TSEA26 – Design of Embedded DSP Processors
Lab Manual

Andreas Ehliar, Johan Eilert, Oscar Gustafsson, Per Karlström, Di Wu

January 16, 2025

Lab 0 – Introduction

This chapter will present important aspects of the lab environment. Read this chapter carefully and use it as a reference if you get in trouble during the labs. You can also ask a lab assistant for help if you cannot find the answer in the lab compendium.

0.1 Formatting

To assist in where to execute which command, background coloring is used. The options are:

- `shell> command` : Execute `command` in a shell prompt
- `ModelSim> command` : Execute `command` in ModelSim prompt
- `sim> command` : Execute `command` in the Senior simulator `srsim`

0.2 Lab Groups and Lab Examination

The labs are supposed to be done in groups consisting of up to two students. It is important that both students in a lab group are active and take part in all aspects of the labs, including the preparations and the oral examination. Since the labs are a part of the examination, cooperation between groups is not allowed. If you have any questions about this, feel free to ask the lab assistants.

When you have finished a lab, you will have to show a lab assistant your work. At this point both members of the lab group must be prepared to answer whatever questions the lab assistant may have regarding your work[1].

0.3 Get Started

Before you start you need to perform a number of steps to setup the correct environment to run the labs. If you run into problems or suspicious error messages at any step described in this chapter, do not hesitate to ask the lab assistants!

To get started, first you need to download the lab files from Lisam. Unpacking the downloaded file with

[1] Be aware that the lab assistant may choose to pass only one member in the group if only one member in the group is able to answer questions about the lab work.

```
shell> unzip labX.zip
```

will create a sub directory called `labX`. There are three lab files, one for lab 1, one for labs 2 and 3, and one for lab 4. The lab manual assumes that you have unpacked these in a directory called `tsea26`. See Section 0.6 for more information about the directory for labs 2 and 3.

You also need to execute

```
shell> module add courses/TSEA26
```

to get access to the assembler and the instruction set simulator used in this course. In addition, you need to execute

```
shell> module add prog/matlab/
```

to get access to MATLAB or

```
shell> module add prog/anaconda3/
```

to get access to (a newer) Python.

Later on, you need to exectute

```
shell> module add prog/modelsim
```

to get access to ModelSim. To synthesize the designs, you must log in to `only-da` using `ssh` and execute

```
shell> module add synopsys/dc2019.03
```

.

0.4 Senior Assembler (`srasm`)

The assembler takes as input a text file and translates it into a binary format that the processor understands. To run the assembler is straightforward. For any file you want to assemble give the following command:

```
shell> srasm anyfile.asm
```

The assembler will then output a file called `anyfile.hex` which contains the binary data for the program memory.

0.5 Senior Simulator (`srsim`)

The simulator is used to simulate the programs in software before they are run on the processor. The simulator takes as input a `.hex` file generated by Senior assembler.

To run the simulator for any program you want to test, give the following command:

```
shell> srsim file.hex
```

The simulator will now start, and you will see the simulator command line (`sim>`). Now try the command `sim> help` (or `sim> h`) to get a list of available commands for you to use. For

example, the command `sim> r 8`, will run the program for 8 clock cycles.

0.6 Setting Up Lab 2 and 3

The two most important directories for lab two and three are `lab2-3/rtl/` where the RTL code of the DSP processor can be found and `lab2-3/asm/lab2/` where the assembly language program templates resides.

You need to decide whether to use VHDL or Verilog by changing to the `lab2-3/rtl/` directory and executing

```
shell> make use_vhdl
```

or

```
shell> make use_verilog
```

This will move the template files for the selected language into the RTL directory and delete the template files for the other language.

You may also need to execute

```
shell> module add prog/modelsim
```

to get access to the RTL simulator ModelSim (this is necessary if you receive an error that `vlib` cannot be found). To synthesize the designs, you will use Synopsys Design Compiler, although currently this requires that you log in to a different machine, see Section 2.4.3.

0.7 Verification with ModelSim

When you have implemented a new hardware module you need to test it to see if it works as expected.

Testing your implementation can be initialized with the following command (to be executed in the `tsea26/lab2-3` directory), where ξ is the current lab number:

```
shell> make lab_vsim TEST=labξ/module.asm
```

This will assemble and run the program *module*`.asm` found in `tsea26/lab2-3/asm/labξ/` using the instruction set simulator `srsim`.

