02131 Assignment 2: The ECG processor

Michael Reibel Boesen, Jan Madsen, Linas Kaminskas,

Karsten Juul Frederiksen, Simon Graverholt Søkilde

September 30, 2015

1 Introduction

At Medembed you're now given the task of implementing a small embedded processor, which can execute the QRS algorithm from A1. However, in order to achieve proof-of-concept for the processor you will start by implementing only one of the filters in the processor in order to demonstrate the performance (in terms of area, speed and power) of the processor you build. To build the processor you will use the Hardware Description Language (HDL) called Gezel.

The task of designing and implementing the processor will be divided into the following parts:

- **A2.1:** Building the processor modules.
- **A2.2:** Designing the instruction set
- **A2.3:** Implementing the controller
- **A2.4:** System integration
- **A2.5:** Performance analysis

1.1 Deadline

By **November 10 at 23:59** you have to deliver an electronic version of the report and source files as an archive uploaded on CampusNet at the location:

`CampusNet/Assignments/Assignment2`.

1.2 Notes & Hints

Keep the file `Filesharing/Literature/GezelBasicSyntax.pdf` and `Filesharing/Literature/Gezel.pdf` close at hand.

1.3 Expectations

The recent years we have experienced that the students have had a hard time of determining whether they are behind or on track, time-wise. Table 1 should guide you into checking that you are right on track. You will have 5 TA supervised labs to complete this exercise. Please note that we **also** expect that you work on this as part of your preparation for the course! You will probably not be able to finish the assignment if you only use labs to work on it. A2 is the hardest of the three assignments you will do - it should not be taken lightly.

Date	**Complete**
30/09	A2.1
07/10	A2.2, A2.3
21/10	A2.3
28/10	A2.3, A2.4
04/11	A2.4, A2.5

Table 1: Table of expectations

2 Project overview

As we talked about in lecture 4, a processor contains two main parts: Control and Datapath. The datapath typically consists of an ALU, register file and some multiplexers. The control consists of an instruction memory from which to fetch the instructions and a program counter (PC) to pick the next instruction. Figure 1 shows this processor schematic [1]. In order to define the behavior of the controller the designer first needs to define the processors instruction set. The processor's instruction set defines what the processor is capable of doing and it is very dependent on:

1. The application to be implemented
2. Available processor modules and their configuration.

Item 2 can be an issue since the available processor modules also depends on the instruction set. Since the processor we're designing is very application specific one way to define the instruction set is simply to translate the filter implementation you have into suitable instructions as was done during the lectures. Once the instruction set is defined the controller can be implemented.

Finally, once the controller is implemented the processor can be integrated with the rest of the embedded system. In our case the entire embedded system can be seen in Figure 2. The data memory (i.e. where the processor is picking up the filter data from) is external to the processor. In order for the processor to fetch data from the data memory the processor needs to adhere to the communication protocol implemented by the bus.

[1] Note that multiplexers are not shown as it depends on the design of the processor and the instruction set.

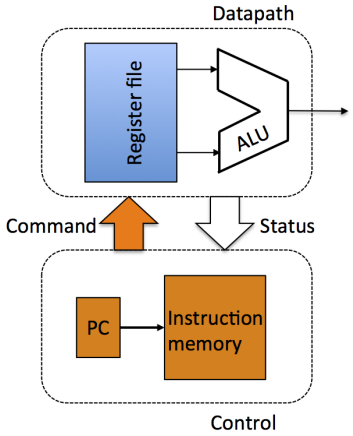


Figure 1: Processor schematic.

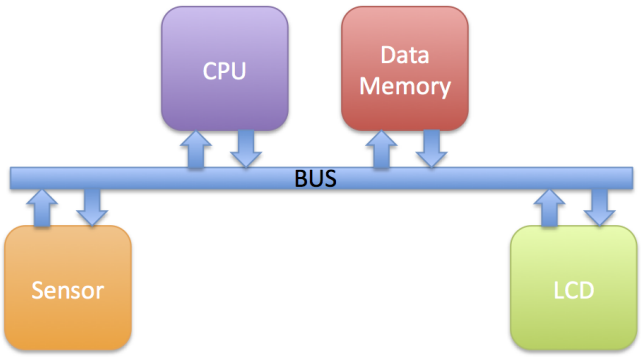


Figure 2: Our ECG scanner embedded system

Finally, once we have the complete system we can analyze the system with respect to performance.

