The overall goal is to implement the Mic-1 processor described in Tanenbaum’s book. We shall progress in steps towards the goal, each step being the result of one week’s assignment for work in the databar. The overall plan is as follows

1. Implementation of low-level components
   You will have to build and test Gezel-models of a tiny ALU, a register, and a multiplexer (-bus) and combine them to a model of a small datapath.

2. The Mic-1 datapath
   This is merely a quantitative extension of the first step: more registers and a more complex ALU (including status bits).

3. Combination with a model of the Mic-1 microcontroller unit
   Depending on experiences you might be asked to build the microcontroller unit or to modify a provided one. The controller will have to be combined and tested with the Mic-1 datapath and a component that corresponds to an initialised microinstruction memory.

4. The final step is to synthesize the model and configure an FPGA
   This step depends on our success with testing the tools for synthesis and configuration. We are optimistic about it, but need to express reservations that may require us to give up the final step and just ask for a report on the previous steps.

You must in the end document your work on all steps. The report will be evaluated and count towards a final grade for the course. Details about this will be provided later.

1 Step 1: Low-level components

Mic-1 contains several components, some being so similar that it can be hard to keep them apart at first. This step introduce a pre-Mic-1 datapath with a very simple and regular structure which is close to that of Mic-1.
1.1 Task 1: Gezel components

The first task towards building the Mic-1 microcontroller is to get you familiar with the Gezel language and simulation tool. For this, you should carefully read sections 1, 2, 3 and 4 in the Gezel Language Reference Manual 1.7.

1. Combinatorial adder

Implement the adder from last week’s lecture, including the test component and the system. In Gezel the complete design has to be described in a single file, using the extension .fdl, e.g. adder.fdl. When you have typed in the code, use the following command in the E-databar to simulate it,

    prompt> ~g02130/bin/fdlsim adder.fdl 10

fdlsim runs the simulator and 10 indicates that in this simulation we want it to run for 10 cycles.

2. Multiplexer

When designing a datapath, we quite often have the situation where we want the input of a component to come from the output of one of two or more other components. In order to do so, we need a component which is able to route one of several inputs to a single output, i.e. a multiplexer. In Tanenbaum’s book, subsection 3.2.2 on page 147, you can find the description of a multiplexer expressed in terms of logic gates and bit-wires. In Gezel we may express such functions using logic operations on bit-vectors.

Device an algorithm which is able to select one of four inputs, all represented as 32-bit two’s complement. Implement and test the multiplexer in Gezel.

3. ALU version 1

Most computers contain a single component which is able to perform a number of logic and arithmetic functions, called an Arithmetic Logic Unit or ALU. Extend the simple adder such that it is able to provide the following arithmetic functions,

<table>
<thead>
<tr>
<th>select</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x+y</td>
</tr>
<tr>
<td>01</td>
<td>x-y</td>
</tr>
<tr>
<td>10</td>
<td>x+1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Input and output should all be represented as 32-bit two’s complement.

4. ALU version 2

The controller of an FSMD makes, among others, its decisions based on status signals from the datapath. Quite often, these signals are obtained from the ALU based on the result of a computation.
Extend the ALU to provide two 1-bit status signals; \( N \) which is set to the value of the most significant bit of the result of a computation, and \( Z \) which is set to ‘1’ if the result is zero.

5. **ALU version 3**

Extend the ALU to support the full set of functions as presented in fig.4-2, page 234 in Tanenbaum’s book.

*Hint, combine the 6 1-bit input signals, which controls the function which will be performed, into a single non-signed integer of 6-bits. The sequence of bits in this integer should correspond to the sequence in the figure, i.e. \( F_0, F_1, ENA, ENB, INV A, INC \).*

6. **Register**

In many circuits it is necessary to sample the value of a signal at a specific point in time. A register is such a component. Every time the clock signals makes a transition from ‘0’ to ‘1’ (called the rising edge of the clock), the register samples its input and stores it. This is the way that Gezel handles variables (\texttt{reg}) within a datapath. Often we want to be able to control whether or not the input should be sampled and stored on a particular clock edge.