For example, to test the saturation module in lab2, run the following command:

```
shell> make lab_vsim TEST=lab2/saturation.asm
```

If the process stops with an error message, you must correct the assembly language program or in the RTL code. The error message will hopefully give you enough information.

If there are no errors, the RTL simulator (ModelSim) will load. It will automatically load the test program into the DSP program memory. You can start the RTL simulation with `ModelSim> run -all`. The simulation will stop when the simulator encounters the instruction `out 0x12,...`, or when it

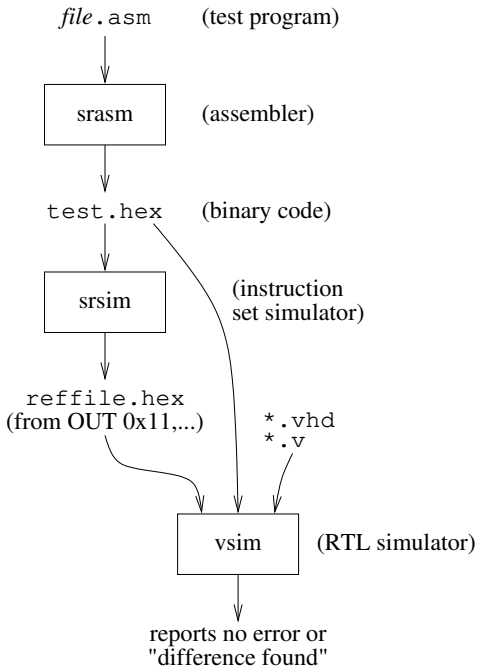

Figure 1: A graphical view of the verification flow described in Section 0.7.

encounters the instruction `out 0x11,...` that outputs data that do not match that in the reference file generated by the instruction set simulator.

Some people may find it useful to know what is actually happening during the execution of the `make` command. The entire flow is also shown in Fig. 1.

1. First, the specified test program is assembled:

   ```
   shell> srasm asm/labξ/module.asm test.hex
   ```

   This creates `test.hex` which is the machine language version of your program.

2. Then, the instruction set simulator is run in batch (unattended) mode:

   ```
   shell> srsim -r test.hex
   ```

   For each "`out 0x11,...`"-instruction, the simulator will write a line to the file `IOS0011`.

3. Then, the output file is renamed:

   ```
   shell> mv IOS0011 reffile.hex
   ```

   This file is used by the RTL simulator as the reference file.

4. Then, all RTL files that have changed are compiled:

   ```
   shell> make compile_rtl
   ```

   This will run `vcom` and `vlog` to compile the VHDL and Verilog code, respectively.

5. Finally, the RTL simulator (ModelSim) is started:

   ```
   shell> vsim dsp_system_top
   ```

   The RTL simulator will load the test program (`test.hex`) and the reference output file (`reffile.hex`). Every time the RTL simulator encounters an "`out 0x11,...`"-instruction during the execution of the program, the data is compared to that in the reference file. If there is a mismatch, the RTL simulation will be stopped.

0.7.1 Going Step by Step

A step by step procedure for compiling the code and working with ModelSim is presented below. If you want, you can try this procedure with a simple program called `simple.asm` residing in the directory `asm/lab3`. All commands require that you execute them from the root of the `tsea26/lab2-3` directory.

1. **Assemble:** Assemble `simple.asm`

   ```
   shell> srasm asm/lab3/simple.asm test.hex
   ```

2. **Simulate code:** Simulate the generated `asm/lab3/simple.hex` in batch mode

   ```
   shell> srsim -r test.hex
   ```

3. **Create reference file:** Create the reference file for the test bench by renaming `IOS0011` to `reffile.hex`

   ```
   shell> mv IOS0011 reffile.hex
   ```

4. **Initialize VHDL/Verilog:** If you have not done this before. Initialize the hardware with either VHDL or Verilog skeleton files.

   ```
   shell> make use_verilog
   ```

   (for Verilog)

   ```
   shell> make use_vhdl
   ```

   (for VDHL)

5. **Compile hardware:** Compile all hardware modules.

   ```
   shell> make compile_rtl
   ```

6. **Start ModelSim:** Start ModelSim and load your design.

   ```
   shell> make vsim
   ```

Note that this make rule also compiles all the files, thus the previous step was not really necessary.

