COMP22111

Processor Microarchitecture

Lecture Notes

Part 1: Introduction

(approx. 2 lectures)

Dr Paul Nutter

Email: p.nutter@manchester.ac.uk
Office: IT119 (Ground floor IT building)

Version 2014

About these notes

These notes form part of the teaching materials for the second year course unit COMP22111: Processor Microarchitecture (formerly known as VLSI Design, and COMP20241).

The module aims to give a view of the role of a digital hardware designer, taking an idea and implementing it as a silicon chip. A processor is a representative example of logic used in today's chips, also giving further insight into how computers actually *work*. Having completed the module you should have developed the confidence to be able to take a concept and realise it in hardware. You should also appreciate the test and verification processes involved so that your chips work efficiently and reliably ... first time, every time!

This module aims to develop the two key aspects of COMP12111, namely hardware design and microprocessors. COMP12111 gave an overview of the hardware development process; COMP22111 builds on these skills to introduce and exercise industrially relevant hardware skills with a design flow from concept to implementation. It used microprocessors as design examples to illustrate and reinforce how machine code, output from a compiler, is interpreted and executed by a computer.

The lectures support the laboratory exercises where you will complete a design for a 16-bit microprocessor, the STUMP, a RISC machine similar in concept to an ARM processor.

Organisation of these notes

These handouts contain copies of the lecture slides as well as "extra" information relevant to the material being taught. All material is examinable.

As well as providing information, I also aim to provide examples in the notes in order for you to gauge your own level of understanding; examples are clearly identified.

Q.

What is your name?

References

I acknowledge the following references that I have used to prepare these notes:

1. COMP22111 Lecture Notes (pre-2011) by Dr Linda Brackenbury, School of Computer Science, The University of Manchester.
2. "ARM System Architecture", S Furber, Addison Wesley, ISBN 0-201-40352-8
3. "Principles of Computer Hardware", A Clementa, 4th Edition, Oxford University Press, ISBN 978-0-19-927313-3.
4. "The Verilog hardware description language", Thomas & Moorby, 5th Edition, Springer, ISBN 978-0-387-84930-0.
5. "Digital VLSI Systems Design", S Ramachandran, Springer, ISBN 978-1-4020-5828-8, (ebook available to download via JRULM)

Assessment

This course unit is assessed by a formal examination and laboratory work. The breakdown of the assessment is

- exam – 55%
- lab – 45%

The examination lasts 2hrs and you must answer three questions out of the four questions provided. The questions are split evenly between the two halves of the course unit (2 from PWN, 2 from JDG).

This course was modified extensively for the 2011-2012 academic year, and is undergoing some updates for the 2013-2014 academic year. Hence, when looking at past papers (all of which are available of the University intranet) take care when looking at papers from earlier years. If the question looks unfamiliar, the chances are the subject matter is no longer covered. No sample answers are given for past papers. However, I will be happy to discuss any questions relating to the material I cover in lectures, providing some attempt to answer them has been made beforehand.

Course Website

Further information, such as copies of these notes, copies of the slides, corrections to the notes, etc, can be found on the course unit website

http://studentnet.cs.manchester.ac.uk/ugt/2014/COMP22111/

If anything is incorrect on the course website, or there is anything you would like to see there (apart from your exam questions ☺) then let me know.

Additional material can be found on the School of Computer Science Engineering Wiki:

http://wiki.cs.manchester.ac.uk/engineering/

If you're stuck ...

Then please come and see me ... I'll be happy to answer any questions relating to the module (well, my course material certainly). Ask questions during the lecture or catch me at the end of lectures, or in the lab, or come and see me in my office. If possible, please email me to check my availability, that way I'll make sure I set aside enough time to answer your questions. If you knock on my door out of the blue, then I may not have the time to see you!

Further information, such as copies of these workbooks, copies of the slides, extra notes, corrections to the notes, etc, can be found on the course unit Blackboard site.

Finally, if you notice any mistakes, then please let me know ... I'm not perfect and my notes definitely aren't ... ☺.

MANCHESTER 1824
The University of Manchester

COMP22111

Processor Microarchitecture

Dr Paul W Nutter &
Dr Jim Garside

School of Computer Science

Notes:

Course Aim

The module aims to give a view of the role of a digital hardware designer, taking an idea and implementing it as a silicon chip. A processor is a representative example of logic used in today's chips, also giving further insight into how computers actually *work*.

Learning Outcomes

The learning outcomes of the course unit are to:

- have a knowledge and understanding of the process of designing complex (VLSI) chips
- have an understanding of the different design stages and representations of complex circuits
- have an understanding of the major architectural and performance factors to be considered in the design of a large integrated circuit
- have an understanding and appreciation of the problems arising out of the rapid change of technology and increase in design complexity
- be able to design a 16-bit RISC processor at the upper levels of the design process, and have experience of the tools to test and debug the design
- have a knowledge and understanding of industry-standard hardware description languages.

Syllabus (PWN material)

The first half of this course unit covers the following:

Introduction (approx. 2 lectures)
Introduction, overview of top down design hierarchy, design choices, recap of sequential systems design.

Introduction to RISC processors & the Stump Processor (approx. 3 lectures)
RISC versus CISC, load/store architecture, non-pipelined behaviour, instruction set. MU0 as a design example. ISA of the Stump processor.

Verilog Hardware Description Language (approx. 4 lectures)
Principles, structure, features and syntax, scheduling, test bench.

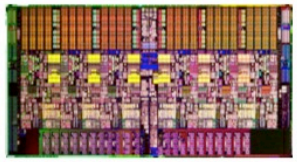
Implementation of the Stump (approx. 2 lectures)
Block partitioning, datapath occupancy. RTL design. Implementing processors in Verilog (using MU0 as an example).

Introduction

MANCHESTER 1824
The University of Manchester

How do we build complex digital systems?

- what steps do we follow when designing complex systems
- use concepts such as abstraction & hierarchy
- top-down approach to design



Intel i7 six core processor

What will we be doing?

- use a RISC processor as a design example
- understand the design process and follow it in a practical application
- learn more about the use of Hardware Description Languages: Verilog
- learn about RTL design and its application
- the use of test benches to confirm the validity of a design

COMP22111: Processor Microarchitecture Part 1

Notes:

Staff

The course unit is taught by:

Dr Paul Nutter
Room IT 113 (Ground Floor IT Building)
Telephone: (0161) 275 5709 (Internal: 55709)
Email: p.nutter@manchester.ac.uk

Dr Jim Garside
Room IT 211 (2nd floor IT building – south side of the building)
Telephone: (0161) 275 6143 (Internal: 56143)
Email: jgarside@cs.man.ac.uk

If you need to contact a member of staff regarding material covered in this course unit, or the laboratory exercises, then in the first instance use email to arrange a suitable date/time. Do not rely on us being available in our offices when you need to speak to us.

You can catch us at the end of lectures, or during the laboratory, but this doesn't leave much time to discuss any problems in any great detail.

Website

Information about the course unit, course materials etc. can be found at the course unit website:

http://studentnet.cs.manchester.ac.uk/ugt/2014/COMP22111/

If you find any errors in these notes, or on the course website, please let me know!

Textbooks

There are many textbooks that cover aspects of the material described here, as well as number of web resources.

Some useful sources include:

1. COMP22111 Lecture Notes (pre-2011) by Dr Linda Brackenbury, School of Computer Science, The University of Manchester.
2. "ARM System Architecture", S Furber, Addison Wesley, ISBN 0-201-40352-8
3. "Principles of Computer Hardware", A Clementa, 4th Edition, Oxford University Press, ISBN 978-0-19-927313-3.
4. "Digital VLSI Systems Design", S Ramachandran, Springer, ISBN 978-1-4020-5828-8, (ebook available to download via JRULM)
5. http://www.asic-world.com/verilog/intro.html provides a good introductory overview to Verilog.
6. "The Verilog Hardware Description Language", Thomas & Moorby, 5th Edition, Springer, ISBN 978-0-387-84930-0
7. "Digital Design: An Enbedded Systems Approach Using Verilog", P J Ashenden, Morgan Kaufmann, Elsevier, ISBN 978-0-12-369527-7

I do not recommended you buy any of these books, they are all available in the School library or the main University library. All the information you will need will be given in these handouts.

What does it involve?