3 Running and Debugging Gezel

This section contains some hints about how to run and debug in Gezel. We also looked at this during lecture 5. Feel free to ignore this now and jump straight to section 4.

3.1 Running Gezel code

In order to run your Gezel code located in a file named `code.fdl` type in a terminal:

`fdlsim code.fdl` *cycles*

where *cycles* is an integer that indicates how many clock cycles you would like to simulate. An example could be `fdlsim code.fdl 10`, this would execute the Gezel code in a file named `code.fdl` for 10 clock cycles.

3.2 Debugging

In order to debug your Gezel code you have two options:

1. Use `$display` statements.
2. Use `$trace` statements and GTKwave to display the waveforms.

3.3 `$display` statement

You can insert a `$display` statement in your code (inside `sfg` or `always` blocks) whenever you want to check the value of a variable at that particular point. This corresponds to outputting something using e.g. `System.out.println` in Java. The syntax is `$display(format, string, var, string, var, ...,)`. `format` can be `$bin`, `$dec`, `$hex` for binary, decimal and hexadecimal representation, respectively. `string` is a string and `var` is the name of the register or signal you want to display. Finally, special directives can be inserted as the `$var`. For instance, writing `$display($dec, "Cycle: ", $cycle)` would output the cycle count at this point. For more information see http://rijndael.ece.vt.edu/gezel2/tools.html.

3.4 Waveform

`$display` statements has a big drawback by the fact that it can be difficult to distinguish at what cycle the individual variables has what value. This is due to the inherent parallelism in hardware design. Therefore a better way to debug your Gezel code is to use waveforms. In order to see the waveform of a signal you first need to trace it. This is done by inserting a `$trace` directive anywhere in your datapath where the signal is that you want to trace. The `$trace` directive have the following syntax: `$trace(var, "filename.txt")`. Where `var` is the name of the register or signal you want to trace and `"filename.txt"` is a string containing the path of the file, which you want to store the trace. If you want to trace more than one signal you simply insert another `$trace` directive. Once you simulate your hardware design using the `fdlsim` command a trace directive will store the values of the signal/register at each cycle in a file.

In order to view the waveform you would also need to include the directive `$option "vcd"` at the top of your Gezel file. This tells the `fdlsim` to create a `TRACE.vcd` file, which is a file that contains the waveform. Opening this file with the command `gtkwave TRACE.vcd &` will open the waveform. Figure 3 shows a list of the signals you have traced using the `$trace` directive. You can drag and drop those you want to view to the signal area and then you can view the signal waveform in the signal area. In this case we are viewing a signal named `PC` in the datapath named `instFetch`.

For the software students this might be very weird and difficult at first, but using the waveform will give you a much clearer overview over what is happening in your system, because you can see your signals/register change with each clock cycle.