7. **View wave:** ModelSim should now have started with no errors. If ModelSim did not start or reported errors some of the previous steps were not properly completed. To view the wave window, type the following in ModelSim
   `ModelSim> view wave`

8. **Add signals:** Add a number of useful signals to the wave window as follows:
   `ModelSim> do senior_signals.do`

9. **Simulate hardware:** Run the simulation as follows:
   `ModelSim> run -all`

10. **Say No!:** The simulation should run with no errors and ModelSim will prompt if you want to finish. Answer no and look at the wave window. Almost at the top, under INSTR PIPE you can find the current PC value and where the various instructions are in the pipeline. A number of other useful signals has been recorded also. Look around here and familiarize yourself with what signals has been added for you. Depending on your success with this lab you might have to add a number of other signals

11. **Rerun:** If you discover an error in the RTL code, do not close ModelSim. Fix the error in the RTL code, then run

    ```
    shell> make compile_rtl
    ```

    and restart the simulation in ModelSim.
    `ModelSim> restart -f`
    Finally run the simulation again (`ModelSim> run -all`)

It is often useful to log more signals than it is meaningful to have in the wave window (avoid information overload). You can, if you logged more signals, run your simulation and afterwards add the signals you need to look at. To accomplish this, you can use the log command in ModelSim
`ModelSim> log -r /*`
It is often useful to perform a restart, log, and run all after each other:
`ModelSim> restart -f; log -r /*; run -all`

0.8 Important Makefile Rules

The procedures described in Section 0.7 is run using `make` with a `Makefile` in the root of the `tsea26/lab2-3` directory. `make` is a tool often used when compiling programs. A `Makefile` has a number of rules for how to reach a specific goal. You can think of a `Makefile` as a recipe for how

to cook programs and make as the chef. A more thorough description of how `make` works is outside the scope of this course, but it is sufficient to know that you run `make` by typing

```
shell> make
```

and after `make` the rule you want `make` to run.

The most important `make` rules are presented here. There are more rules in the `Makefile` and if you are interested in what they are, feel free to take a look in the `Makefile`.

`compile_rtl:` Compiles all Verilog and VHDL files needed for RTL simulation.

```
shell> make compile_rtl
```

`vsim:` Runs `compile_rtl` then starts ModelSim and loads `dsp_system_top` as the top module. Now use ModelSim to simulate, view the wave window and so on.

```
shell> make vsim
```

`decoder_vsim:` Compiles and prepares all files necessary for simulating the full MPEG2 decoder with the RTL code. ModelSim is then executed in batch mode and if the simulation is successfully completed, the output from the RTL code is converted to `rtlaudio.wav`, which you can listen to.

```
shell> make decoder_vsim
```

`lab_vsim:` Compiles and prepares all files necessary for simulating an assembly file specified by the `TEST` variable. `TEST` must be the name of a file under the `asm` directory. If the test program needs input data, a file named `$TEST.in` in the same directory as `$TEST` will be used for input. ModelSim is then started with the `dsp_system_top` as the top module

```
shell> make lab_vsim TEST="lab directory"/"test program.asm"
```

`use_verilog:` Copies all relevant Verilog source code skeletons to the main RTL folder. Use this rule if you want to work with Verilog code. This needs to be done once for each time you use a new release of the lab package.

```
shell> make use_verilog
```

`use_vhdl:` Copies all relevant VHDL source code skeletons to the main RTL folder. Use this rule if you want to work with VHDL code. This needs to be done once for each time you use a new release of the lab package.

```
shell> make use_vhdl
```

Table 1: Top level modules.

Instance	Description
`clock_generate`	Generates a clock signal
`reset_generate`	Generates a reset signal
`dsp_core`	The DSP processor Senior
`mem`	Models of all the memories; DM0, DM1, and PM
`write_out_file`	Writes data to `rtloutput.hex` through port `0x11`
`read_in_file`	Reads data from `IOS0010` through port `0x10`
`read_ref_file`	Reads reference data from `reffile.hex` when data is written through port `0x11`

0.9 The System

The processor is described in `dsp_core.v`. In order to run the processor core, additional hardware is needed, i.e. memories and a clock generator etc. All this support hardware has been modeled for you and are instantiated together with Senior in `dsp_system_top.v`. The instantiated modules are briefly described in Table 1. It is advised that you take a look into `dsp_system_top.v` and familiarize yourself with the simulation top level module.