MANCHESTER 1824
The University of Manchester

Lectures

- 22 in total, 3 per week in weeks 1 & 2, 2 per week for weeks 3 to 11
- Monday 12pm and Thursday 2pm, Tuesday 10am (weeks 1-2 only)
- split 50:50 between Jim and Paul

Laboratories

- 9 scheduled labs, each 2 hours, Tues 10-12pm (F), or Thurs 11-1pm (G).
- start in week 3, Tootill 1

Assessment

- through completion of laboratory exercises
- examination in January – 2 hours
- examination 55%, laboratory 45%

In the lab you will be using the Cadence tools - these are commercially sensitive for which must agree to the end-user agreement.

Notes:

What will you be doing?

In the first half of the course the labs and lectures are closely linked. It is important that you attend ALL the lectures and ALL the labs! The lectures will cover the information you will need to successfully complete the lab.

Timetable of Activities

Lectures and laboratories are timetable as given below:

Semester Week	Labs	Lectures	
		Monday 12pm	Thursday 2pm
1	No Lab	Lecture 1 (22/9) PWN	Lecture 2 (25/9) PWN
2	No Lab	Lecture 3 (29/9) PWN	Lecture 4 (2/10) PWN
3	Lab 1 (w/c 6/10)	Lecture 5 (6/10) PWN	Lecture 6 1(9/10) PWN
4	Lab 2 (w/c 13/10)	Lecture 7 (13/10) PWN	Lecture 8 (16/10) PWN
5	Lab 3 (w/c 20/10)	Lecture 9 (20/10) PWN	Lecture 10 (23/10) PWN
6	Reading Week – No lectures/labs		
7	Lab 4 (w/c 3/11)	Lecture 11 (3/11) PWN	Lecture 12 (6/11) JDG
8	Lab 5 (w/c 10/11)	Lecture 13 (10/11) JDG	Lecture 14 (13/11) JDG
9	Lab 6 (w/c 17/11)	Lecture 15 (17/11) JDG	Lecture 16 (20/11) JDG
10	Lab 7 (w/c 24/11)	Lecture 17 (24/11) JDG	Lecture 18 (27/11) JDG
11	Lab 8 (w/c 1/12)	Lecture 19 (1/12) JDG	Lecture 20 (4/12) JDG
12	Lab 9 (w/c 8/12)	Lecture 21 (8/12) JDG	Lecture 22 (11/12) JDG

PWN – Paul Nutter, JDG – Jim Garside

Due to increased student numbers we will be running 2 lab sessions per week. Tuesday 10am-12pm for group F, and Thursday 11am-1pm for group G. Please make sure you **ONLY ATTEND your scheduled lab session**.

You will have lots to do in the laboratories, so it is important that you prepare thoroughly beforehand. Make sure you have read the lab manual COMPLETELY before the first scheduled lab; this will help you make a good start on the exercises. Remember, the more you hand in the more marks you will get!

The lab has strict deadlines associated with it (particularly the optional exercises). Extensions are available. Each lab exercises involve the design of components of the Stump processor, which come together at the end to implement a working processor. Consequently, it is important that you complete each exercise.

Assessment

The course unit is assessed by lab work (45%) and examination (55% each). The examination consists of 4 questions, of which you must answer 3.

The questions are split equally across the two halves of the course unit – 2 questions will be set by PWN, 2 questions will be set by JDG.

Introduction to the Lab

MANCHESTER 1824
The University of Manchester

- To understand the design of a complex systems, you will design a (complex) processor in the laboratory – from concept to implementation.
- What will you be designing?

  … a 16-bit ARM-based RISC processor - the **Stump**
- You will be introduced to the Stump instruction set
- Exercises will build on the design of the Stump, resulting in the implementation of a fully working processor.
- There is no schematic entry in this lab. All designs will be produced and implemented form Verilog descriptions.

Notes:

Laboratory

Lab timetable:

Week	Ex.	Subject	Deadline lab w/c
1		No lab	
2		No lab	
3	1	Familiarisation with the Stump ISA	6/10
4	2	The Stump ALU	
5	2	The Stump ALU	20/10
	2x	The Stump Shifter (Optional)	20/10
6	-	Reading Week	
7	3	The Stump Processor Datapath	
8	3	The Stump Processor Datapath	10/11
9	4	The Stump FSM	17/11
10	5	The Stump Control & Testing	
11	5	The Stump Control & Testing	1/12
12	6	Stump Synthesis	8/12
	6x	Stump Expansion	8/12
		Total	

We have teaching assistants (demonstrators) present in the lab to help you with any problems you are experiencing; these are typically PhD students in the school. Teaching assistants are not present to solve problems for you, but to guide you in the right direction and offer advice where necessary.

Cadence

In the lab you will be using the Cadence Design Framework environment that you have used in the first year. The Cadence tools are industry standard tools that are very expensive and IP sensitive. The University gains access to these tools through an educational agreement. Details regarding the end user agreement for using this software can be found at

http://www.europractice.stfc.ac.uk/eua/univs/cadence.pdf

Of particular importance are your obligations to keep the Cadence tools confidential and that the University's Cadence tools may NOT be used for any commercial purposes. You will be asked to agree to the terms of the end user agreement when you run the tools for the first time.

Introduction to the Lab

MANCHESTER 1824
The University of Manchester

- Work must be submitted by the deadline associated with each exercise.
- Submission is using the script "22111submit", much the same as for COMP12111. The mechanism is the same.
- When work is submitted ARCADE records the submission date/time. Late penalties will be applied to work that is submitted after the deadline.
- As in COMP12111, we are introducing demonstrations in COMP22111, so you need to print out labprint sheets via the 22111submit script.
- Laboratory work is individually assessed. We take academic malpractice seriously. Anyone found copying/sharing work will be penalised according to University regulations on academic malpractice.

Notes:

Lab Deadlines

The deadline for submission of lab exercises is the end of the lab with the deadline associated with it. If you feel you cannot make a deadline then extensions are available until the START of the next scheduled lab. However, extensions are not automatic, you must ask for one and they will not be granted unless you have made sufficient progress with the exercise.

Submission Process

As in COMP12111 we have introduced a submission process for COMP22111 using a custom submission script. To submit your work you must run the script

22111submit

from the command prompt. You will be then asked to select the exercise you would like to submit, i.e. ex1, ex2, ex3x etc. Once you have selected the exercise, you are then presented four options:

a) to type "submit" to submit your work first the first time
b) to type "submit-again" to submit your work again (be careful as work submitted again after the deadline will be considered late)
c) to type "submit-diff" to check your current design against that which you have previously submitted
d) to type "labprint" to produce the labprint sheet for demonstrating your work in the lab.

Academic Malpractice

Laboratory work in COMP22111 is individually assessed. Academic malpractice, i.e. plagiarism, sharing, or copying work, will not be accepted. Any student suspected in engaging in academic malpractice will be penalised according to University regulations. Further information can be found here:

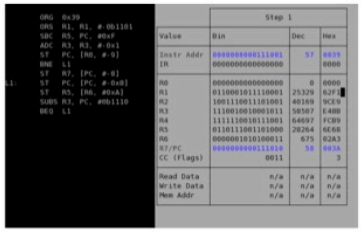
- University guidance for students on Plagiarism and Academic Malpractice:
  http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=2870
- School specific information:
  http://studentnet.cs.manchester.ac.uk/assessment/plagiarism.php

Lab exercises …

MANCHESTER 1824
The University of Manchester

Ex 1: Familiarisation with the Stump ISA

- you are given some randomly generated Stump code
- use the 'Treadmill' tool to record the status of registers etc. as each instruction is executed
- submit the file containing your answers for marking



Objectives

- to gain familiarity of the Stump ISA
- to understand the Stump instructions and their operation

Notes:

Lab Exercises

The lab exercises require you to design, implement, simulate and synthesize a fully working (simple) 16-bit RISC processor – the Stump.

Preparation

Preparation outside the timetabled laboratory classes is **necessary** and **expected**. Students who wish to make good progress in the laboratory time when help is available should not only read the relevant material for each week **before** coming to the lab but should also do further work on stages of the design outside this. Remember, you are expected to spend the **same** amount of time on preparation as you spend in the scheduled lab time. In addition, the lab work and lectures are closely integrated, so important and useful information about lab exercises is given in lectures; so attendance at lectures is closely linked to good progress in the lab!