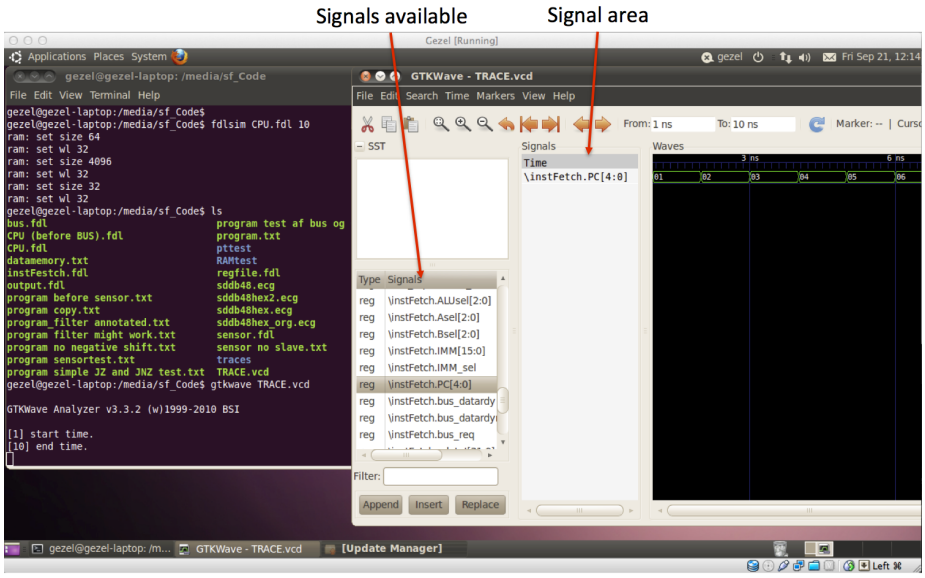


Figure 3: Tracing a signal using GTKwave

4 A2.1: Building the processor modules

To start building the processor and in order to familiarize yourself with Gezel you will start off by implementing the following small processor modules.

4.1 Program counter

The program counter (PC) is a simple counter that will count from 0 to xx depending on the size of your program. In order to simplify your implementation let's say the PC is supposed to count from 0 to 31 (you can always extend it later on if you need to). Figure 4 shows the block diagram for the PC. When writing your code observe that PC *always* does the same calculation in each clock cycle. Once you've built it verify that it is working using `$display` and Waveforms using GTKwave.

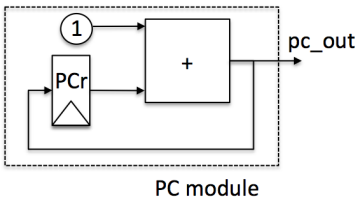


Figure 4: Block diagram of PC.

The code below should get you started:

```
1 dp PC(out pc_out : ns(5))
```

```
2 {
3    always{
4       $display($dec, "Cycle: ", $cycle); // note : For debuging purposes.
5       $display($dec, "pc_out:", pc_out); // note : For debugging purposes
      .
6       // todo : Insert code here!
7    }
8 }
9
10 system PCsystem
11 {
12    PC(pc_out);
13 }
```

In order to run it, see subsection 3.1. When implementing the block diagram on Figure 4 pay attention to when the addition happens and where the output of the addition is led.

4.2 Adder

An adder is a simple component that, not surprisingly, performs an addition of two values as seen in Figure 5.

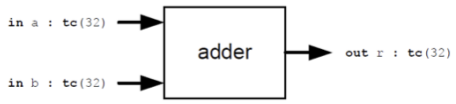


Figure 5: Adder schematic.

Implement and test such a component where the input and the outputs are of type 32-bit two's complement.

4.2.1 Testbench

In order to test your component it might be helpful to write a testbench. A testbench is a simple Gezel component that provides the inputs to the components under test and picks up the results. For the adder a testbench schematic can be seen in Figure 6.

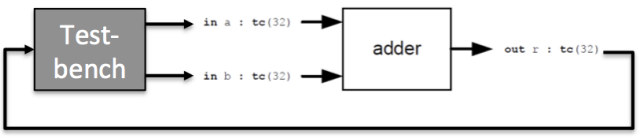


Figure 6: Adder and it's testbench schematic.

