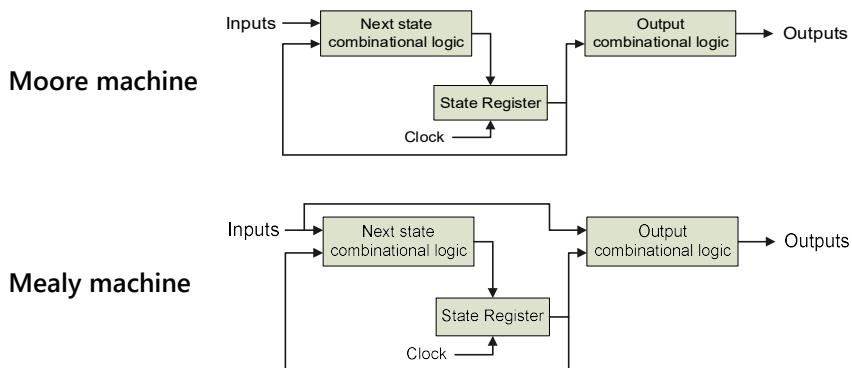
- Sequential circuits: are essentially combinational circuits (eg: adder, mux) plus memory elements (eg: register)
 - The memory elements store the "state" of the circuit
 - The combinational logic can be further separated into
 - Next state combinational logic, which implements the state transition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



135

General Form of a Sequential Circuit

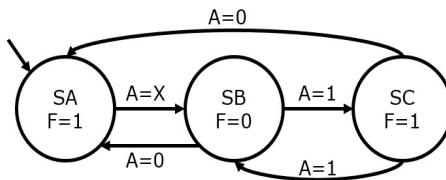
- Based on this general form, we can capture sequential circuit behaviour using two types of FSM models
 - Moore Model: Outputs depend on current state only
 - A FSM that fits the Moore Model definition is a Moore Machine
 - Mealy Model: Outputs depend on current state and inputs
 - A FSM that fits the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



136

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 (i.e. in the state circle), so this diagram represents a Moore type FSM

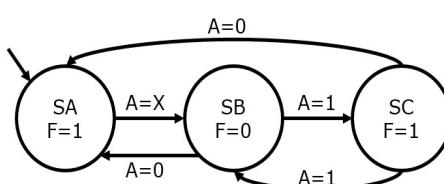


137

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

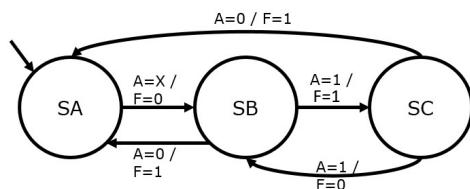
Current State	Input(s)	Next State	Output(s)
SA	X	SB	1
SB	0	SA	0
	1	SC	
SC	0	SA	1
	1	SB	



138

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 provided on the transition arrows (based on its originating state, and current inputs)
 - 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)	Next State	Output(s)
SA	X	SB	0
SB	0	SA	1
	1	SC	1
SC	0	SA	1
	1	SB	0

139

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 transitions
 - Step 2 to 4 might take a couple of iterations. Once you become more familiar with the design process, these steps can be simplified

140

Moore FSM example

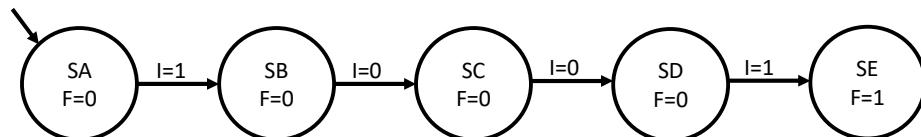
Example: Design a lock that will open if a particular binary input pattern is detected. The lock receives a single-bit input stream I , and assumes that every clock cycle a new bit arrives. Let's say that the unlock code is "1001". If we receive this bit sequence, then the output $F = 1$ (unlock), otherwise, 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
- Each transition is caused by a certain input condition to move from the current state to the next state at the active clock edge
- We know input is I and output is F and for Moore FSM output is only associated with states

141

Moore FSM example

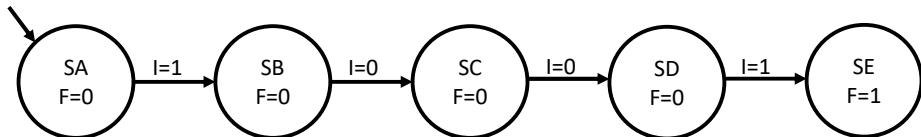
- Step 1: List the states and the most obvious transitions
- Step 2: Add input(s) and output(s) for those transitions and states
- Since we are detecting the pattern "1001", four transitions are needed for these four consecutive input bits, so we need five states
 - The following state transition diagram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



142

Moore FSM example

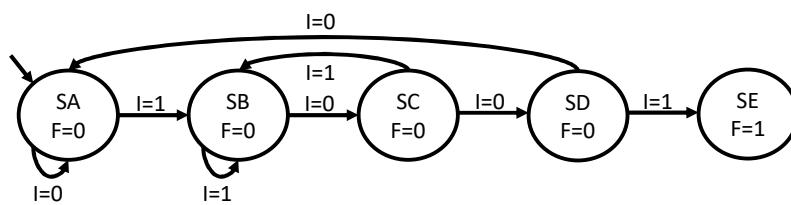
- Each state should have its unique meaning
 - State SA is the initial state, means nothing has been received, output is 0
 - State SB represents the case when a single '1' was received in the previous clock cycle, output is 0
 - State SC represents when the pattern "10" was received in the previous 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 "1001" was received in the previous four consecutive clock cycles, output is 1



143

Moore FSM example

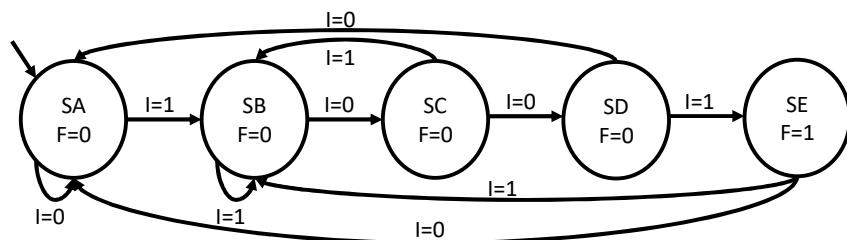
- Now let's analyse the behaviour and add more transition arrows
- Step 3: Create and label more transitions for uncovered cases
 - Self loops: the transition goes back to the same state. There are two cases:
 - In the initial state (SA) if we keep receiving bit 0, then it is like nothing has been received.
 - 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



144

Moore FSM example

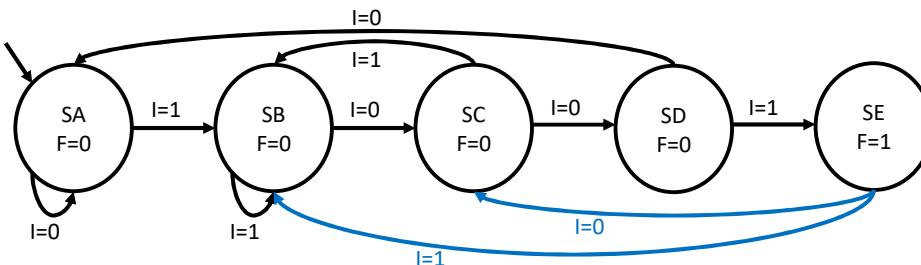
- For each state, there should be the right amount of transition arrows according to the number of input bits, for example
 - 1 bit input, will have at most two transition arrows (i.e. $I=0$ and $I=1$)
 - 2 bits input, will have at most four transition arrows (i.e. $XY=00, 01, 10, 11$)
 - So for each 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



145

Alternative Moore FSM Example

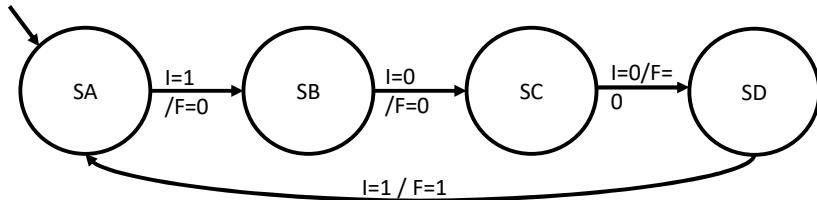
- Step 4: Refine the FSM to make sure all cases are covered
- Alternatively, rather than reset the lock to start a new sequence, what if we want to continuously check for the correct pattern (overlapping)?
 - State SE will transit to different states according to the input value
 - If $I=0$, then state 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



146

Mealy FSM example

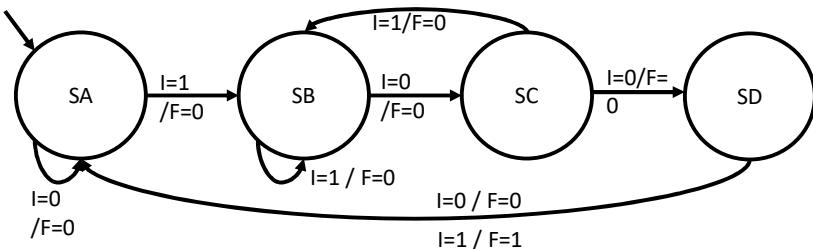
- Step 1: List the states and the most obvious transitions
- Step 2: Add input(s) and output(s) for those transitions and states
- In Mealy FSM, output is generated based on the current input and current state
 - When in state SD, i.e. we have already received "100" in the previous 3 consecutive clock 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 to generate output F=1
 - For the same system, Mealy machine requires fewer states, operates faster and requires less resources compared to Moore machine



147

Mealy FSM example

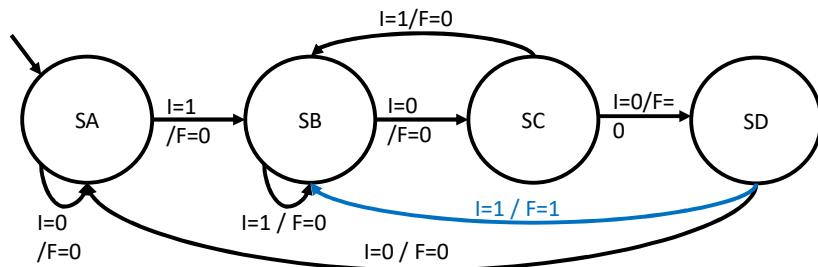
- Step 3: Create and label more transitions for uncovered cases
- Step 4: Refine the FSM to make sure all cases are covered
- If we would like to detect non-overlapping sequence, i.e. once a pattern is detected, we restart to detect a new sequence
 - So in state SD, it does not matter whether the input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



148

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 $I=0$



149

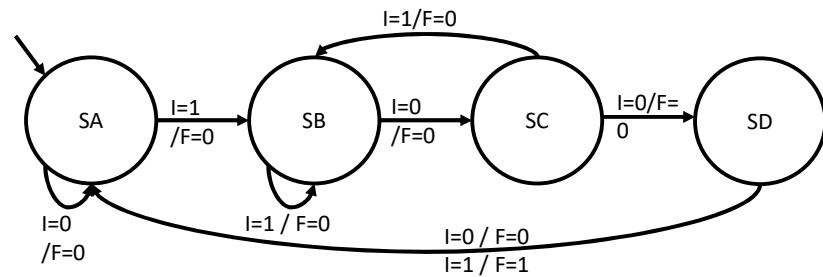
Sequential Circuit Design

- 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. Decide the implementation architecture:
 - a. Encode the states with a unique value to each state (the values can be in different coding scheme, typically binary)
 - b. 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

150

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: Decide the architecture

- We have 4 states, so we need 2 bits state variables to uniquely identify each state (assuming binary encoded states), which will be stored in a 2-bit register

151

Sequential Circuit Design: Electronic Lock Example

- Step 3: Convert the FSM diagram into a state table including encoded state values

Current State	Current Input	Next State	Output
$S_1 S_0$	I	$nS_1 nS_0$	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

152

Sequential Circuit Design: Electronic Lock Example

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

state transition functions

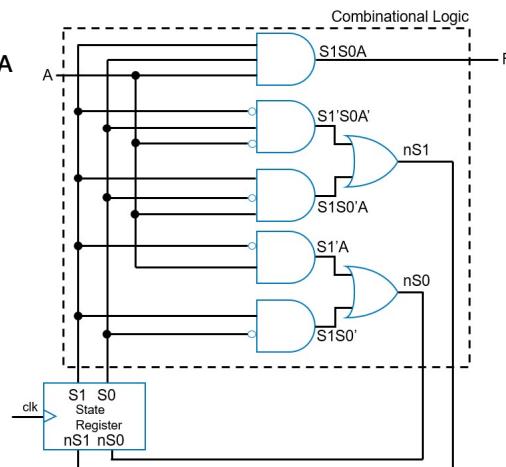
$$nS_1 = S_1'S_0A' + S_1S_0'A'$$

$$\begin{aligned} nS_0 &= S_1'S_0'A + S_1'S_0A + S_1S_0'A' + S_1S_0'A \\ &= S_1'A + S_1S_0 \end{aligned}$$

Output function

$$F = S_1S_0A \text{ (Mealy)}$$

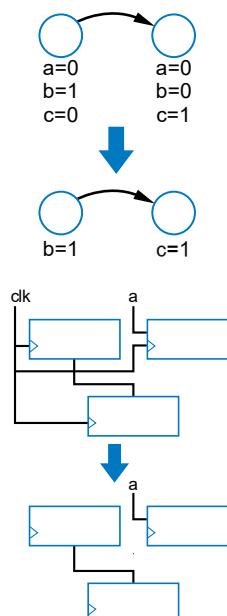
Current State	Current Input	Next State	Output
S1S0	A	nS1nS0	F
00 (SA)	0	00 (SA)	0
	1	01 (SB)	0
01 (SB)	0	10 (SC)	0
	1	01 (SB)	0
10 (SC)	0	11 (SD)	0
	1	01 (SB)	0
11 (SD)	0	00 (SA)	0
	1	00 (SA)	1



153

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



154

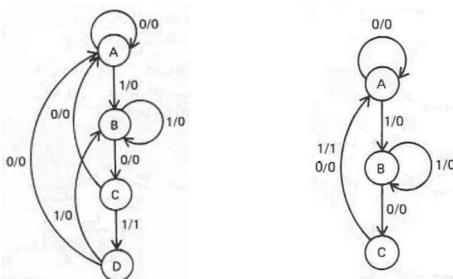
State Minimisation

- 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

155

State Minimisation Example

- Let's have look at a 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)



156

State Minimisation Example

- There are many approaches to ensure optimal FSM design, we are going to use a simple approach called row matching
- Consider the following state table
 - We can easily spot that state B and D, and state C and E are identical
 - They have identical outputs
 - They transition to equivalent next states in every possible input combination
 - Only one state of multiple equivalent states needs to be kept. Therefore, replace state D by state B, and state E by state C

Present state	Next State		Output	
	I=0	I=1	I=0	I=1
A	B	C	1	0
B	D	E	1	0
C	A	F	1	1
D	D	E	1	0
E	A	F	1	1
F	A	A	0	0

157

State Minimisation Example

- With equivalent states removed, we have a new state table
- The row matching approach is iterative, so we need to look at the state table again
 - This time, state A and B are found to be equivalent
 - Apply row matching until you are certain that all states are not equivalent
- In this example, we reduced from 6 to 3 states, so we will only need 2-bit state variables, rather than 3-bit

Present state	Next State		Output	
	I=0	I=1	I=0	I=1
A	B	C	1	0
B	B	C	1	0
C	A	F	1	1
F	A	A	0	0

Present state	Next State		Output	
	I=0	I=1	I=0	I=1
A	A	C	1	0
C	A	F	1	1
F	A	A	0	0

158

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

159

COMPSYS201 – Part II Fundamentals of Computer Engineering

Register Transfer Level (RTL) Design

Dr. Kevin Wang
kevin.wang@auckland.ac.nz

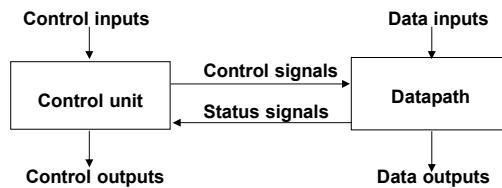
Some slides in this section are modified from:
Digital Design, with RTL Design, VHDL, and Verilog, 2nd edition, Copyright © 2010 Frank Vahid (<http://www.ddvahid.com>)
Fundamentals of Digital Logic with VHDL Design, 3/e, Copyright © McGraw Hill



160

Introduction

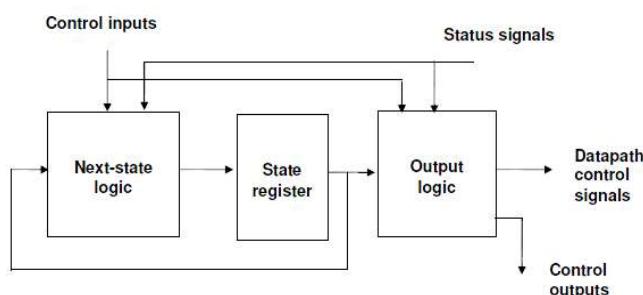
- What is Register Transfer Level (RTL) design?
- An abstraction and methodology for partitioning and designing digital systems, which consists of
 - Datapath: the body of a digital system which does all the computation
 - Control unit: the brain of a digital system which controls what will be performed by the body (i.e. the datapath) via control signals
 - Datapath may generate status signals according to the results (or status) of its operations and provide feedback to the control unit
 - Datapath may load external data input and produce data output (typically multiples of bytes)
 - Control unit may accept external control input and produce control output (typically multiples of bits)



161

Control unit

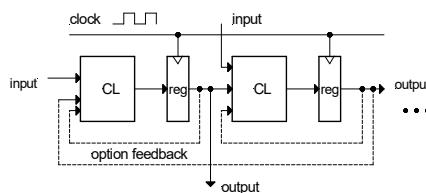
- Often control units are designed as FSMs and consists of
 - Next state logic
 - Output logic
 - State register
- Control unit :
 - takes external control inputs and feedback status signals from datapath as inputs to the next state and output logic
 - generates control signals to datapath to control data source, data destination, and operations
 - generates control outputs to external components



162

Datapaths

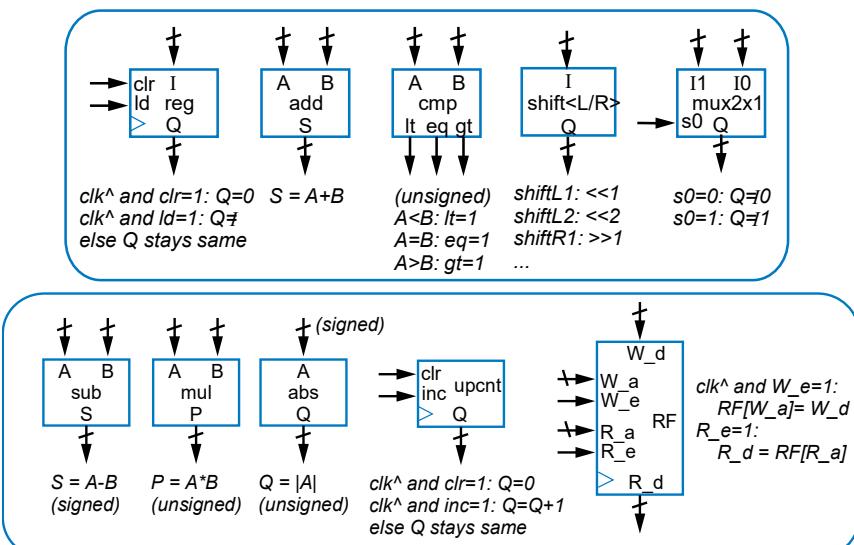
- Datapath is the functional unit of a digital system
 - ONLY does what control unit (the brain) asks
 - Consists of storage (e.g. registers, memories), processing (e.g. arithmetic/logical units), and routing (e.g. multiplexers and buses)
 - takes the values from storage units (typically registers), performs computation in the combinational units, and returns the results to storage units during each state
 - Registers -> combinational units -> registers (i.e. register transfers)
- The operand values are stored in storage components
 - They are fetched from storage after one active (e.g. rising) clock edge – i.e. transfer can only happen at active clock edges
 - Results are stored back into storage at the following active clock edge
 - The values are transformed in between combinational units during two consecutive active clock edges, to be processed



163

Typical Datapath Components

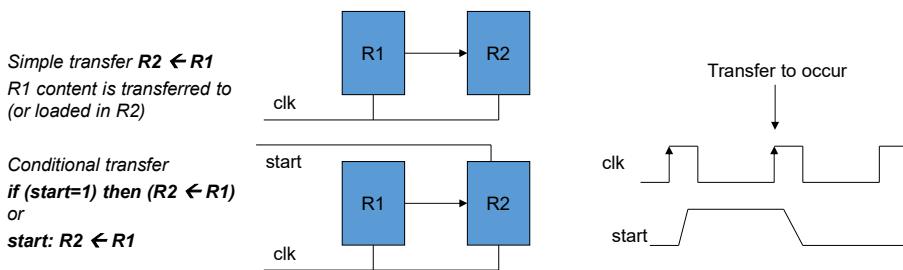
- Library of components that perform “standardised” operations



164

Register Transfer Operation Examples

- Data flows in a datapath represent register transfer operations (RTOs)
- RTOs (or microoperations) specify:
 - Source register(s)
 - Destination register(s)
 - Operation
- Control unit generates control signals which determines whether a specific RTO happens or not



165

Typical Types of RTOs

- Simple transfer RTOs and their notations.
- Transfer (i.e. that arrow) happens only at active clock edges

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	$AR, R2, DR, IR$
Parentheses	Denotes a part of a register	$R2(1), R2(7:0), AR(L)$
Arrow	Denotes transfer of data	$R1 \leftarrow R2$
Comma	Separates simultaneous transfers	$R1 \leftarrow R2, R2 \leftarrow R1$
Square brackets	Specifies an address for memory	$DR \leftarrow M[AR]$

- Arithmetic RTOs, typically involves combinational units in between the source and destination registers

Symbolic designation	Description
$R0 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R0
$R2 \leftarrow \overline{R2}$	Complement of the contents of R2 (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement of the contents of R2
$R0 \leftarrow R1 + \overline{R2} + 1$	R1 plus 2's complement of R2 transferred to R0 (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of R1 (count up)
$R1 \leftarrow R1 - 1$	Decrement the contents of R1 (count down)

166

Typical Types of RTOs (cont'd)

- Logic & shift RTOs. Similar to arithmetic RTOs which involves combinational units in between registers transfer

Symbolic designation	Description
$R0 \leftarrow \bar{R1}$	Logical bitwise NOT (1's complement)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

Eight-bit examples			
Type	Symbolic designation	Source $R2$	After shift: Destination $R1$
shift left	$R1 \leftarrow sl\ R2$	10011110	00111100
shift right	$R1 \leftarrow sr\ R2$	11100101	01110010

167

Register Transfers – Multiplexer and Bus-based

- Design compromise between performance and cost

- (a) Mux-based, point to point connection with dedicated wires, or
- (b) Common input from a mux with a common bus
- No. of buses and muxes can vary depending on app. requirements

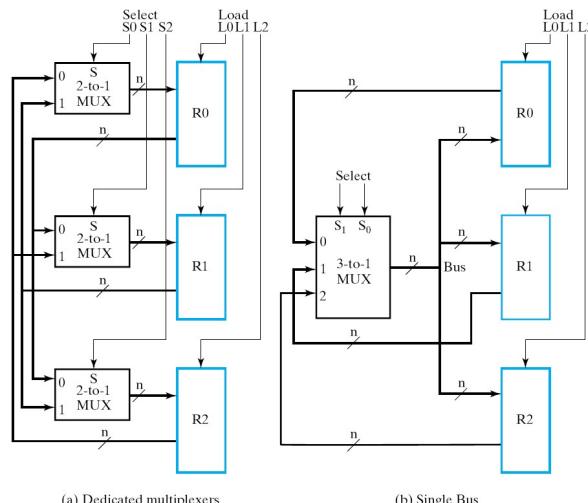
Transfer examples:

$R0 \leftarrow R1$

$R1 \leftarrow R0, R2 \leftarrow R0$

Is this possible to do:

$R1 \leftarrow R0, R0 \leftarrow R1$

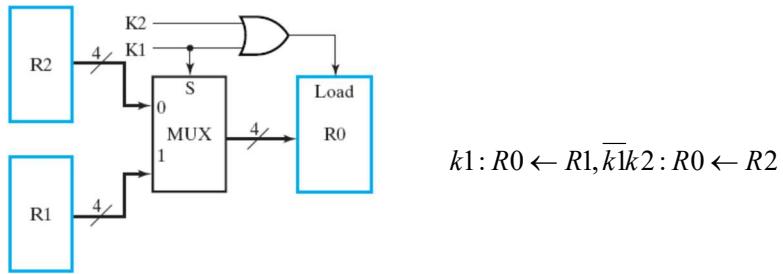


168

RTO Example – Conditional Transfer Operation

- Conditional transfer RTO

- RTOs only happen when certain condition is met, similar to if statement in programming languages
- Example: if ($k_1=1$) then ($R_0 \leftarrow R_1$) else if ($k_2=1$) then ($R_0 \leftarrow R_2$)

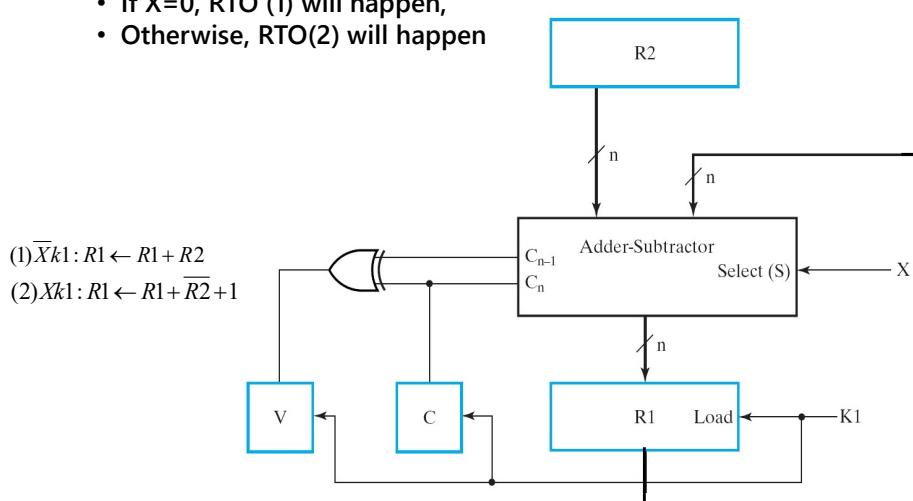


169

RTO Example – Conditional Arithmetic Operation

- Example, adder/subtractor RTOs

- Control signal K1 decides whether the RTO happens or not
- Control signal X decides which RTO (1 or 2) should happen
 - If $X=0$, RTO (1) will happen,
 - Otherwise, RTO(2) will happen



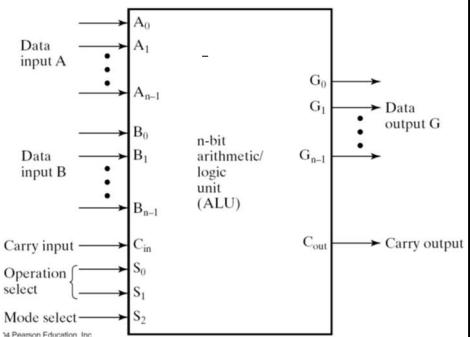
170

Datapath Arithmetic and Logic Unit (ALU)

- ALU is
 - the most common combinational unit in a datapath, which provides most of the basic arithmetic and logic operations
 - constructed in modular fashion where each bit is functionally identical to form a multi-bit ALU
 - Input: data and operation control signals to perform
 - Output: result of operation and status information

Function Table for ALU

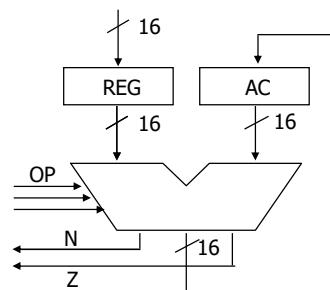
Operation Select				Operation	Function
S ₂	S ₁	S ₀	C _m		
0	0	0	0	$G \leftarrow A$	Transfer A
0	0	0	1	$G \leftarrow A + 1$	Increment A
0	0	1	0	$G \leftarrow A + B$	Addition
0	0	1	1	$G \leftarrow A + B + 1$	Add with carry input of 1
0	.	.	.	$G \leftarrow A + \bar{B}$	A plus 1's complement of B
0	1	0	1	$G \leftarrow A + \bar{B} + 1$	Subtraction
0	1	1	0	$G \leftarrow A + 1$	Decrement A
0	1	1	1	$G \leftarrow A$	Transfer A
1	X	0	0	$G \leftarrow A \wedge B$	AND
1	X	0	1	$G \leftarrow A \vee B$	OR
1	X	1	0	$G \leftarrow A \oplus B$	XOR
1	X	1	1	$G \leftarrow \bar{A}$	NOT (1's complement)



171

Simple Datapath Example (ALU + Registers)

- Accumulator (AC)
 - Special register
 - One of the inputs to ALU
 - Output of ALU is also stored back in accumulator
- One-input Operation
 - The other operand and destination is the accumulator register
 - AC <- AC op REG



172

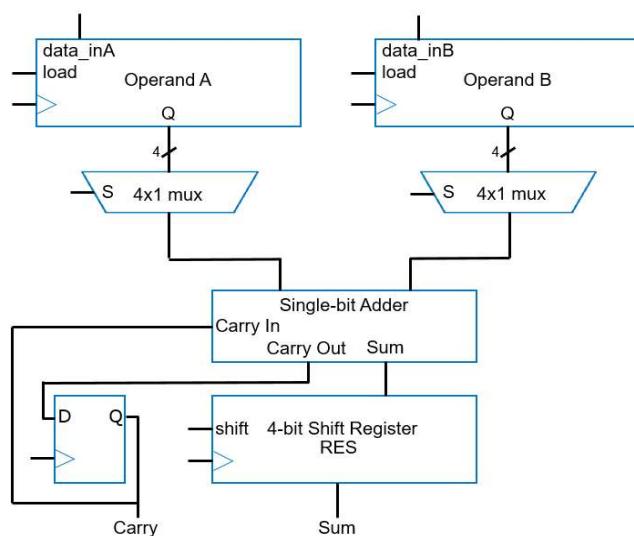
RTL Design Example

Example: A serial adder is a sequential circuit which adds each corresponding bit pair one after another in order like doing addition by hand, starting from the LSb and iterating to the MSb. Design a 4-bit serial adder using the RTL approach.

- What do we need to design the datapath?
 - Since we are adding one bit pair at a time, most likely we will need a single bit full adder?
 - For a 4-bit serial adder, we probably will need a few 4-bit registers to take inputs and store output in sequence?
 - Carry generated by each bit pair addition needs to be stored and feed into the next bit pair addition?
 - We may also need multiplexers to select the right input bit pair to add?
 - Try and link all these components together

173

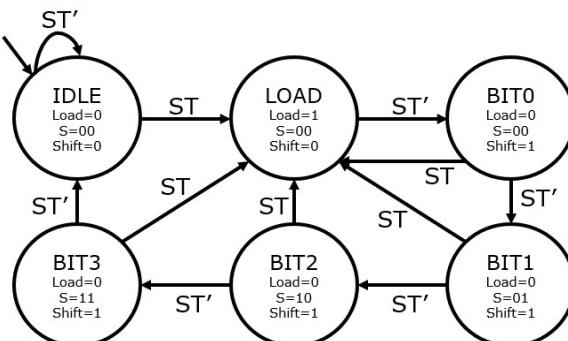
Serial Adder Datapath



174

Serial Adder Control Unit

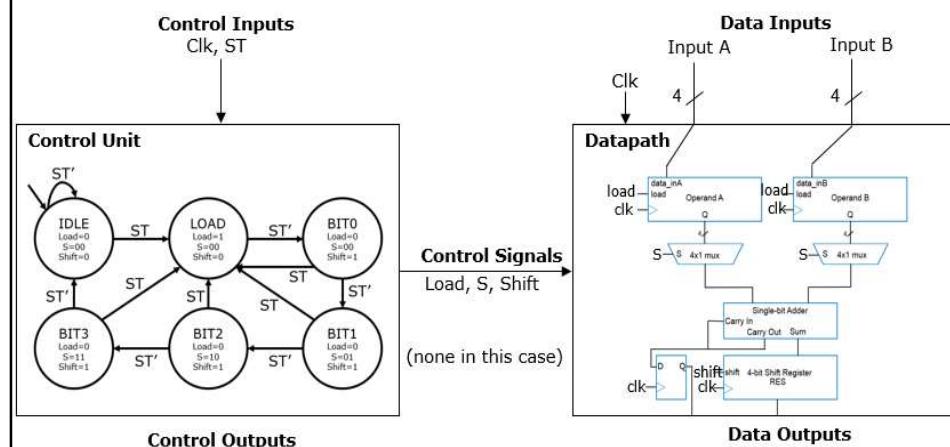
- Based on the datapath, we need to generate the following control signals to properly perform serial addition:
 - load: load enable signal that triggers two input registers to fetch two 4-bit numbers
 - S: select signal to choose which bit pair from the source registers to add
 - shift: shift enable signal that triggers the shift register shift one bit to the right and take 1-bit sum coming out of the 1-bit full adder



175

Serial Adder Design

- Link the datapath and the control unit together to form the serial adder system

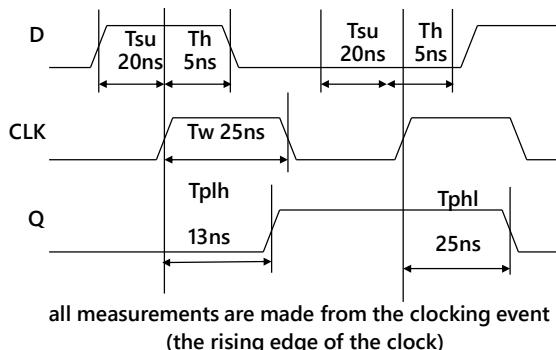


176

Recap: Flip-flop Timing Parameters

- For a positive edge-triggered D flip-flop

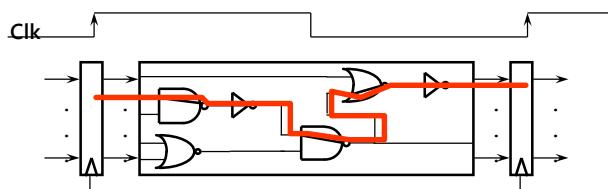
- tsu (setup time) – the minimum time that input D signal must be stable prior to the active edge of the Clock signal
- th (hold time) – the minimum time that input D signal must remain stable after the active edge of the Clock signal
- tcQ – the propagation delay for input signal D to appear at the output Q, usually given as a range



177

Critical Path and Min. Clock Period

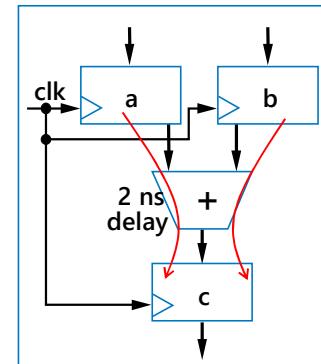
- All memory elements are controlled by the same clock edge
 - For source memory, inputs are fetched immediately after the clock active edge, and available at the output pin after t_{cQ}
 - For destination memory, after processing, all inputs MUST be stable before the next clock active edge to satisfy the setup time
- Critical path: the slowest path between any two storage devices, which includes
 - t_{cQ} (source memory)
 - combinational logic delay of the critical path
 - t_{su} (destination memory)
- The minimum clock period must be greater than or equal to the sum of all three timing components



178

Determining Clock Frequency – Simplified Case

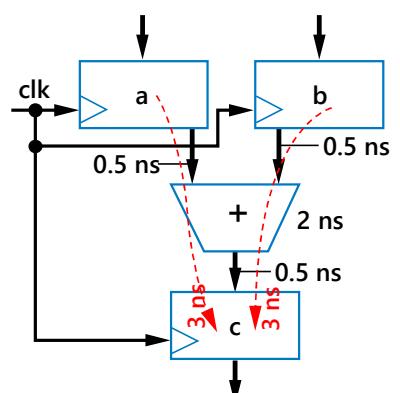
- Designers of digital circuits often want the fastest performance
 - Want high clock frequency
- The maximum clock frequency is limited by the longest register-to-register delay
 - Which represents the minimum clock period
 - If clock is any faster, setup time of the destination register c might be violated
 - Incorrect data may be stored into register (metastability)
 - Longest path is 2 ns in this example
 - Ignoring wire delays, and register setup and hold times, for simplicity
 - Minimum clock period is 2ns, the maximum clock frequency is $1/2\text{ns} = 500\text{MHz}$



179

Critical Path Considering Wire Delays

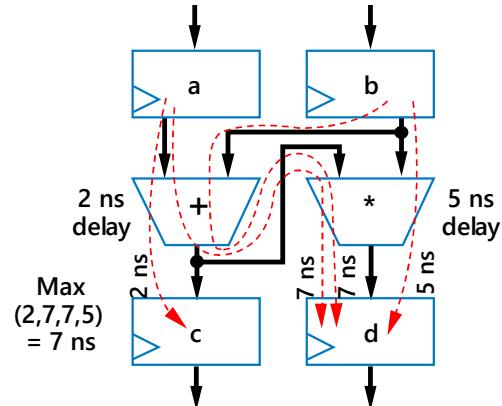
- Real wires have delay too
 - Must include in critical path
- Example shows two paths
 - Each is $0.5 + 2 + 0.5 = 3 \text{ ns}$
- Trend
 - 1980s/1990s: Wire delays were tiny compared to logic delays
 - But wire delays are not shrinking as fast as logic delays
- Should also consider register propagation delay (of source registers a or b) and setup times (of destination register c)



180

Critical Path Example

- Example shows four paths
 - a to c through +: 2 ns
 - a to d through + and *: 7 ns
 - b to d through + and *: 7 ns
 - b to d through *: 5 ns
- Longest path is thus 7 ns
 - Min. period allowable
- Fastest frequency
 - $1 / 7 \text{ ns} = 142 \text{ MHz}$



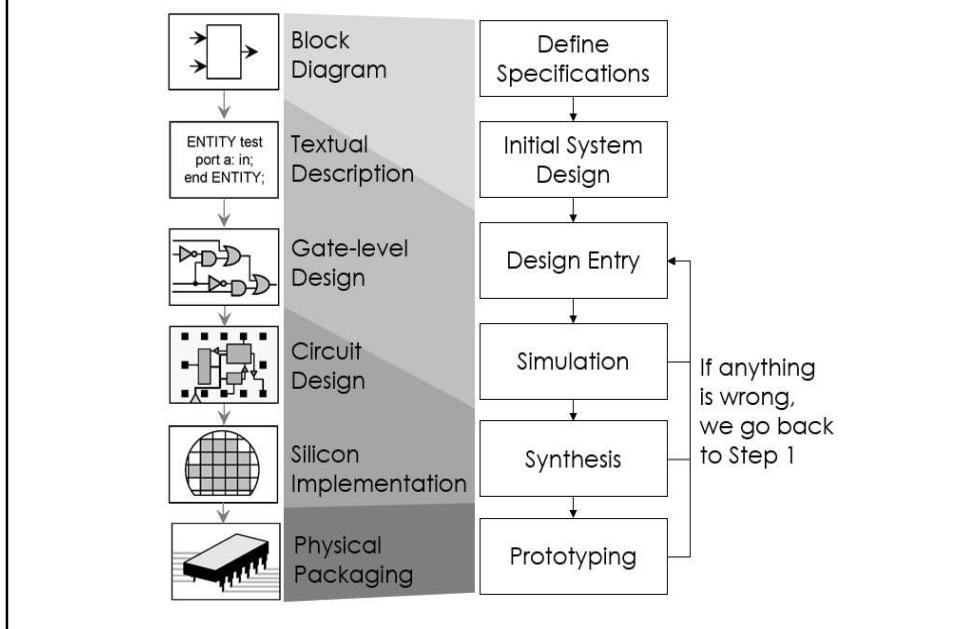
181

Learning Outcomes

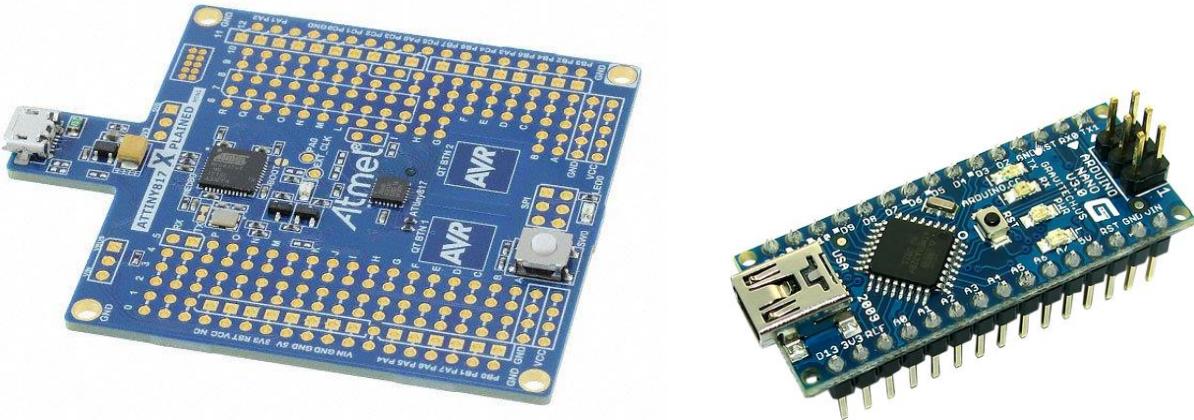
- The register transfer level design paradigm with control unit and datapath abstraction
- Typical types of register transfer operations and how to read, express and implement them
- Arithmetic and logic unit
- Basic timing analysis

182

Concluding Remarks – Digital Design Flow



183



COMPSYS201 - 2021

Fundamentals of Computer Engineering

Part. 2 - Microcontrollers

**Department of Electrical, Computer, and Software
Engineering, the University of Auckland.**

© Dr. Bill Collis 2021

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

Contents

2.1.	Introduction	96
2.2.	Learning outcomes and success criteria	97
2.3.	The way of the program	98
2.4.	Review your prior learning.....	98
2.5.	Atmel and the AVR range of 8 bit microcontrollers	99
2.5.1.	AVR block diagram,.....	99
2.5.2.	What is a computer/ microprocessor /microcontroller?.....	100
2.5.3.	The Arduino Nano development board	101
2.5.4.	AVR memory types	102
2.5.5.	Number ‘types’	102
2.5.6.	External peripherals - inputs or outputs?	104
2.5.7.	LO Understand the ES as an automaton	105
2.6.	Code Style	106
2.7.	The interrelatedness of hardware and software in ES’s.....	107
2.7.1.	IO port overview	108
2.7.2.	IO peripherals, macros & bit masking	109
2.7.3.	Macros	109
2.7.4.	unpacking macros	110
2.7.5.	How can a single uC hardware pin be used as both an input and an output?	111
2.7.6.	The internal pullup resistor	113
2.7.7.	State	114
2.7.8.	ES development overview.....	115
2.8.	Describe the ES as reactive and responsive to its environment	118
2.8.1.	Die Program (dice - plural, die - singular).....	120
2.8.2.	LO: Understand the importance of transparent software practices	121
2.8.3.	Apply a formal approach to writing program code	122
2.8.4.	Polling.....	123
2.8.5.	Hardware topic: Switch issues.....	124
2.9.	Memory mapped IO and Pointers	126
2.9.1.	Volatile	127
2.10.	Timer/Counters	128
2.10.1.	AVR Timers	129
2.10.2.	Timer example – ‘heart beat’ led flash	130

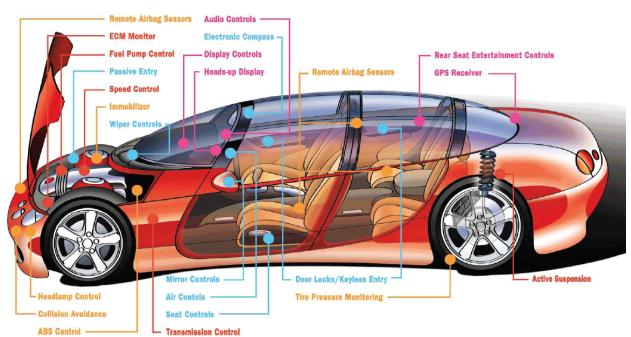
2.10.3.	The ‘about’ 1 second timer	131
2.10.4.	The ‘exact’ 1 second timer	132
2.10.5.	The ‘exact’ 1 second flash direct hardware output timer.....	133
2.10.6.	Make a short duration pulse at a regular time interval.....	134
2.10.7.	Timer Registers.....	136
2.10.8.	Use ICP1 to measure pulse width.....	138
2.11.	External Interrupts.....	144
2.11.1.	Interrupt complications	148
2.12.	Serial Data Communications	150
2.12.1.	External serial and parallel peripherals	150
2.12.2.	Serial communication protocols.....	151
2.12.3.	ATMega328 internal communication hardware datasheet section 20	153
2.12.4.	Serial communications example code	154
2.13.	The ADC	155
2.13.1.	The voltage divider	155
2.13.2.	ES analog sensors	158
2.13.3.	The conversion process.....	159
2.13.4.	AVR ADC inputs	162
2.13.5.	ADC Channel select.....	163
2.13.6.	Sampling rate	165
2.13.7.	ADC modes.....	166
2.14.	State machine programming.....	169
2.14.1.	Coding state machines in C	171
2.14.2.	Electronic safe state machine	172
2.15.	Driving output devices.....	174
2.15.1.	Driving devices with higher current / voltage requirements	176
2.15.2.	The logic MOSFET	176
2.15.3.	Inductive Loads.....	178
2.16.	Fitness-for-purpose	179
2.17.	Useful resources.....	180

2.1.Introduction

Embedded systems (**ES**) are at the centre of the exponential increase of smart devices, the IoT (internet of things) and TinyML (embedded machine learning applications) happening 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. Common ES's have at their heart microcontrollers (**uC**) - other digital controllers include FPGAs, ASICs, and microprocessors. ES's are ubiquitous (everywhere) and so much so, that we are generally unaware of them. Over 250 billion of these devices automate, balance, maximise, minimise, regulate and control our everyday lives better than we can ourselves! A washing machine measures the volume and dirtiness 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 reorder more paper for us before the roll has run out! While the number of smart phone and internet users are now growing at small rates every year, the number of ES is increasing by around 20% every year.



This part of the course is an introduction to uC's and their programming, and how as part of ES's they react to, and control, the environment. Because ES's come in so many varied forms, with so many different features and have so many diverse 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 ECSE 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.microchip.com

2.2.Learning outcomes and success criteria

Course 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 (i.e.how registers work)
- with reference to a datasheet, write code that manipulates/checks single bits in any uC register

LO 2: understand the ES as an automaton

- describe three common C code variants used to make an ES into an automaton
- explain ES's with regard to unattended consistent and reliable operation
- explain how an ES can operate with the illusion of concurrency
- explain state - the interrelatedness of the past, present and future conditions of an ES

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 an analog 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 high quality software practices

- apply a formal approach to writing program code through use of a template, functions and naming conventions that makes code transparent and therefore maintainable
- write comments in code that explain system function and not code syntax
- write single task self-contained functions

LO 5: recognise 'fitness-for-purpose' factors that guide 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 common variable types and their limitations
- explain over/under flow of variables and their implications
- interface an LED to a uC, taking into account the DC characteristics of each



LO 6: use 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 engineer by working with complex tasks with multiple relationships



2.3.The way of the program

We use mathematical models to express the dynamic relationships within the real world around us; whether this is the flow of charge, clouds, or continents. Programs are the ways of calculating these models using special languages (C, Java, Python...); and the computer is a numerical processing device that can interpret these languages and carry out computations of our models faster than we could do it ourselves. The embedded system takes this one step further, it allows us to measure changes in the real world as they occur, process these numerical representations using our models and control our world in real time.