Marks

This course has more labs and less lectures than other courses and the overall lab and exam mark is weighted accordingly. Students are expected to work individually and independently. Hence work resulting from collaborative efforts will result in the mark awarded for the work being equally split amongst the contributors. As the COMP22111 lab forms a significant contribution to the overall course mark, it is in your interests to invest the time in obtaining a good lab mark!
The laboratory for COMP22111 has been completely redesigned for 2012/13 with the development of new tools for some of the exercises; a vacation student put these together.

Exercise 1

Exercise 1 will involve the investigation of a short code listing for the Stump processor. Each student will have a unique code listing that contains **most** of the Stump instructions. The task is to go through each line of code and identify the changes to the system registers, flags, memory etc. after the execution of each instruction. You will do this using a linux application written specifically for this lab called “Treadmill”.

Lab exercises …

MANCHESTER 1824
The University of Manchester

Ex 2: The Stump ALU

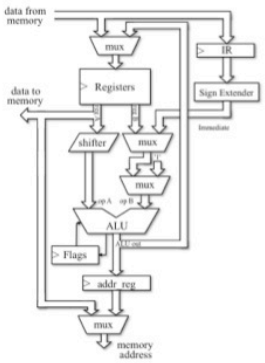
- start to create your own working processor
- create your own synthesizable Verilog module
- simulate using the test code provided

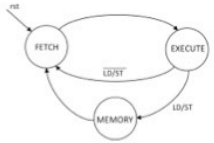
Ex 3: The Stump Processor Datapath

- create a Verilog structural implementation of the Stump RTL datapath

Ex 4: The Stump fsm

- first part of implementing the control for the Stump
- test using your own test stimulus





Notes:

Exercise 2

Exercise 2 will involve the implementation of a Verilog module for the Stump alu. The module is purely combinatorial in nature and is required to implement the arithmetic functions ADD and SUBtract, as well as logical functions AND and OR. It also provides the flags:

N – the negative flag – set if the alu result is negative if the number is interpreted as a 2's complement number, 0 otherwise.

Z – the zero flag – set if the result is zero.

V – the overflow flag – indicates if the result from the addition/subtraction is 'wrong' if interpreted as a 2's complement number.

C – the carry flag – set if there is a carry out from the most significant bit of the result (bit 15), else is 0.

Exercise 3

In exercise 3 you will implement the Stump datapath in Verilog. In COMP12111 you were given a schematic implementation of the MU0 datapath, which you completed. However, it is also possible to describe the *structure* of a datapath using components defined as modules and identifying the connections (nets) between these components. In this exercise, you will be using *structural* Verilog to describe the Stump datapath, rather than using a schematic as you have seen previously.

Exercise 4

A processor contains a finite state machine to cycle through the "states" the processor can reside in. In the case of the Stump it has three states:

fetch – read the word pointed to by the address held in the PC into the instruction register, and increment the PC by 1.

execute – decode the instruction held in the instruction register, perform the required operation and writeback to the result.

memory – memory access for load and store operations.

The fsm will cycle through these states at every positive clock edge, only entering the memory state only for a load/store operation. The output from the fsm is the current state code.

The Stump fsm is very simple and resides in the Stump control block. You must implement and test its operation. Remember, an fsm is a sequential systm so your Verilog code should include non-blocking assignments.

Lab exercises …

MANCHESTER 1824
The University of Manchester

Ex 5: The Stump Control & Testing

- implement behavioural Verilog to control the operation of the Stump datapath
- test the complete Stump processor using the test scripts provided

Ex 6: Stump Synthesis

- synthesize your Stump processor design & download
- demonstrate code running on the processor

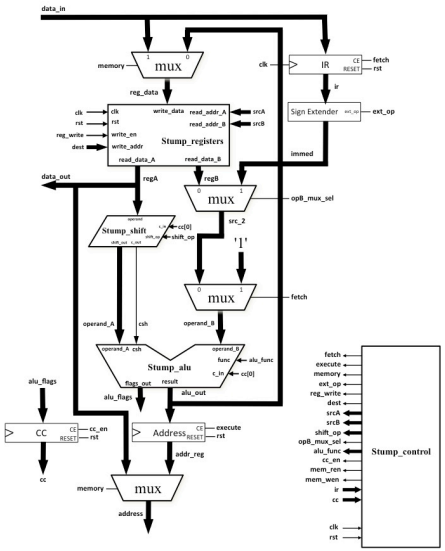
Optional Exercises

- count a small amount of marks
- Exercise 2x – The Stump shifter – Verilog module
- Exercise 6x - Stump Expansion – enhance the instruction set, speed up the implementation …
- Meant to challenge students who are ahead. Fixed deadlines with no extensions.

Notes:

Exercise 5

Following on from Exercise 4 you will write the Verilog code to implement the control block for the Stump processor. The control block controls the signals to the datapath depending on the current state of the fsm, and the instruction being executed. The full RTL view of the Stump datapath including control signals is shown opposite. You are required to produce the Verilog code to assert the control signals when required. Testing of you complete Stump is performed using four test files that are providing for you. These test files are working Stump code that you must run by simulating your complete Stump processor, and view the output from these in order to determine whether your processor is working as required. Each test will test certain aspects of the Stump design, such as the operation of the alu and flags, branches etc.



Exercise 6

Synthesize and implement your working Stump processor on the FPGA boards available in the lab. If you have time, write some of your own simple code to run on the processor.

Optional Exercises

There are two optional exercises:

- Exercise 2x – The Stump Shifter,
- Exercise 6x – Stump Expansion.

There are few marks associated with both these exercises. They are there to challenge students who are ahead in the lab. There are fixed deadlines for these exercises with no extensions. The shifter provided already contains a working shifter design, which you must comment out when creating your own. The design provided is inefficient and produced in gate-level structural Verilog.

In the case of Exercise 2x you will need to produce a Verilog module for the shifter unit in the Stump datapath.

In the case of Exercise 6x you have the flexibility to take the design of the Stump further, although we only suggest you attempt this lab if you are at least a WEEK AHEAD of schedule.

Part 1 – Introduction

MANCHESTER 1824
The University of Manchester

In this course we are going to look at how we design complex systems using the processor as an example complex system.

A processor is a sequential system: it contains a datapath – to process data, and control (fsm) – to issue commands to the datapath to control its behaviour.

Processors are an example of a VLSI system – very large scale integration – lots of transistors in a single device – Moore's law.

Year	No. of Transistors	Clock MHz	Geometry	Area mm²	
1971	2300	0.1	10µm	20	
1978	29000	5	3µm	33	
1985	0.275M	16	1.5µm	52	
1993	3.1M	50	0.8µm	76	Pentium
1999	22M	500	0.25µm	100	
2003	135M	1700	130nm	131	
2007	500M	2600	65nm	300	
2010	1170M	3300	32nm	500	i7 6-core
2030?	50000M?	5000?	<5nm?	>1000?	

COMP22111: Processor Microarchitecture Part 1

Notes:

What is this course about?

In COMP22111 we will be looking at the process of designing complex digital systems – VLSI - Very Large System Integration, where systems have a large number of transistors (>1 million) on a single, silicon integrated circuit, i.e. 32-bit processors and upwards. Development of such complex systems from start (specification) to finish (production) requires a large range of skills: applications, choice of IC type and technology, architectures, block design, logic design, testing, timing and control, circuits, layout, placement, fabrication, packaging, testing and documentation! There are many complex stages (and some iteration) required in the design of such systems, all of which must be managed efficiently. How do you do that?

Evolution of Silicon Technology

The complexity of systems has grown considerably over the last 50+ years as the demands of today's applications has required the integration of more and more complex functions (data processing, telecommunications etc) in ever smaller devices. The relative cheapness of today's microprocessors has fuelled the growth in their use in a variety of everyday devices, ranging from electric toasters to cars … the list of applications is endless. In fact, modern mobile phones may contain 5 or more processors to enable calls to be made, to allow complex finger gestures on the touchscreen, to allow wireless communication with headsets via Bluetooth etc.

