UCD School of Electrical and Electronic Engineering

EEEN40280
Digital and Embedded Systems

System on Chip Design

Introduction AHB-Lite 连接到各种外围模块

In this assignment you will build a complete embedded system on an FPGA, with an ARM Cortex-M0 processor, connected to various peripheral blocks using an AHB-Lite bus, running software written mostly in C. Most of the peripheral blocks are provided, but you will probably want to design one or two more. You will also have to integrate the blocks that are provided into the system, making the necessary connections to the bus and other signals.

Requirements

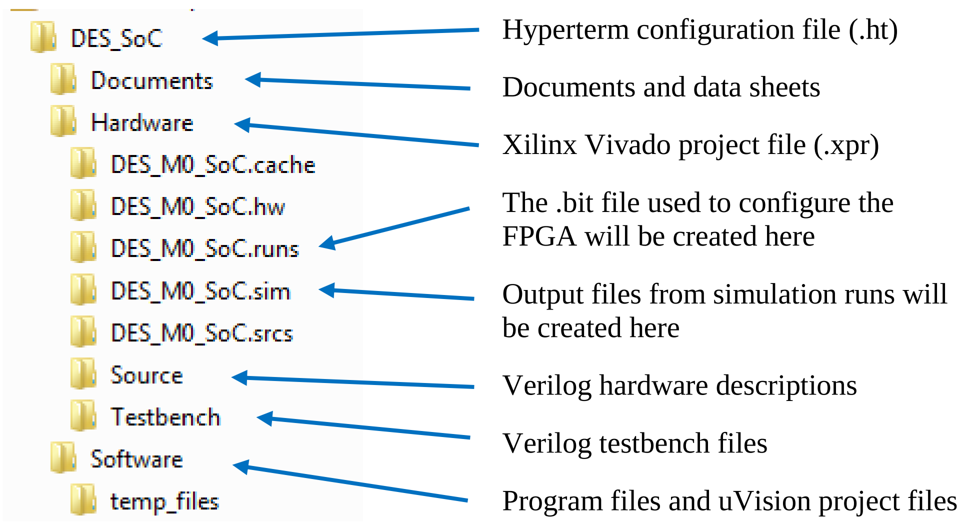
There is a 3-axis accelerometer on the Nexys-4 board, an Analog Devices ADXL362. The minimum requirement for this assignment is to use the accelerometer to measure acceleration in one axis, and display this information on the LEDs on the Nexys-4 board. This is discussed further in the System Design section on page 6. It should be possible to meet this requirement with a team of two students, using the five lab sessions scheduled. However, you will have to work as a team, with each person taking responsibility for one part of the design.

If you prefer, you can form a team of three or four people, and extend the system to use the accelerometer data to control an icon on a VGA display, or to control the frequency of an audio output. The hardware that you would need for either of these extensions is available on the Nexys-4 board. Grading of the assignment will depend on what you have achieved in relation to the size of your team.
加速计数据用于控制VGA显示器上的图标， 或控制音频的频率 输出