2.4.Review your prior learning

Take some time to review the material below before this part of the course begins. This will mean you will be comfortable learning the new knowledge that is introduced.

EngGen131 review

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 23: Casting

page 24: unary operators

page 25: special assignment operators and operator precedence

page 31: Arrays (basic types)

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 69-77: pointers and parameter passing

If it weren't for C we'd all be
programming in Basic or obol



ElectEng101 review

This part of the course relies upon material from EE101, review sections

- 3.2.4 Number Systems
- 3.2.5 ADC
- 3.2.8 Embedded Systems

CompSys201 digital section review

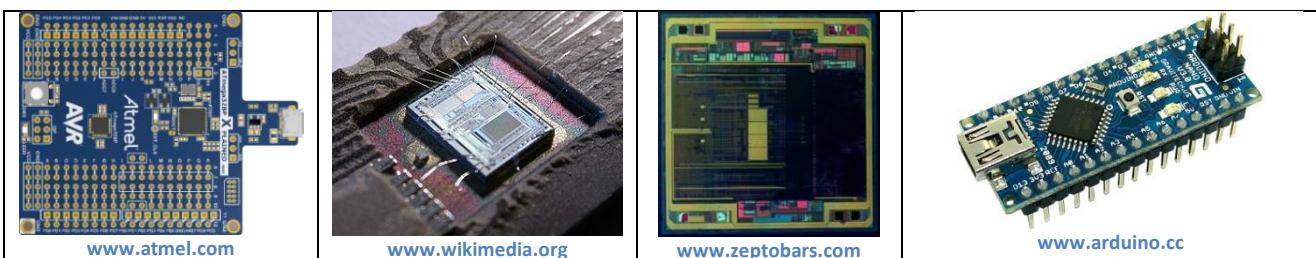
This part of the course relies upon the material in the first part of the course, review the material on:

- Latch/Register
- Counters/dividers
- Comparators

2.5. Atmel and the AVR range of 8 bit microcontrollers

Atmel is one of many brands of microcontrollers. Other brands include: Texas, Cypress, Renasas, ST, TI, National, NXP, Freescale, and Microchip (who now own AVR) also make PICs

- 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
- The Atmel brand includes several different ranges of uC's - AVR8, UC3(32 bit) and ARM
- The 8 bit AVR range includes several sub types: ATTiny, ATmega, ATxmega, AT90
- 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 Nano Board from aliexpress.com for about _____

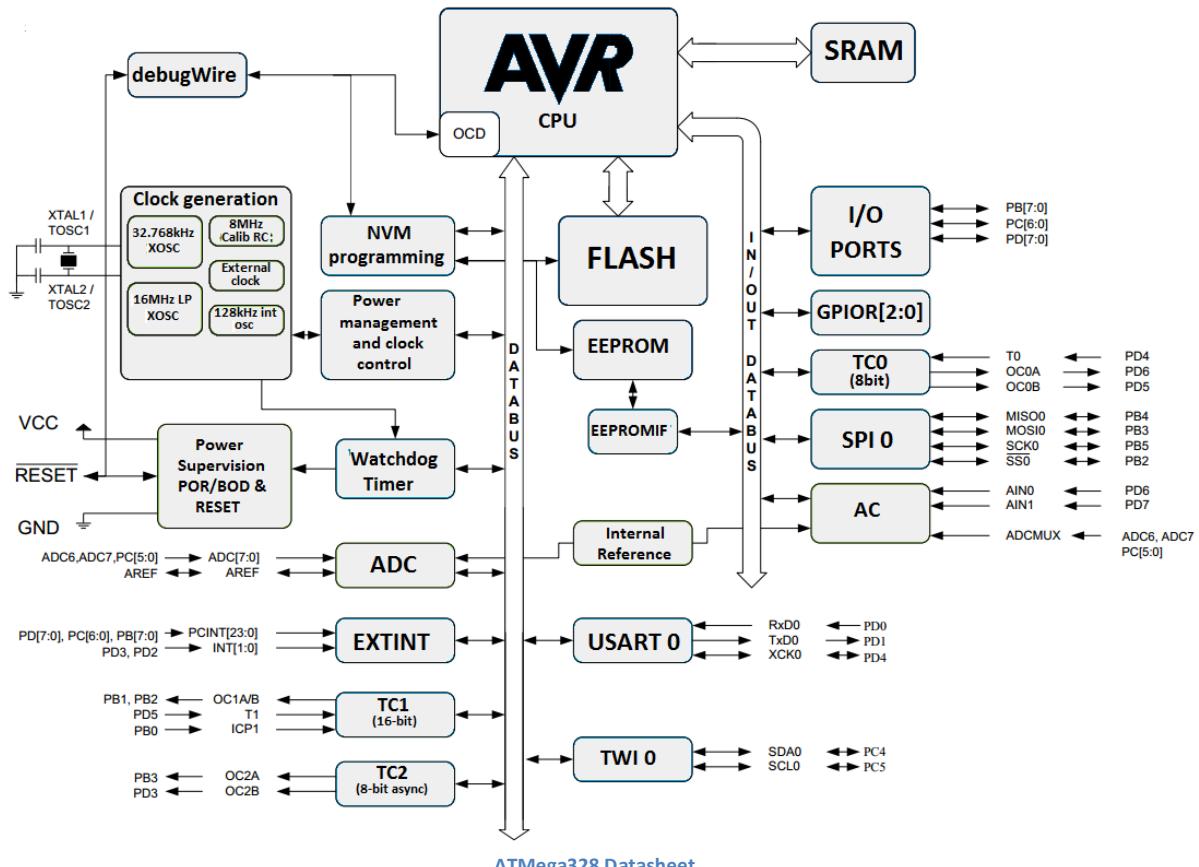


In ElectEng 101 you were introduced to the ESP32 DevKit, we use the AVR in this course because it is less complex (it is still very complex) and is easier to learn about the inside workings of a Microcontroller.

2.5.1. AVR block diagram,

The AVR block diagram shows the CPU and the internal peripherals (parts of the chip that are not the CPU, and handle specific jobs) e.g. _____

Figure 4-1. Block Diagram

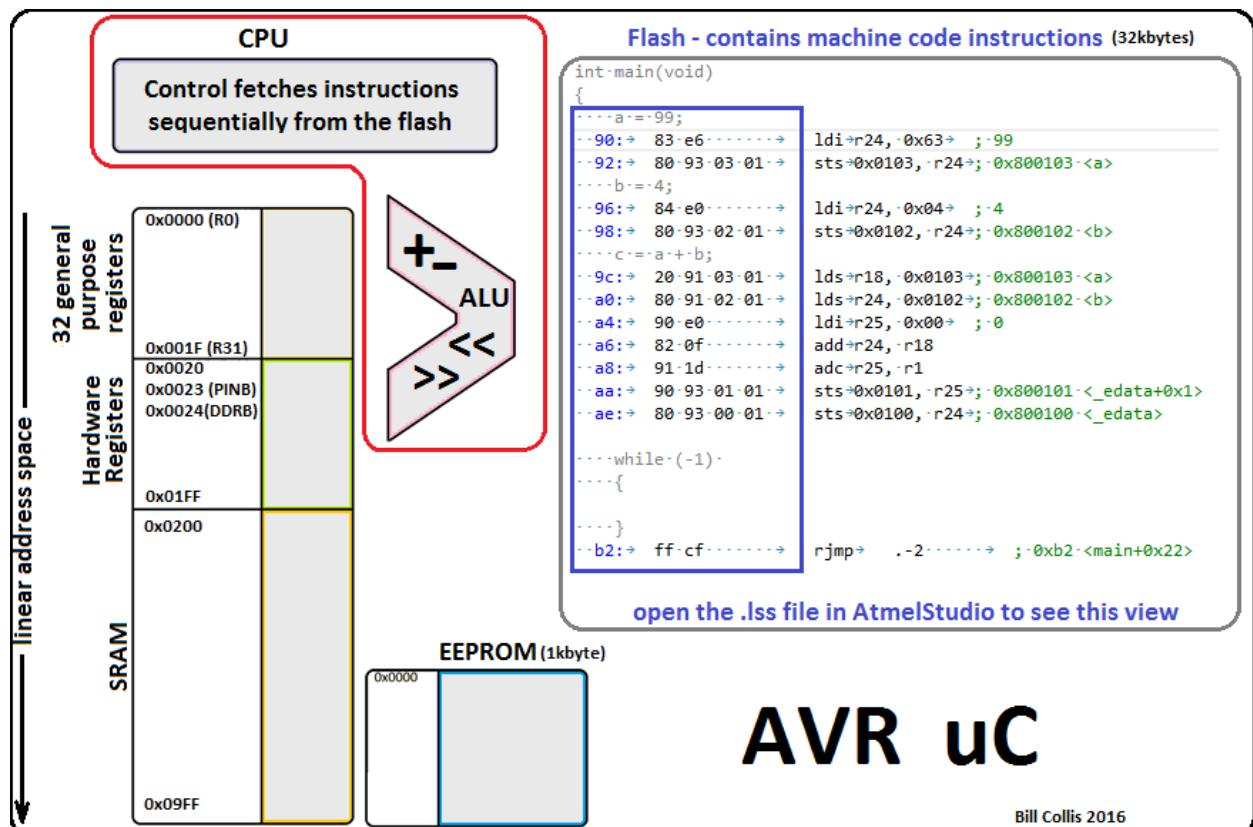


2.5.2. What is a computer/ microprocessor /microcontroller?

Computers have microprocessors with external peripheral devices; such as ram, hard drive, displays etc. A microcontroller is a single computer, with a range of integrated peripherals, all inside one IC. Both a microprocessor and a microcontroller contain a CPU (central processing unit) which has a control unit and an ALU (arithmetic logic unit). Internal microcontroller peripherals include ADCs, Timers etc, and an array of registers to control them along with several different types of memory: SRAM, FLASH and EEPROM.

```
//C code example
uint8_t a = 0;
uint8_t b = 0;
uint16_t c = 0;
int main(void)
{
    a = 99;
    b = 4;
    c = a + b;
    while (1) { ... } //loop forever
}
```

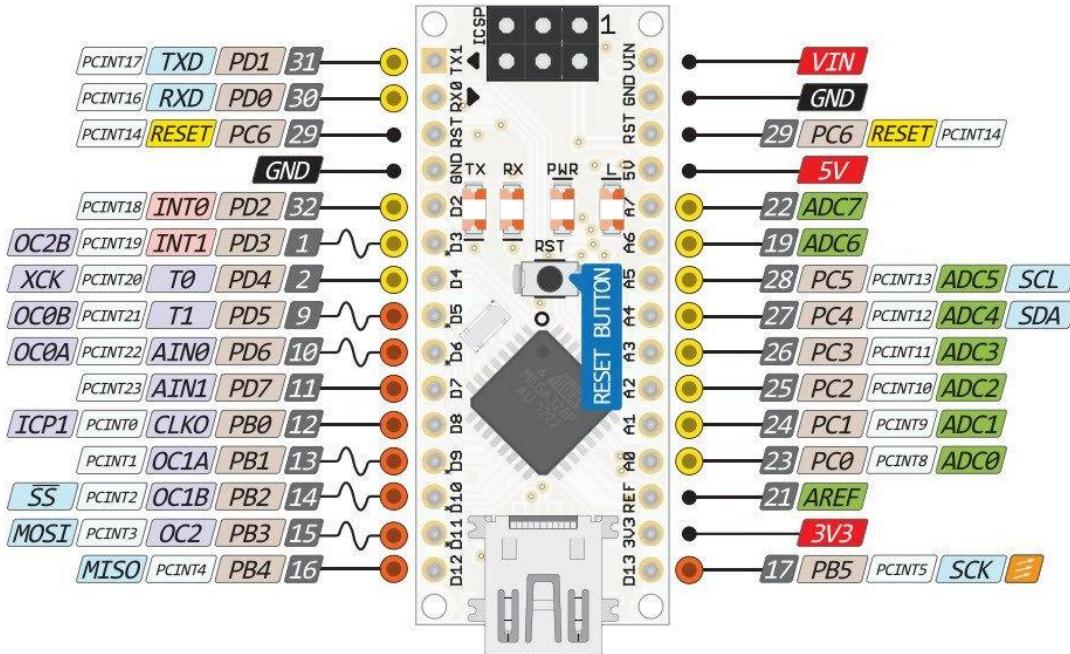
You need to know a few things: C programs are not stored in the uC – they are compiled into machine code and that is stored in the Flash. The human readable version of machine code is called assembler code or assembly language (assembler commands: ldi, sts, lds, add, adc are described in the datasheet). 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 work on registers so take only 1 clock cycle. Thus the AVR is a RISC (reduced instruction set computer) as opposed to a CISC (complex instruction set computer) where a single instruction may be more powerful containing multiple read, write and ALU operations, and take multiple cycles. The CPU control section fetches instructions sequentially from the flash (program memory) and decodes them. Take the simple program to add 2 numbers. The binary machine code stored in Flash (starting at address 0x0090) is: 0x83, 0xe6, 0x80, 0x93, 0x03, ... The machine code can be viewed in the .hex file – however a more intelligible view of flash is found in the .lss file.



The AVR has a contiguous address space: the general purpose registers, the hardware registers and the SRAM are part of the same address space; whereas EEPROM and FLASH are separate address spaces. We refer to the IO as being memory-mapped IO – because it's addressed like it was the same as the SRAM.



2.5.3. The Arduino Nano development board



www.atmel.com

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

I/O PORTS – are made up of I/O (input/output) pins – places we can connect devices to:

I/P devices: _____

O/P devices: _____



ATmega328P has 3 I/O 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 development 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, input devices sense the real world, output devices control the real world. An initial aspect of using a pin is to set up which direction it will be; i.e. – input – to read an input device or as an output to control an output device.

ATmega328P datasheet: <http://www.microchip.com/wwwproducts/en/ATmega328P>

All datasheet references will be to the complete datasheet, we will use version DS40002061B

2.5.4. 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) -_____
- Registers
- EEPROM – stores long term data
 - the ATmega 328P has a _____ EEPROM -_____

2.5.5. Number 'types'

www.gansle.com

When writing programs for a uC, computer languages offer us the ability to constrain a number to a defined range so that we can make the most of the limited RAM we have. 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.

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 (*) unsigned 64 bit
- int8_t signed 8 bit
- int16_t signed 16 bit
- int32_t signed 32 bit
- int64_t (*) signed 64 bit ()
- float/double (***) 32 bit - 1.2E-38 to 3.4E+38



Explain the difference between signed and unsigned?

What might be the implications of a bad variable choice?

What is the problem with using global variables in your program?

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

Guidelines for type selection:

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

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 critical importance of social (non-technical) factors in 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

One of the critical issues facing embedded systems engineers is the incorrect use of variables.

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?



2.5.6. External peripherals - inputs or outputs?

Research these devices, whether they are inputs or outputs and what they are used for.



	Device	Input or Output	Use
1	Thermistor		
2	LCD		
3	GPS receiver		
4	Accelerometer		
5	Microswitch		
6	LDR		
7	Humidity sensor		
8	Potentiometer (pot)		
9	Solenoid –electromechanical		
10	Solenoid valve		
11	Strain gauge		
12	FSR		
13	DC Motor		
14	LED		
15	7 segment display		
15	Photo diode		
16	keypad		

2.5.7. 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 usually 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*

```
//c programs begin executing at the main function
int main(void) {
    // setup code is placed here, it is only executed once
    ...
    ...

    //code in this while loop will run forever
    while (1) {      //
        ...
        ...
        ...
    }      //jump back to the while and do the loop code again

    //your code will never get here
}
```



Research the alternatives to while(1){}

2.6. Code Style

A design-pattern is a common approach to a common problem in software, an anti-pattern is the opposite, it is a poor programming practice (bad habit) that is ineffective and counterproductive (ultimately works against your efforts in the end).

Anti-patterns lead to software bugs. It is cheaper and easier to prevent bugs from creeping into your programs than it is to find and fix them once you realise they are there. Gansle says that 56% of 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 debug and modify - even if you are the only person who will ever read it.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand"

Martin Fowler



Use design patterns.

- 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_COEFFICIENT 324`
- Use nouns for variable names and verbs for function names
- Make all names descriptive
- No magic numbers
- 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 modular and small 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 or function from a system perspective, avoid stating the obvious about syntax unless it is complex or abstract
- Let the compiler do the optimization; most of the time our efforts at optimization fail us.
- Use local variables whenever possible. A change to a global variable breaks everywhere it is used. Use the narrowest scope possible for all variables.

(http://www.koopman.us/bess/chap19_globals.pdf)



pvisorsoftware.com

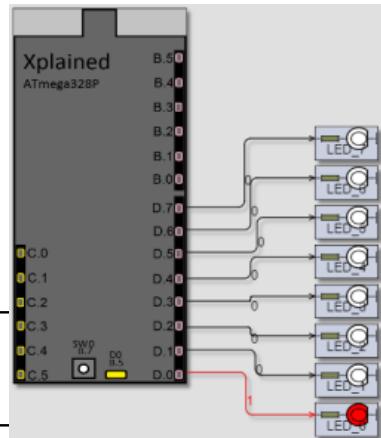
2.7.The interrelatedness of hardware and software in ES's

A core learning objective is to work fluidly between hardware and software. This involves understanding registers. Registers are used to control internal peripheral devices such as IO ports, and they are used to transfer data between all the peripheral devices and the CPU. Each IO Port peripheral has _____ registers

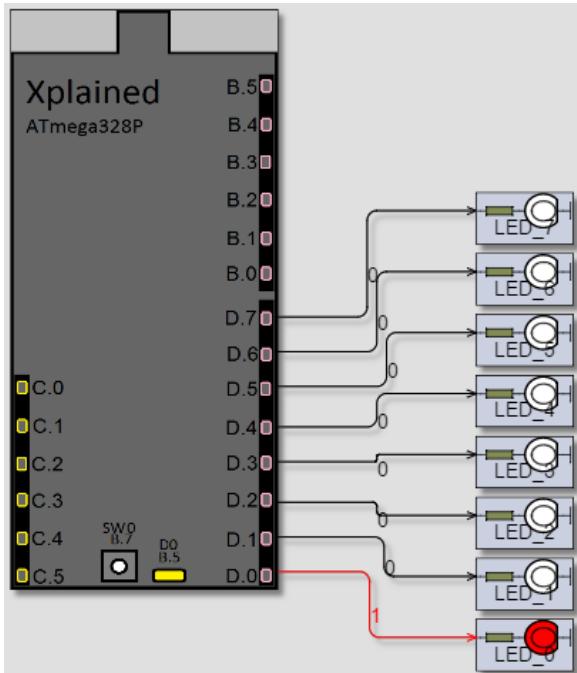
Describe how program commands work that link hardware and software

```
***** 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; //shortcut to 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 = 0b1110000;       //
        //

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



www.byyourcommand.net



`_delay_ms(TIME_DELAY);`
If these lines of code were removed what would the Cylon look like?



Software that controls IO hardware peripherals

DDRD = 0xff;

1. make all 8 pins of portD outputs
2. by writing to Data Direction register D
3. alternatives:
 - a. DDRD = 0b_____
 - b. DDRD = _____

Software that transfers data from the CPU to the IO hardware peripheral

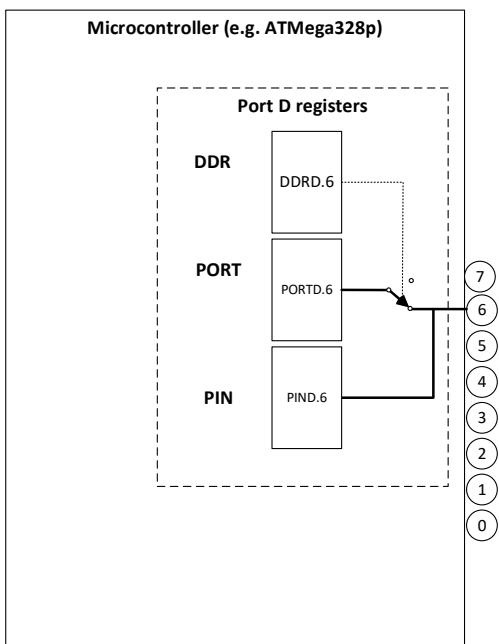
PORTD = 0b00000001;

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.

2.7.1. IO port overview



What is a register?

A register is a binary storage device (made of flip-flops). Registers can be single or multibit, and simple, shift, or multifunction. Here we mean a multifunction 8-bit register. It will have a reset (clear) function and of course a clock input. When the uC powers up, the reset lines on all the registers in the uC are set briefly, so the stored values in all registers are 0. When you want a new value in a register, e.g. PORTD= 0b10011000; the CPU will put the value 10011000 on the 8 bit data bus, then pulse the clock line for that specific register, and the new value will be stored in the register.

2.7.2. IO peripherals, macros & bit masking

It is not usual to control all pins of an IO port peripheral at one time; we actually set and clear individual pins to control individual IO devices. We do this using an understanding of **Boolean identities linked to an understanding of registers.**

1. $X \text{ or } 1 = 1$
2. $X \text{ or } 0 = X$
3. $X \text{ and } 0 = 0$
4. $X \text{ and } 1 = X$ where 'X' means _____

For example: In the pedestrian lights controller in this diagram we will want to control different lights at different times of the sequence so we must learn how to control each pin of a port register on its own. To control one output such as the GRN_LED we actually follow this process:

- read all 8 bits of register PORTB,
- change bit 2 (either high or low) while keeping all the other 7 bits unchanged *** requires boolean identities knowledge
- write the resulting value back into register PORTB.

To turn the LED on we need to make bit 2 '1' (high) **and** keep the other bits unchanged – identities 1 and 2 will help us. **PORTB = PORTB | 0b00000100; // the '| is the C command for a bitwise OR**

We are forcing only bit 2 to be high, it does not matter what value it has to start with, the effect of using boolean identity 1 will be only PORTB bit 2 will go high. The other bits are 'OR'ed with 0 (Boolean identity 2) so they remain unchanged; 0 stays as 0, 1 stays as 1, i.e. we don't care what t was to start with – it will be the same afterwards.

To turn the LED off we need to make bit 2 '0' (low) **and** keep the other bits unchanged – identities 3 and 4 will help us. **PORTB = PORTB & 0b11111011; //the '&' is the C command for a bitwise AND**

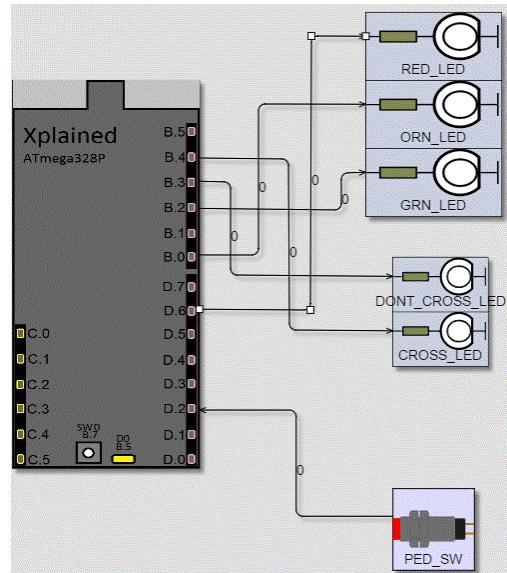
Here we are forcing bit 2 low, it does not matter what value it was to start with, the effect of using Boolean identity 3 will be only PORTB bit 2 will go low. The other bits are 'AND'ed with 1 (Boolean identity 4) so they remain unchanged, 0 stays as 0, 1 stays as 1.

2.7.3. Macros

While the statements above work perfectly, for code transparency reasons we must not go 'littering' our code with statements like those above it makes it totally unreadable and almost impossible to debug. We learn early on to write macros at the top of your program that keep all the hardware control clearly transparent **and** in one place, making it easy to make changes later on.

- `#define SET_GRN_LED PORTB|=(1<<B2) // macro to turn GRN_LED on`
- `#define CLR_GRN_LED PORTB&=~(1<<B2) // macro to turn GRN_LED on`

Throughout our code we use the macros SET_GRN_LED /CLR_GRN_LED, that way if we ever moved the GRN_LED to another pin we only need to change it in one place in the code and that is easily identifiable as they are all in one place.



2.7.4. unpacking macros

The Anatomy of an output macro

- The pre-processor replaces the macro
- read the port register
- change only the required bit(s) using bitwise logic and Boolean identities
- write the new value back to the port register

Output macros