A test bench consists of a number of signal flow graphs that each cycle inputs new data to the component under test. You may use this code sample as a starting point for your testbench. You might even enhance this code to also

test that the result coming back are correct by testing `R` against a value, which you know is the right one.

```
dp adder(...)
{
  // todo : Insert code here!
}

dp testAdder(out A, B:tc(32);
             in R:tc(32))
{
  always{
    $display($dec, "Cycle:", $cycle, ", A=", A , ", B=", B , ", R=", R);
  }

  sfg test_0 { A=3;  B=6; }
  sfg test_1 { A=23; B=17; }
  sfg test_2 { A=12; B=0; }
}

// note : State machine to control the adder testbench.
fsm f_testbench(testAdder)
{
  initial s0;          // begin with state s0
  state s1, s2;        // other states are: s1, s2
  @s0 (test_0) -> s1; // run test_0 and go to s1
  @s1 (test_1) -> s2; // run test_1 and go to s2
  @s2 (test_2) -> s0; // run test_2 and go to s0
}

system myFirstSystem
{
  adder(A, B, R);
  testAdder(A, B, R);
}
```

4.3 Multiplexer

When we design a datapath, we quite often have the situation where we want the input of a component to come from the output of 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 Gezel we may express such functions using the selection operator `?`.

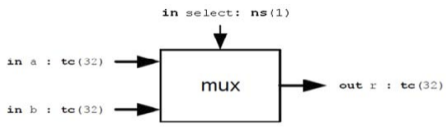


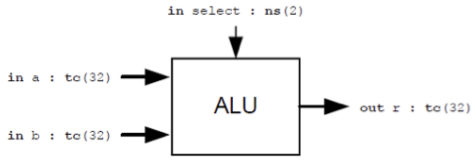
Figure 7: Multiplexer schematic.

Figure 7 shows the schematic of a multiplexer. Depending on the value of

the input signal `select`, the value of the output `r` is either equal to the value of the input: `a` or `b`. You should first develop an algorithm which is able to select one of two inputs, both represented as 32-bit two's complement. Afterwards, implement and test the unit in Gezel.

4.4 Arithmetic Logic Unit

Most computers contain a single component which is able to perform a number of arithmetic and logic functions, called an Arithmetic and Logic Unit (ALU), shown in Figure 9.



Select	Function
00	r = a - b
01	r = b - a
10	r = a
11	r = b

Figure 8: ALU schematic and function.

This simple ALU extends the adder from subsection 4.2 with a 2-bit unsigned input `select` that determines the function applied to the two inputs according to the table above. Input and output should all be represented as 32-bit two's complement. Implement and simulate this component.

4.4.1 ALU extension: Status flags

After you get the basic ALU to work you should extend the ALU with status flags. The controller of an FSMD makes decisions based, among others, on status signals from the datapath. Quite often these signals are obtained from the ALU based on the result of a computation.

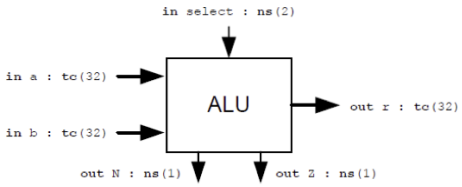


Figure 9: The ALU extended with status flags.

You should extend the ALU to provide two 1-bit status flags: negative `N`, which is set to the value of the most significant bit of the result of a computation (i.e. whether the signal is negative or not); and zero `Z` which is set to `1` if the result is zero.

4.5 Register file

Most processors have a set of registers which they use to store data and retrieve data from which it is current operating on. This allows the processor to save

time when it is doing the calculations, because it would otherwise need to fetch the data from the external memory, which will take many more clock cycles when compared to reading the value from a register.

Figure 10 shows a schematic of the register file. It should have the following functionality:

- `asel` selects between 8 different registers and outputs the contents of the selected register to `a`.
- `bsel` selects between the same 8 registers as `asel` does and outputs the contents of the selected register to `b`.
- If `storeenable` is set to `1`, store `storedata` at the register indicated by `storesel`.
- The register addressed with the `asel` / `bsel` `000` should always return the value `0`.

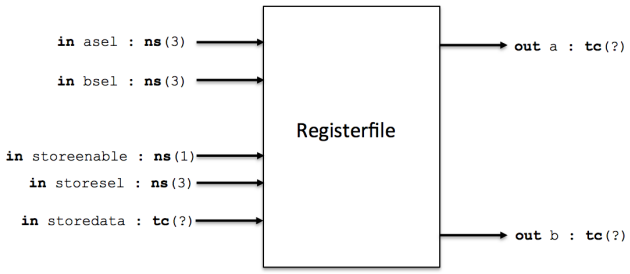


Figure 10: Register file schematic.