Some things to note about `dsp_system_top`:

- The instruction pipeline is visible to ease debugging in ModelSim
- Two extra output ports are defined: port `0x21` and `0x22`. Port `0x21` will output the value written to the port to ModelSim but will not have any effect in `srsim`, writing to this port will also cause the simulation to stop, in contrast to finish as would happen if writing to port `0x12`. Port `0x22` will simply output a value to ModelSim, this is good to use when debugging code for example.
- When writing to port `0x11` the data written will automatically be compared to data from `reffile.hex`. If the data does not match, the simulation will stop and report the error. To constantly output data makes the simulation run slower. Thus, if running the full decoder, make sure to comment out the line reporting what is being compared.

Lab 1 – Firmware

1.1 General Description

Digital signal processing (DSP) is everywhere in our daily life. Meanwhile, DSP can be meant as quite different things ranging from MATLAB/Python programming to hardware implementation. In a real-world project, engineers usually start from a high-level model of the algorithm. However, this is not enough because high-level models in e.g. MATLAB or Python are usually not real-time. Therefore, the algorithm must be implemented using e.g. DSP processors or fixed-functional hardware, which is one of the major tasks of embedded system design.

The purpose of lab 1 is to go through the top-down flow from MATLAB/Python modeling to assembly programming using the DSP processor **Senior** presented in this course. Although the lab is based on a simple example, the same flow can be applied to other more complicated DSP applications. Through this lab, you should be able to learn basic skills of assembly programming using the **Senior** instruction set. Furthermore, you should be aware of the finite-length error introduced in digital signal processing in the real-world implementation.

1.2 Task 1: Create a Lowpass Filter

You can do this part in either MATLAB or Python.

1.2.1 Matlab

Download and decompress the source code for lab 1 from Lisam. Start MATLAB by typing

```
shell> matlab
```

Find the MATLAB file `lab1.m` in the uncompressed directory. This file is a demonstration of how 50 Hz line noise can be removed from an ECG signal by using a lowpass filter.

Change `lab1.m` so that the `fir1()` call creates the coefficients for a lowpass filter with a cut-off frequency of 15 Hz. To see exactly how `fir1()` works, type `help fir1` in MATLAB, although the following excerpt from the help page should be enough for this lab:

> *B = FIR1(N,Wn) designs an N'th order lowpass FIR digital filter and returns the filter coefficients in length N+1 vector B. The cut-off frequency Wn must be between 0 < Wn < 1.0, with 1.0 corresponding to half the sample rate.*

After you have changed the script, run the `lab1.m` script. If everything is correct, you should see two windows with a time-domain and frequency-domain analysis of the signals. In the time-domain window you should be able to see that the ECG signal has now been filtered and the frequency-domain analysis should also reflect the fact that the energy at 50 Hz has been significantly reduced.

1.2.2 Python

To get access to SciPy which is needed by the Python script, run

```
shell> module load prog/anaconda3/
```

Download and decompress the source code for lab 1 from Lisam. Open an editor with the provided file `lab1.py` in the uncompressed directory. This file is a demonstration of how 50 Hz line noise can be removed from an ECG signal by using a lowpass filter.

Change `lab1.py` so that the `firwin()` call creates the coefficients for a lowpass filter with a cut-off frequency of 15 Hz. To see exactly how `firwin()` works, type

```
1 import scipy
2 help(scipy.signal.firwin)
```

in a Python prompt, although the following excerpt from the help page should be enough for this lab:

> *firwin(numtaps, cutoff, width=None, window='hamming', pass_zero=True, scale=True, nyq=None, fs=None)*
> *FIR filter design using the window method.*
> ...
> *cutoff : float or 1-D array-like*
> *Cutoff frequency of filter (expressed in the same units as fs) OR an array of cutoff frequencies (that is, band edges). In the latter case, the frequencies in cutoff should be positive and monotonically increasing between 0 and fs/2. The values 0 and fs/2 must not be included in cutoff.*

After you have changed the script, run the `lab1.py` script. Note that you must run it with Python 3. To execute in interactive mode so that you can run the `lab1()` command again, you can start as

```
shell> python3 -i lab1.py
```

If everything is correct, you should see two windows with a time-domain and frequency-domain analysis of the signals. In the time-domain window you should be able to see that the ECG signal has now been filtered and the frequency-domain analysis should also reflect the fact that the energy at 50 Hz has been significantly reduced.

1.3 Task 2: Run and Modify a Simple Assembly Program

Find the assembly source file `helloworld.asm` which contains a small assembler program example. Use the assembler to convert the assembly source file to the binary code which can be understood and executed by the instruction set simulator.

