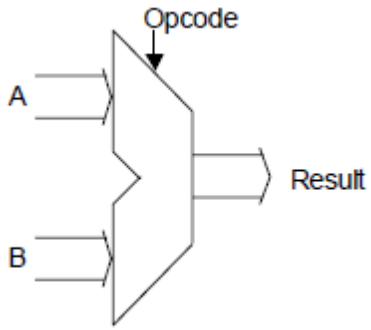
Overview
In this project you will build a simple part of a microprocessor. Firstly you will build two main blocks: the ALU and the register file, then you will connect them together and run a simple machine code program on them, and to verify at the end that your result is correct.

In order to ensure that your design is related to yourself, various aspects of the design will be dictated by your student ID numbers. These include the machine code used by your design, the contents of the register file, and the purpose of the program that you will use to demonstrate the correctness of your design. This means that every student should have a unique design.

You have to work individually in this project and demonstrate your work to me and submit a report about your design and implementation of this project.

Part 1

The ALU
Write a VHDL description of an ALU with two 32-bit inputs, *A* and *B*, and a 32-bit output *Result*.



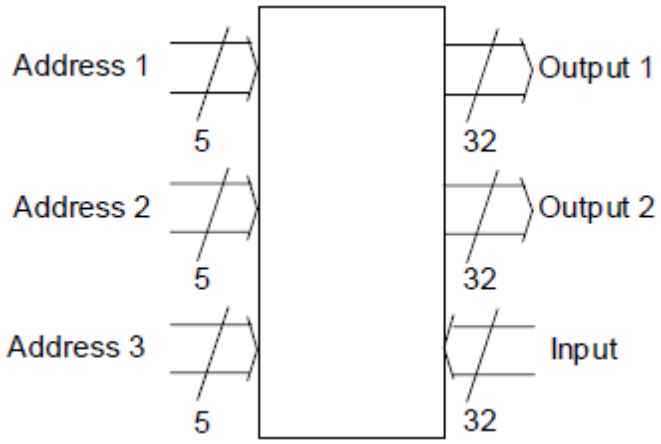
The result is derived from one or both of the inputs according to the value of a 6-bit opcode. The operations that the ALU can perform are listed below:
• a + b
• a - b
• |a| (i.e. the absolute value of a)
• -a
• max (a, b) (i.e. the maximum of a and b)
• min (a, b) (i.e. the minimum of a and b)
• avg(a,b) (i.e. the average of a and b – the integer part only and remainder is ignored)
• not a
• a or b
• a and b
• a xor b

The opcode that will be used to represent each of these operations is determined by the last digit of your student ID number. The table below shows which opcode you should use in your design for each instruction.

Digit of ID no.	1
a + b	6
a – b	8
|a|	10
-a	12
max(a,b)	14
min (a,b)	11
avg(a,b)	13
not a	15
a or b	2
a and b	3
a xor b	9

The register file

Inside a modern processor there is a very small amount of memory that is used to hold the operands that it is presently working on. This is called the *register file*, and normally has the following appearance.



This is a very small fast RAM, typically holding 32 x 32-bit words, and therefore requiring a 5-bit address to select out one of the 32-bit words. It is unlike normal RAM in that it can process three addresses at the same time, two of which are always read operations, and one of which is always written to.

Output 1 produces the item within the register file that is address by *Address 1*. Similarly *Output 2* produces the item within the register file that is address by *Address 2*. Input is used to supply a value that is written into the location addressed by *Address 3*.

The initial values stored in the register file are determined by the second-from-last digit of your student ID, and are shown in the table below:

ID/ Location	3
0	0
1	12996
2	11490
3	7070
4	6026
5	3322
6	10344
7	6734
8	15834
9	15314
10	6000
11	12196
12	11290
13	13350
14	2086
15	6734
16	7430
17	14102
18	13200
19	3264
20	2368
21	15846
22	11710
23	14736
24	5338
25	5544
26	1852
27	3898
28	16252
29	1048
30	5642
31	0

(N.B. these values are in *denary (i.e. base 10)*. You will need to convert them to binary or hexadecimal.)

Part 2

In this part you will connect the ALU with the RAM to form a simple microprocessor.

Clocking the register file

The register file that you created in the first part was a combinational circuit. This causes some serious problems if, for example, address 2 and address 3 are the same. The circuit

would read output 2 from the location referenced by address 2, at the same time that the input is over-writing that location.

These problems can be solved by synchronising the register file to a clock. You will need to add an extra input named *clock*, and give the register file the following behaviour:

On the rising edge of the clock:
• *Output 1* produces the item within the register file that is address by *Address 1*.
• *Output 2* produces the item within the register file that is address by *Address 2*.
• *Input* is used to supply a value that is written into the location addressed by *Address 3*.

Under all other circumstances:
• the outputs are held constant at the values they assumed during the last rising edge of the clock.

You should check your design thoroughly, and in particular make sure that it behaves sensibly when the address for the input is the same as the address of one of the outputs.