Implement and test this component. Note, that the only constraint we put on the implementation of this component is that you are only allowed to use 8 registers, where one of the registers always should be 0.

4.6 Instruction memory

The final component (except for the controller) we will need is an instruction memory. Gezel comes with a set of IP blocks that can be used to instantiate various predefined components. One of these IP blocks is a RAM block. Now we will create a component that can read data from a specified address, which will be used as our instruction memory. The instruction memory can be instantiated with the following template:

```
1  ipblock instmem(in address : ns(5);
2                  in wr,rd   : ns(1);
3                  in idata   : ns(32);
4                  out odata  : ns(32))
5  {
6      iptype "ram";
7      ipparm "size=64";
8      ipparm "wl=32";
9      ipparm "file=program.txt";
10 }
```

This creates an IP block with the name `instmem` and the ports as seen above. The `address` is used to indicate what address to interact with in the IP block. If `wr` is set to `1`, then write `idata` to the `address`. If `rd` is set to `1`, then read out the value stored at the `address` to the `odata` port. (You might think about what should happen if `wr` and `rd` are both set to `0` or what should happen if both are set to `1`). Furthermore, the IP block have a number of parameters `size`, `wl`, `file`. The `size` indicates the number of entries in the RAM block. `wl` indicates the word length (i.e. the size of each entry in bits). The `file` parameter indicates a file path, which will be used to initialize the contents of the RAM block. The file need to contain two columns, where the first column is the address and the second column is the data. In this example the data values are in hexadecimal format and the word length is 32 bits.

```
0 20200000
1 20400000
2 20600E38
3 80A30801
4 1000001D
5 20800020
```

You must now implement a component that can read a specified address from the memory IP block and output that. The figure below shows a schematic. Note, that once you have defined your instruction set you might want to revisit your specification of your memory IP block in order to optimize the configuration to your design. Figure 11 shows the schematic of this block. To no

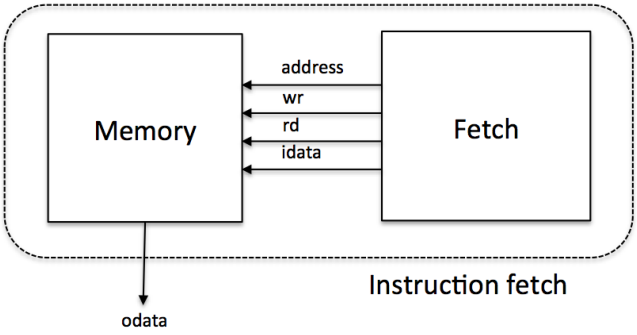


Figure 11: Instruction fetch schematic.

surprise, the reason why we built the PC component in subsection 4.1 was that it should be used to pick the next instruction to be executed. After you can read the data from a specific address you should integrate the PC component into this component, such that the PC is used to select the next instruction to be executed.

5 A2.2: Designing the instruction set

The filter you will be implementing is the Moving Window Integration. In this part of the project you should take your code and manually compile it into

assembler instructions. Lecture 4 already covered various ways of compiling program code to assembler instructions. You are free to design any instructions you may like but remember that you need to implement them later as well. It might be a good idea to look into Chapter 3.1 to 3.4 in Frank Vahids book.

A few hints:

- Input data can be picked up from an external memory.
- Our implementation had a program of 29 instructions with an instruction set size of 9. However, this might depend very much on your software implementation of the filter!

Once you have compiled your filter into assembler instructions you can extract the different instructions from that - and voila - you have your instruction set.