Read through the example file, then try to use the `srasm` assembler (Section 0.4) to convert the assembly source file to the binary code which can be understood and executed by the instruction set simulator (Section 0.5).

Start `srsim` and run one instruction at a time until the program is finished. This can be done in the following way:

- Tell the simulator to run one cycle `sim> r 1`
- Press enter to repeat the last command. (Hold down enter to single step faster.) Notice that some of the registers that have been printed to the screen have changed.
- If you want to see where the PC is located, you can use the `l` command to list the source code surrounding the current program counter.
- You can see a few other commands that the simulator has by using the `sim> h` command.
- If you want to run the simulator until it reaches the end of the program, use the `sim> g` command.
- Take note of how many clock cycles that were used.
- Press `sim> q` to exit the simulator when done.

Open the `IOS0011` file and verify that they contain the expected output, that is, all numbers from 0 to 42.

Change the program so that a `repeat`-based loop is used instead of a loop which uses normal conditional branches. When done correctly, you will be able to run the same program in less than 100 clock cycles.

1.4 Task 3: Implement a Single Sample 32-Tap FIR Filter in Assembler

In this task you are supposed to implement an interrupt handler friendly version of an FIR filter. Being interrupt friendly means that the FIR filter may not modify any register unless it can restore that register to its original value.

Open `lab1.asm` and try to understand what the file is doing:

1. Setup the stack pointer
2. Setup FIR kernel parameters
3. Initialize the sanity checker (which sets almost all registers to known values)

4. For 1000 samples, call the `fir_kernel` subroutine. This subroutine should read one sample from the input port (`0x10`), add this to the ring buffer, filter it using a 32-tap FIR filter, and write one sample to the output port (`0x11`).

5. Run the sanity checker that makes sure that (almost) all registers contain the same value they were set to in the beginning.

6. Quit the program (by writing to port `0x13`)

In order to do this you need to perform at least the following tasks:

1. Convert the filter coefficients you created in MATLAB/Python into suitable fixed point constants and enter them into `lab1.asm`

2. Finish the `fir_kernel` function

3. Assemble and run `lab1.asm`

4. Verify that the output of the filter is correct. You can do this by running the `lab1.m` (`lab1.py`) script in MATLAB (Python). The relative error should be very small (less than 0.001) if everything works correctly.

1.5 Task 4: Scaling the Coefficients

Once you have gotten the FIR filter working you should ensure that you get as small relative error as possible. Two things are required for this: You need to scale the coefficients so that you utilize as much as possible of the available 16 bits. You also need to round the coefficients correctly.

1.6 Task 5: Performance Measurement

You now need to measure the performance of your FIR filter under the assumption that it is running as an interrupt handler. In other words, the clock cycles used by the top-level loop and the sanity-check code is unimportant. You must thus separate how much time this code takes by running `lab1.asm` with an empty `handle_sample` which only contains a `ret` instruction.

If the processor is running at, say, 10 MHz, how much of the CPU time is used by the interrupt handler?

165353 - 77353 = 88000
88000/10M = 0.0088 sec

Discuss the latency of your signal processing system.

Latency is constant with the Pushing and popping and MAC operations hapening every time

Approximately how many clock cycles does it take from an interrupt is generated to the time when a new sample is presented on the output of the D/A converter under the assumption that the Senior Processor is connected as shown in Fig. 1.1?

