Chapter 2

Lab 2 - Datapath

2.1 Overview

During lab 2 you will complete the RTL code of the ALU and MAC datapaths of the DSP core and write a set of small test programs to verify your implementation. Most of the datapath code has already been written and verified by us, but a few critical parts are either missing or implemented in a suboptimal manner. You can use either VHDL or Verilog in this lab.

Besides datapath implementation, the lab also covers module level verification. In order to verify your modules, you will need to implement and run assembly language programs that test your modules. The programs will be executed both by the `srsim` instruction set simulator and on the actual processor RTL code. The output from the instruction set simulator will be saved to a file which is used by the RTL simulator to check that the RTL code produces the same results.

Note that the test programs *cannot use jumps or any other kind of flow control instructions* since the control path in the processor is not finished yet (you will implement it in a later lab). For now, the control path will always increment PC after each instruction and the effect of jumps is undefined.

Section 2.2 gives a step-by-step list on how to do the implementation and testing.

Section 0.6 gives more details on the verification step.

Section 2.3 gives hints on how to write good test programs.

Section 2.4 gives a detailed specification of the modules.

2.2 Work Flow

2.2.1 Implementing and Verifying (Testing) the Modules

The basic procedure for each module is as follows:

1. Choose a module that you wish to implement. The modules are described in Section 2.4.
2. Open the file *module*`.v` or *module*`.vhd` (depending on your previous choice between VHDL and Verilog) and write your implementation. Note that the module input and output port declarations are already present and should not be changed. The files are located in the `lab2-3/rtl/` directory.
3. Implement an assembly language program to test your module. There are a few template files provided with the lab that you can use if you want. The test should be as thorough as possible to test for as many different errors as possible so that you are sure that your module works for "all possible" inputs.

The principle behind the verification is as follows. Perform a computation that uses the module you want to test (for example, if you want to test the rounding module you should use an instruction that performs rounding). Output the result to a file with an `out 0x11,...` instruction as shown in the template files. In this way the result from the rounding operation can be compared between the instruction set simulator and the RTL code simulator.

Hints on how to write good test programs are given in Section 2.3.

4. Compile the module and run the assembly language program using the RTL simulator as described in Section 0.6.
5. The RTL simulator will stop at the first mismatch between the instruction set simulator output and the RTL simulator output, or when the assembly program has ended. A mismatch usually means that the RTL code needs to be fixed.

2.2.2 File List

You must modify the following files:

`saturation.v` *or* `saturation.vhd`: The RTL code of the saturation module.

`saturation.asm`: Assembly language code to test the saturation module.

`mac_dp.v` *or* `mac_dp.vhd`: The RTL code of the datapath of the MAC unit.

`rounding_vector.asm`: Assembly language code to test the rounding code

`adder_ctrl.v` *or* `adder_ctrl.vhd`: The RTL code of the adder mux control module.

`min_max_ctrl.v` *or* `min_max_ctrl.vhd`: The RTL code of the MIN/MAX mux control module.

`alu_test.asm`: Assembly language code to test the adder mux and MIN/MAX control modules.

You should not need to modify any other existing files, but you can create more assembly language test programs if you need.

2.3 Module Verification

Module verification is to try to prove the correct function of a module versus the specification. This can be approached by comparing against a "known good" implementation like in this lab. The results from the RTL code can be compared to those of the instruction set simulator.

An important concept in module verification is that of corner cases. Basically, the corner cases are the set of inputs that trigger or very nearly trigger special cases in the implementation or inputs that are near the numerical limits. The purpose of testing the corner cases is to make sure that the special cases are triggered when they should and only then. Writing good test programs for module verification is very much about identifying corner cases.

For example, for a 16-bit absolute value computation the corner cases would be the very largest positive value `0x7fff`, the largest negative value `0x8000` (should trigger the special case saturation), the almost-largest negative value `0x8001` (should not trigger saturation). It can also be good idea to try zero and one positive and one negative "arbitrary" values such as `123` and `-123` to cover the general cases. An absolute function that gives the correct result in all these cases is likely to be correctly implemented.

Another important aspect of testing is that each test should give as much information as possible, or in the worst case at least give *some* information.

Tests should have as few false positives as possible.

For example, to test whether the ADD instruction really performs an addition it is not very useful to try 4+0 since this input will give identical results for ADD, SUB, or OR. A more suitable test case could be 4+6.

Test only relevant things.

This goes without saying really. Things that are not relevant to the module does not need to be tested at this point. For example, in this lab you will mostly implement control signals for muxes and that is what should be tested, it is not necessary to test if the adder can produce a correct carry-out signal or not.