6 A2.3: Implementing the controller

Now that you have the instruction set the first you need to do is to connect your components (from A2.1) according to your instruction set. Note that depending on your instruction set you might need 0 or more multiplexers. In order to this you will need a block diagram in order to keep an overview of how your components should be connected. **Please show your block diagram to the TAs before continuing with the implementation - it might save you hours of trouble!**

Once you have connected all components you should be able to distinguish control signals and data signals. Control signals are the signals, which configure a particular component, such as for instance the `select` signal of the ALU in subsection 4.4. These are the signals your controller need to define. However, depending on your instruction set you might need to add more components. For instance if you need to implement a Jump instruction you might need another multiplexer in your PC component such that it might end up looking like Figure 12. It will then be the job of the controller to set the correct `sel` signal depending on whether a Jump instruction is the current instruction or not. A

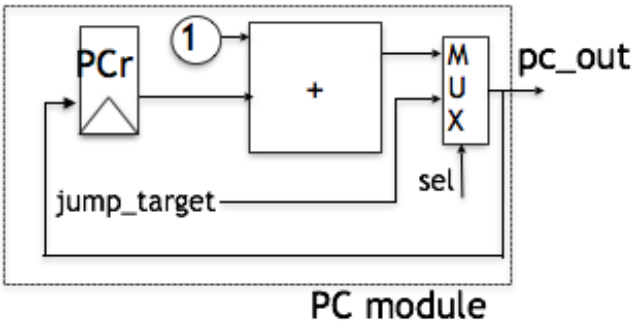


Figure 12: PC with multiplexer to handle a jump.

few hints:

- It might be beneficial to have an `ADDI` instruction (ADD Immediate), which basically adds a constant to a specified register and saves the result in another register (it is an Immediate type instruction as explained in Lecture 4). This instruction can be used to initialize the contents of a register, or move the contents of a register to other registers.
- Start off by implementing one instruction at a time and test that it does as you expect. Once you've implemented the second instruction, make sure to also test the first instruction to make sure that you didn't screw this one up. Continue like that.
- For now, ignore any instructions that uses the external data memory. Implement this *after* you have connected your processor to the embedded systems platform (see the next section).

7 A2.4: System Integration

Now that you have most of your instruction set working you should start connecting your processor to the rest of the embedded system. Among other things you will need access to the external data memory in order to pick up the data for the filter. In this part of the assignment you will implement the instructions necessary to handle that.

First, you must connect your processor to the bus. The bus and the components connected to it can be found on CampusNet at `FileSharing/Labs/Assignment2/Platform.fdl`[2]. The bus uses a master/slave protocol, where your processor is the master. This means that the processor should be connected to a master bus interface, which handles the communication protocol. The master bus interface is seen in Figure 13. The processor can communicate

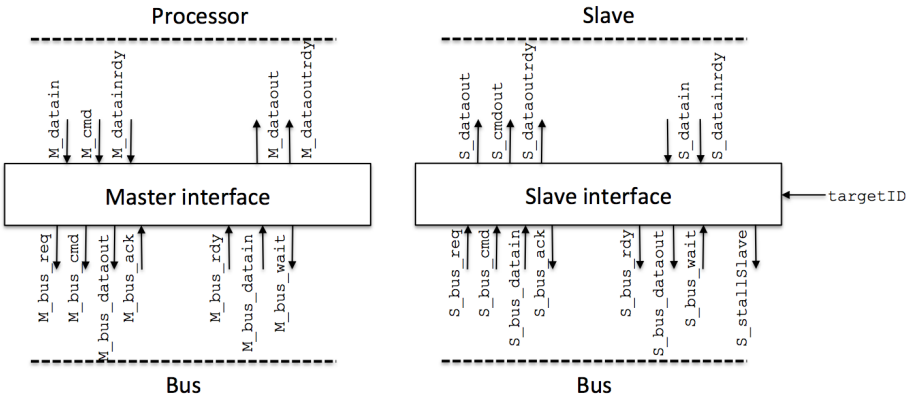


Figure 13: Master bus interface

two different types of data to the units on the bus: `cmd` and `data`. `cmd` identifies the command that the processor wants the slave to execute and `data` contains

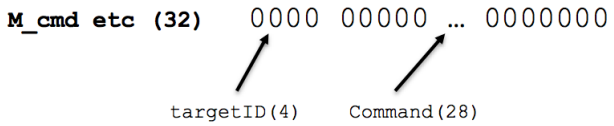
[2]Remember to also download `datamemory.txt` and `sddb48hex.ecg` as well as create folder named `traces` at the location where you are using the `Platform.fdl`.

the data related to the command. Note that sometimes a command might not need any data.