Pre 1960s	discrete components only, single transistors
Early 1960s	logic gates introduced, few logic gates, the flip-flop (small scale integration – SSI)
Late 1960s	more complex devices introduced, such as counters, multiplexers, adders (medium scale integration – MSI)
70s	First 8-bit microprocessors introduced, ROM, RAM (large scale integration – LSI)
Early 80s	16-bit and 32-bit processors introduced, on-chip cache, architecture advances (very large scale integration – VLSI)
Late 80s	more on-chip functions, more memory
90s	embedded processors, integration of more complex functions, e.g. floating point number representation, more memory, pipelining, concurrency
00s	systems-on-chip (SoC) introduced –functionality implemented on a single device

The level of integration (measured by the number of logic gates or transistors) has been accelerating to the point where today's processors have billions of transistors on a single silicon integrated circuit. Since the introduction of the first Intel Pentium processor, to today's high end desktop processors, such as the Intel i7 six-core processor, the number of transistors on chip has increased over 350x, along with an increase in clock speed of over 60x, and an increase in the chip area of over 6x. How has this level of integration been achieved? Through the miniaturisation of components, where advances in fabrication technologies have allowed feature sizes of around 32nm (over 3000th the width of a human hair) - a 25x reduction in the size of the device. How long will this continue? It is difficult to say, but the scaling of features sizes will soon reach a physical limit. What happens then is anyone's guess!

The monolithic integration of a large number of functions on a single chip provides:

- decreasing area/volume
- higher reliability
- higher speed
- significant cost savings as everything is on one single ic.

The problem with such complex designs is the effort required to design them. The design complexity grows exponentially with the number of transistors to be integrated. How do we manage this complexity? As this growth in complexity translates into an increase in the design cycle time.

Managing Complexity

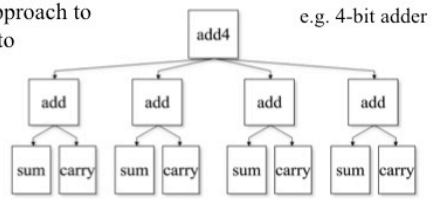
The design complexity of logic devices increases exponentially with the number of transistors – this translates to an increase in design time and cost!

How do we manage complex designs?

- use suitable design methodologies
- use abstraction & hierarchy
- use CAD tools

Hierarchy - divide and conquer approach to design … break a design down into simpler components … reuse!

e.g. 4-bit adder



COMP22111: Processor Microarchitecture Part 1

Notes:

Designing Complex Systems

The real problem in chip design is what to put in the very large space available and how to manage the complexity of the design.

We can manage complex designs using sensible design methodologies, such as top-down, using concepts such as abstraction and hierarchy, and by using powerful computer aided design (CAD) tools. In the lab you will be using Cadence, an industry leading CAD tool, to design a 16-bit RISC processor.

How do you start to design a complex system?

- the design process involves taking an idea and implementing it in some manner
- the design process is complex and can be a bit of an art
- design involves the continual refinement of the representation of something
- we can think of the design process as a top-down refinement of a specification

What places constraints on the design?

- time to design
- performance (speed)
- area requirements
- power requirements
- technology
- …

Hierarchy

Hierarchy is a "divide and conquer" approach to design. This is a common approach that is used in hardware as well as software design. For example, in the case of software design, instead of writing a complex programme in one long code listing, you would break the problem into procedures, or functions, and separate files that you code independently, which you include and reuse. You can follow a similar approach in hardware design where you can identify similar functional blocks and implement them with a view to using many instances of the block in the final design. In addition, we can use hierarchy to simplify a more complex design by breaking it down into more simpler, repeated components. A good example is a 4-bit adder, as shown below. Rather than implementing the 4-bit adder directly, we can recognise that it is constructed from four 1-bit adders, and that each 1-bit adder is composed of a sum and carry logic block. The design process then simply involves designing the sum and carry blocks and use these to implement a 1-bit adder, which is then replicated in the implementation of a 4-bit adder.

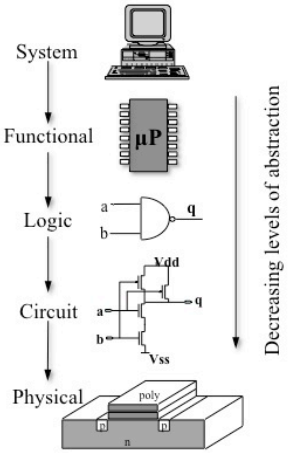
Managing Complexity

MANCHESTER 1824
The University of Manchester

Top down approach to design encapsulates abstraction

Abstraction – helps the designer by hiding the complexity

We can use a full adder block in a design without knowing how it has been implemented



Notes:

Top-down Approach to Design and Abstraction

Abstraction is a very important aspect of any design process and it allows the designer to hide the complexity in a design in order to simplify the design process. It frees the designer from having to remember, or even understand, the low-level details of a component. For example, the designer knows that an adder will add two numbers together but does not need to know how the adder has been implemented internally.

For example, a logic circuit should have the same logical behaviour whatever the technology it is implemented in, so detail of the implementation can be extracted out during the design process in order to ensure the functional design is correct. However, at later stages of the design the technology used to implement the circuit becomes important as it will affect the performance of the circuit, but it should have no effect on the initial design.

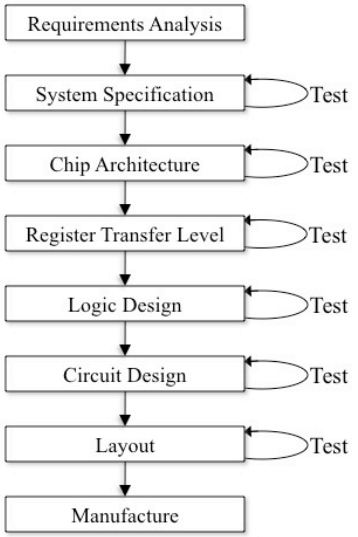
Abstraction allows us to treat our design as a black box for which we know its function, interface etc, but we don't necessarily care about its implementation.

The process of abstraction can be viewed in the top-down approach to design, which encapsulates the concept of abstraction very nicely. The design process is then a process of refining an idea through many levels of detail. The figure illustrates the top-down approach to design where the level of abstraction decreases as we progress down through the design stages, i.e. the amount of "detail" in the design increases as we progress through to arrive at the final physical implementation. Some iteration is often required to ensure the final design meets the original specification.

You can view abstraction being used at the various levels of the design hierarchy.

VLSI design flow

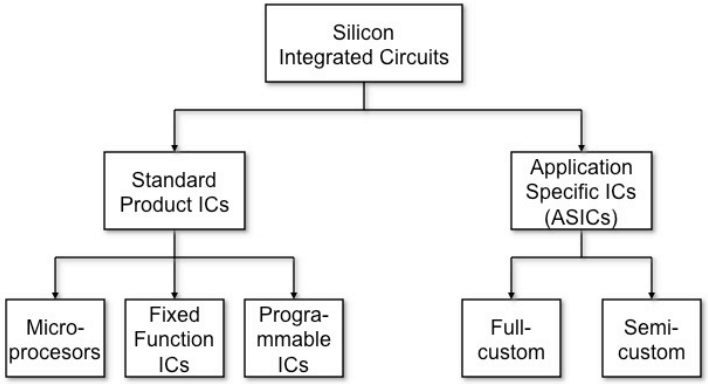
The VLSI design flow is an extension of the top-down approach to design, however, here we have introduced further detail to illustrate how the more important aspects of the design are incorporated into the design flow at the various levels of abstraction. The design process is evolutionary in nature and starts with a set of requirements to which the design is developed and tested against. Verification of the design at EACH stage is extremely important, even at the system specification stage, as failure to implement a valid design that satisfies the original system requirements (even at the earlier stages of the design process) can prove costly in time and expense to re-design at a later stage – this will increase the time to market which will have a further impact on cost! Although the diagram shows the design developing in a linear, stage-by-stage fashion, there can, in fact, be many iterations back and forth, particularly between steps. The checking for correctness at every level is very important and all levels should be checked against the SAME test program for correct functionality.



Design Choices

MANCHESTER 1824
The University of Manchester

You need to decide how the design is to be implemented …



Notes:

Design Choices

As a designer you have a number of choices for the actual implementation of a design. Although implementation doesn't become an issue until later in the design flow, it is important to make you choice early in the design process as it can affect the design process as a whole.

Most designs today are implemented in silicon, however, there are a number of choices as to the implementation we choose, such as:

Full Custom

- design your own cells of gates/flip flops

Advantages

- total flexibility
- fully speed, area and power optimised
- easily protect intellectual property rights (IPR)

Disadvantages

- long design time
- expensive to rework
- need high volume product

Semi Custom (or Standard Cell)

- use predefined and characterised standard cells of functions, cells should butt together for power and ground, so just need to add signal routing (low level)

Advantages

- reduced design time
- less chance of error
- can partially optimise the design
- easily protect IPR

Disadvantages

- limited flexibility
- expensive to rework
- need high volume product

Programmable Logic Devices (PLD)

- PLD is a general name for a digital integrated circuit capable of being programmed to provide a variety of different functions. These devices use an array of logic cells which each provide the same universal logic function. Signal routing is through a switchbox approach with the routing pattern held in (re-programmable) RAM. Field programmable gate arrays (FPGAs) are the current state-of-the-art programmable device, and are what you will be using in the lab to implement your designs.