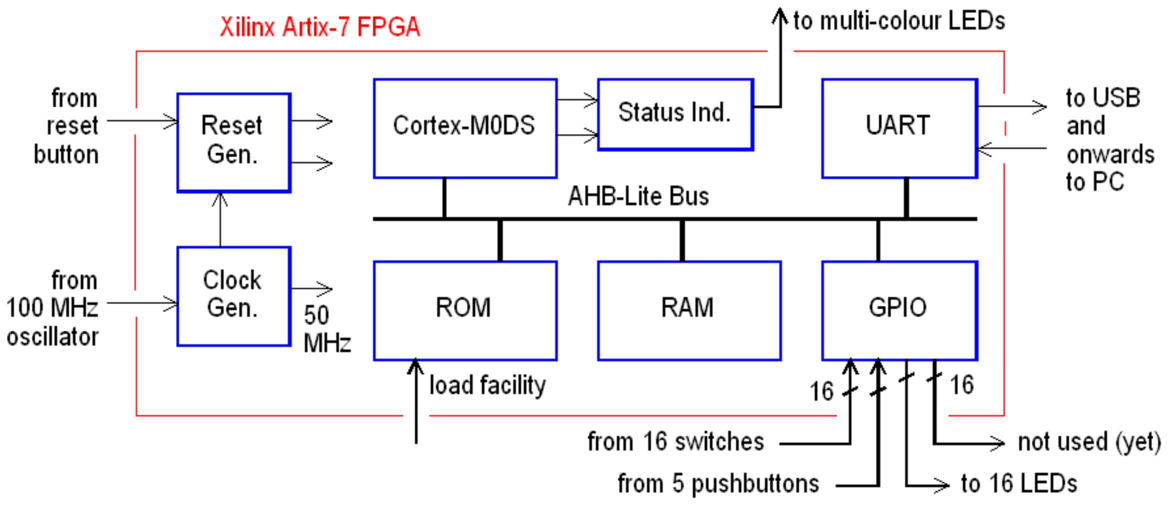
Part 1: Initial System

Before starting on the main assignment, you should build the basic system and test it, using the software provided. Then submit a brief interim report, at the end of the first lab session.

Download the `DES_SoC.zip` file on Blackboard, and unzip it into the correct folder on the lab computer: `Documents\EmbeddedSystems\???day`. Take care to use the folder names when extracting the files – this operation builds a directory structure for the entire assignment.


DES_SoC
Documents
Hardware
DES_M0_SoC.cache
DES_M0_SoC.hw
DES_M0_SoC.runs
DES_M0_SoC.sim
DES_M0_SoC.srcs
Source
Testbench
Software
temp_files
Hyperterm configuration file (.ht)
Documents and data sheets
Xilinx Vivado project file (.xpr)
The .bit file used to configure the FPGA will be created here
Output files from simulation runs will be created here
Verilog hardware descriptions
Verilog testbench files
Program files and uVision project files

Start the Xilinx Vivado software, and open the project `DES_M0_SoC`. When the project opens, you should see the Verilog modules arranged in the appropriate hierarchy in the Project manager window. Modules that are not yet included in the design will be shown separately.



Block Diagram

The top-level block diagram of the test system that you will build is shown above. All of the blocks shown here are provided – these are listed below, with brief descriptions.

AHBliteTop.v

This is the top-level module, in which all the individual modules in the system should be instantiated and connected together. It is provided in incomplete form – the data memory (RAM) and the GPIO and UART peripherals are missing. A simple testbench is provided, which provides the clock and reset signals needed to simulate the system.

clock_gen.v

Clock generator – provides a 50 MHz clock for all the hardware and the AHB-Lite bus, using the 100 MHz clock input from the oscillator on the Nexys-4 board. count edges?

reset_gen.v

Reset generator. This provides a hardware reset signal, active at power on or if the CPU RESET button is pressed. It keeps reset active until the clock generator is running. It also provides a separate reset signal for the processor and AHB-Lite bus, which conforms to the Cortex-M0 reset requirements, and can be requested by software.

status_ind.v

Status indicator – uses two multi-colour LEDS on the Nexys-4 board to display status signals.

Colour	**LD17 – on the left**	**LD16 – on the right**
Blue	Processor and AHB bus reset active	Processor sleeping (wait for interrupt)
Red	Processor lockup	ROM loader active (see below)

CORTEXM0DS.v

This is a wrapper for the Cortex-M0 DesignStart processor, provided by ARM. It makes the CPU registers visible in simulation. This module has been synthesised already, to save time.

AHBDCD.v

Address decoder, defines the address map. It selects the appropriate slave on the AHB-Lite bus in response to the address input. It also controls the multiplexer block, below. You will have to modify the decoder block as you add more slaves to the system.

AHBMUX.v
Multiplexers for signals from the slaves. You should not have to modify this block, but you will have to connect more slave signals to it.

AHBprom.v
Program memory, 32 kByte, read-only from the AHB-Lite bus. The memory is constructed using block RAM components on the FPGA. This memory contains the machine code for a simple test program. You can change the program later, as this program memory includes hardware (ROM loader) to allow new program code to be loaded through the serial port after the FPGA has been configured. See instructions later.

AHBbram.v 因为此程序内存包括 硬件（ROM加载程序）允许新的程序代码通过串口
Data memory, 16 kByte, with read and write access from the AHB-Lite bus. It supports byte, half-word and word (8, 16 and 32-bit) writes. This memory also uses block RAM on the FPGA.