Test every possible corner of the implementation.

Ideally all possible bugs should be tested for. This can be achieved by applying all possible input combinations to each possible internal state of the module. In practice this is usually impossible to do because it would take far too long time. Instead corner cases and random testing is used to speed up the testing while still giving very high test coverage.

Investigate code coverage to find missing corner cases.

Code coverage is a feature which is present in many simulation tools which allows you to determine whether all statements in a source file have been executed. If a certain line hasn't been executed this may indicate that you have missed a corner case in your test bench. (However, some statements shouldn't be executed, such as assertions or other error handling code.) Of course, merely executing all statements is not a guarantee that the module is correct, your test benches needs to take this into account and output data so that all functionality is tested.

In the Makefile which is used for lab 2 and 3, the coverage option is enabled for branches, conditions, statements, and signal toggling when Modelsim is started. After running a test case you can inspect the code coverage by looking at the coverage window in Modelsim.

Test things once.

Redundant tests will not give any extra information, they will not uncover any additional bugs that were not already found by earlier tests. In practice this is nearly impossible to satisfy. For example, even though corner cases are designed to have at least some non-overlapping test coverage, they also often have a large overlap on other parts. (However, it is of course better to include some tests that you think may be redundant than to forget about an important corner case.)

2.4 Description of The Modules

2.4.1 Modules `adder_ctrl` and `min_max_ctrl`

The first task is to implement two modules that generate control signals for three muxes in the arithmetic data path. In order to solve this task, you need to know how a single adder can be multiplexed to perform various functions such as addition, subtraction, absolute value, minimum, and maximum.

In the first part, you should implement the control signals for the adder input mux and the carry-in selection mux. In the second part, you should implement the mux control signal for the MIN and MAX function. Note that the MIN and MAX functions use the adder to compare the numbers.

For all operations, the inputs a and b should be treated as two's complement (signed) numbers.

Note that the calculation of absolute value should not perform saturation. This is performed by another circuit not shown here. Also not shown is that the operand b will automatically be set appropriately for those operations that require a constant.

All outputs not related to the currently computed function are don't-care signals. For example, the value at the MIN/MAX output is irrelevant when an ADD or SUB operation is requested.

A schematic is given in Figure 2.1. A list of the ALU functions is given in Table 2.1. A list of input and output signals for the adder input mux is given in Table 2.2, and a list of input and output signals for the MIN/MAX mux is given in Table 2.3.

You should also write assembler code to test the functionality of your ALU. At a minimum, you should aim for 100% statement coverage in `adder_ctrl` and `min_max_ctrl`, although some code (such as assertions) can be excluded from the coverage metrics. Depending on how you write your code you may also need to look at the other coverage types as well.

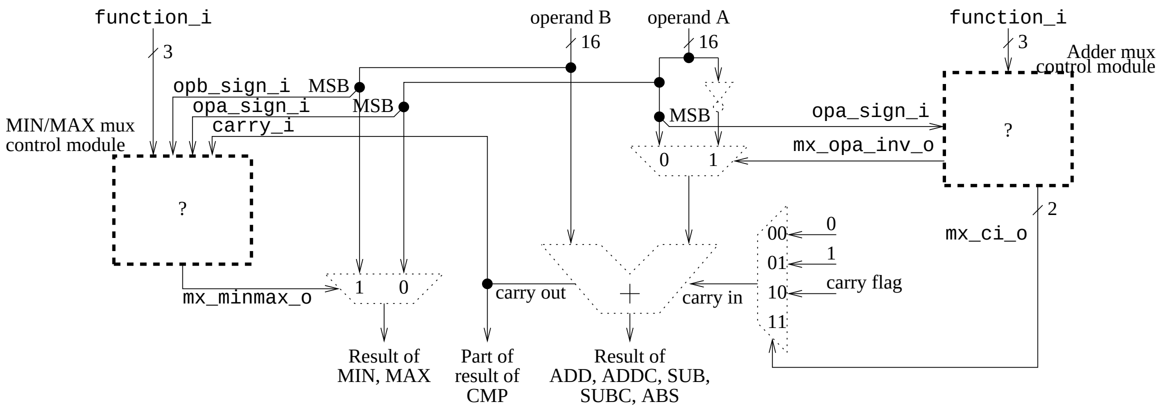

Figure 2.1: The Arithmetic Unit and MIN/MAX modules.

2.4.2 Module `saturation`

The second task is to create the saturation module in the multiply-and-accumulate unit, the MAC. To solve this task, you need to know about saturation and when and how it is performed.