Advantages

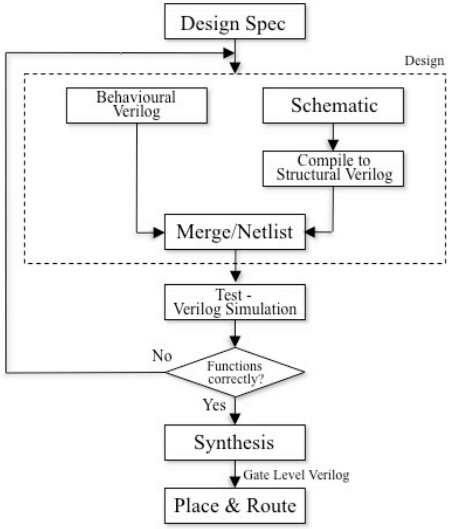
- excellent for prototyping (even complex) designs (such as processors)
- fast development time
- easy to change design (in terms of design effort and cost)
- low design costs
- reasonable speed
- decreased time to product

Disadvantages

- limited flexibility
- low volume product
- less protected IPR

Design Process

In our labs we use Xilinx field programmable gate arrays – the design is determined by this device



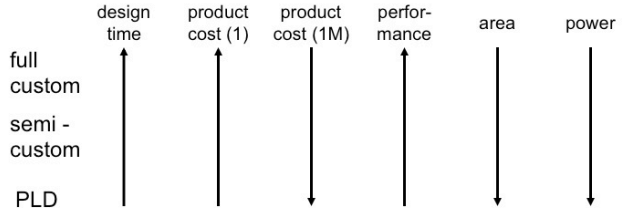
Notes:

Design Approaches

The constraints in any design include:

- time to design
- cost (which is related to design time and number of units produced)
- performance (speed)
- area requirements
- power requirements
- technology

The constraints are affected by the technology you choose:



Comparing the design approaches it is clear that the approach depends on your constraints.

If you want a device that is cheap, quick to produce and you aren't bothered about the performance, then a programmable device would be the best option. If you want something low power, low cost and fast, then you may be struggling as the three are mutually exclusive!

In the lab we have experimental boards with Xilinx FPGAs on them. These are programmable devices that offer a rapid route for prototyping designs, as well for use in production. The CAD tools allow designs to be easily downloaded to the FPGA for testing, and can be easily updated if they don't conform to the required specification. How do we go from a paper design (be it schematic or a functional Verilog description) to implementation on an FPGA? We will have a look soon. The design approach differs considerably depending on the design. In the case of the examples shown, the processor has an irregular structure and the design approach is often full custom in order to optimise area, speed and power. The FPGA, on the other hand, is a regular structure that contains repeated cells of the same functionality, hence a semi-custom approach design is often used where the design effort is spend on optimising the performance of a single block.

Nowadays, the approach often taken is standard cell design (semi-custom) combined with external IP blocks. Full custom would only be used if there were time/area/power critical sections or very high product volume, such as the processor above. An FPGA is often used to prototype a design in order to test functionality before it goes in to mass production.

The trend is now moving away from the gate level design of a system (unless off the shelf), and behaviour is described in a Hardware Description Language (HDL) and Computer Aided Design (CAD) tools are used to translate (synthesize) the design to logic in the targeted family. In COMP22111 we will be using the standard cell approach.

Sequential Systems … a recap

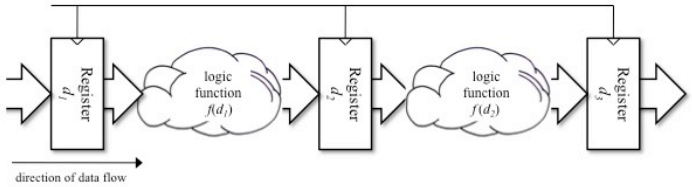
The majority of digital systems are ***sequential*** in nature

… events are sequenced with respect to an external clock.

Why?

In a sequential system we have registers to store data and combinatorial logic to performs operations on the stored data.

Can be viewed as a pipeline of registers and combinatorial logic blocks – ***register transfer level*** (RTL) – part of the design hierarchy



Notes:

Sequential Systems

Combinatorial logic is capable of processing an arbitrary number of inputs to produce an arbitrary number of outputs; it can also evaluate any function. However, this assumes that specialised, complex, reliable, (expensive) logic can be defined for each function required. In practice this is infeasible.

In order to reduce a complex job to a manageable size it can be broken down into smaller jobs (hierarchy). One way of approaching this is to repeat simple hardware, such as in an n-bit adder. A different approach is to use the same hardware in repeated steps to process different parts of the problem; this is the approach that a computer takes, solving the problem one instruction at a time. This calls for **sequential** logic.

To solve a problem in a series of time steps results from earlier steps much be held and, sometimes, used as inputs to later calculations. As an example consider long multiplication.

You can probably multiply all the single digit numbers in your head. Very few people can multiply four digit numbers mentally – at least without going through some more complex process (long multiplication and remembering the intermediate stages rather than using paper doesn't count!) Instead, the task is broken down into smaller, tractable multiplications and the result of each stored. Finally, the results are recalled and totalled.

These intermediate results are held as part of the **state** of the system. A machine which worked this way would need the requisite number of flip-flops/latches to hold these numbers until they were needed. Furthermore, this is not the whole state of the calculation as it is also necessary to "keep track" of which steps have been done, determine what to do next, and know when the result is ready. There is thus some state stored in the **control** logic, as well as the **data processing** logic.

As another consideration, remember that many jobs are interactive over time; it is therefore necessary to have some means of controlling progress through time. Think of controlling a washing machine, or playing a video game.

Sequential Systems – A Summary

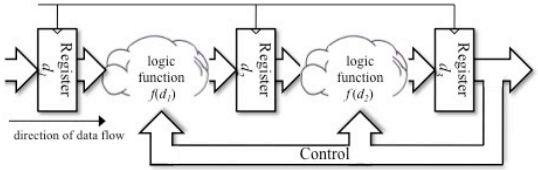
- In general, the outputs from sequential logic circuits depend upon a function of their input values, as well as an arbitrary number of previous input values.
- In other words, the outputs are a function of the history (or sequence) of previous input values, as well as the current input values.
- The usual way of simplifying this view is to regard a sequential logic circuit as having a current state. The sequential circuit stores one or more binary digits (bits) which determine which state it is in.
- Thus, the output of a sequential circuit is a function of the current inputs and the current state.
- The state may (optionally) change under some combinations of current state and inputs. This allows the state to sequence through many different possible values.
- Indeed, useful sequential circuits can be made where there are no external inputs at all. The outputs, and the next state, are solely determined by the current state.

RTL Datapath & Control

The RTL system:

- A set of registers with combinatorial logic blocks
- Operations are performed on the data stored in the registers
- External c**ontrol** supervises the sequence of operations in the system

So the system can be divided into ***data*** and ***control*** paths. The data elements form the **datapath**.



In general the control tells the datapath what to do – and when to do it.

The datapath supplies values that determines the behaviour of the control.

Notes:

RTL Datapath

The RTL design reflects the general sequential system, where data flows through an arrangement of registers and combinatorial logic. Registers hold data being manipulated by combinatorial logic. At the next clock pulse the changed data is loaded into another register (or could be the same!) for further manipulation. As data flows along this path (the "**datapath**") decision are made as to how the data should be operated on. A separate control block monitors the data asserts control signals to the datapath to control how the data should be manipulated.

A clock is normally free-running but it may not be desirable for the register to change every cycle. Usually, therefore, there will be a clock enable (CE) input, which is also common to the flip-flops in one register. Different clock enables control different registers, whereas the clock will typically be the same signal in all the registers in the system.

More rarely, other inputs are also present. For example, there may be a 'clear' signal that sets all the flip-flops to a known value; this may be used for initialising the system. (It has not been shown here.)