AHBgpio.v
General-purpose input-output block. This provides two 16-bit input ports and two 16-bit output ports, with byte writes possible to the output ports. Full details, including an RTL diagram, are in a separate document, in the `DES_SoC\Documents` folder. A testbench for this block is also provided, which simulates some AHB transactions.

AHBuart2.v 无奇偶校验
UART – asynchronous serial interface block. This block can transmit and receive at 19200 bit/s, in groups of 8 data bits with one start bit, no parity and one stop bit. The transmit and receive paths each have 16-byte buffers or queues. A status register provides bits to indicate if these buffers are full or empty. Each of these bits can be enabled to cause an interrupt, by setting appropriate bits in the UART control register. Full details are in a separate document, in the `DES_SoC\Documents` folder. A testbench is provided.

Constraints file
Along with all the hardware blocks, a constraint file is provided: `Nexys4_Master.xdc`. This defines the pin numbers and signal voltages for all the connections to the FPGA. The signals not used in the test system are commented out. You will need to modify this file later, to include the signals from the accelerometer, and maybe some other signals.

Block Simulation
In Vivado, select the `sim_gpio` simulation. Click on Run Simulation and choose a behavioural simulation.

Expand the simulation timing diagram window, and zoom to fit the entire waveforms. You should be able to see some AHB transactions being performed by the testbench – writing to registers and reading from registers in the GPIO block. Examine the Verilog testbench to see what is supposed to happen. Examine the documentation and the Verilog description for the GPIO block to see how it works.

System Integration
The system that you have been given is not complete – some of the slaves are not connected to the AHB-Lite bus. You need to build a complete system, as shown on page 2, and then run some test programs on it.

Add RAM block
Edit the top-level module, and add the `AHBbram` block to the system. Instantiate the block and connect its ports to the appropriate AHB bus signals. These are mostly shared signals, but there are separate signals, already defined, for its slave select signal and for its output signals – these wires are already connected to the address decoder and multiplexer blocks. You need to identify all the relevant wires and connect them to the appropriate ports on the `AHBbram` block.

If you have done this correctly, you should see the RAM block appear in the hierarchy. If you expand it, you will see a block RAM, which was generated from the IP catalog.

At this stage, you should be able to Open Elaborated Design, and see the block diagram of the system so far. You will see some warnings about unconnected ports, and you will also see these on the block diagram – these are for the blocks that you have yet to add.

Add GPIO block
Now add the GPIO block to your system. Instantiate it and connect its bus ports to the bus signals as before. This time, you will have to create wires for the signals specific to this block.

You will also have to connect the other ports on the block. The test program expects the `gpio_out0` port to connect to the LEDs on the Nexys-4 board. The ROM loader also uses the LEDs when it is active, so a multiplexer allows the two blocks to share the LEDs. Connect the `gpio_out0` port to the multiplexer input, using existing wires called `led_gpio`.

Connect the signals from the switches on the Nexys4 board, `sw`, to port `gpio_in0`. Connect the signals from the push-buttons to the least significant bits of `gpio_in1`, with 0s on the other bits. There is already a signal called `buttons` that combines the 5 button signals.

You can check your work by opening the elaborated design again…

Add UART block
Next add the UART block. The software uses this block to communicate with the PC.

Its serial transmit and receive ports connect to ports of the top-level module, called `RsTx` and `RsRx`. You also need to connect its interrupt request line to one of the `IRQ` inputs of the processor – the software expects this interrupt at `IRQ` bit 1.

Address decoding
Now modify the address decoder block to position the new slaves at the correct places in the address map. The address decoder is designed to check only the 8 most significant bits of the address, so each slave gets a 16 MB range of addresses – far more than it needs.

You should see that the program memory (ROM) is already located at address `0x00000000`, and the data memory (RAM) at `0x20000000`. You need to add logic to put the GPIO block at `0x50000000` and the UART block at `0x51000000`.

Check the elaborated design again. Note that you can expand any block to see what is inside it.

Implementing Your System on FPGA
In Synthesis Settings, set `fsm_extraction` off. Then Run Synthesis. You can select to run it in the background if you want to get on with other tasks while it is running.

When the synthesis has run, view the synthesis report. You will find many warnings, hidden among even more information messages. You can also see these warnings on the Messages tab at the bottom of the Vivado window. You need to check that all of these are acceptable before you proceed. If you are not sure, ask for help!