```
SET_RED_LED;
```



```
CLR_RED_LED;
```

The anatomy of an input decision?

- pre-processor replaces the macro
- read the pin register
- isolate the required bit(s) using bitwise logic and Boolean identities
- write the new value to the port register

```
if ( PED_SW_IS_LOW ) { }
```

```
if (PED_SW_IS_HIGH) { }
```

2.7.5. How can a single uC hardware pin be used as both an input and an output?

Describe how program commands work that link hardware and software

Each IO pin has three bits, one from each of three 8-bit registers connected to it:

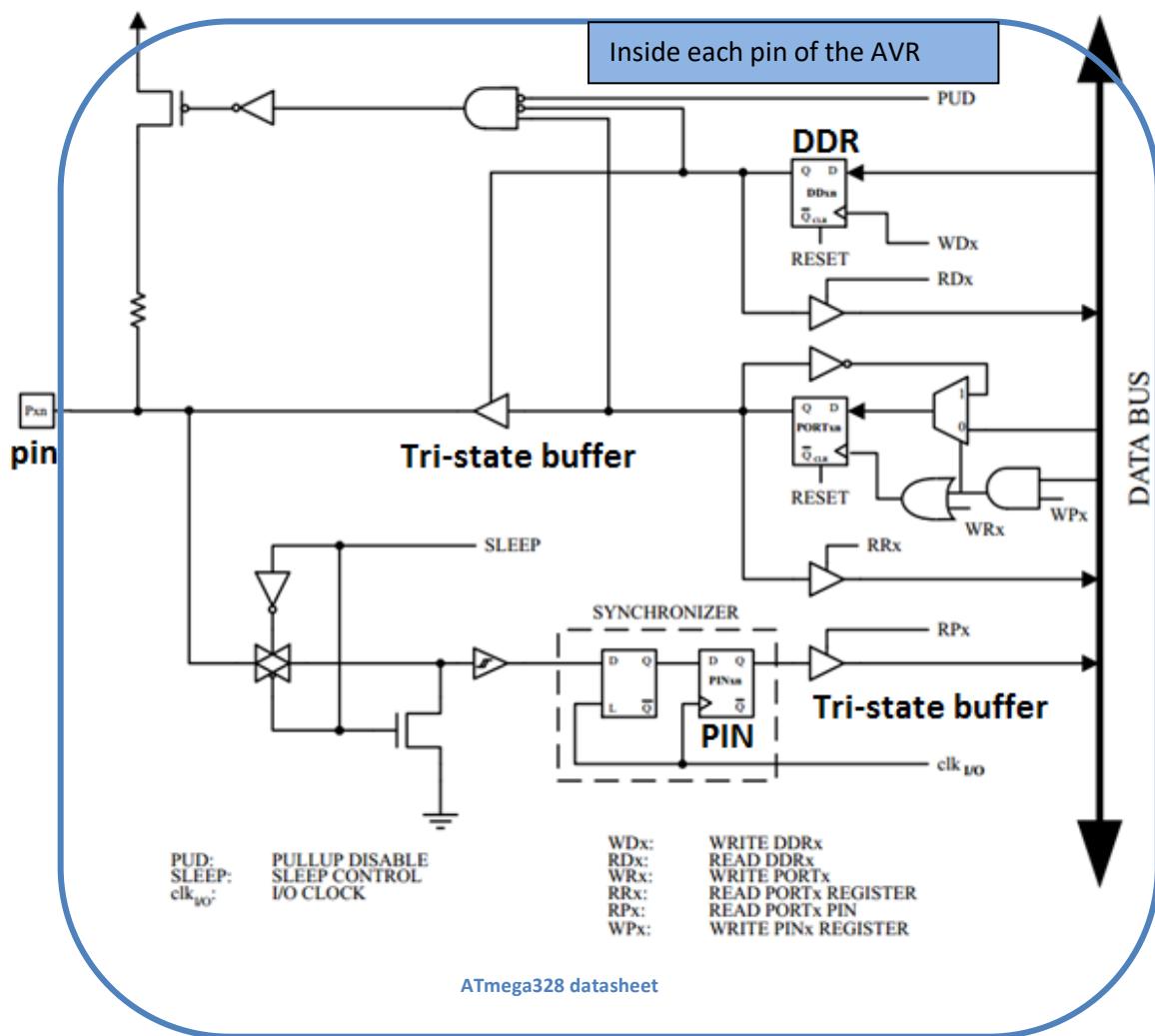
- PORTx.n e.g.
 - PINx.n e.g.
 - DDRx.n e.g.

How a μ C pin works as an INPUT

Figure 18-2. General Digital I/O⁽¹⁾

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)



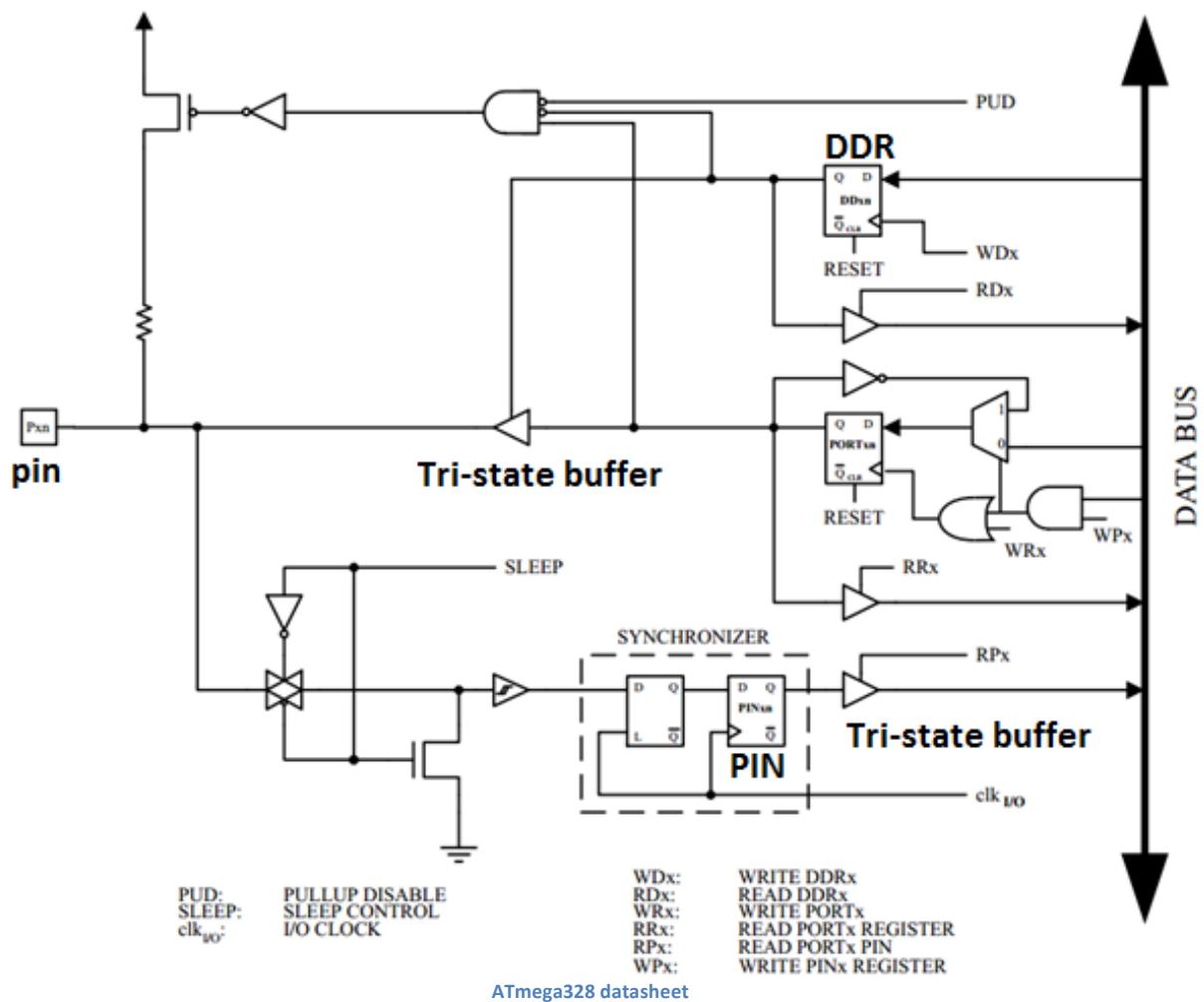
The initial power-up state of all pins is as inputs- the DDR (data direction register) output is 0. This means the **tri-state buffer** has a high Z output (it's like a tap is off) and signals from the PORT register cannot go through to the pin. The signal (1 or 0) from the physical pin comes through 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⁾



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 $\text{DDRA} |= (1 \ll \text{A5})$. Or we could make all 8 bits of a port outputs all at once by writing $\text{DDRA}=0xFF$. At any stage in our program we can drive the physical pin high or low using the commands $\text{PORTA} |= (1 \ll \text{A5})$ or $\text{PORTA} \&= \sim(1 \ll \text{A5})$. 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.

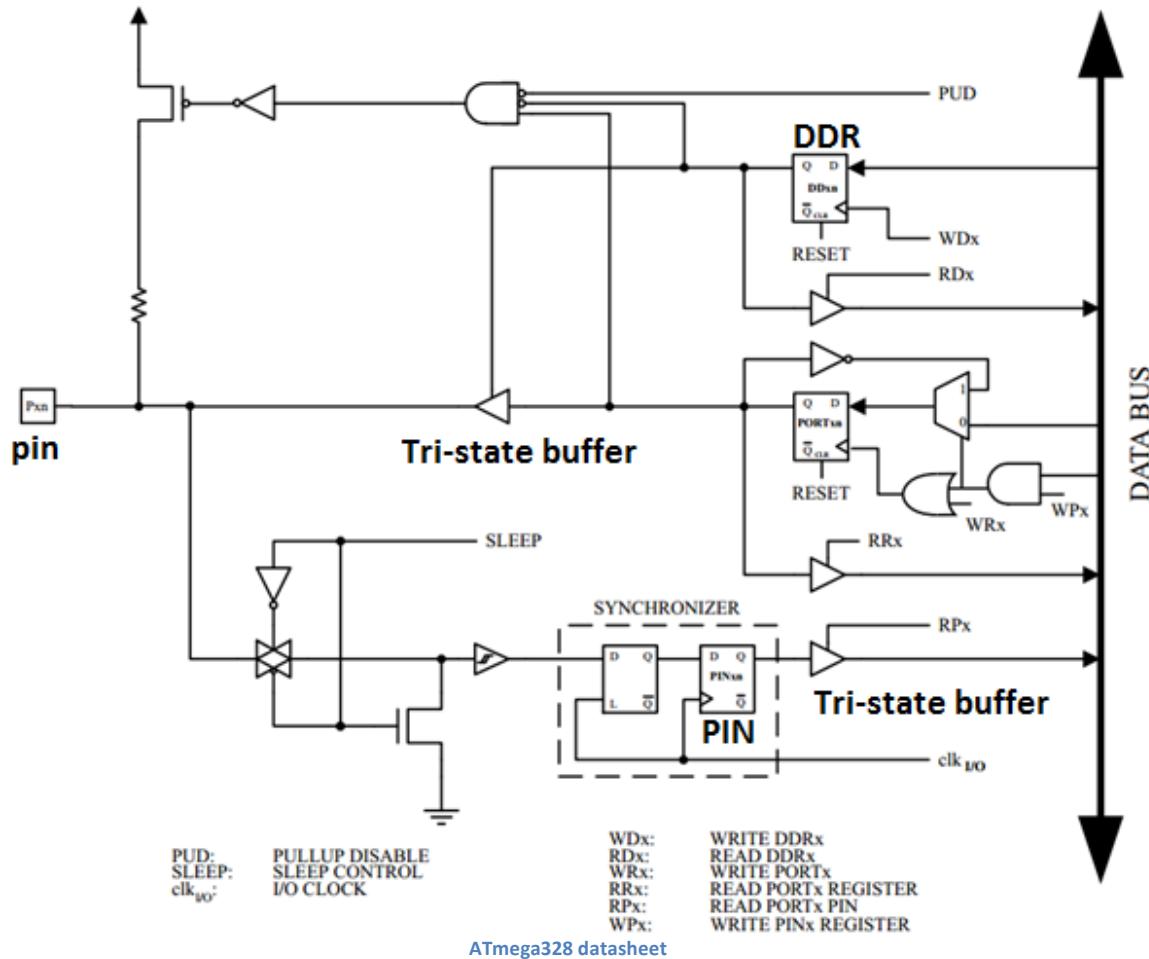
A special effect of this hardware is that the PIN register always reflects the state of the hardware pin even when the port is setup as an output. So you can check the state of an output device at any stage in your program by reading the PIN register - can you see how this could be used in the quiz game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

Confusing terminology alert - pin & port: 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', 'PIN 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.

2.7.6. The internal pullup resistor

Pullup resistors are such common components in ES's that uC designers often add internal pullup resistors to every I/O pin. These can be switched in and out using software commands.

Figure 18-2. General Digital I/O⁽¹⁾



Configuring the internal pullup resistor.

1. The pin must be configured as an input
 2. Turn on the ...



There is a good reason for providing the internal pullup resistor. Floating (unconnected) pins can become troublesome for uC's. An unconnected pin can detect stray electric charge (noise) surrounding the uC which can cause the input circuit to flip state between high and low rapidly. Switching like this causes unwanted power consumption.

There are at least two solutions to this problem:

1. After setting up all your I/O registers set the pullup resistor for any unused pins
 2. Add your own external pullup resistors - why??

2.7.7. State

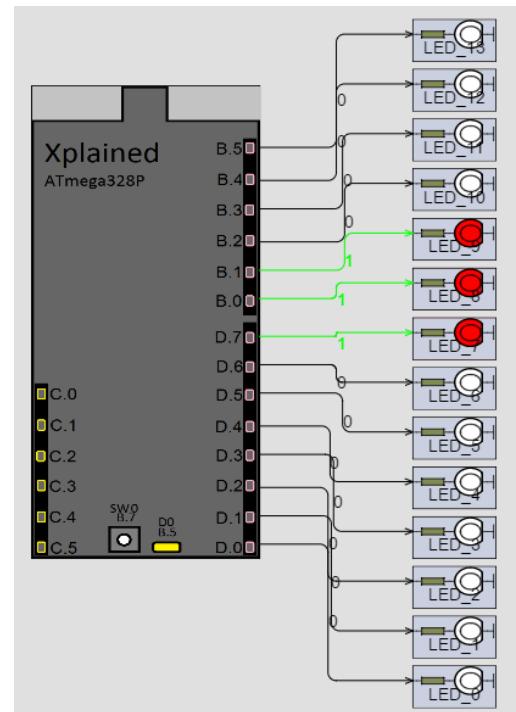
- Explain state - the interrelatedness of the past, present and future conditions of an ES

A core understanding of the ES as an automaton is that it is a sequential device; instructions are carried out one after another in a sequence. This has a significant side effect; the concept of state. The future state (condition) of the system is entirely dependent upon the current state and what has happened previously.

In this exercise the concept of state can be seen when transitioning our Cylon program to more LEDs and having to work with two ports at once, and making the pattern run smoothly when LEDs on both ports must be controlled in one step.

The Cylon14

```
while (1) {
    PORTD = 0b00000001; //turn on the first LED
    _delay_ms(TIME_DELAY);
    PORTD = 0b00000011; //turn on TWO LEDs
    _delay_ms(TIME_DELAY);
    PORTD = 0b00000111; //turn on THREE LEDs
    _delay_ms(TIME_DELAY);
    PORTD = 0b00001110; //shift pattern by one
    _delay_ms(TIME_DELAY);
    PORTD = 0b00011100; //shift pattern by one
    _delay_ms(TIME_DELAY);
    PORTD = 0b00111000; //shift pattern by one
    _delay_ms(TIME_DELAY);
    PORTD = 0b01110000;
    _delay_ms(TIME_DELAY);
    PORTD = 0b11100000;
```

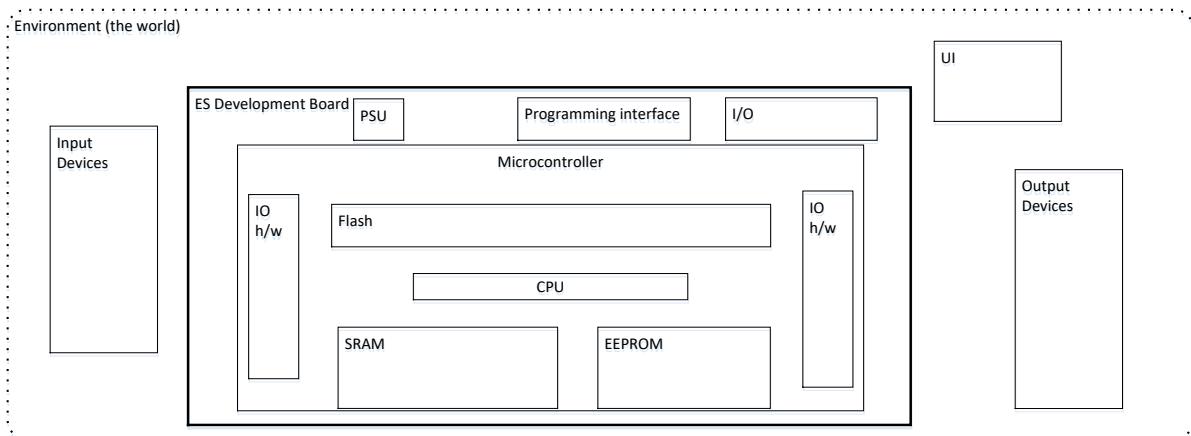
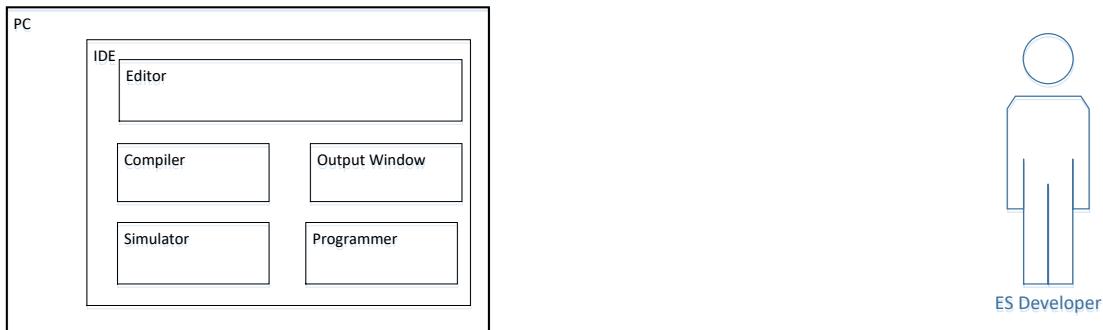


2.7.8. ES development overview

So far you have used a simulator to develop your ideas around simple AVR uC based ES's,

When developing ES software a number of processes are encountered

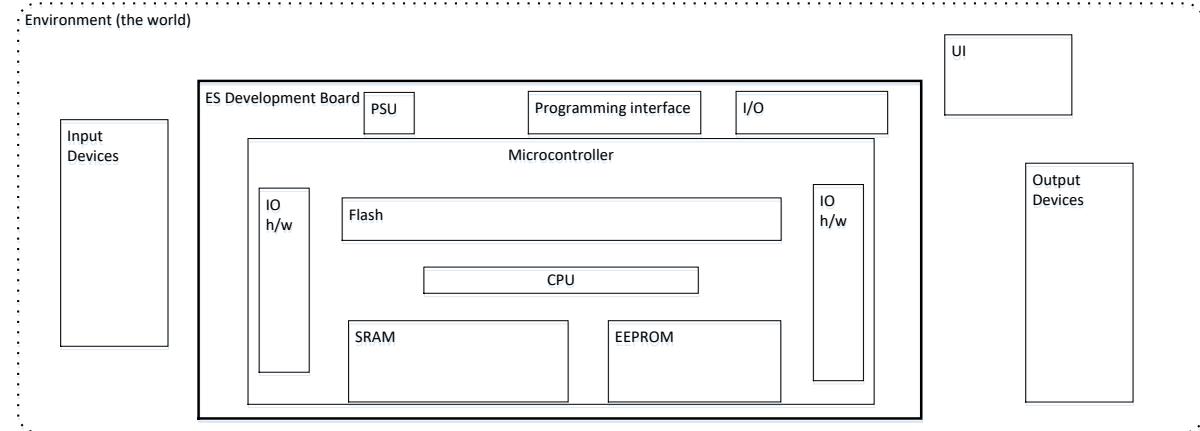
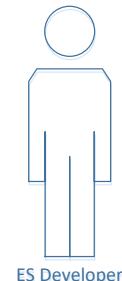
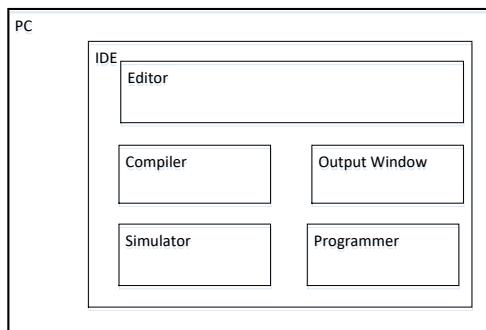
1. IDE – Atmel Studio 7
2. Writing code
3. Syntax errors
4. Compiling
5. uploading



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

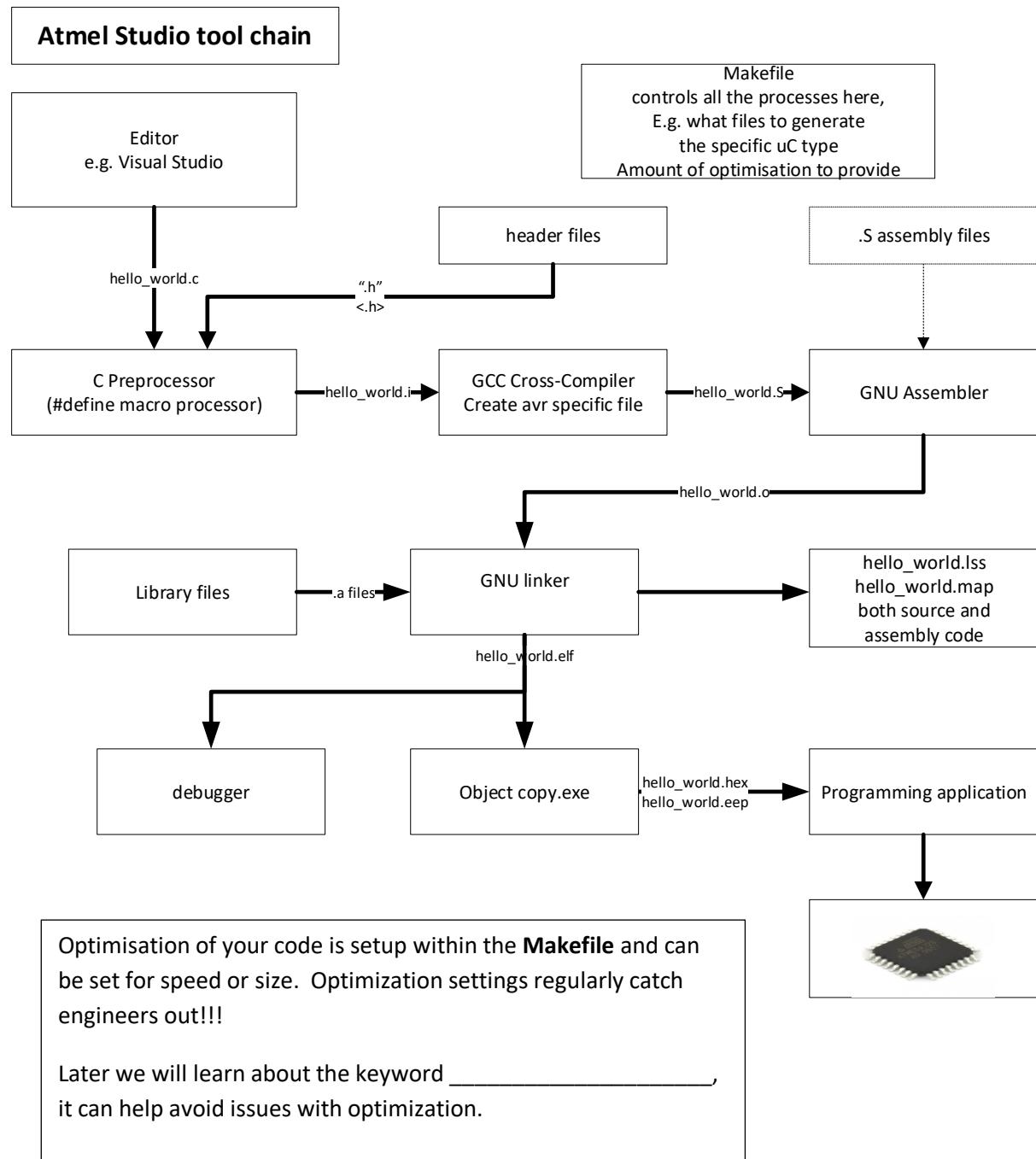


What could we add to an ES to help assist debugging?



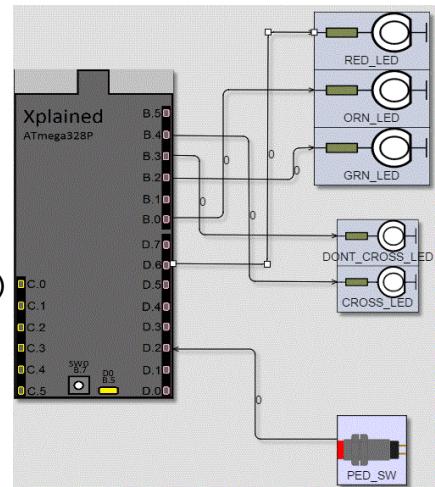
How we get from C to hex binary file in ATTEL Studio 7.

Describe the functions of an IDE - the process of compilation



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.     DDRD |= (1 << PD6); //D6 set as output
26.     // Activate internal pull-up resistor
27.     PORTD |= (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)
```

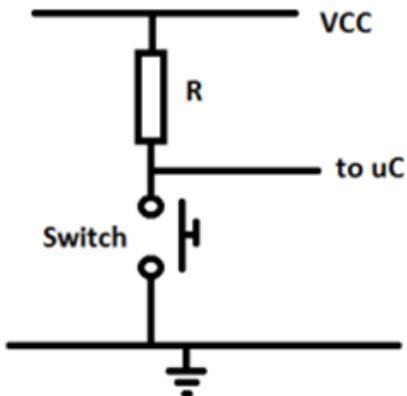


Which line(s) of code is/are part of making the ES responsive to its environment?