Registers – a recap

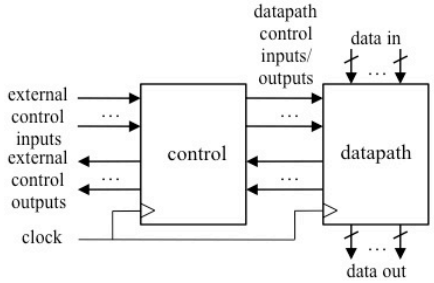
A **register** is a storage device that is used to hold binary data and is constructed from a group of flip-flops, each capable of storing a single bit. Thus, if you want to store a 32-bit word you would need a chain of 32 flip-flops.

A typical register is constructed from a number of flip-flops (D-types in the slide), which each have their own input and output bit but all have the same common clock input; this enforces synchronisation – all the bits switch at the same time.

RTL Design

RTL design follows a two-step process: capture the desired behaviour, and then convert that behaviour to a circuit design.

Behaviour is captured using a **finite state machine** (**fsm**), which is then converted to a controller/ datapath architecture,



- the **datapath** realises the data operations required by the FSM,
- the **control** issues signals to control the operation of the datapath
- the current state of the control/datapath and the state of any input signals will determines the next state

Notes:

RTL Design

A standard architecture is shown in the slide that consists of a controller connected to a datapath. Both elements affect the operation of the other:

- the **datapath** manipulates data in registers according to the system's requirements, and
- the **controller** initiates sequences of commands to control the operation of the datapath according to the requirements at a particular instance in time.

The control logic uses status signals it receives from the datapath to determine the sequence of control signals.

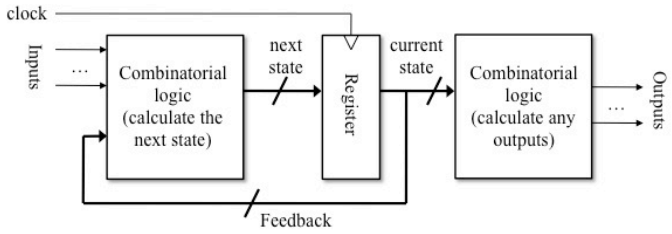
The control logic is generally a sequential circuit that determines the **state** of the digital system that dictate the control commands. At any instance in time the **current state** determines a prescribed set of commands, and (maybe) determines the status of any **outputs** to the digital system. Depending upon the current state and the status of any external input signals, the controller determines the **next state** to initiate the next set of operations.

The controller is a **finite state machine** (**FSM**) that has a finite number of operational states and moves between these states depending upon the current state of the machine and the status of any input signals.

Control – the finite state machine (fsm)

The University of Manchester

The **fsm** is a sequential machine that moves from one 'state' to another according to some well-defined rules.



- **next state** is determined from **current state** and the state of any inputs
- the state is held in a **register**
- **input signals** influence the behaviour of the system
- **outputs** are derived from the current state and possibly the state of the inputs
- visualise the operation of the fsm using the **state transition** diagram

Notes:

Finite State Machines

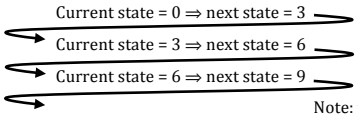
When we looked at designing combinatorial circuits we simply took the Boolean expression and implemented it as a circuit using logic gates - generally in the form of a sum of products expression. However, in the case of sequential systems we cannot use a Boolean equation. Instead we need an approach that will describe the behaviour of the design over time – the **finite state machine**, or **fsm**, allows us to do this.

In a state machine 'what happens next' is determined by a set of rules and is influenced by 'where we are now'. The FSM consists of a set of **states** representing every possible situation the system can be in. The FSM is then in only one state at any particular time, which is known as the **current state** (or present state). The **next state** is determined by the current state and the status of any input signals to the FSM. A combinatorial logic block – known as the next state logic – determines the next state, which the state machine moves to at the next clock pulse. The machine simply moves between different states depending upon the various conditions that determine the transition between these states. Any output signals are determined from the current state of the machine and possibly (although not shown in the slide) the status of any input signals. Again, a combinatorial logic block performs this operation.

The current state of the machine is held as a binary code (so there are a set of valid binary codes to determine each possible state of the system) in a register acting as a memory element. The next state (calculated by the next state logic) is latched into the register on the next clock pulse and becomes the current state of the controller.

Example

Rule: counting in threes



... Note: this is **not** a finite state machine.

A finite state machine – abbreviated FSM – works on this principle, but the number of states is finite; for most real machines the number of states is also quite small.

Implementation

- To remember its history the machine must contain some state information – what state am I currently in? – and consequently a device to hold the current state (i.e. a register).
- To represent each state unambiguously the register must contain at least $\log_2$(states) flip-flops. (N flip-flops allow the representation of 2^N states.)

Number of states	Number of flip-flops required
1	0
2	1
3-4	2
5-8	3
9-16	4
$(2^{N-1}+1)$-2^N	N

Q.

There are two types of FSM: Mealy and Moore machines. See what you can find out about them (hint: the example in the slide is a Moore machine) and produce a sketch of the Mealy and Moore machines noting the differences.

The Synchronous Paradigm

When does a state machine move from one state to the next?

- this could happen as a result of any input change
- in practice it is normal to change only in on a **clock transition**

This is synchronous design – a very useful design simplification:

- the state machine is easy to design, 'all' you have to do is look at:
  - where you are now (current state)
  - what are your inputs
- … and then write down where you want to be next.
- define all state transitions (correctly!) and your system will work ☺.

Notes:

State Transition Diagram

The state transition diagram[1] is used to graphically represent a sequential system and depicts all the possible states in of a system, the influence of input signals and the condition of any output signals. In the state transition diagram there is a labelled circle to represent each of the valid states in the system. The movement between states (on a clock pulse) is identified by arrows between states to illustrate the paths, or transitions, from one state to another. The transitions between states are often determined by external signals that form inputs to the system (although this is isn't the case for the counter shown in the slide). The current state is represented by a binary value in each node that represents the binary number stored in the register (see previous slide).

The Design Process

The complete design process for simple sequential systems usually works something like this:

- From a system specification decide on the inputs and the outputs, and determine the different internal states required. Draw a **state transition diagram**.
- Perform **state assignment** (i.e. label each state with a unique number – typically binary).
  - State assignment is arbitrary but the particular state assignment chosen may influence the complexity of the finished design; as you gain experience optimal state assignment becomes easier.
  - Often some of the state bits can be used as output bits directly.

In a manual process, proceed as follows:

- Complete a **state transition table**.
- **Extract** (and simplify) all the **logic equations** – Karnaugh maps may help.
- Draw the **logic diagram** as **gates** and connect to the appropriate flip-flops.
- Test!

If using a HDL:

- Usually the FSM can be translated directly into HDL code
- Compile
- Test!
- Download to system (if FPGA based)

The Synchronous Paradigm

There are a number of benefits to this paradigm:

- Combinatorial logic can produce output 'glitches' as signals race down different paths. These can be allowed to settle before the clock transition occurs.
- Changes from different parts of the system that arrive in the same clock cycle can be regarded as 'simultaneous'.
- Outputs change 'simultaneously' (in the same cycle) if desired.
- The timing of the system is easy to predict by counting the number of clock cycles needed in the state diagram.
- The design tools are geared towards synchronous machines, that being 'normal' practice.

[1] also known as a state diagram, state graph, or bubble graph

Datapath & Control

The RTL model is useful for showing the movement of data and the operations performed on the data; however, it doesn't provide a good overview of the "control" structure.

Likewise, the FSM model demonstrates the control of a system but doesn't necessarily illustrate the movement of data.

Therefore, a typical sequential system is best handled with a combination of the two approaches:

- the part of the system which **handles the data** where the behaviour is largely independent of the data value is **modelled at RTL**
- the part of the system that **controls the movement of data** is modelled as an **fsm** (or for more complex designs, several, possibly interacting fsms).

Notes:

Splitting a System into Data and Control Elements

A typical digital system can be divided into *data* and *control* parts. The data elements collectively form what is often known as the **datapath** (the term "data" is usually reserved for the values contained therein) – control is usually referred to as just **control**. Naturally these two parts of the system interact. In general, the control tells the datapath what to do – and when to do it; the datapath has less influence on the control but will occasionally supply values that influence its behaviour.

For example a simple **processor** fetches, decodes and executes instructions:

- **Fetch**: output an address, input an instruction
- **Decode**: decide what the instruction should do
- **Execute**: change the processor's register (or control) state
  - the details of execution depend on the instruction fetched

The **fetch** will normally involve some control sequencing but is totally independent of the value (instruction) fetched.