The input to the saturation unit is a 40-bit value (8 guard bits and 32 data bits) and a "perform saturation" signal (1 bit) that selects between the two function of the saturation circuit: perform saturation (1) or pass the data through unmodified (0). The output is also a 40-bit value. When saturation is performed the nine most significant bits of the output will be equal (that is, nine "zeros" or nine "ones"). There is also an output that indicates whether saturation was performed or not. This output should be

Code	Mnemonic	Description	Operation
000	ADD	Add	$b + a$
001	ADDC	Add with carry	$b + a + C$
010	SUB	Subtract	$b - a$
011	SUBC	Subtract, add carry	$b - a - 1 + C$
100	ABS	Absolute value (no saturation)	$\|a\|$
101	CMP	Compare (subtract, set flags only)	$b - a$
110	MAX	Select maximum value (signed)	$max(b, a)$
111	MIN	Select minimum value (signed)	$min(b, a)$

Table 2.1: The functions that should be supported by the ALU.

Signal	Dir	Size (bits)
`function_i`	in	3
`opa_sign_i`	in	1
`mx_opa_inv_o`	out	1
`mx_ci_o`	out	2

Table 2.2: Input and output signals of the `adder_ctrl` module.

Signal	Dir	Size (bits)
`function_i`	in	3
`opa_sign_i`	in	1
`opb_sign_i`	in	1
`carry_i`	in	1
`mx_minmax_o`	out	1

Table 2.3: Input and output signals of the `min_max_ctrl` module.

one when saturation was actually performed and the data was modified by the module, it is zero when the data is unmodified. The output should be generated purely combinatorially, there should be no flip-flops in the module.

A schematic overview is given in Figure 2.2. A list of input and output signals is given in Table 2.4. Finally, you need to write assembler code to test your saturation module. At a minimum, your assembler code should provide suitable stimuli so that your saturation unit gets 100% statement coverage.

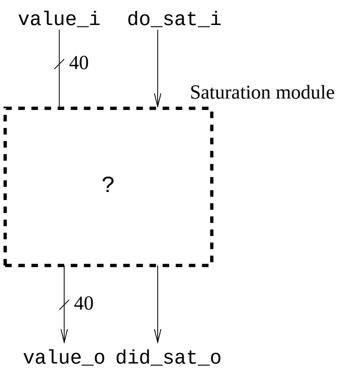

Figure 2.2: The `saturation` module.

Signal	Dir	Size (bits)
`value_i`	in	40
`do_sat_i`	in	1
`value_o`	out	40
`did_sat_o`	out	1

Table 2.4: Input and output signals of the `saturation` module.

2.4.3 Module `mac_dp`

The final task in lab 2 is to minimize the area of the MAC unit contained in the file `mac_dp.v` or `mac_dp.vhd`. As it is currently written, the datapath uses three adders. Your goal is to modify the datapath shown in Figure 2.3 so that all operations listed in Table 2.5 can be implemented even though only one adder is used. Note that the operation names do not always correspond exactly to an assembler instruction. For example, the `MOVE` operation in `mac_dp` is used for both the `move`, `movel`, and `postop` instruction. Similarly, the `ADD` operation is used for the `addl` instruction.

In order to finish this task you will first need to create a schematic which shows how you plan to modify the datapath. *You should also bring this schematic to the lab so that you can show it to the lab assistant.* You should probably also familiarize yourself with the MAC unit source code and identify all parts in Figure 2.3.

If you should now check the area of the MAC unit by synthesizing it using the command `make synth_macdp` to run the synthesis script. (You may need to run the command `module add synopsys/2008.09` before `make synth_macdp` will work.) If you are feeling ambitious you may also want to look into the TCL script used by the synthesis tool and change the timing constraints. This will allow you to determine whether the timing constraints have an impact on the final area or not.

After determining the area of the unoptimized module you need to modify the RTL code so that it matches the schematic you created previously. Once this is done you need to verify that your changed module works correctly by creating one or several test cases in assembler. Similarly to the other modules in this lab, you should, at a minimum, aim for 100 percent statement coverage of the `mac_dp` module.

Finally, once you have verified that the module works correctly you should synthesize the module again to determine whether your changes have improved the area, and if so, how much. You could also look at the timing report to see whether you can still fulfill the timing requirements.