Then Run Implementation. This may take some time.

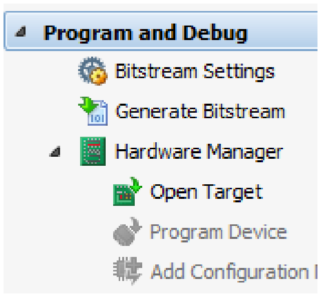
While you are waiting, connect the Nexys-4 board to the PC and switch it on. You should see it demonstrate some of its features as a self-test.

When Implementation completes, with no warnings, Generate Bitstream. This produces the `.bit` file that you need to download to the FPGA to configure it with your design.

HyperTerm
Run HyperTerm – Start menu, All Programs. On the File menu, open the connection file in the project folder: `Nexys4.ht`. This attempts to connect to a device on port 5 – if it fails, you may need to select a different port. Leave the window open.

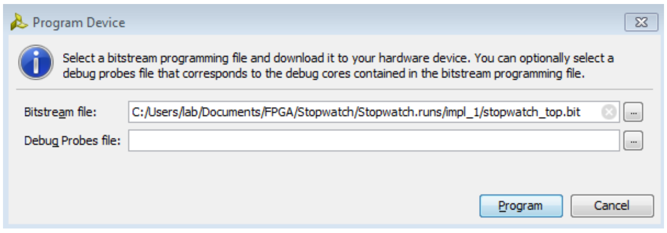
If you need to make a new connection, choose a port with a number greater than 3. Configure the connection for 19200 bit/s, 8 data, no parity, 1 stop bit, no handshake. Save the settings.

Configure FPGA
To configure the FPGA, use the Hardware Manager.



Select Open Target. The first time you do this, you should choose Open New Target. Accept the defaults, and the Nexys4 board should be found and identified. The FPGA on the board should also be identified.

If you do this again, when you click Open Target you should see the previous FPGA board listed – just select it. If there is more than one board, the top one is usually the most recently used – there is a label on the back of each board with the last six digits of the board number.


Program Device

Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.

Bitstream file: C:/Users/lab/Documents/FPGA/Stopwatch/Stopwatch.runs/impl_1/stopwatch_top.bit

Debug Probes file:

When you have opened the target board, click Program Device, and select the only device offered. The correct bitstream file will usually be offered. This file should have the name of your top-level module, with extension `.bit`. You do not need a Debug Probes File. If all is correct, click Program.

Your design should now be on the FPGA, and ready for testing. If you switch off the board, the configuration will be lost, and you will have to download the `.bit` file again: Program Device. There is no need to repeat synthesis and implementation if the design has not changed.

Testing
When the FPGA has been configured, press the `CPU RESET` button. You should see a message appear in the HyperTerm window. If you type characters here, they will be echoed back to you. When you press return, the whole sentence will be repeated, with the letters modified.

The LEDs on the Nexys-4 board should show each character code as it is received. When you press return, they will flash a pattern that depends on the switch settings. You will also notice the right-most status LED (LD16) showing blue most of the time – this indicates that the processor is sleeping, waiting for an interrupt.

Software
Run the uVision4 software. Open the `DES_M0_SoC_Basic_Uart` project located in `DES_SoC\Software`. Check the project options – on the Target tab, you should see the ROM and RAM areas defined. Check the path to the `.sfr` file, and check that it points to the file on your computer.

Examine the code. This is the program that was already loaded into the program memory in your system, and is now running on the processor in the FPGA on the Nexys-4 board. You should be able to understand why the system is behaving as it is.

Note the "critical section", where interrupts are disabled to prevent any change to the `RxBuf` array while it is being processed. Interrupts should only be disabled for a short time, so the printing of the `TxBuf` array, which will take a long time, is done outside the critical section.

Modify the program
When you understand the software provided, change it. Make it send a different welcome message, so you will recognise the new version. Modify it so that it reports the number of characters that you have typed, and also reports the state of the switches as a number.

Build your new program. You should find an output file called `ROMcode.hex` in the Software folder. Examine this – it contains the 32-bit words that should be in the program memory.

The first word defines the initial value of the stack pointer. The second is the address of the first instruction to be executed after reset. The 18th word should be the address of the interrupt handler for the UART interrupt.