The **decode** may be different for different instructions, but there will be fewer than 2^{32} *categories* of operation[1] so there will still be a large amount of commonality (e.g. the control probably doesn't care *which* registers are being used).

Execution may have a (small) number of different behaviours. However, the controller may order an ADD instruction but will not care what numbers (from the 264 possible combinations) the datapath adds.

Register Transfer Level (**RTL**) is a means of exploiting this separation of data and control in order to simplify the design process. RTL effectively ignores the different values (states) of the data, instead treating them as individual variables. RTL is therefore a hierarchical 'level' of abstraction 'higher' than a gate level design.

Of course, the subsystems are not completely separate. The datapath will influence the control at the decode stage because it supplies the instruction. In a few cases there is also influence during execution: an example would be a *conditional branch*; an instruction (BNE) could specify "Branch if Not Equal" (to zero) – IF the specified data value is not zero a jump would occur, ELSE the instruction would be ignored.

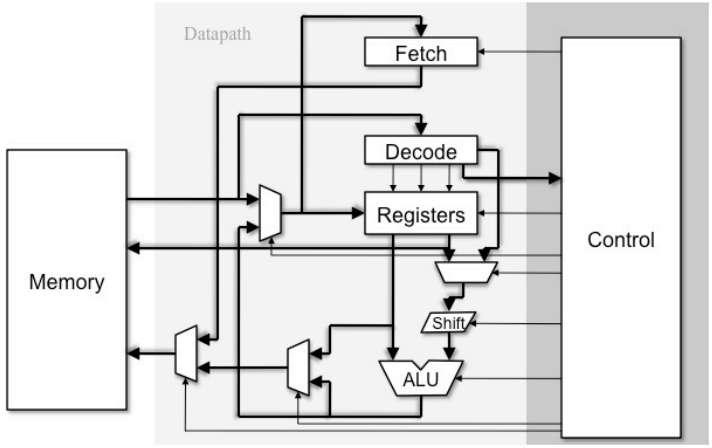
However, the two subsystems separate at well-defined boundaries and this is important in keeping the complexity manageable.

Further subdivision of the datapath may be done: an obvious subdivision here would be to build separate units for each instruction processing stage. This approach is common in high-performance processors.

The control may also be subdivided as appropriate to keep its complexity under control.

[1] Tacitly assuming a 32-bit RISC processor

ARM Processor Architecture



COMP22111: Processor Microarchitecture Part 1

Notes:

ARM Microprocessor

The slide shows a possible implementation of an ARM microprocessor, similar to the simpler versions available. The processor occupies the right hand three-quarters of the figure, the memory to the left hand side.

The ARM is a 32-bit microprocessor, so the thick lines normally show 32-bit buses. The thinner lines show control paths.

The datapath has been shown subdivided into its major blocks such as instruction fetch, instruction decode, the register bank and the Arithmetic Logic Unit (ALU), which performs the actual calculations. Multiplexers allow the selection of inputs at various places.

The control logic is less easy to draw at this scale, so has been concealed.

We will produce a similar, in simpler, picture for a smaller microprocessor later.

Don't try and memorise the design. However, later in the course or at revision time, you might like to consider how some of the ARM instructions you have used in COMP15111 use the various buses shown here.

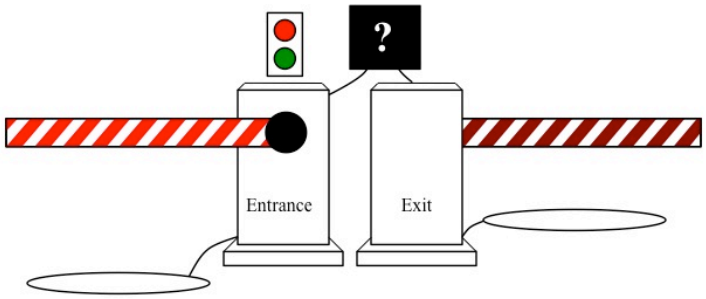
Here are a few to try:

- `SUB  R0,  R1,  R2`
- `LDR  R5,  [R6, R7]`
- `STR  R8,  [R9],     #&20`

RTL Design Example

MANCHESTER 1824
The University of Manchester

A car-park controller



- Allow cars in, one at a time, unless the car park is full
- Allow cars out, one at a time
- Illuminate a sign (and disable the entry barrier) when the car park is full

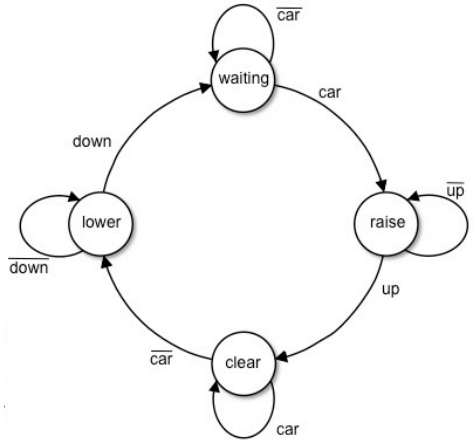
(we aren't going to design the system, just recap the process …)

Notes:

Car Park Example

This example follows form the simple, single barrier example covered in COMP12111. In that example we were designing a simple system (with no datapath) that controlled the open and closing of a car park barrier as a car approached:

Where we designed a simple 4 state finite state machine to give us the required behaviour.



We assigned state codes to each state: waiting = 00, raise = 01, clear = 10, lower = 11, and produced a state transition diagram, that could then be used to produce a Verilog module for the controller.

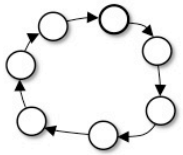
This example adds additional functionality whereby a complete car solution is developed where two barriers allow cars to enter and leave the car park, and the additional system being developed should keep track of the number of cars in the car park.

Partition the design

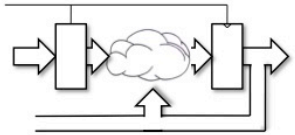
MANCHESTER 1824
The University of Manchester

Recall:

- FSM to describe control behaviour – state transition diagram



- RTL design to describe the datapath



Make sure you understand the problem, the specification, before you design the machine!

Notes:

Design Process

To design a digital system (such as a computer processor) it is first necessary to specify its behaviour. The **specification** should be sufficiently rigorous to determine the behaviour in any combination of internal and input states. This should also help determine the **interfaces**, i.e. the inputs and outputs (usually abbreviated to **I/O**).

When this is done it should be relatively straightforward to determine the requirements for the datapath. The number of variables that must be stored will indicate the minimum number of registers required and the processing operations on these variables will be enumerated. Note that while it is impossible to produce a working system with fewer registers than variables it is perfectly possible to produce one with extra registers; although this is sometimes valuable when attempting to increase performance (techniques such as *pipelining* will be discussed later) it increases the control complexity and is usually undesirable.

A good rule is

"Things should be made as simple as possible – but no simpler."

-A. Einstein

In this simple example we are simply designing a controller that takes input signals and provides output signals that drive directly other aspects of the system as a whole. There is no datapath as we are not manipulating any "data". The modified example later however, where the car park controller has input and output barriers and counts the number of cars in the car park requires a datapath as in that case we have data – the number of cars.

State Transition Diagram

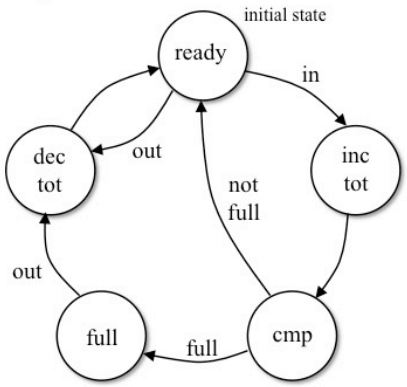
The car park needs to keep count of the number of cars

The 'count' can be treated as a data item and operations on it performed by a datapath.

A state transition diagram could be:

where
- input "in" indicates a car entering,
- input "out" a car exiting, and
- output "full" is true when the car park is full

- Use a constant to represent the "maximum capacity".
- The "ready" and "full" states are exited only when "out" or "in" is true.



This is only an indication

Notes:

Car park controller specification

Specification

A controller for access to the car park:

- Cars may leave at any time.
  - Cars attempting to leave will be sensed, and counted out.
- Cars may enter at any time, unless the car park is full.
  - Cars attempting to enter will be sensed
    if the car park is not full they will be admitted and counted in
    else the car park is full so they will not be admitted
- If the car park is full, switch on the sign.

