The **Microelectronics Training Center**

The MTC is an initiative within the INVOMEC division

Industrialization & Training in Microelectronics

Lab-exercise

Lab 4:

VHDL basics: Register file

Cluster: Cluster1
Module: Module1e

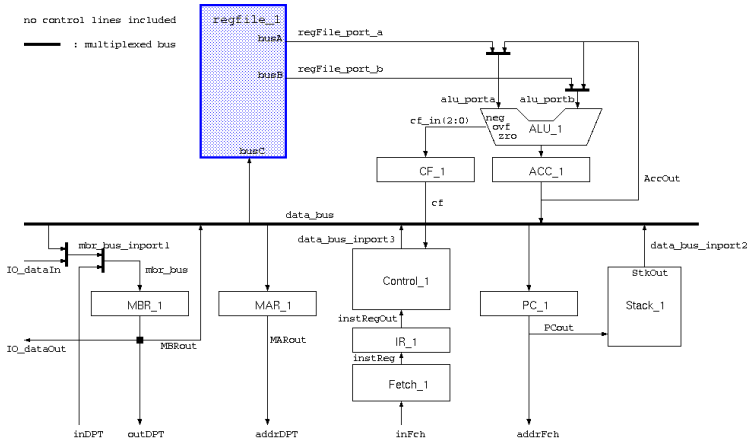
Target group: Students

Version: 1.1
Date: 21/03/06
Author: Osman Allam
Modified by: Geert Vanwijnsberghe
History : 1/12/06 : testbench added

This material was developed with support of the European Social Fund.
ESF: Prevent and combat unemployment by promoting employability, entrepreneurship, adaptability and equal opportunities between women and men, and by investment in people.
http://www.esf-agentschap.be

Introduction

Register files are common in most computer systems. The structure of the register file has significant impact on the performance of the microprocessor. The register file in Micro6 holds ALU operands and results in addition to providing support for register indexed and register stack addressing modes.



Objectives

After completing this module, you will be able to:

- Build relatively complex circuits;
- Understand different technique for structural modeling.

Knowledge background

- Basic knowledge of VHDL;
- Basic knowledge of HDL synthesis.

Classification

- Level: 2
- Duration: 45 minutes

Input

VHDL template

The lab

The register file is a structural model in which you will use some of the units you have developed in previous modules. The register file is composed of the following registers.

1. 28 General-Purpose registers
A 32-bit register each. These are used to hold data manipulated by the ALU.

2. 3 Index registers
A 9-bit register each. These are used for register-indexed addressing mode where the address of the data to be read or written is the sum of the contents of an index register and the contents of a general purpose register.

3. Stack pointer
A 9-bit updown-counter. The stack pointer controls the memory stack. This is a segment in the main memory. Data can be pushed and popped from the stack by the instructions PUSH and POP respectively.

Structural description

Describing a system with a structural model means expressing the system in terms of its building blocks. The building blocks themselves can be composed of sub-systems, and so on. The components of the bottom level of the hierarchy must be expressed only in terms of their behavior.

VHDL provides 2 ways of structural description:

1. Entity instantiation
Also known as **direct instantiation** because you directly specify which entity and which architecture to instantiate. You have to provide the following information for each instance:
    a. The library where the entity is compiled;
    b. The name of the entity;
    c. The name of the architecture you want to use.

```
<instance_label>: entity <library_name>.<entity_name>
(<architecture_name>)
  generic map (
    -- generic association
  )
  port map (
    -- port association
  );
```


2. Component instantiation
Before you can use this kind of instantiation, you have to declare components. You can do that either in the declarative part of the architecture or in a separate package (recommended).

a. Default binding

This is the simplest form of component instantiation. However, there are some restrictions on using it:

- The component name and interface must match those of the entity.
- The entity must be compiled in a visible library.

In this kind of binding, component instances are bound to entities of the same name as the components. In case of multiple architectures the last compiled one is associated with the instance.

```
<instance_label: <component_name>
  generic map (
    -- generic association
  )
  port map (
    -- port association
  );
```

b. Configuration specification

If you want to use a specific entity-architecture pair for a particular instance, then you may use configuration specification. You provide the same information as with direct binding but you write them in the declarative part of the architecture using the following syntax:

```
For <instance_name>:<component_name> use entity
<library_name>.<entity_name> (<architecture_name>)
  generic map (
    -- generic association
  )
  port map (
    -- port association
  );
```

c. Configuration declaration

Configuration declaration provides a means to defer specifying the binding of component instances. You can do that by declaring a configuration and associate it with your entity. You can have as many configurations for a given entity as you want. Every configuration is a separate design unit, which can be compiled and loaded by the HDL simulator.
Configurations are a very powerful mechanism for component instantiations in different levels of the hierarchy. Unfortunately, many synthesis tools do not support configurations but you are encouraged to use them for simulation.