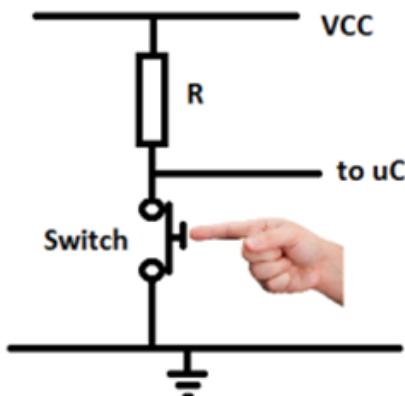


The pullup resistor is a critical component in making the input responsive. When the switch is:

- **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<<PD2)

if ( SWITCH_IS_HIGH ){
```

```
}
```

Draw a pulldown circuit



2.8.1. 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
}
```

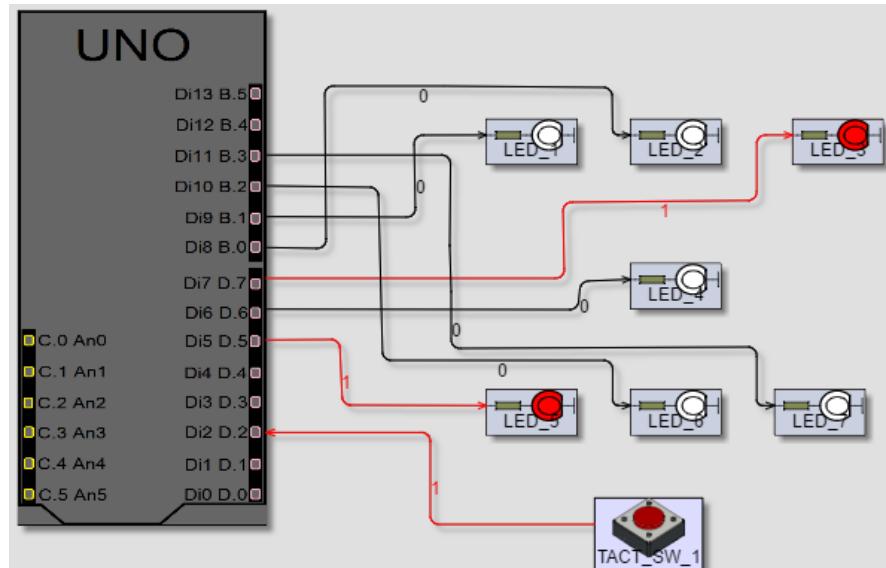
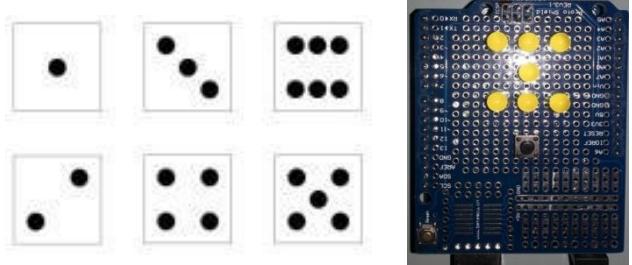
Making code transparent with functions

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?



2.8.2. 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...

***** 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; //turn on first LED
        _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 = 0b11100000;
        ...
    }
} //end while(1)
} //end of main
```



www.freefoodphotos.com

When a colleague opens up your code, what does 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 documents.

2.8.3. Apply a formal approach to writing program code

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 and 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. }
```

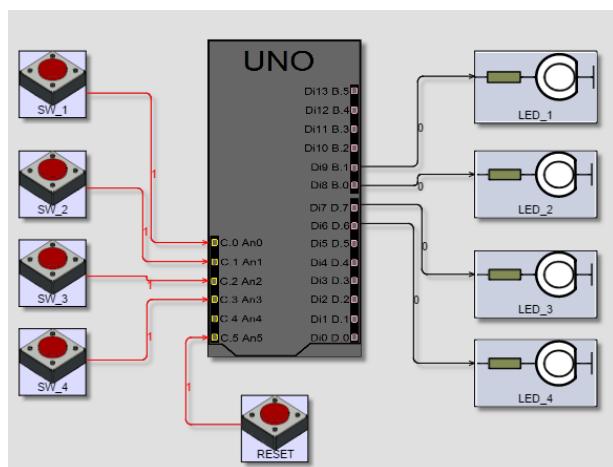
2.8.4. Polling

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, SET student evaluations) to see what is happening. When a uC has to check multiple inputs or variables to see if they have changed, this means that the program has to **poll** or regularly check each pin. Critically polling must be at a rate that is fast enough not to miss any event. When polling input pins, the rate of polling relates to how fast the environment might change state. If we poll switch inputs faster than the environment can change them, then no environmental change will ever be missed and we will have the illusion of concurrency. If we do not poll fast enough then the state of an input might change and then change back before we have polled it, that's bad as we might miss an important change in the environment. In this example polling is demonstrated by sequentially checking four contestants' switches. The illusion of concurrency is demonstrated by the rate of polling being so high that it appears no contestant switch press is ever missed, which might give an unfair advantage to another contestant.



www.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.

while (1) {



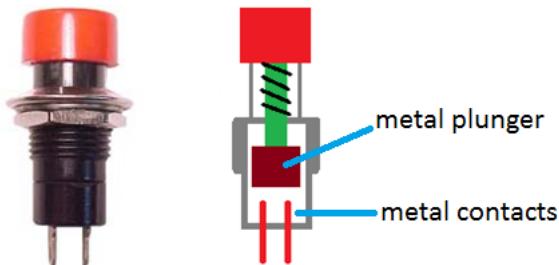
A 'cheat' for this game might be to favour a particular switch by polling it more often!!!



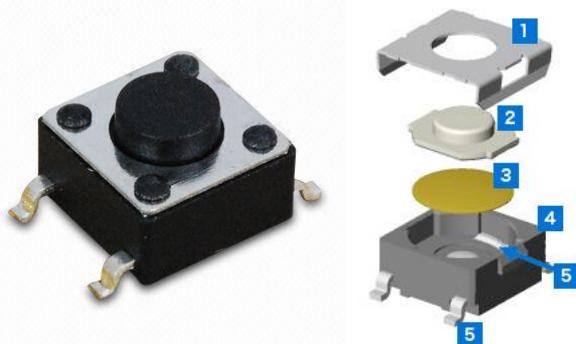
2.8.5. Hardware topic: Switch issues

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 _____

<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?



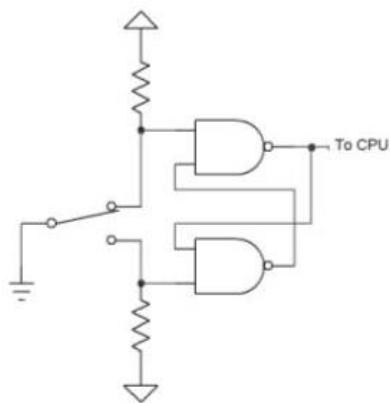
De-bouncing switches

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

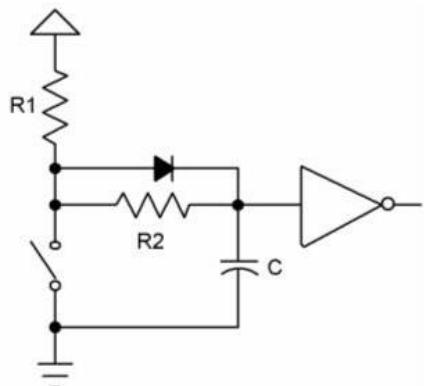
Hardware solutions

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

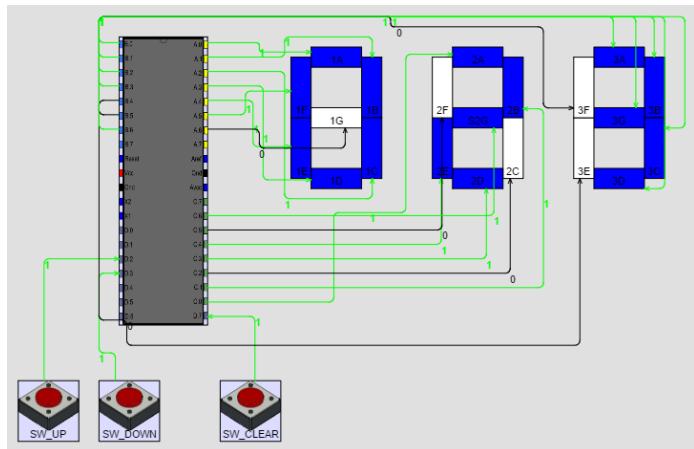
Often a short delay using `_delay_ms(...)` will be sufficient, sometimes more complex timing methods are used. We will look at using simple delays to solve issues in specific contexts.

2.9. Memory mapped IO and Pointers

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

In ENGGEN131 pointers were introduced. Think about what pointers are for, to store the address of a variable. In an 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 SRAM or FLASH memory. 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.



We **could** write three different display functions

```
void disp(int16_t number){
    //split the number into 3 digits
    dispHundreds (hundreds);
    dispTens (tens);
    dispUnits (units);
}
void dispHundreds(uint8_t digit){
    if(digit == 0){
        PORTA = 0b11000000; //PORT A
    }
    if(digit == 1...
    if(digit == 2...
}
void dispTens(uint8_t digit){
    if(digit == 0){
        PORTC = 0b11000000; //PORT C
    }
    if(digit == 1...
    if(digit == 2...
}
void dispUnits(uint8_t digit){
    if(digit == 0){
        PORTB = 0b11000000; //PORT B
    }
    if(digit == 1...
    if(digit == 2...
} (but this is very wasteful of our FLASH memory)
```

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**

$$9/3 = 3 \quad 9 \% 3 = 0$$

$$10/3 = 3 \quad 10 \% 3 = 1$$

$$11/3 = 3 \quad 11 \% 3 = 2$$

$$12/3 = 4 \quad 12 \% 3 = 0$$

$$\text{modulus} = A - B * (A/B)$$

no rounding

e.g. if count = 246

$$\text{hundreds} = \text{count} / 100 =$$

$$\text{tens} = \text{count} \% 100$$

$$\text{tens} = \text{tens} / 10$$

$$\text{units} = \dots$$

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
    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	v		
0x102			
0x101	o		
0x100			
...	a		
0x29	t	PIND	
0x28	i	PORTC	
0x27	l	DDRC	
0x27	e	PINC	

Com

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.

2.10. 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. As commands are processed sequentially, and can take various paths in software, due to conditional (if) statements, it can be impossible to predict how long even a simple program loop can take; making it impossible to create accurate timing.

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



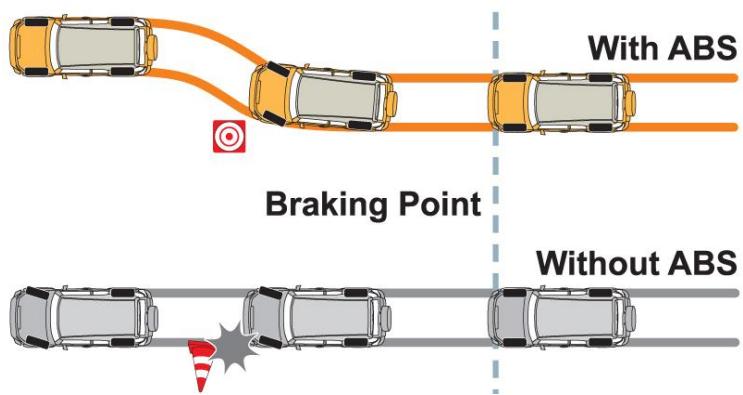
Some events in the real world need to be accurately controlled within tight timing constraints- sometimes in the order of 10^{-6} or 10^{-9} seconds. An example is a petrol engine, controlling the timing of the explosion of fuel in the cylinder is critical for power, fuel efficiency and emission reduction.

A microcontroller is required to control the timing of output devices such as this not only with great accuracy but reactively (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 the speed of each wheel of a vehicle needs to be measured accurately for the ABS (anti-lock braking system) to work. A microcontroller can measure the speed of each wheel then determine whether one has a different speed to the others to see if the car is at risk of losing traction (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



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

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 and measure the wheel speed, meaning the uC will never miss an event.

2.10.1. 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 slightly differently to describe their different function.

Timers are complex - and allow complex relational problems to be solved

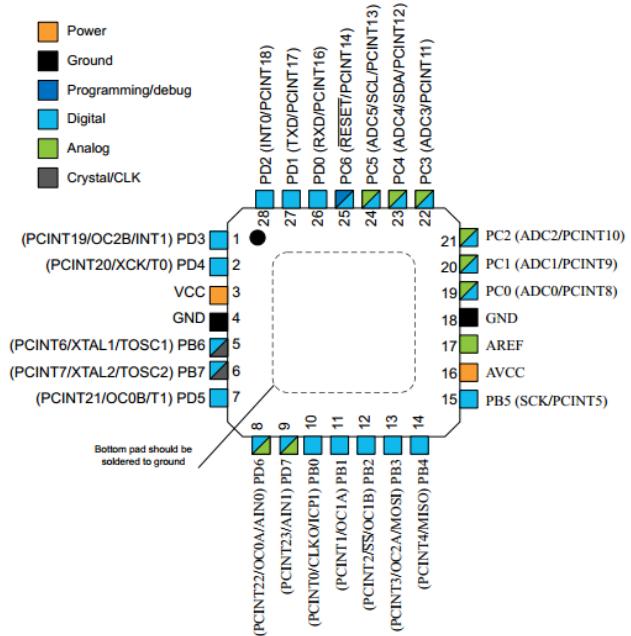
- AVR's generally have
 - two 8 bit timer/counters
 - one 16 bit timer/counter
- clocking of the timer can be configured to be from:
 - the CPU clock source
 - the CPU clock source divided down (via a pre-scaler/divider)
 - Timer0 and Timer1 can be clocked via a related pin
 - Timer2 can be clocked from 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 some timers



Figure 5-2. 28-pin MLF Top View

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



2.10.2. Timer example – ‘heart beat’ led flash

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

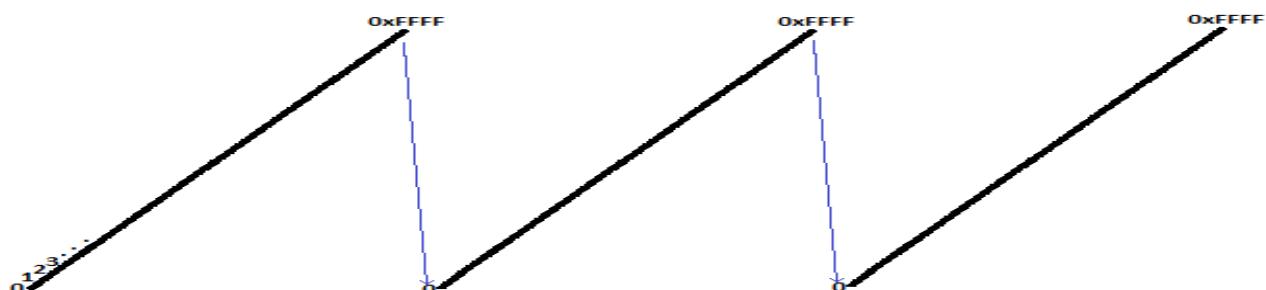
How to accurately control the timing of an external device without involving counting in the programs main loop.

One example of this ‘heart beat’ LED on the front of a piece of equipment. Users need to feel confidence that a piece of equipment is functioning, a steady lit LED is often used to indicate that power is on; a flashing LED makes users think that the equipment is actually working (like it’s 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. This diagram is used to show the counting process from 0 to 65536 and then overflow back to 0 in a 16 bit timer.

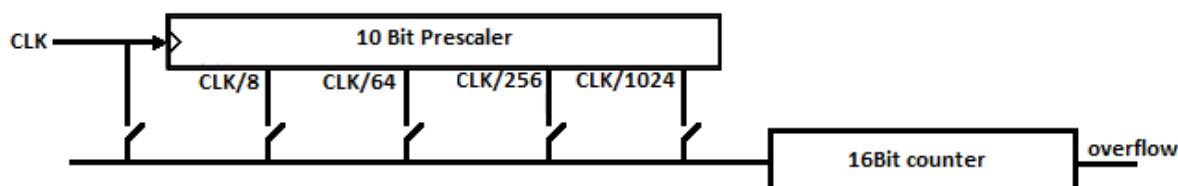


The rate of the counting is dependent upon F_CPU, the clock frequency of an Xplained mini, Arduino Nano, or Arduino Uno is 16MHz (if you are using another development board it may be different – so watch out)

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 (another counter/divider) for dividing the crystal frequency down before it goes to the main timer counter.



16Bit timer counter

Clock	No prescale	/8	/64	/256	/1024
TCNT1 input	16Mhz	2MHz			
Time period of 1 count	0.0625 uS 62.5 nS	0.5uS 500nS			
Overflow (65536 counts)	4,096 mS	32.768mS			

2.10.3. 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)					

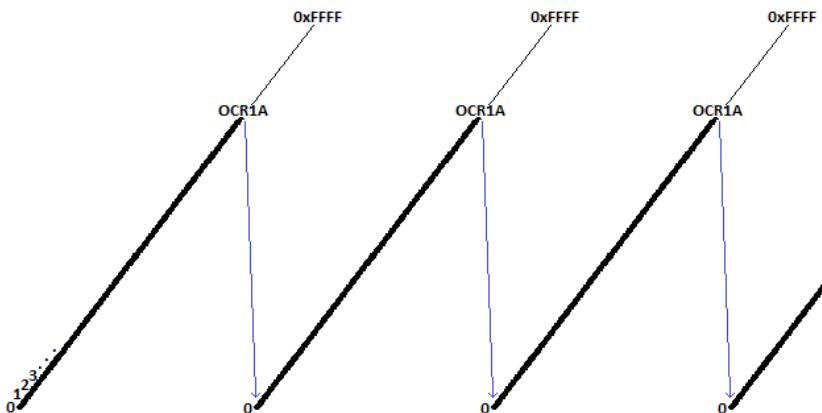
Can you think of a way to use an 8 bit timer to get a delay of about 1 second?