Interface

The controller gets inputs from two sensors and outputs control signals:

- Sensors
  - An input sensor
  - An output sensor
- Actuators
  - A 'full' light
  - An input barrier control
  - An output barrier control

(The following design neglects the barrier controls. You are encouraged to add outputs to control them.)

Deductions

We have made the following deductions about the design:

- The occupancy of the car park must be deduced from the number of cars entering and leaving.
- Some form of controllable up/down counter will be needed
- The data path width must be great enough to accommodate the largest count values – equivalent to the capacity of the car park, e.g. for 800 parking spaces, (at least) 10 bits are needed

We can now sketch a preliminary state diagram, as shown in the slide, with 5 states.

The difference in the design of this controller is that we now have some **data** to deal with – the number of cars in the car park. Hence, we will be required to design the *datapath* to handle this.

Datapath Design - RTL

The informal state transition diagram suggests that we need to be able to:

- keep a running total of the number of cars
  - use a register to hold a value
- increment the total
- decrement the total
  - some form of arithmetic unit to increment (add 1) or decrement (subtract 1) to form a running total as cars enter or leave the car park
- compare the total with the maximum allowed capacity
  - we could compare by subtracting the running total from the maximum allowed capacity, this would then give an indication whether the maximum number of cars has been reached.
  - we could use the same arithmetic unit we use to increment/decrement the count to determine whether the maximum has been reached.

Notes:

Datapath Design

Time to think about the datapath. What do we need the system to do?

- Keep track of the number of cars
- Determine when the car park is full

We need to keep a running total of the number of cars in the car park, s clearly the controller needs a single variable to keep track (and count) the number of cars. An *N*-bit register should suffice for this purpose. At this stage we need not be worried about the size of the register (*N*), although it should be chosen so that it can count at least to the maximum number of spaces available.

The system clock runs freely, i.e. continuously, so we will need a clock enable (ce) signal to 'freeze' the register when no activity is required (i.e. no cars are entering or leaving the car park).

Evaluation functions are required including incrementing (+) and decrementing (-) the register that stores the running car total. To do this the number '1' may be fed to the second port of the adder/subtractor; the running total is passed to the other port.

Subtraction is done by: $a - b = a + \bar{b} + 1$, i.e. invert the bits and put a carry into the least significant bit) - remember 2's complement arithmetic!

A slightly less obvious function is the ability to compare the count with the car park's capacity.

Comparison

There are various ways of comparing values. If a comparison alone is required a specialist unit can be built. However, in this design we already require the ability to add and subtract and a comparison can be done simply by subtracting.

To compare A and B, subtract one from the other

Result (A-B)	Conclusion
zero	A = B
positive (not zero)	A > B
negative	A < B

The positive/negative result can be determined by testing the sign 'bit' (the most significant bit) of the result (in 2's complement representation).

To compare with the "max" value, the current total must be made available to the adder/subtractor.

Datapath Design - RTL

MANCHESTER 1824
The University of Manchester

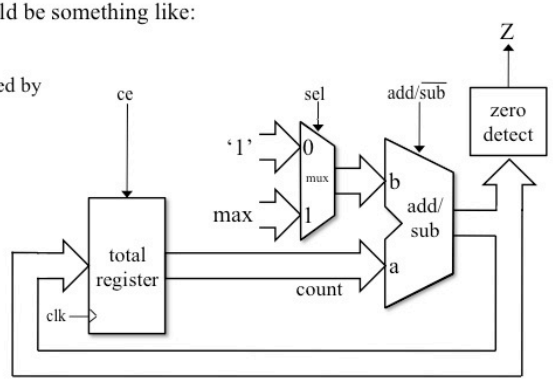
An datapath design could be something like:

The 'datapath' is controlled by 3 input wires:

- ce
- sel
- add/sub

and produces one control output:

- Z



The data buses are N bits wide, where $2^N >$ maximum number of cars (max).

Notes:

Datapath Design

So our initial datapath design includes:

- A register to hold the N-bit value for the number of cars in the car park – a clock enable (ce) maintains control over when the contents are updated.
- An *N*-bit bus.
- An adder/subtractor to increment/decrement the car running total (result is stored back to the register)
- One input to the adder/subtractor comes from the current total register, the other input depends upon whether we are incrementing/decrementing the total, or we are determining whether the maximum number of cars has been reached.
- Selection between '1' and 'max' when incrementing/decrementing or determining whether the car park is full is done through a multiplexer (mux) before the adder/subtractor.
- A zero detector circuit to determine whether the maximum number of cars has been reached (from the result of subtracting the current total from the maximum).
- Control signals
  - ce – clock enable for the register (output from the controller)
  - sel – selection signal for the mux – '0' selects the input '1' in order to add/subtract one from the running total when incrementing/decrementing the running total (output from the controller), and 'max' when determining whether the maximum number of cars has been reached.
  - $\overline{\text{add/sub}}$ - to select addition or subtraction in the adder/subtractor (output from the controller)
  - Z – result of the zero detector – goes high when the maximum number of cars has been reached (input to the controller) – can be used to control the car park "full" light.

System Design

We have designed the datapath, we now have to complete the FSM design by producing a state transition table, from this we can move on to a Verilog implementation, which we will look at later.

Q. You can have a go at this part in order to recap the process.

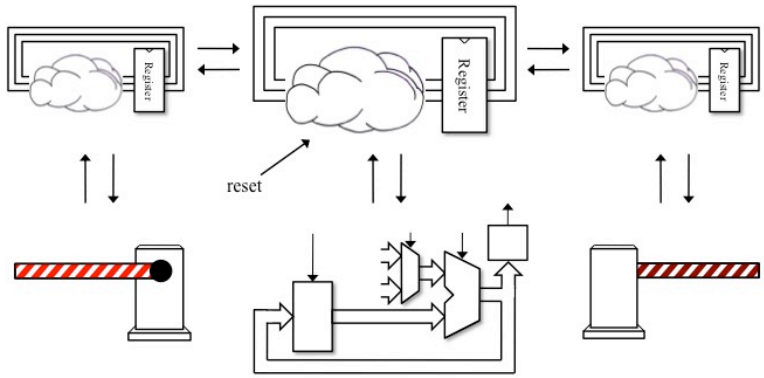
State Codes/State Vectors

We have 5 states so we need a 3-bit code to uniquely identify the 5 states (with 3 redundant codes). We could use optimization techniques (not covered here) to assign the state codes to states in an optimal manner, or we could do a simple assignment: ready – 000, inc tot – 001, cmp – 010, full – 011, dec tot – 100 (101, 110, and 111 unused).

Q. Do you think the way you assign state codes will affect the design?

Q. Another method of assigning state codes is one-hot assignment – what does this involve and why is it useful? What is the overhead of one-hot assignment?

The Final Implementation



Each barrier has a fsm, along with the fsm counting the number of cars.

Notes:

Final Implementation

The final design of our (more complex) car park controller will have 3 finite state machines to:

1. Control the input barrier
2. Control the output barrier
3. Keep track of cars entering and leaving and keep a count of the number of cars in the car park.

We only need a datapath for the third controller as this is keeping track of data – the number of cars in the car park.

Reset

The only thing we haven't discussed, but is shown in the diagram opposite is that of system reset. It is important that any system has a reset procedure. In most cases this results in the system being out in a known state, such as the initial state when the reset signal goes high.

If the reset occurs independent of the system clock, then it is *asynchronous*. Otherwise, it is *synchronous* and occurs at the next rising clock edge.

Summary

MANCHESTER 1824
The University of Manchester

When designing sequential systems …

- **Understand** the problem.
- Produce a **specification**, maybe sketch a state diagram; determine the **interfaces**.
- If necessary and appropriate, **partition** the design to **simplify** the problem.
- Identify the flows of data; sketch an **RTL datapath**; list the control signals.
- If it helps, draw **timing diagrams** (we haven't gone this far).
- Formalise the **state diagram**; verify that the specification is fulfilled.
- Design the **fsm** that drives the control signals (process described here).
- **Implement**

We will be following this process to develop the Stump processor, as well as revisiting the MU0 processor as our design example later …

Notes:

Next

We will look at RISC processors. We will recap the design of MU0, a processor you are already familiar with. We will then move on to the design of the Stump processor, starting with the design of the instruction set.