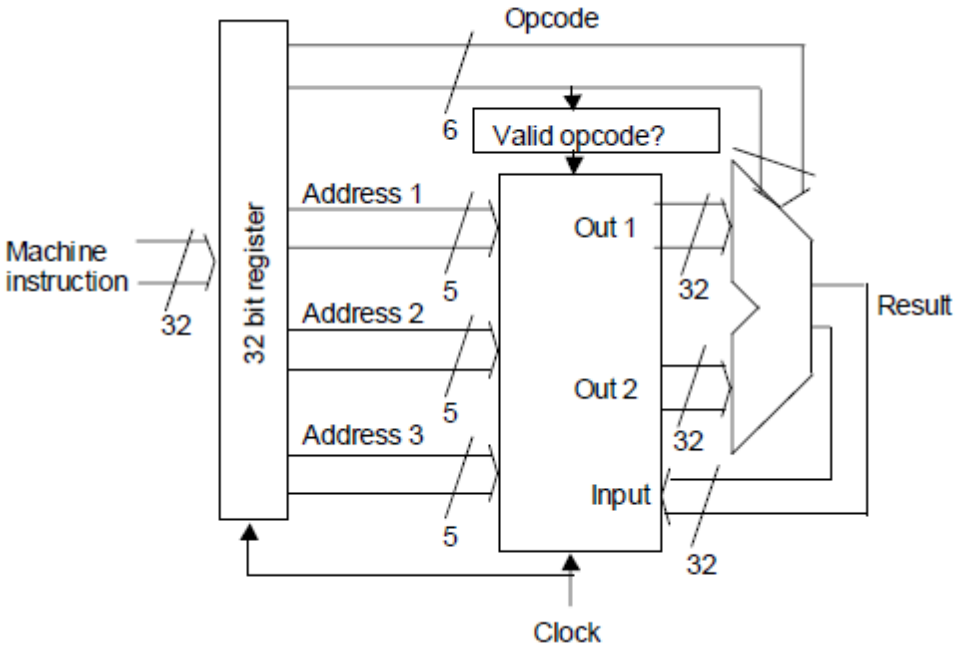
Enabling the register file

The register file that you created is always sensitive to its inputs, even when the inputs have garbage values. This can cause problems because when the simulation initializes (which corresponds to the real hardware being switched on) all the values of the logic signals initializes to some random garbage value (denoted 'U' in VHDL, but in real life either a '1' or a '0' chosen at random).

Give the register file an enable input. When the enable input=1 the register file will operate normally. Otherwise the register file will ignore its inputs, and will not update its outputs.

Creating the core of the microprocessor

Now create a testbench that contains an instance of the register file and an instance of the ALU connected like this:



Machine instructions are supplied to this arrangement in the form of 32-bit numbers. The format of these instructions is as follows:
• The first 6 bits identify the opcode
• The next 5 bits identify first source register
• The next 5 bits identify second source register
• The next 5 bits identify destination register
• The final 11 bits are unused

So, for example, if you want to add the contents of register 1 and register 2 and put the result into register 3, then the machine instruction would be as follows:
• The first 6 bits supply the opcode for the *add* instruction
• The next 5 bits would address register 1, and the next 5 would address register 2
• The next 5 bits address register 3.
• The remaining bits are unused, and should be set to zero.

The *enable* signal to the register should go high when the *opcode* contains a valid value, and should be low otherwise.

Test out your design by supplying machine instructions to it, and check that the operation performed is correct. Make sure you understand the timing of instructions, and in particular the relationship between the clock cycle on which the instruction occurs, and the clock cycle on which the appropriate result is written to register.

Running a program

Now create a stream of machine instructions that will act as a small program. The program must act as follows:

If the third-form-last digit of your student ID is 1, 4 or 7:
You should write a machine code to find the maximum number stored in addresses from1 to 30. The values of registers 1-30 should not be altered by your program, but the value of register 0 may be over-written if you wish. Verify that your result is correct automatically, also introduce error to show that you will be reported about.

If the third-form-last digit of your student ID is 2, 5 or 8:
You should write a machine code to find the minimum number stored in addresses from1 to 30. The values of registers 1-30 should not be altered by your program, but the value of register 0 may be over-written if you wish. Verify that your result is correct automatically, also introduce error to show that you will be reported about.

If the third-form-last digit of your student ID is 0, 3, 6 or 9:
You should write a machine code to find the sum of the numbers stored in addresses from1 to 30. The values of registers 1-30 should not be altered by your program, but the value of register 0 may be over-written if you wish. Verify that your result is correct automatically, also introduce error to show that you will be reported about.

You should try to make your program as efficient as possible.

Synchronising the data

All parts of the design that are synchronized to the clock should use *only the rising edge* of the clock. If you use both the rising edge and the falling edge, then it's much easier to do the design in simulation, but the real life hardware would be very expensive and complicated to manufacture. You will therefore lose a substantial number of marks if you use both edges of the clock.

No ops
You may find it useful to create a no-op instruction, i.e. an instruction that instructs the microprocessor to "do nothing" for the next clock cycle, and allocate one of your un-used opcodes to this instruction.