Build a component (register) which has a single 32-bit input and a single 32-bit output and a 1-bit load signal which tells the register to keep its current value (load = ‘0’) or sample its input (load = ‘1’);

*Note, as the word ”register” is a key word in VHDL and hence, reserved, you should use a different name, e.g. registerA.*

### 1.2 Task 2: Gezel datapath

The next task towards building the Mic-1 microcontroller is to get you familiar with writing FSMD’s using the Gezel language and simulation tool. For this, you should carefully read the rest of the Gezel Language Reference Manual 1.7.

1. **Hierarchy and module instantiation**

It is important that you have a clear understanding on how a datapath may be composed out of other datapath modules, i.e. how to create hierarchy and instantiate and use modules in Gezel. Figure 1 may help you. Write the Gezel code which uses the basic components developed in task 1, to implement the datapath as outlined in figure 1. Note that its structure fits Tanenbaum’s Figure 4-1 (page 233).

2. **Interface**

The datapath is controlled by an input command of 9-bits,

<table>
<thead>
<tr>
<th>bit</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-3</td>
<td>controls the ALU (fig.4-2, 206)</td>
</tr>
<tr>
<td>2</td>
<td>load on regA</td>
</tr>
<tr>
<td>1</td>
<td>load on regB</td>
</tr>
<tr>
<td>0</td>
<td>mux select 1=left, 0=right</td>
</tr>
</tbody>
</table>
3. Greatest Common Divisor (GCD)

In order to functionally test the datapath, we will implement and run a simple GCD program using the datapath. A C-like version of the program may look like the following,

```c
int gcd( int x, y) {
    while (x!=y) {
        if (x<y) y = y-x;
        else    x = x-y;
    }
    return x;
}
```

Use this program and the code for the datapath and its modules (including your drawing of the datapath) to devise a set of operations which are necessary to compute GCD. Note, you will only have two input ports; one for 32-bit data and one for 9-bit control of the datapath. Assume that the input values of GCD are always integers greater than 0.

4. Implement and functionally test the GCD algorithm

The following Gezel code implements an FSMD. The `test_datapath` is controlled by a finite state machine (FSM) which implements the GCD algorithm. Try to understand the code and then connect it to your datapath and run the simulation.

```gezel
Especially you should make sure that your implementation conforms to the assumptions expressed by the use datapath(...);
construct (line 7).
```
dp test_datapath {
  sig s_z, s_x : tc(32);
  sig s_command : ns(9);
  sig s_status : ns(2);
  reg r_status : ns(2);
  reg r_result : tc(32);

  use datapath(s_x,s_z,s_command,s_status);

  // utility actions
  sfg rep {
    r_status = s_status;
    $display($cycle, ": command=",$bin,s_command, 
    "(din=",$dec,s_x," dout=",$dec,s_z, 
    " (N,Z)=","r_status[1],","r_status[0], 
    ")=","r_status,"));
  }
  sfg finish {
    $display($cycle, ": end, x=",$dec,s_z);
    $finish;
  }
  sfg idle {s_command = 0b010100001; s_x = 0;}

  // The following are the actions necessary to compute GCD
  // read x ('A) and y ('B)
  sfg a0 {s_command=0b010100010; s_x=24; $display($sfg);} // B=24
  sfg a1 {s_command=0b010100100; s_x=15; $display($sfg);} // A=15
  // test B-A
  sfg a2 {s_command=0b111111001; s_x= 0; $display($sfg);} // B-A
  // B-A positive:
  sfg a3 {s_command=0b111111101; s_x= 0; $display($sfg);} // B=B-A
  sfg a4 {s_command=0b111111111; s_x= 0; $display($sfg);} // A=B-A
  sfg a5 {s_command=0b110111111; s_x= 0; $display($sfg);} // A=-A
}

fsm test(test_datapath) {
  initial s0;
  state s1, s2, s3, s4, s5;
  // The GCD program,
  //
  @s0 (rep,a0) -> s1;
  @s1 (rep,a1) -> s2;
  @s2 (rep,a2) -> s3;
  @s3 if (r_status[0])
    then (idle,finish) -> s0;
    else if (r_status[1]) then (rep,a4) -> s4;
    else (rep,a3) -> s2;
  @s4 (rep,a5) -> s2;
}