Operation name	opcode	Explanation
`CLR`	0	`mac_result = 0`
`ADD`	1	`mac_result = mac_operanda + mac_operandb`
`SUB`	2	`mac_result = mac_operanda - mac_operandb`
`CMP`	3	`mac_result = mac_operanda - mac_operandb`
`NEG`	4	`mac_result = 0 - mac_operandb`
`ABS`	5	`mac_result = abs(mac_operandb)`
`MUL`	6	`mac_result = mul_opa * mul_opb`
`MAC`	7	`mac_result = mac_operanda + mul_opa * mul_opb`
`MDM`	8	`mac_result = mac_operanda - mul_opa * mul_opb`
`MOVE`	9	`mac_result = mac_operandb`
`MOVE_ROUND`	10	`mac_result = round(mac_operandb)`
`NOP`	11	`mac_result = 0`

Table 2.5: All operations that should be supported by the `mac_dp` module.

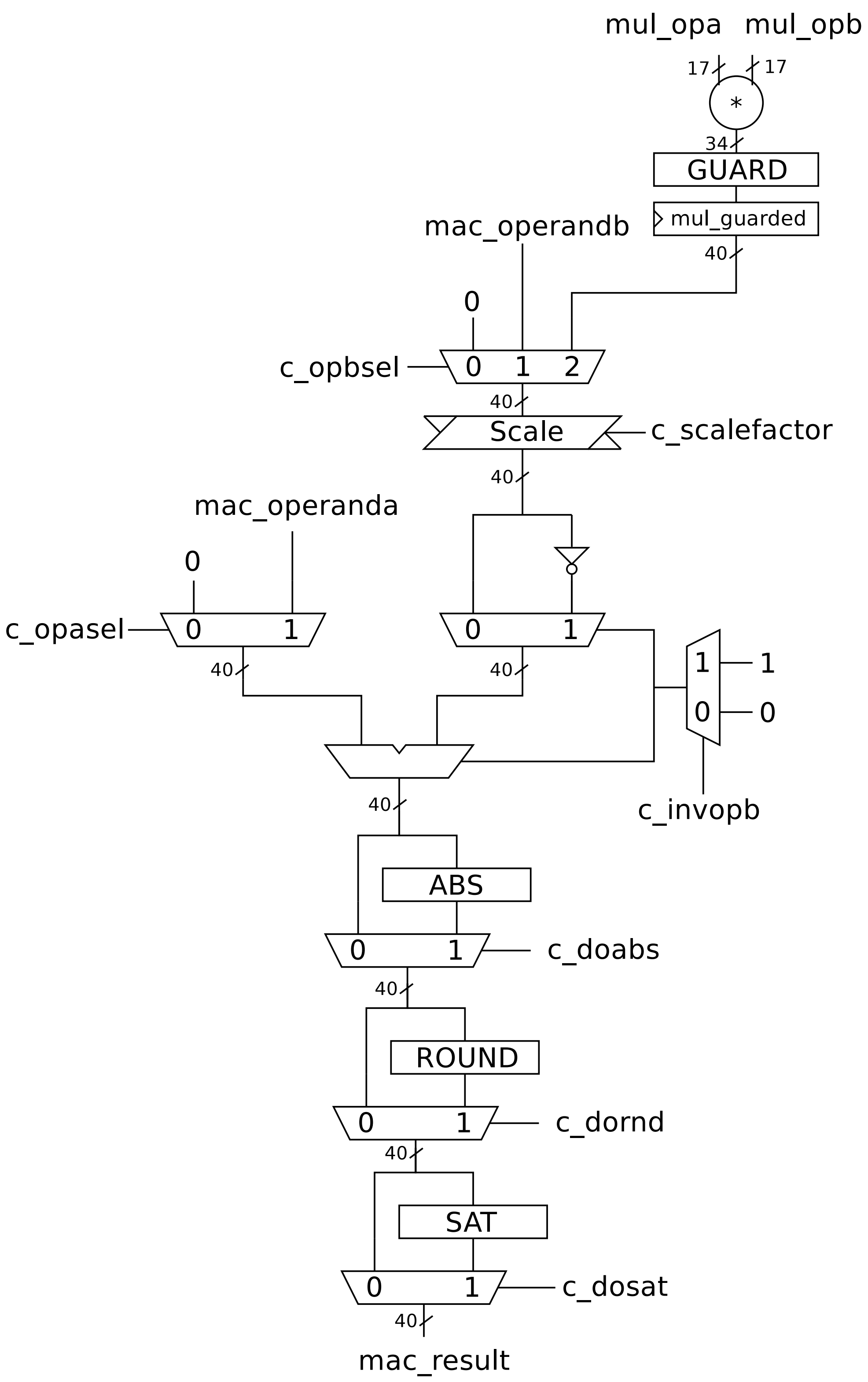

Figure 2.3: The unoptimized `mac_dp` module. (Not shown: logic for the control table and the overflow flag.)