The simplest syntax rules for a configuration declaration is shown below.

```
Configuration <identifier> of <h_entity> is
For <h_architecture1>
For <instance_name1>:<component_name1> use entity
<library_name>.<entity_name> (<architecture_name>);
```

```
End for; -- end of binding information.
.. -- binding information for other component instances
end for; -- end of h_architecture1
.. -- binding information for other architectures.
End configuration;
```

Where `h_entity` and `h_architecture1` are the entity and architecture of the higher level structural model respectively.

Generate statement

A generate statement provides a mechanism for iterative or conditional inclusion of a portion of code.

The portion of code must be concurrent statement(s) only. This means that generate statements may not be used inside processes, functions or procedures.

In this module, we will explain using generate statements to instantiate several instances of the same entity. Remember that a component instantiation is a concurrent statement.

```
<generate_label>: for x in <range> generate
  <instance_label>: <component_name> port map (
    -- port associations
    );
end generate;
```

`<generate_label>` and `<instance_label>` are mandatory. Every *generate* and component instantiation statement must be labeled.
x is the range parameter.

The above code makes use of **default binding.**

Exercise

Using the components you have designed in previous labs, construct a register file containing the following components:

- 28 General-purpose registers: 32 bits wide;
- 3 Index registers: 9 bits wide;
- 1 Stack pointer: 9 bits wide updown-counter.

The register file has one input port (busC). The target register for write is selected by the selection lines (selC).
The register file has 2 output ports (busA, busB). Selection lines (selA, selB) are used to select which register output is connected to each one of these ports respectively.
Writing into all 32 registers is done by setting the address of the register on selC and by asserting the wr input. The selected register will be updated at the next rising edge of the clock.
Incrementing or decrementing the Stackpointer is achieved by asserting the stkInc or stkDec input. Also this action will be performed at the rising edge of the clock.

For all components, use default component instantiation as shown above. Use *generate* statements for repetitive instantiation of the general-purpose registers.

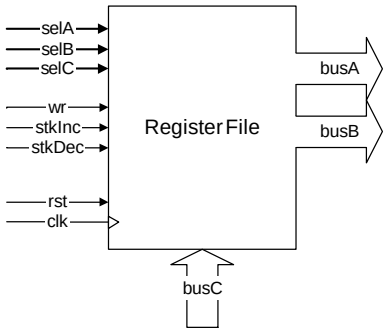


Figure 1: Symbol register file

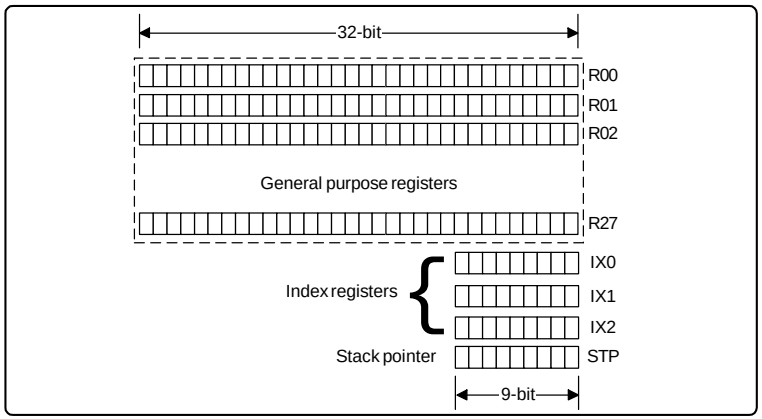


Figure 2: Registers

In the attached template (file: `cpu.vhd`), the code below makes arbitrary signal assignments. They have no impact on the operation of the register file, but they are useful in synthesis to avoid generating unnecessarily long registers.

```
reg_int (28)(31 downto 9) <= (others => '0');
reg_int (29)(31 downto 9) <= (others => '0');
reg_int (30)(31 downto 9) <= (others => '0');
reg_int (31)(31 downto 9) <= (others => '1');  -- see below
```

The Stackpointer contains an address value located in the Stack Segment area. In our micro6 microprocessor this area is fixed and equal to 0xE00 – 0xFFF. This means that the initial value of the Stackpointer has to be 0xE00. This is the reason why the most significant bits of the Stackpointer are fixed to '1'. In the above figure only the bits are show that can be modified synchronously by the stkInc and stkDec inputs.

Use the template provided in the file `cpu.vhd` to describe the complete register file. The tb_regFile.vhd should be used to verify the correct behaviour of the register file.

Have a close look at the testbench file since it shows some techniques to write testbenches in an efficient way.