In order to transfer data from the processor to the bus the following happens:

1. The processor puts the relevant data in `M_datain` and relevant command at `M_cmd`.
2. The processor sets `M_datainrdy` to '1'.
3. The master interface detects the assertion of `M_datainrdy` and transmits `M_cmd` to `M_bus_cmd` and `M_datain` to `M_bus_dataout` as well as the setting `M_bus_req` to '1'.
4. Based on the `S_bus_cmd` (which for all slave interfaces is connected to `M_bus_cmd`) a particular slave interface reacts on this request by setting `S_bus_ack` to '1' and sends the command to the slave unit. The `M_cmd` has the form as seen in Figure 14. The figure shows that the first 4 bits of the `M_cmd` is used to identify the slave, which should respond to this data. This is defined by the `targetID` signal, which is defined when the slave bus interface is initialized and will never change value. The figure also shows the `targetID` value for each of the 3 slaves in the system.
5. The master interface detects that `S_bus_ack` is '1' and sets `M_bus_req` to '0'.
6. The slave interface detects this and sets `S_bus_ack` to '0' - hence the data transmission is complete.

Figure 15 shows how it looks like when looking at the waveform. The signals beginning with `bus.xx` are the lower left signals in the master interface from figure Figure 13. The signals beginning with `bus_toplevel.DM_xx` are the top left signals of the slave interface. Observe that the signal `ackCPU` is actually being set by the `S_bus_ack` signal.



targetID	Unit
1	Sensor
2	Data Memory **Command 0x8000000** = Write (MSB indicate write, rest indicate ADDR) **Command 0x0000000** = Read (M_datain should contain ADDR)
3	LCD

Figure 14: Command signal format

Our Master-Slave bus protocol always expects that whenever the master sends data to the slave, the slave will always respond with data. This means

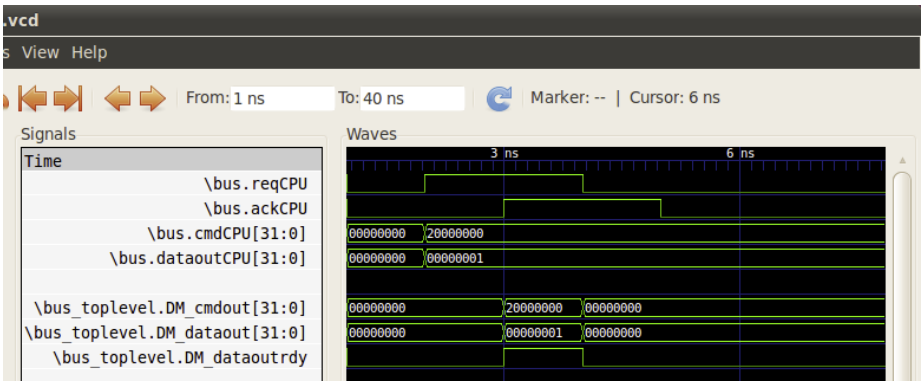


Figure 15: Master data transmission example

that your processor needs to support that it is waiting for the data to get back. For instance if you execute a `Load` instruction which should fetch data from the external data memory, the processor would need some way of waiting for data to get back. This is exactly what the `M_dataout` and `M_dataoutrdy` is used for.