Download to program memory
On the Nexys-4 board, press the black pushbutton marked `BTNU`. While holding this down, press and release the reset button. Then release `BTNU`. This activates the ROM loader hardware, which keeps the processor in reset. The status indicators should show blue and red.

In HyperTerm, on the Transfer menu, select Send Text File… Select the `ROMcode.hex` file – you will have to change the type of file to "All Files" to see this. Opening the `ROMcode.hex` file will cause HyperTerm to send the file, byte by byte. The ROM loader should receive it – you should see the LEDs on the Nexys-4 board displaying a count of the words received.

When the file has been sent, click in the HyperTerm window and press Q on the keyboard. This tells the ROM loader that the transfer is complete. The status indicators should change as the processor starts running your new program.

Interim Report
When you have completed the initial tasks, and before you start on the system design, submit your modified Verilog files: `AHBliteTop.v` and `AHBDCD.v` for review. Also submit your modified program file, `main.c` for review. Use Notepad++ to print just the **relevant sections** of these files, staple them together and write your names on the first page.

These pages will be returned with feedback, but will not be graded. One submission per team is sufficient.

Part 2: System Design
The data sheets for the Analog Devices ADXL362 accelerometer and for the Nexys-4 board are in the `DES_SoC\Documents` folder, and also on Brightspace. Section 13 of the Nexys-4 data sheet describes how the accelerometer is connected to the FPGA. The accelerometer is mounted on the underside of the circuit board, marked as IC13. It is oriented so that the Z axis is perpendicular to the circuit board, and the Y axis is parallel to the line of 16 switches.

Look at the data sheet for the accelerometer. There is no need to read every word at this stage – you just need to get an overview of what information is available from the accelerometer, and how you can get access to this information.

Section 9 of the Nexys-4 data sheet describes the LEDs on the board – both the line of 16 separate LEDs that you have used already, and the 8-digit 7-segment display. The minimum requirement in this assignment is to display acceleration on these LEDs in some way. Ideas:

- You could use the line of 16 LEDs to represent acceleration in one axis, using either a one-hot code (one LED on, in a position that varies with acceleration) or a thermometer code (a set of adjacent LEDs on, with the number of LEDs turned on varying with acceleration.

- You could use the 8-digit display to show acceleration in numerical form. Four digits would give adequate resolution, even with a sign symbol, so you could use the eight digits to display acceleration in two axes.

- You could also use the 8-digit display to show acceleration in two axes in a graphical way, lighting one segment at a time, with the position of the segment varying with acceleration. Note that if you only light one segment, the current in it might be excessive, so you could drive the LED with a square wave, at a frequency high enough to avoid visible flickering.

If you want to make the system more flexible, you can use some of the switches on the board to decide what information to display.

As you have seen in the sample program, you can send information to the PC, to be displayed in HyperTerm. This could be useful in testing. You could also send commands from HyperTerm to your system, to control how it behaves or to configure it in different ways.

If you have a larger team, and want to extend the system, you can use the VGA output or the audio output provided on the Nexys-4 board. The basic ideas are described below.

Section 8 of the Nexys-4 data sheet describes the VGA port, and the signals that it should provide. (Ignore the description of an old CRT display – LCD display panels with VGA input use the same signals.) You could arrange for the display to show a small rectangle, in a colour that contrasts with the background colour. The position of this rectangle, in two dimensions, could be proportional to the acceleration in two axes. Or the velocity of the rectangle could be proportional to the acceleration – tilt the circuit board more to make the rectangle move faster?

Section 15 of the Nexys-4 data sheet describes the audio output hardware. This is essentially a low-pass filter, which allows you to generate an analogue output signal from a PWM signal. You could arrange to generate a continuous sinusoidal output, at a frequency proportional to acceleration in one axis – a basic musical instrument. If you want to generate distinct musical notes, you could choose the frequency from a finite set, based on acceleration in one axis. Then you could set the amplitude, or turn the sound on or off, based on acceleration in another axis.

Architecture

When you have decided what you want to do, you can start designing a system to do it. First break the problem into a few large blocks, and think about the information flows between the blocks. Then design one block at a time – if necessary, divide a block into smaller blocks…

For example, you will have to connect the accelerometer to the FPGA. The accelerometer uses a SPI interface – it acts as a slave. So your design will need a SPI master interface to communicate with the accelerometer. This will probably be one of the blocks in your design.