2.10.4. 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 to 0, but to some lesser value _____ and then overflows to 0

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
```

2.10.5. 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 hardware 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?



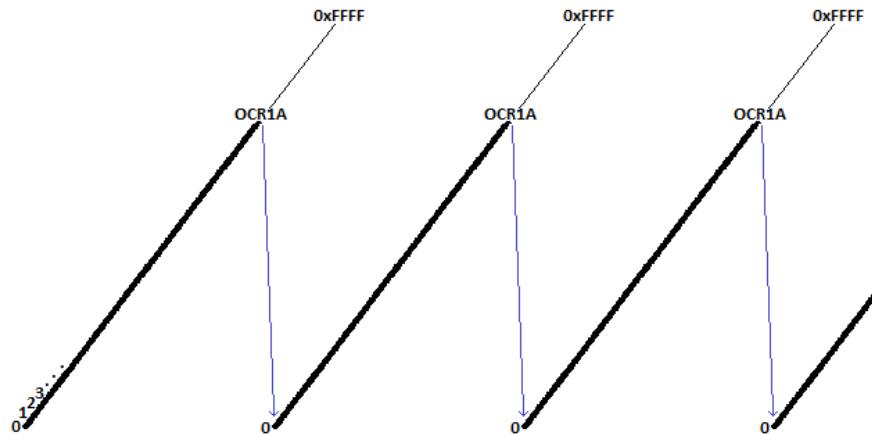
if there is a formula for this then there are calculators to help you figure it out as well, HOWEVER...

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 = _____

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

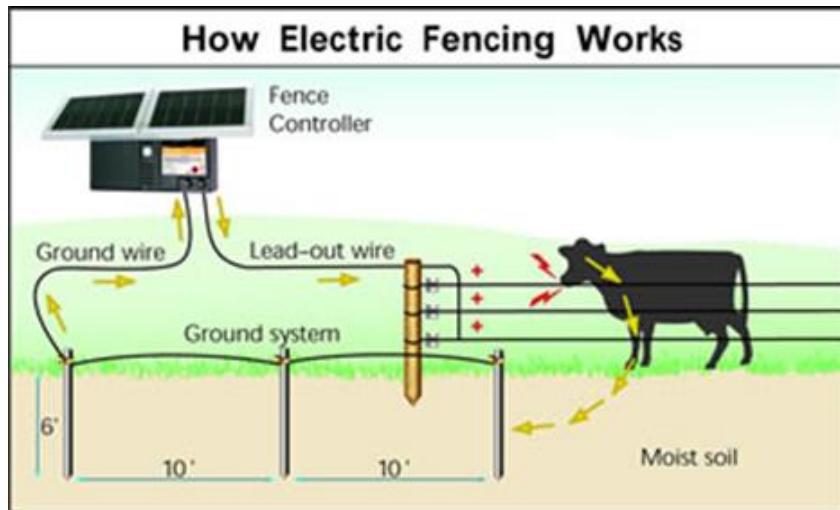
OCR1A and OCR1B registers can be configured to control their respective pins OC1A and OC1B to control two different output devices, or the two interrupts COMPA_vect and COMPB_vect can be configured to provide two different interrupt events. One example that combines the two timer Interrupts would be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



Write code that uses two switches to enter hours and minute for a 24 hour timer



2.10.7. Timer Registers

Each timer in the ATmega328P has a set of registers- these are described in the ATmega328 datasheet (sections 15,16,17,18)

- the timer/counter register itself
 - 8 bit - counts from 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
- TCCROA 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
- OCROA T0 output compare register A the value that TCNT0 will be compared to
- OCROB 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 16.11.8)

TCCR1B – Timer/Counter1 Control Register B								
Bit	7	6	5	4	3	2	1	0
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
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	$\text{clk}_{\text{I/O}}/1$ (No prescaling)
0	1	0	$\text{clk}_{\text{I/O}}/8$ (From prescaler)
0	1	1	$\text{clk}_{\text{I/O}}/64$ (From prescaler)
1	0	0	$\text{clk}_{\text{I/O}}/256$ (From prescaler)
1	0	1	$\text{clk}_{\text{I/O}}/1024$ (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...?



Feeling



then ask questions on PIAZZA e.g.

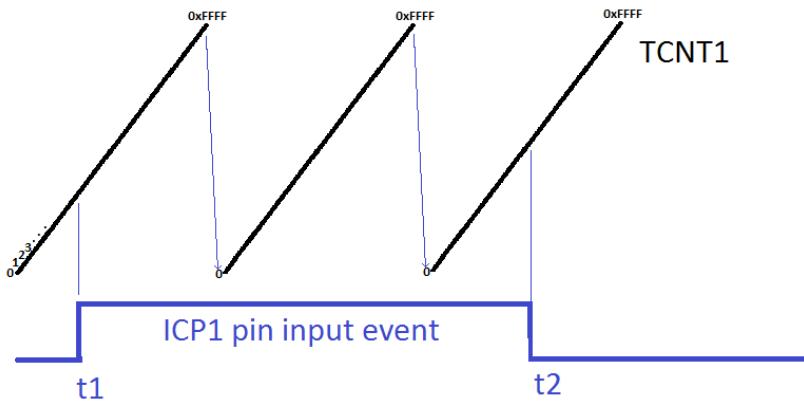
- What is the answer to...?
- This is what I understand/have done... what does ...?
- What does ...mean?
- I am unsure about this concept...

2.10.8. 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 ... 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.

How long is the pulse and what is the algorithm for measuring the pulse length?



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?

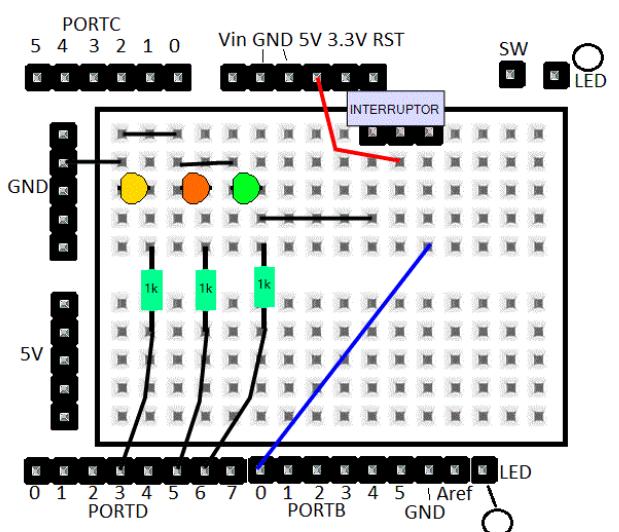
The clock is 16MHz and the prescaler is set to 1024.



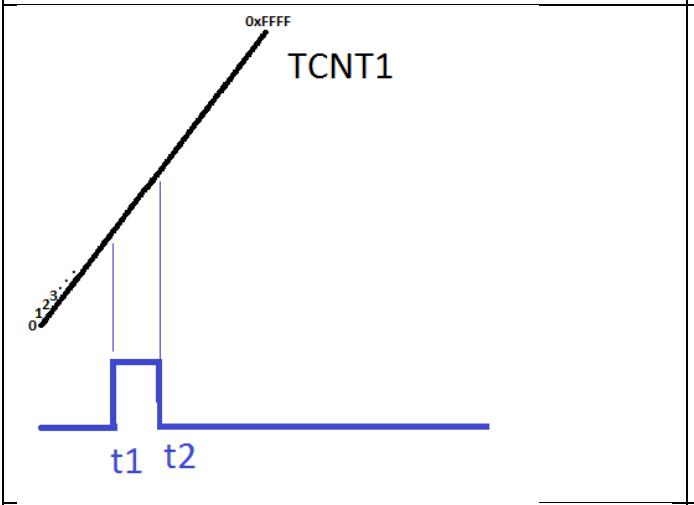
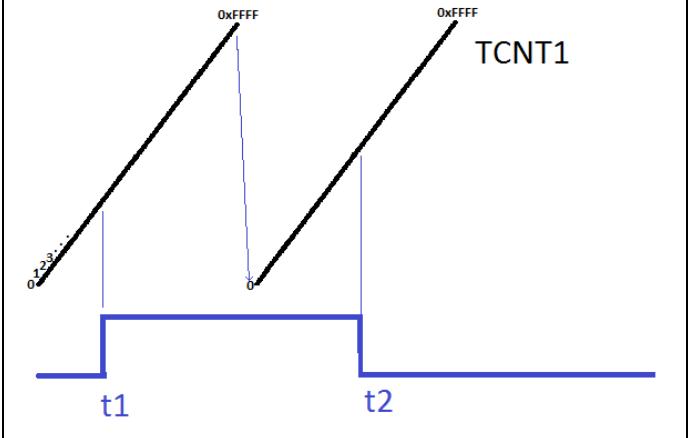
In the lab you will be displaying the time in hundredths of seconds (0.1S to 25.5S) using flashing LEDs

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.

We need an algorithm for the process, there are two complications that have been 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
 <p>A graph showing a digital signal and the value of TCNT1 over time. The digital signal is represented by a blue step function. It starts at 0, rises to 1 at time t_1, and falls back to 0 at time t_2. The TCNT1 value is shown as a black diagonal line starting from 0. At time t_1, it reaches a maximum value of $0xFFFF$ and then begins to decrease linearly. At time t_2, it reaches 0 again. A vertical dashed line is drawn from the start of the pulse to the peak of the TCNT1 curve, and a horizontal dashed line is drawn from the end of the pulse to the peak of the TCNT1 curve.</p>	$\text{Pulse duration} =$
 <p>A graph showing a digital signal and the value of TCNT1 over time. The digital signal is represented by a blue step function. It starts at 0, rises to 1 at time t_1, and falls back to 0 at time t_2. The TCNT1 value is shown as a black diagonal line starting from 0. It reaches a maximum value of $0xFFFF$ at time t_1 and then begins to decrease linearly. It reaches 0 again at time t_2. A vertical dashed line is drawn from the start of the pulse to the peak of the TCNT1 curve, and a horizontal dashed line is drawn from the end of the pulse to the peak of the TCNT1 curve.</p>	$\text{Pulse duration} =$

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)	ICNCl	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 – ICNCl: 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

The algorithm for the pulse width measurement

```
#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 resolution we can get using that prescaler with the 16MHz clock

How much resolution is enough resolution?

This depends entirely upon the situation.



What determines the accuracy of uC timers?



External peripherals	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 <code>_delay_ms()</code> 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 $\frac{1}{2}$ 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.

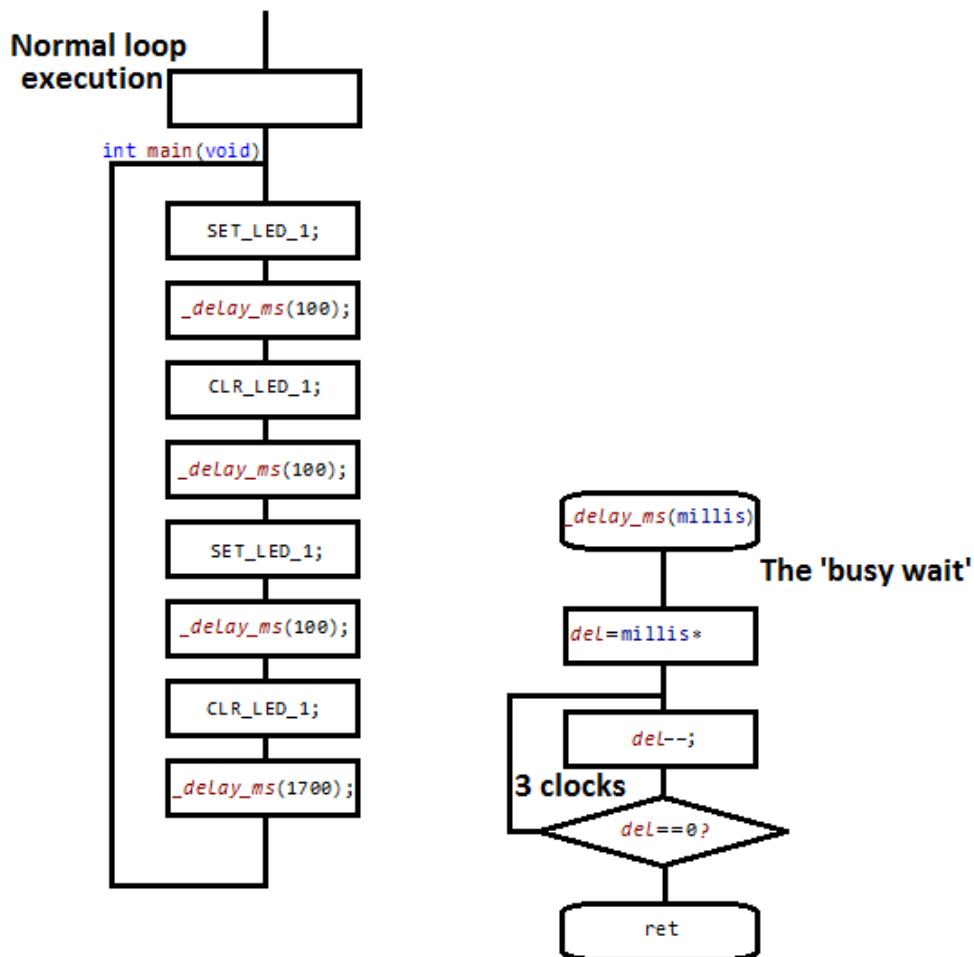
2.11. External Interrupts

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

Microcontrollers have another peripheral called an external interrupt controller, this allows the ES to react immediately to untimed but urgent things that occur around in its environment, without having to poll for them or set up a timer to check them regularly.

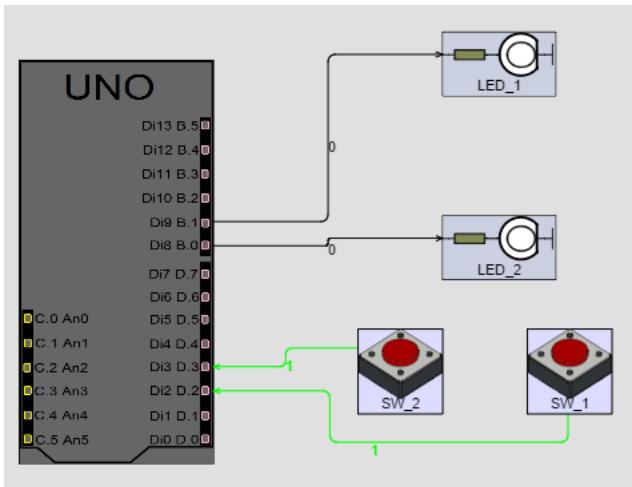
uC's are sequential devices so they can only do one thing at a time; if the uC is busy doing one task it might miss an input of short duration, or if the number of inputs to check is large then even if it checks them all, it may be not be often enough, and miss one, or more. Importantly if the input 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.

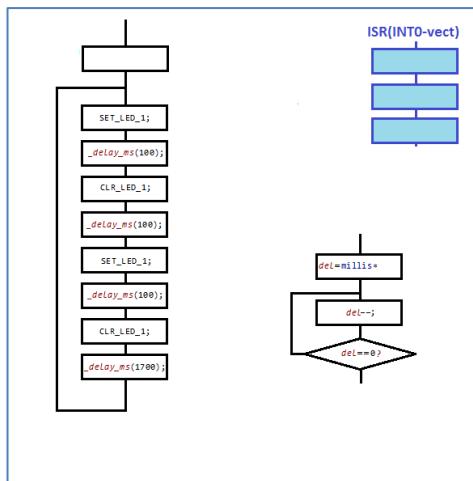
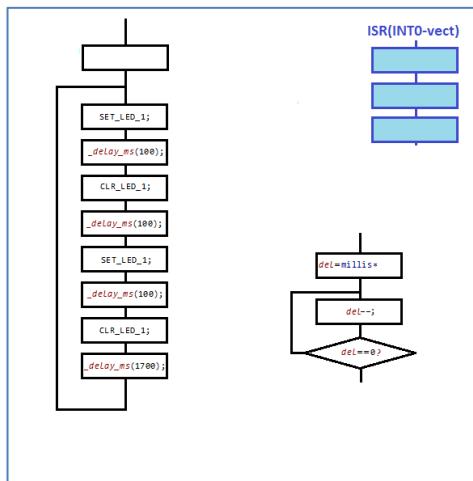


In response to this manufacturers now implement special hardware features in uC's called external interrupts. There are several different interrupt facilities in the AVR. By connecting an input to an external interrupt pin we don't have to poll the pin but let the interrupt do it for us. A hardware interrupt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

```
#define F_CPU 16000000//crystal
/***** 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<<PB1)
#define CLR_LED_2    PORTB &= ~(1<<PB1)
#define TOGGLE_LED_1  PORTB ^= (1<<PB3)
//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) {
    // make these pins outputs
    DDRD &= ~ (1 << PD2);
    DDRD &= ~ (1 << PD3);

    PORTD |= (1 << PD2); //activate internal pullup resistor
    PORTD |= (1 << PD3); //activate internal pullup resistor
    //Interrupt setups

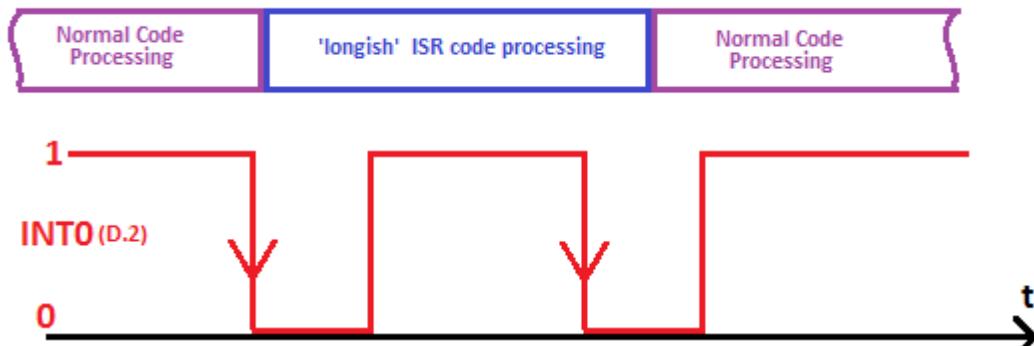
    /**** Loop code ****/
    while (1) {
        SET_LED_2;
        _delay_ms(100);
        CLR_LED_2;
        _delay_ms(100);
        SET_LED_2;
        _delay_ms(100);
        CLR_LED_2;
        _delay_ms(1700);
        if (SW_1_IS_LOW){//poll the switch
            TOGGLE_LED_1;
        }
    } //end while(1)
} //end of main
```

2.11.1. 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).

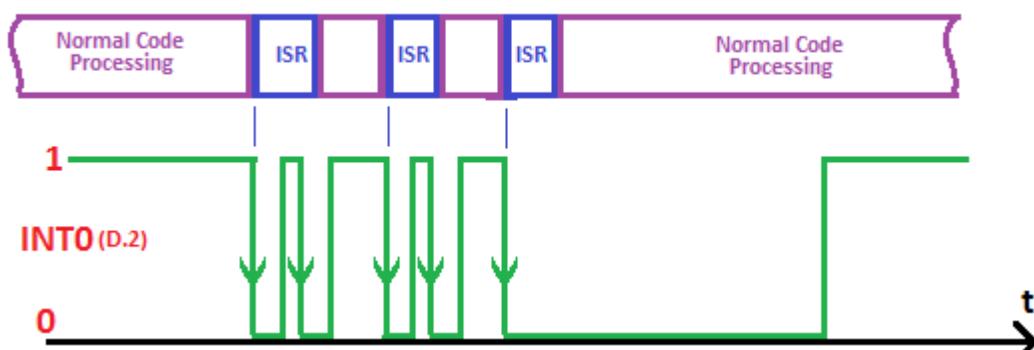


So how long should an ISR be?



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

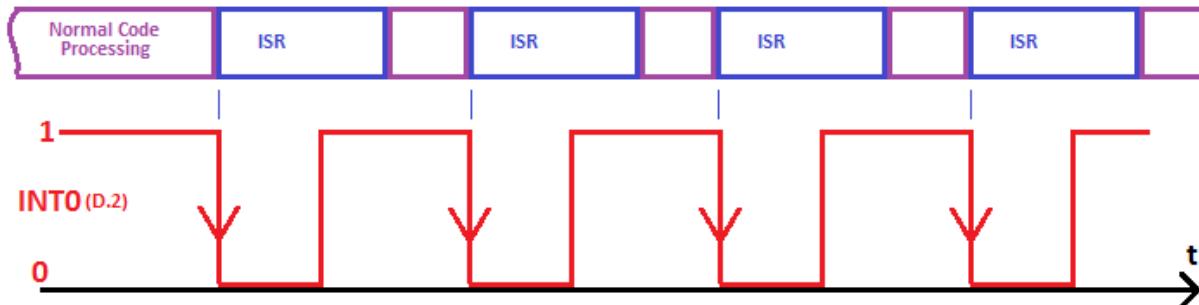


So how short should an ISR be?



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  
}
```

Is atomicity a problem when using only 8 bit types?



2.12. 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 registers for the uC USART peripheral

2.12.1. External serial and parallel peripherals

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 peripherals 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)

2.12.2. Serial communication protocols

There are several different common serial communication protocols. The AVR serial hardware circuits are the:

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

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.

Communication channel types or modes

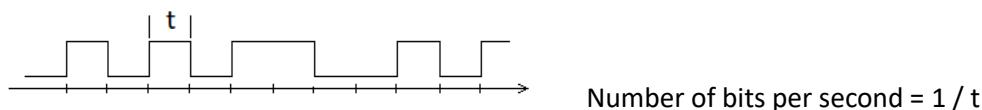
An important aspect of sending and receiving information reliably is that there must be an agreement between the transmitter and receiver as to which direction data is moving and when.

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

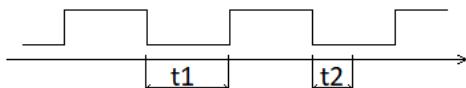
- Simplex - data goes in only one direction TO or FROM the uC
- Half-duplex - data goes both ways but not simultaneously - only one way at a time
- Duplex – data goes both ways simultaneously
- This helpful animation shows the differences between them
 - <http://www.iec-usa.com/Browse05/DTHFDUP.html>

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.



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 t_1 then the data would be _____
2. If the transmitter timing of the above signal was t_2 then the data would be _____

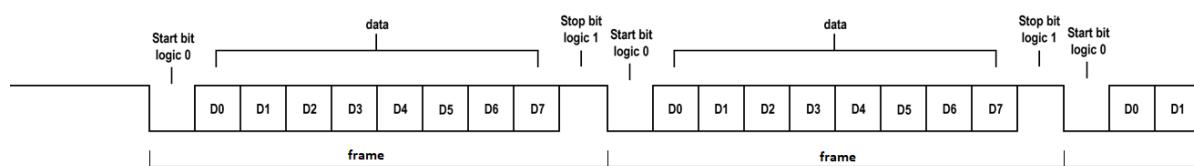
The receiver needs to be able to interpret this signal correctly; the two different ways of achieving this are synchronous and asynchronous methods:

- 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

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.

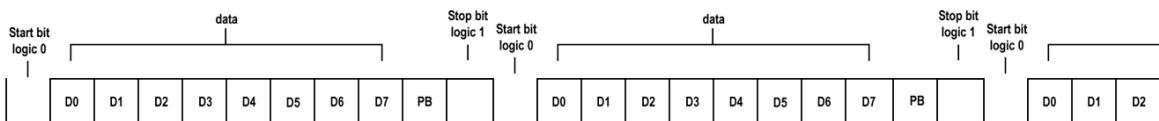
In asynchronous data transfer groups of bits or codes are sent with extra bits as part of a _____ to signal the beginning and end of each frame. The parameters for the frame below are 8 data bits, with 1 start bit (0) and 1 stop bit (1).



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 using a parity bit

Sometimes electrical noise in the environment can interfere with data transmission. In this case a particular bit might get corrupted (changed). One 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	010101010111	even	
B	010111110111	odd	
C	011111010111	odd	
D	010110110111	even	
E	011001010111	odd	



2.12.3. 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**

2.12.4. 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)
    UBRH = 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);
}
```

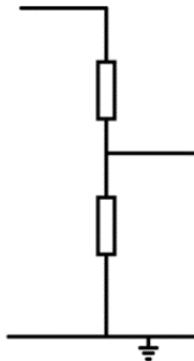
Extremely useful reference: <http://www.fourwalledcubicle.com/AVRArticles.php>

2.13. The ADC

The real world is not digital, it is analog. So for an ES to be responsive and reactive to real world events it must be able to convert information from the analog world to the digital one. This is the function of the internal ADC peripheral

2.13.1. The voltage divider

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



Working in context with voltage dividers is at the core of using ADCs and sensor subsystem design, and this involves understanding their model

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

Develop the models for these expressions

Vout =

Vin =

R1 =

R2 =

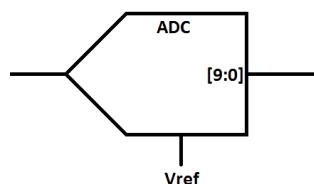
ADC conversion process

- 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 $V_{in} = 0$ the adc output value is _____

When $V_{in} \geq V_{ref}$ 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 model 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)

$V_{ref} = 1.1V, V_{in} = 0.67V$	ADC value =
$V_{in} = 2.2V, V_{ref} = 5.03V$	ADC value =
$ADC \text{ value} = 754, V_{ref} = 5.07V$	$V_{in} =$
$ADC \text{ value} = 432, V_{ref} = 1.1$	$V_{in} =$