Once the slave has received the data from the master (i.e. after the protocol above is completed) the slave will analyze the command and figure out what to do. In fact only the Data Memory is using the command field (for now - more about that in A3). In the case of the Data Memory, it is checking whether the first bit (the Most Significant Bit (MSB)) of the command is 1. If this is the case it interprets this as a write command and it uses the rest of the command bits as address and the `S_dataout` as data to store at that address. If the MSB is 0 it interprets the command as a read and uses `S_dataout` as the address. Once the slave has data ready for the master it will execute the following protocol:

1. The slave will put the data it is returning to the master at the `S_datain` signal.
2. The slave unit sets `S_datainrdy` to '1'.
3. This causes the slave interface to wire the data to `S_bus_dataout` and sets the `S_bus_rdy` signal to '1'.
4. The master interface sees this and sets `M_bus_wait` to '1' and takes the data from `S_bus_dataout` and wires it to the `M_dataout` signal and sets the `M_dataoutrdy` to '1', which signals to the processor that data is ready and that it can continue execution.
5. The slave sees that the master has set its `M_bus_wait` signal to '1' and acknowledges this by setting `S_bus_rdy` signal to '0'. At the same time the master interface sets its `M_bus_wait` signal to '0'. Hereby the transmission is complete.

Figure 16 shows the complete protocol. The bottom 5 signals are related to the slave data transmission. The 3 signals beginning with `bus.xx` are the bottom right signals of the master interface (and consequently also the bottom

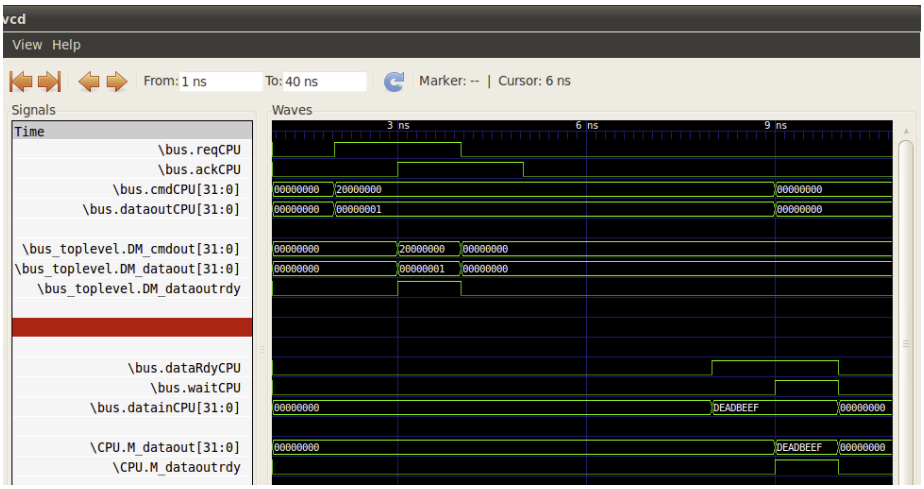


Figure 16: Slave data transmission example

right signals of the slave interface, except the `S_stallSlave`) and the 2 signals starting with `CPU.M_xx` are the top right signals of the master interface.

Please note that Gezel is simulating very slowly. It is not feasible to simulate the full data set from Assignment 1. Therefore, we have put a shorter version of the dataset. You can find this on CampusNet at `FileSharing/Labs/Assignment2`.

8 A2.5: Performance analysis

Finally, analyze the performance of your ECG scanner in terms of speed, area and power and compare it to your analysis in Assignment 1. Consider the following:

Analyzing speed:
Think about, how can you say something about the speed of your implementation. What can you measure? What can you use this for? And how can you compare it to your implementation in A1.

Analyzing area:
What can be used in the Gezel programming language as a measure for area?

Analyzing power:
Gezel can count the number of toggles (i.e. signals switching from '0' to '1') using:

- `$option "profile_toggle_alledge"`
- `$option "profile_toggle_upedge"`
- `$option "profile_display_operations"`
- `$option "profile_display_cycles"`

Read more about it[3] and figure it out for yourself how to do it. There will be said more about this during lecture 8.

[3] http://rijndael.ece.vt.edu/gezel2/tools.html