You could design a hardware SPI interface to connect to the AHB-Lite bus – this could handle all the SPI signals, with the correct timing, and provide some registers that the software could access to control the interface and to transfer bytes (or maybe 16-bit values?) to and from the accelerometer. The software would decide what information to send to the accelerometer, and what information to collect from it – the hardware would provide a means of communication.

Alternatively, you could connect the SPI signals to some input and output signals on the GPIO block, and handle the entire SPI interface in software. In this assignment, the hardware solution would be preferred, but if you cannot design hardware, you could choose the software solution.

Similar decisions will arise if you want to use the 8-digit display, or generate VGA signals or audio signals. How much of the work should be done in hardware, and how much should be done in software?

Another high-level design decision will be on interrupts. What interrupts will you use in your system? You have an example using an interrupt to signal when data has arrived on the UART interface – this is useful because this data arrives at times that are outside your control, when someone presses a key on the keyboard of the PC. You could use another interrupt to signal that data had arrived on the SPI interface. This would not be particularly useful, as this data only arrives when you request it, so you already know that it is coming.

If you want an interrupt at regular intervals, to make your software do something at a known rate, you can use the SysTick timer, which is built in to the processor. This is described in lecture notes on the Cortex-M0. It can be configured to cause interrupts at rates from about 3 Hz upwards.

Block design

When you have completed the top-level design, divide the block design tasks among the members of the team. For example, someone needs to do the detailed design of the SPI master interface, either as hardware or as a set of software functions. That person will need to read about the SPI interface in the accelerometer, and consider bit rate, clock polarity, phase, etc.

Someone else might write the software that communicates with the accelerometer. That person will read different parts of the accelerometer data sheet, to see how to configure the accelerometer, and how to get the information that you want from the accelerometer.

You should also make a plan that will allow the team to complete the rest of the work in the time remaining. You have a total of 5 lab sessions for this assignment, but the first session will probably be mostly occupied with the test system.

If you wish, you can discuss your plan with the staff in the lab at this point, to see if it makes sense or to get advice on possible improvements.

Final Report

You will work on this assignment as part of a team, but you must submit your own report. Block diagrams, RTL diagrams and flow charts may be shared within the team, with credit given to whoever created the diagram. The text of the report should be your own.

The deadline for all reports is 16:00 on Wednesday 1 May. Standard penalties apply for late submission. You should submit your report as a pdf document through the appropriate channel in Brightspace. Please keep the filename short, but include your student number. Avoid any accented letters or characters that are not in the ASCII set.

Your report should start with the usual declaration of authorship – a template will be provided on Brightspace. It should include:

- An overview of your design, with details of the information that you can display, the user interface, etc. Also details of the design process: the main blocks that you identified, your decisions on how each block would be implemented, etc. This section should make clear who worked on each block.

- Details of the blocks that you designed. For software blocks, explain how you broke the problem into separate functions, and what each function does. Use a flow chart or diagram if you wish. For hardware blocks, give an RTL diagram, and explain how the block works and why you designed it that way.

- A brief description of the tests you performed, to verify that your system works as intended. For hardware blocks that you designed, include an example timing diagram from simulation.

The Verilog and program files will be collected separately, but you may want to include extracts from some of these files in your report, to show how you solved a particular problem. In that case, format the code in a fixed-pitch font, so that indentation is preserved. Use a small font, and shorten long lines, so that lines do not wrap. There is no need to leave a blank line between every two lines of code, but a blank line can be useful to separate different sections of code.

One member of the team should collect all the source files used in the assignment: Verilog design files, testbench files and program files. Combine all of these into one zip file. Do **not** include the entire project folder – that will be many megabytes in size, and only a few kilobytes are needed.

This zip file should be submitted through a separate channel in Brightspace. Keep the filename short, but include the student number of whoever is submitting it. Avoid any accented letters or characters that are not in the ASCII set.

Grading

Grading will take into account:

- The capability of the system that you have designed: what can it do?
- The design choices that you made: for example, a hardware SPI interface would be preferred to a software solution;
- The quality of the implementation: well-designed hardware and well-structured software will be expected;
- The quality of the documentation: both the report and the comments on the Verilog and C code;
- The number of students in the team – a larger team will be expected to achieve more.