ADC reference select

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

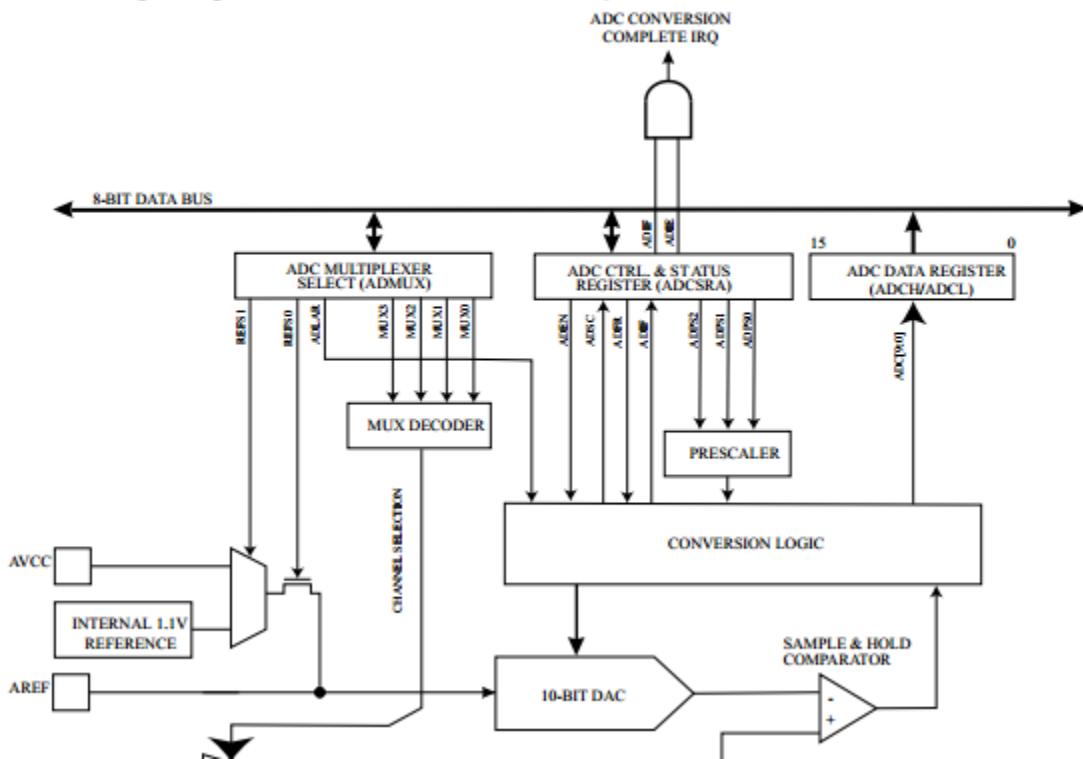


Table 28-3. ADC Voltage Reference Selection

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,

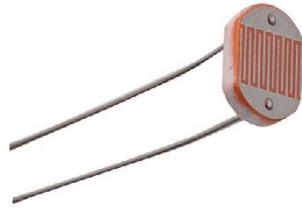
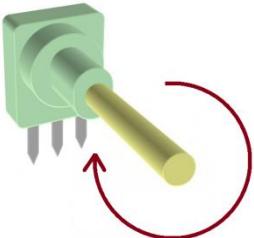
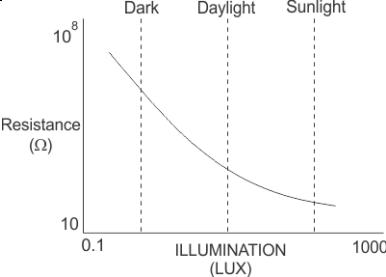
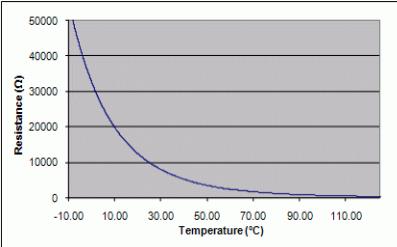
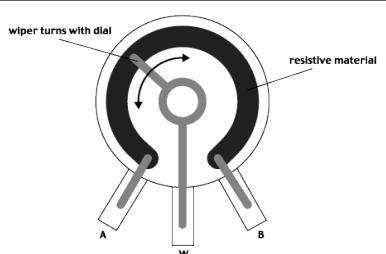
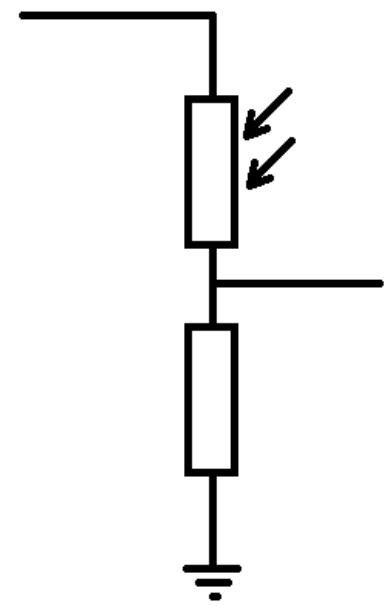
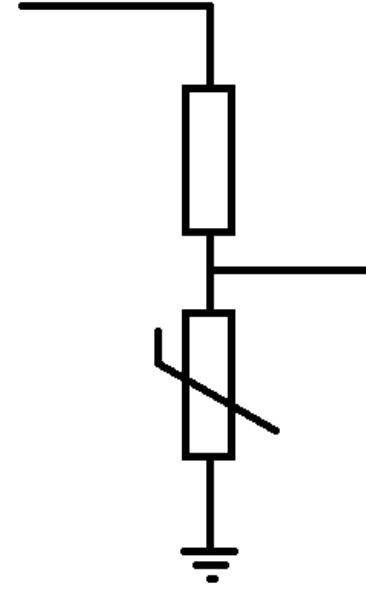
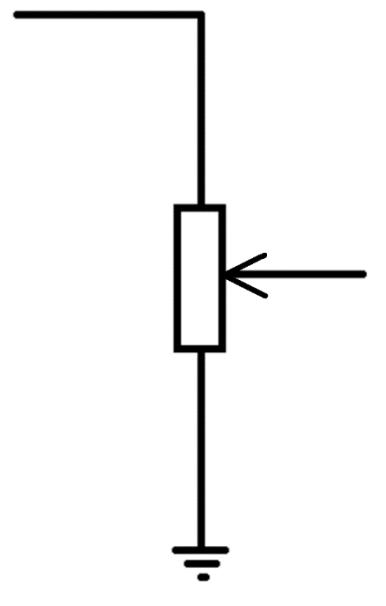
what would be the command(s) to select AVcc as the reference?



2.13.2. ES analog sensors

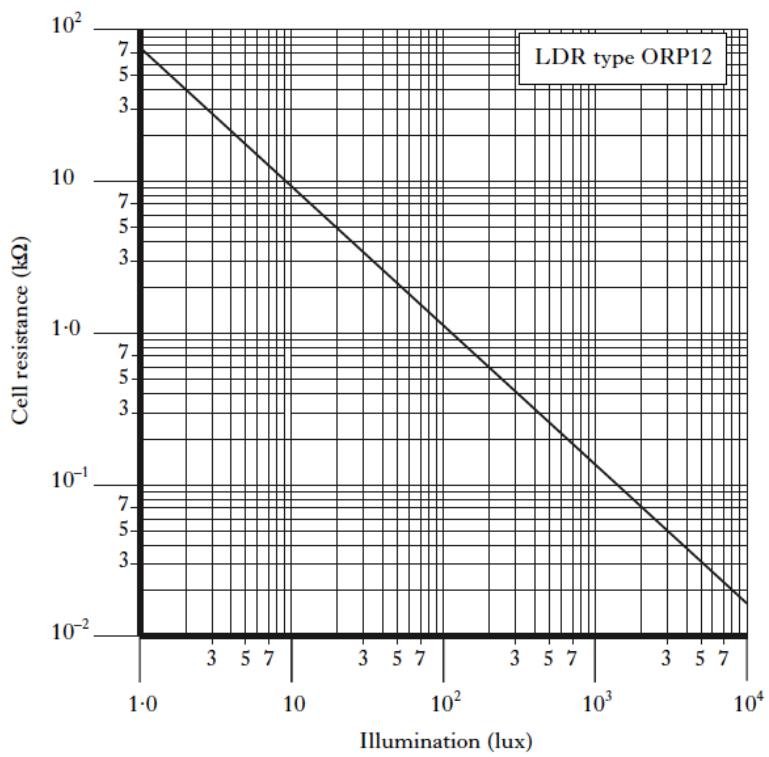
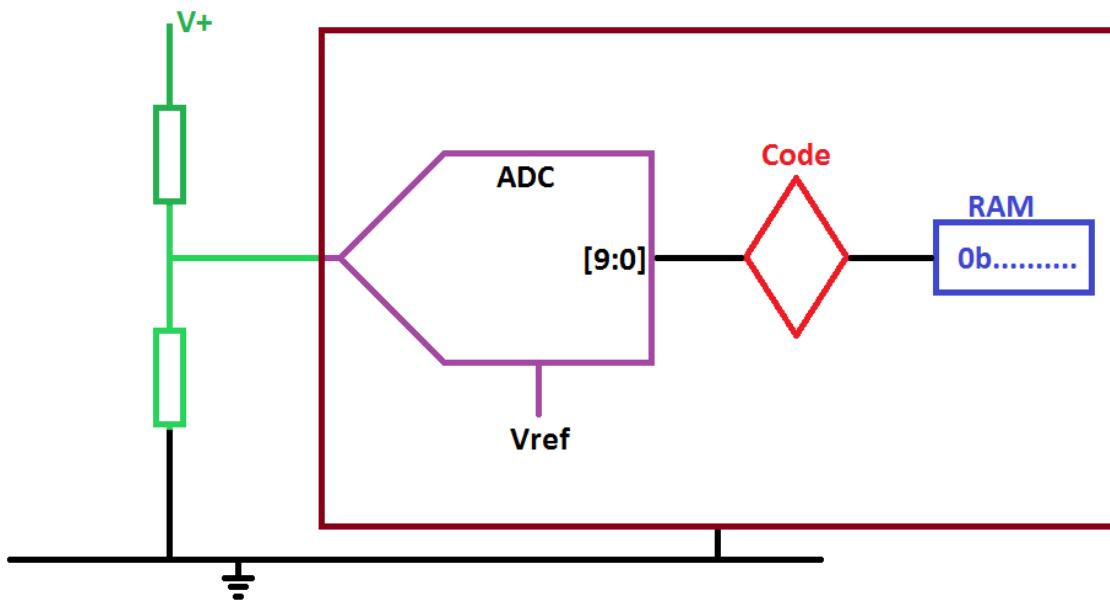
Describe the use of a sensor in a voltage divider circuit

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)
		
 Resistance (Ω) vs Illumination (LUX). The graph shows a logarithmic decrease in resistance from approximately $10^8 \Omega$ at 0.1 LUX to 10Ω at 1000 LUX. Vertical dashed lines mark 'Dark', 'Daylight', and 'Sunlight' conditions.	 Resistance (Ω) vs Temperature ($^{\circ}\text{C}$). The graph shows a exponential decrease in resistance from approximately 50000Ω at -10.00 $^{\circ}\text{C}$ to 1000Ω at 110.00 $^{\circ}\text{C}$.	 wiper turns with dial resistive material A, B A schematic diagram of a potentiometer showing a circular track with resistive material, a wiper arm with a contact point, and terminals A and B. An arrow indicates the wiper turns with the dial.
 A voltage divider circuit with a fixed resistor and a variable photodiode resistor connected between the top terminal and ground. The output voltage is taken across the variable resistor.	 A voltage divider circuit with a fixed resistor and a variable thermistor resistor connected between the top terminal and ground. The output voltage is taken across the variable resistor.	 A simple voltage divider circuit with a fixed resistor and a variable potentiometer resistor connected between the top terminal and ground. The output voltage is taken across the variable resistor.
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.

2.13.3. The conversion process



ORP12 Datasheet, www.alldatasheet.com

$$R1 = \text{LDR in sunlight (400lux)} \quad V_{in} = 5V, R2 = 10k, V_{ref} = 1.1V$$

Which way around should our Vdiv circuit be?

The value converted by the ADC will be:



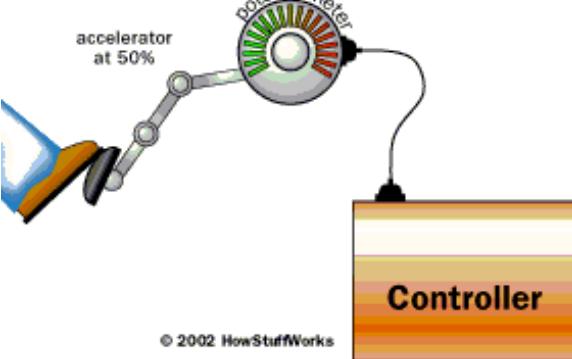
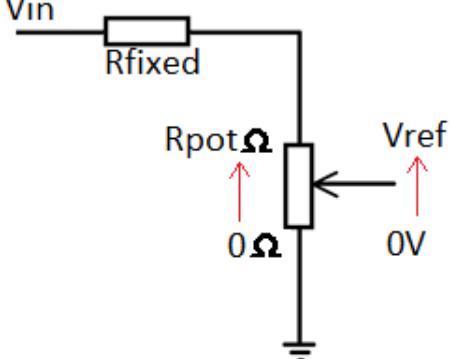
Solve problems of dynamic range and quantization when employing analog sensors

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
 <p>© 2002 HowStuffWorks</p> <p>Car accelerator potentiometer $V_{in} = 12V$, $V_{ref} = 1.1V$, $R_{pot} = 47k$ $R_{fixed} = ??$</p>	 <p>V_{in} → R_{fixed} → R_{pot} (variable from 0Ω to ∞Ω) → V_{ref} → 0V</p>

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’.

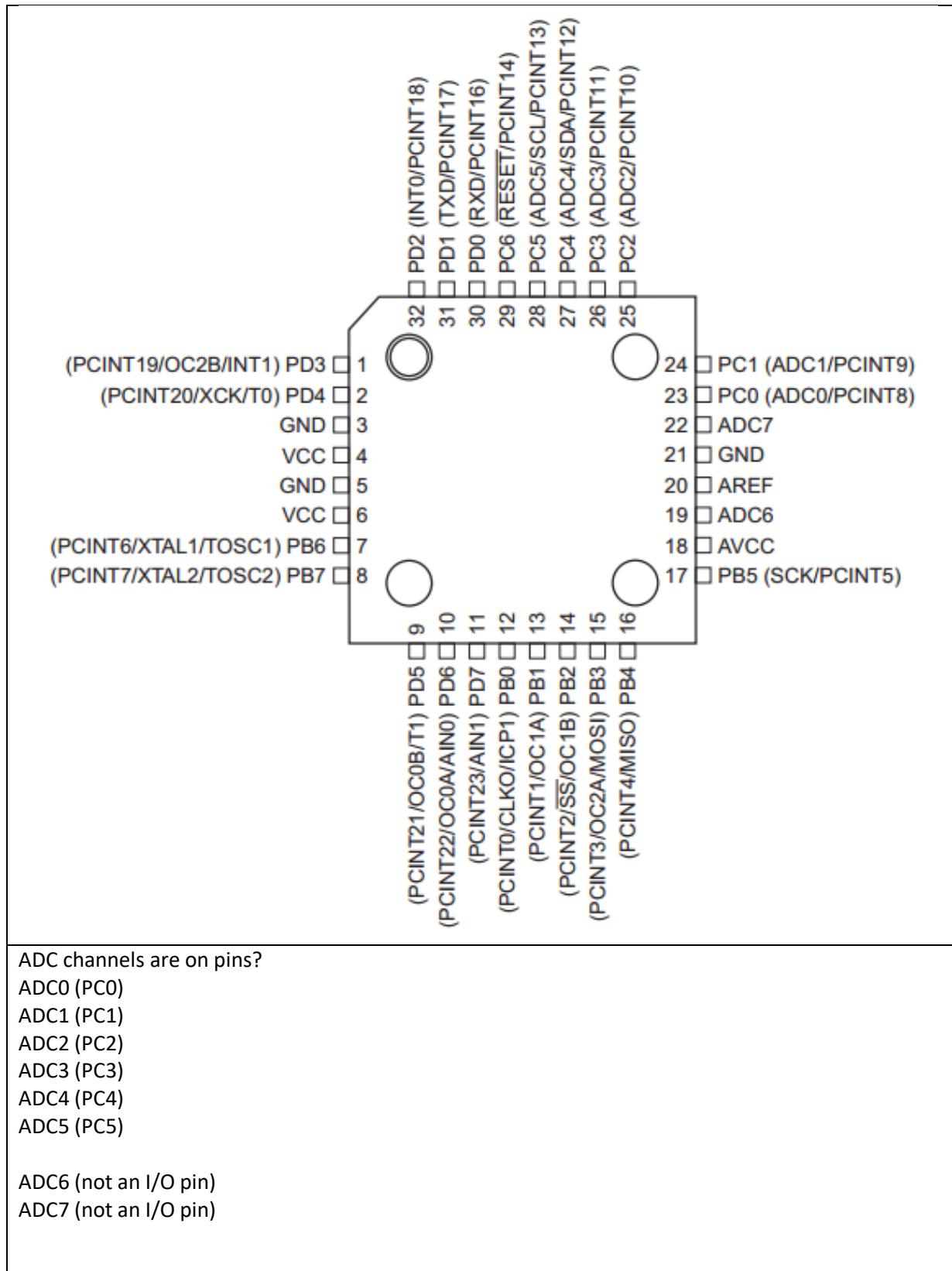
We need to know the situation that an ADC will be used in to make a decision about what are the ‘best’ or minimum attributes for it.



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.

2.13.4. AVR ADC inputs

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

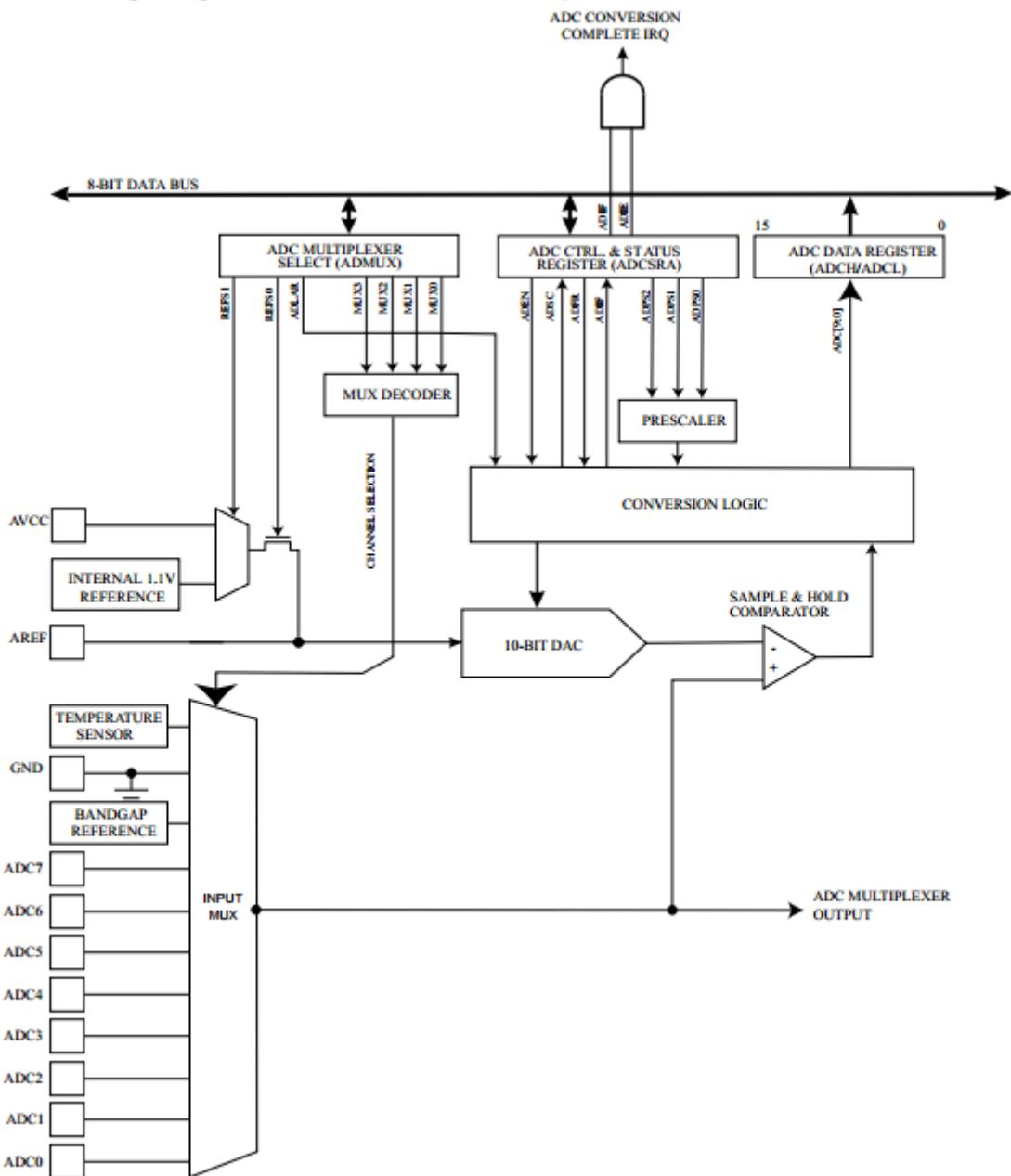


2.13.5. ADC Channel select

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, is made by selecting and clearing the appropriate bits in the ADMUX register, e.g if we wanted to read channel 3 (0011) then we need to set bits 1 and 0. If we wanted to read channel 8(1000) we need to set bit3. However we cannot just set the bits we need to

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 __ for an ADC conversion.



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



2.13.6. Sampling rate

The process of making an analog to digital conversion inside an AVR is via the method known as ‘successive approximation’. This is reasonably complex, involving OpAmp comparisons and electronic switching of resistor networks, so ADC conversions are ‘slow’ compared to manipulating binary numbers. The ADC circuits take 13 clock cycles and requires a clock frequency slower than the CPU frequency e.g. 50kHz to 200kHz. The clock rate is provided by the ADC prescaler which is setup using three bits in register ADCSRA.

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 division factor would meet the specification of 50kHz - 200kHz?

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



What is the command to set this division factor into the prescaler?

2.13.7. 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 (0x7A)	7	6	5	4	3	2	1	0
Read/Write	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Initial Value	0	0	0	0	0	0	0	0

Appreciating that these different modes exist, means that you can design a conversion process for any system that will make it responsive and reactive to its environment.

Datasheet
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.

In addition to

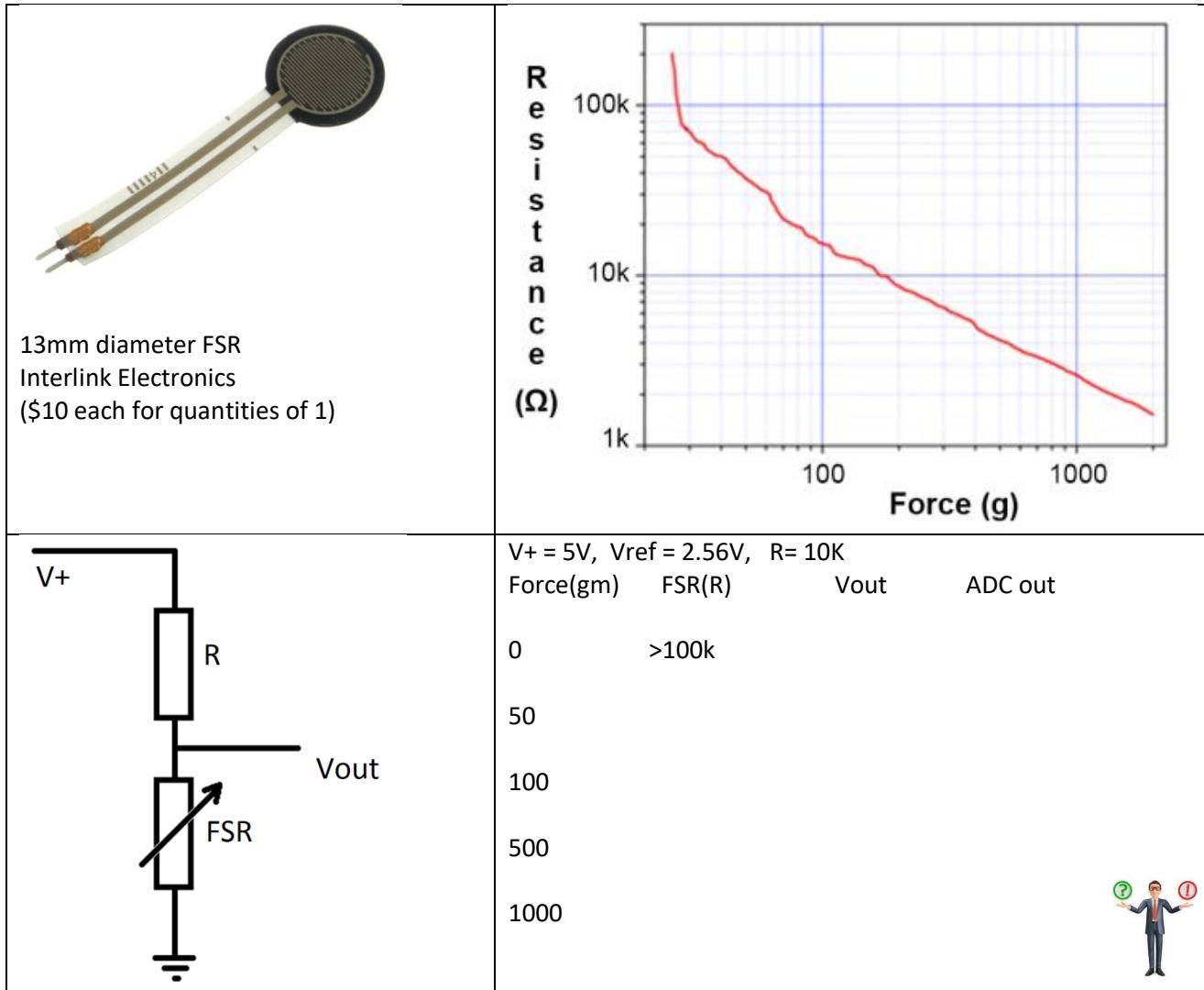
Table 28-6. ADC Auto Trigger Source Selections

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

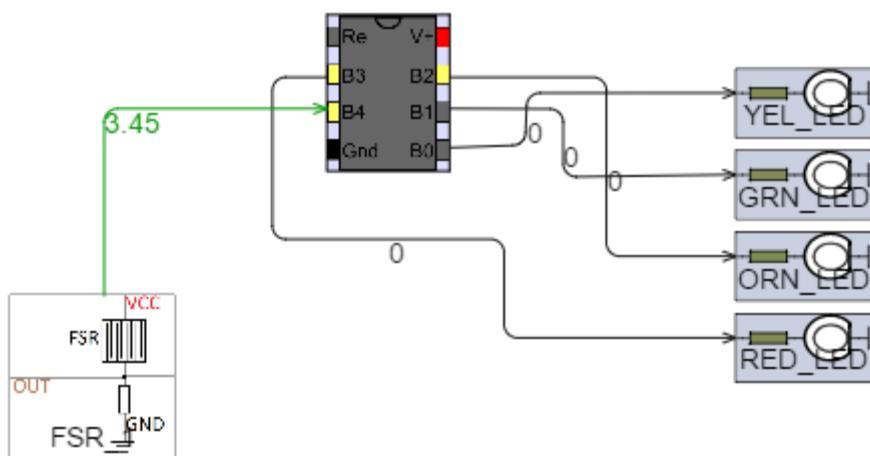
(In register ADCSRB)

Example code to trigger ADC conversion to start on Timer/Counter0 overflow

Sensor - FSR Force Sensitive resistor



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
    //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); //call the function
        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

```

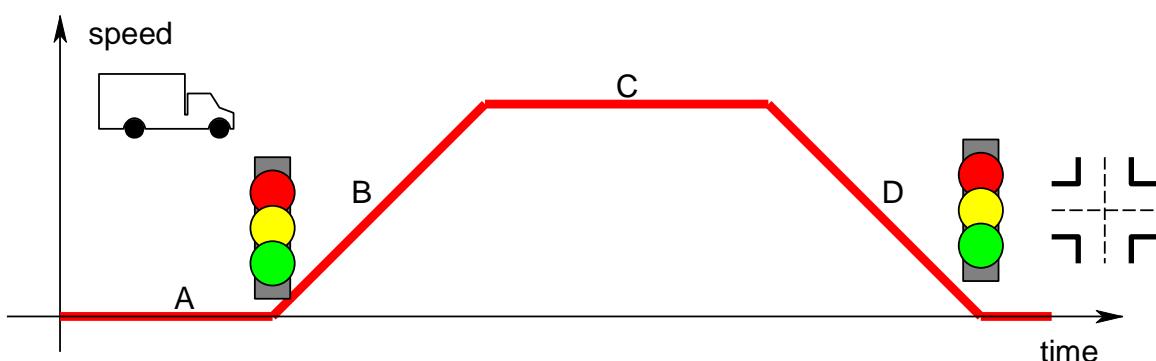
2.14. State machine programming

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. State machines are a very useful tool for non-trivial programs.

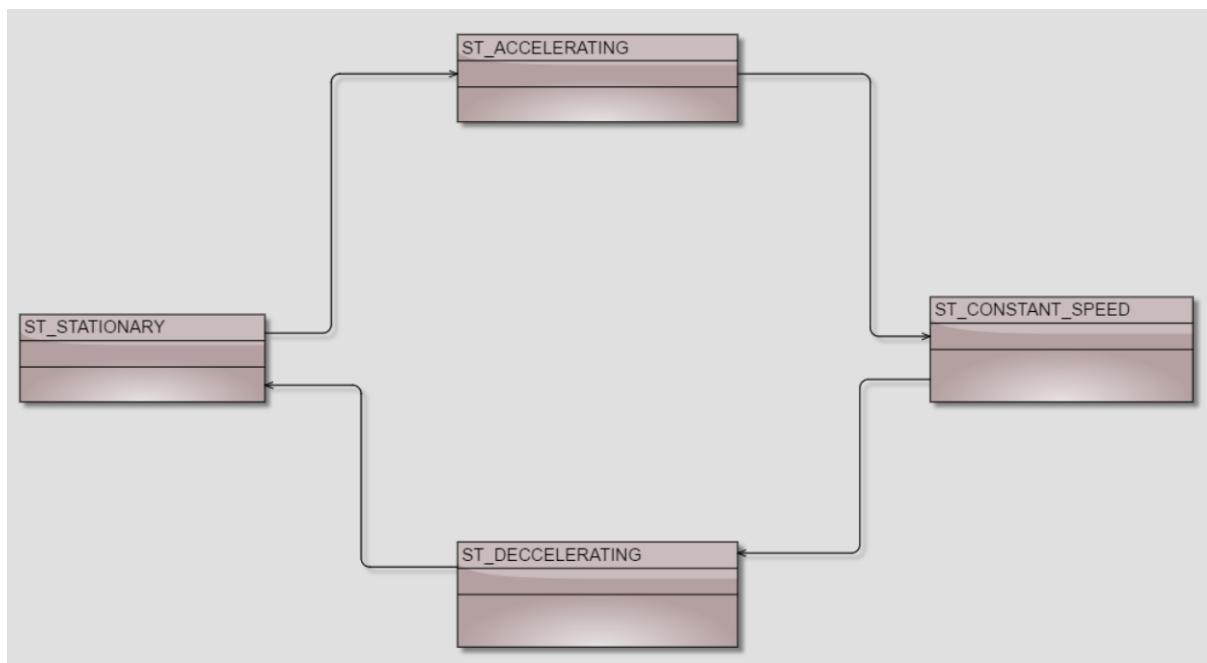
Think of the truck in the graph below as having 4 states.

- 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



A first state diagram for the truck could look like this

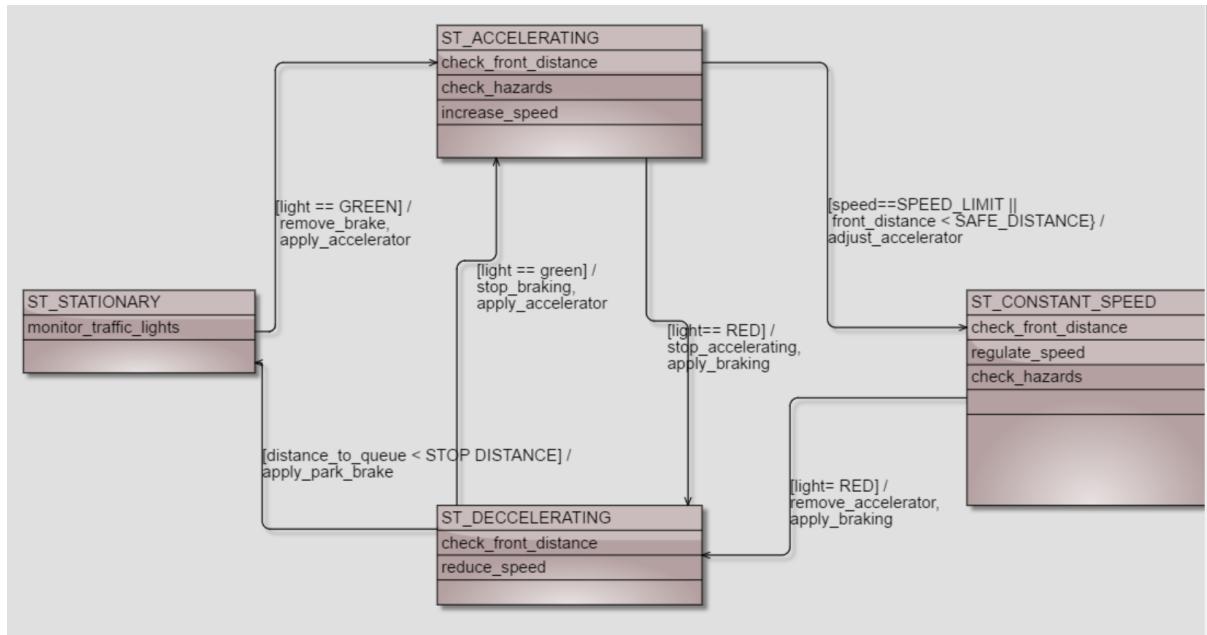


What happens if the truck has not reached constant speed and the driver sees a red light ahead?

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

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

- Pacemaker
- Stopwatch
- Calculator
- Greenhouse
- Traffic lights

2.14.1. Coding state machines in C

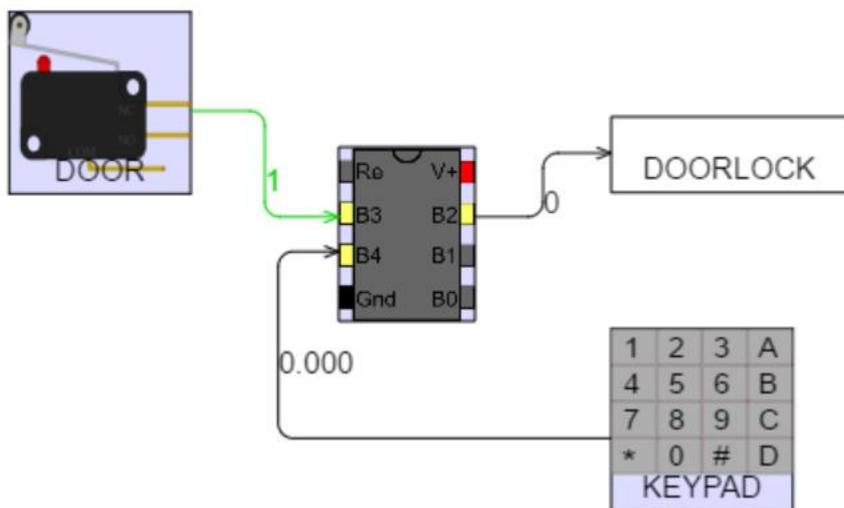
Coding state machines into programming languages is all about clearly identifying and laying out the code to reduce the spaghetti, making it easier to identify sources of bugs and to add new states, transitions and actions.

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

#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){ //... } }
---	--	---

A more comprehensive pattern uses a separate function for each state and calls the appropriate function from a table of function pointers, e.g. <http://www.conman.org/projects/essays/states.html> or <http://johnsantic.com/comp/state.html>

2.14.2. Electronic safe state machine



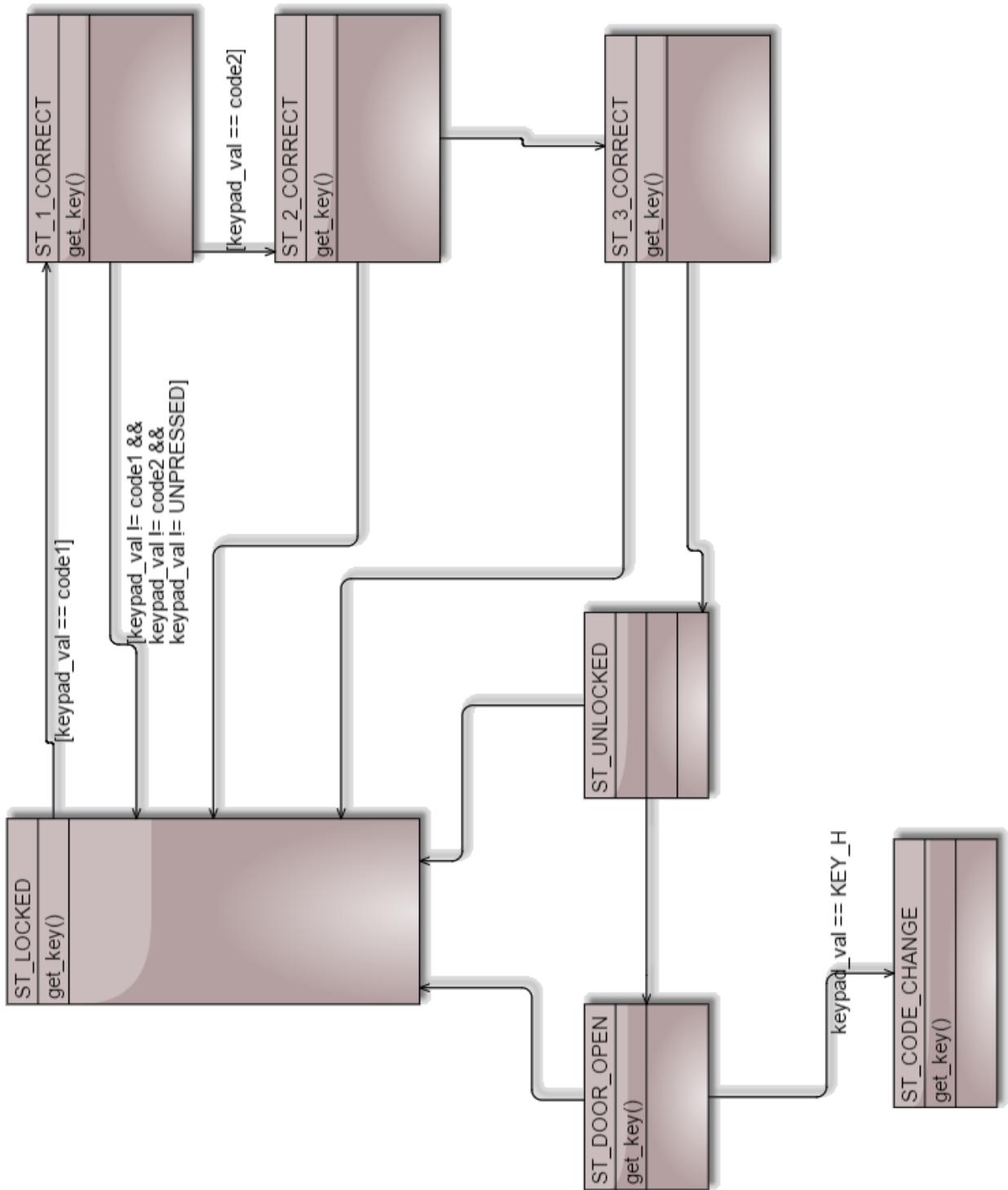
Operating the safe:

- 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 pressed 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();
    }
}

```



2.15. 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 how the characteristics of the devices interrelate with

each other, before we can engineer a solution – actually the equivalent circuit model for each



LED Specifications:

Kingbright, super bright Red BL-BD-BF43R7M 4000mcd

Vf typical: 2.0V at 20mA - this means it will draw 20mA and have approximately 2.0 V across it,

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?

Why does this interrelationship exist?

Could / should we design circuits that use an over-clocked AVR?



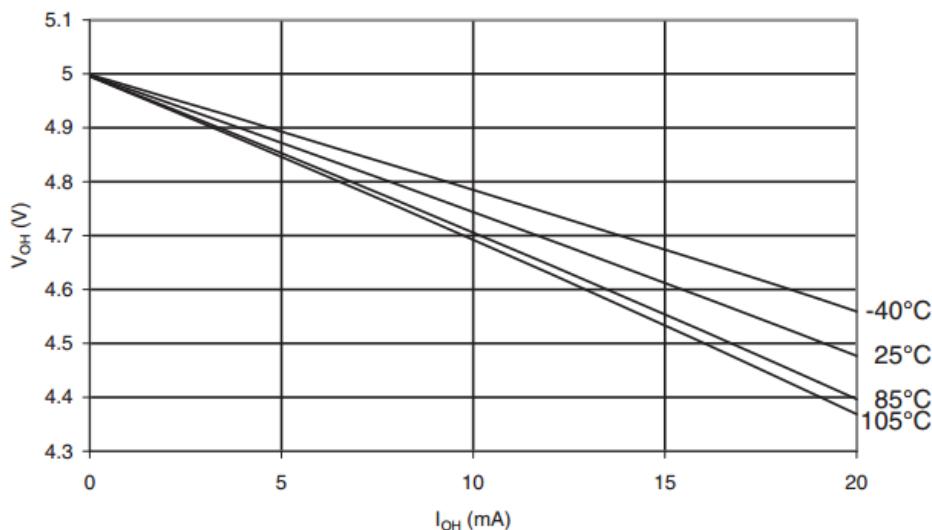
ATmega328P DC characteristics for -40 to +85 degC datasheet section 32.1

Absolute Maximum Ratings

- DC Current per I/O Pin
- DC Current V_{CC} and GND pins

Although the absolute maximum specs are 40mA, the 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

- V_{CC} =
- 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.



2.15.1. Driving devices with higher current / voltage requirements

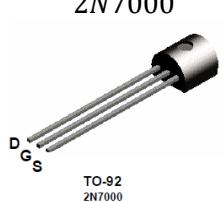
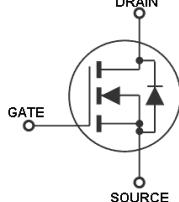
Many output devices require higher power than a microcontroller pin can typically provide.

The AVR can only supply 5V at a maximum current of 40mA. This is too small to drive many output devices. We need an interface device that a microcontroller can switch that can control larger devices. One common device is the logic MOSFET (metal oxide silicon field effect transistor).

2.15.2. The logic MOSFET

There are several different types of transistor –the BJT and the FET are two common ones. The only type considered in this course is the logic MOSFET, a specific type of FET that can be driven by voltages used in 3.3V and 5V logic circuits.

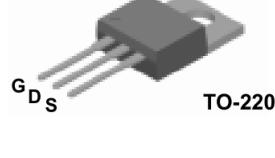
There are many different logic FETs made by very many different manufacturers, one common low power logic FET is the 2N7000.

Physical appearance	Scematic symbol	Characteristics of interest		
 2N7000 TO-92 2N7000		BV_{DSS} / BV_{DS} (max) 60V	$R_{DS(ON)}$ (max) 5.0Ω	$I_{D(ON)}$ (min) 75mA

I_D (continuous)*	I_D (pulsed)	Power Dissipation @ $T_C = 25^\circ C$
200mA	500mA	1W

2N7000 datasheet

One higher power logic FET is the *FQP30N06L*

 FQP30N06L TO-220	Symbol	Parameter	FQP30N06L		
	V_{DSS}	Drain-Source Voltage	60	V	
	I_D	Drain Current - Continuous ($T_C = 25^\circ C$)	32	A	
		- Continuous ($T_C = 100^\circ C$)	22.6	A	
	I_{DM}	Drain Current - Pulsed	128	A	
		Symbol	Parameter	Min	Typ
		$V_{GS(th)}$	Gate Threshold Voltage	1.0	--
		$R_{DS(on)}$	Static Drain-Source On-Resistance	--	0.027
				--	0.035
				0.035	Ω
				0.045	

We connect a Logic FET to a microcontroller like this:

Schematic	Equivalent circuit

The 10k resistor is not necessary for the circuit to work, it is there because when the microcontroller turns on, it takes a finite amount of time for the software to set the pin as an output. During that time the pin is automatically setup to be an input. It could ‘float’ in potential, making the gate of the FET float as well. This means stray electric fields from the circuit around it could turn on the load.

Connecting a $150mW$ load to the circuit above.

What is the current in the load?



What is the voltage V_{DS} ?

What is the power dissipated by the FET?

Replacing the 2N7000 with a high power FET, such as the *FQP30N06L*, and changing to a $150W$ load - what is the current in the load?

What is the voltage V_{DS} ?

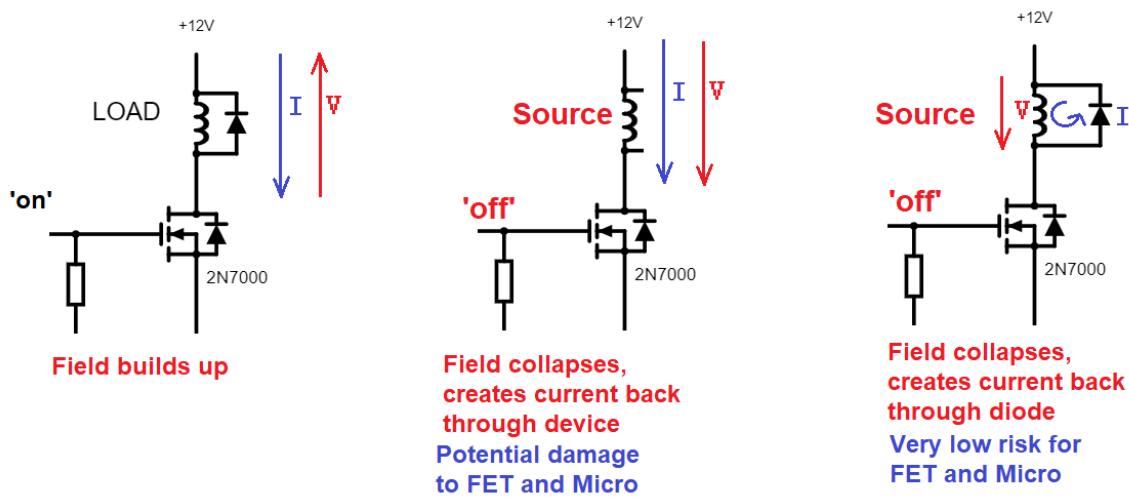
What is the power dissipated by the FET?

2.15.3. Inductive Loads

Faraday's Law states that when the magnetic flux linking a circuit changes, an electromotive force (EMF) is induced in the circuit proportional to the rate of change of the flux linkage (this is the principle behind how generators work).

This however can have catastrophic effects in electronic circuits. If we connect a motor or solenoid (an inductor) to a microcontroller using a FET, when the FET is turned on and there is current in the inductive device and a magnetic field builds up (left diagram below). When the device is turned off the magnetic field collapses (rapidly) and creates an EMF. The effect is that the current tries to be maintained, this turns the inductor from a load to a source, and the voltage across the inductor reverses polarity – we use the term back-EMF to describe this reversed polarity. The inductive device becomes a small ‘battery’ for a short period of time – enough time to create enough current at a high enough voltage to damage the FET and the microcontroller - sometimes spectacularly! Centre diagram below.)

We always connect a ‘flyback’ diode across the load. A diode conducts when the polarity across it is in one direction and does not conduct when the polarity across it is in the other direction, right diagram below)



2.16. Fitness-for-purpose

LO: recognise ‘fitness-for-purpose’ factors that guide uC choice for an ES

Understanding the nature of an ES (the interrelatedness of hardware and software, the unattended automaton, and as reactive and responsive to the environment) allows the engineer a mindset of what will make an ES fit-for-purpose. 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 appropriate device with the required feature set that is cost-effective – and in the process 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 reduced 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 calculations are 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 AVR's 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

2.17. 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
<http://extremeelectronics.co.in/category/avr-tutorials/>

Interrupts
<http://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 hope you find this knowledge useful in your future as engineers.