1214 - 1091 = 123 cycles

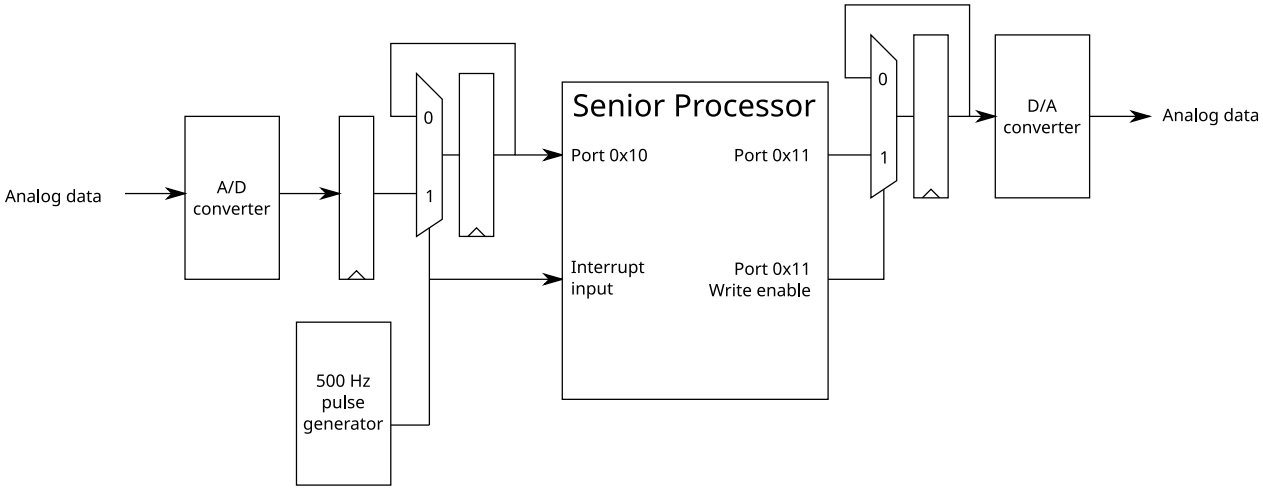

Figure 1.1: The Senior processor as connected to an A/D and D/A converter.

Is the latency always the same in your implementation?

YES

Is there anything you can do to reduce the latency?

Not having as many push and pop by reducing the nr of registers or having constant set registers

1.7 Task 6: Write a 10-Sample 32-Tap Block FIR Implementation

Modify the filter kernel so that the interrupt handler is responsible for processing ten samples during the same interrupt. You can assume that the system is connected as shown in Fig. 1.2. That is, the A/D converter part of the hardware contains a FIFO which stores up to ten incoming samples. (The D/A converter part contains a similar FIFO.)

In order to test your filter kernel, you also need to modify the top level loop so that only 100 iterations are run instead of 1000.

How much CPU time that is used by the interrupt handler?

100853 - 77353 = 23500 cycles

0.00235 sec

Discuss the latency of this implementation.

1106

1.8 Task 7: Reality Check

Discuss the following:

Under what circumstances do you want to use the single sample FIR filter?

When we do not have any memory to store the FIFO

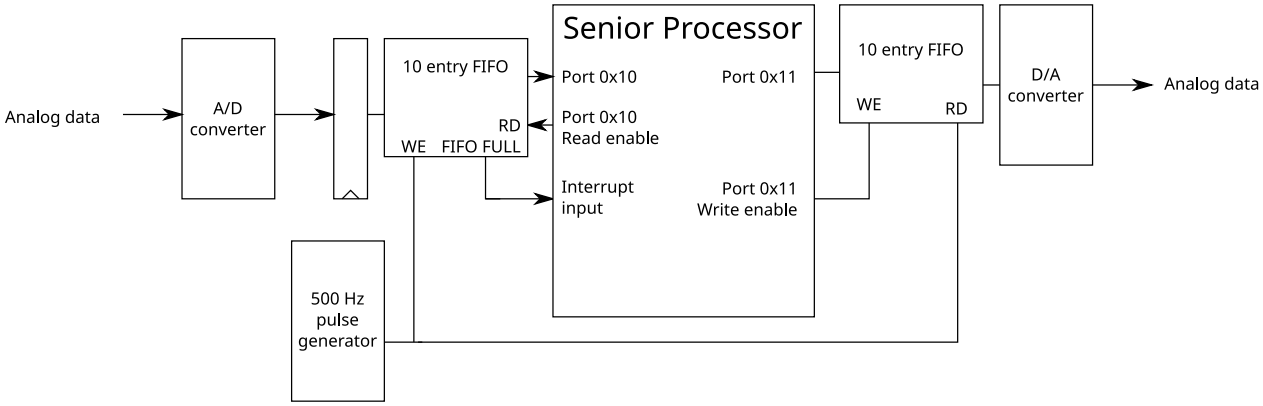

Figure 1.2: The Senior processor as connected to a buffered A/D and D/A converter.

Under what circumstances do you want to use the block FIR filter?

When we do have a memory and do not care about a slower and variable latency

Under what circumstances do you want to run these in an interrupt handler?

Based on your experience in this lab, are there any improvements you want to make to the Senior processor?

Lab 2 – Data Path

2.1 Overview

During lab 2 you will complete the RTL code of the ALU and MAC data paths of the DSP core and write a set of small test programs to verify your implementation. Most of the data path code has already been written and verified, 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 data path 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 0.7 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 Workflow

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 0.7.

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 MAC unit data path

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

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

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

`alu_test.asm`: Assembly language code to test the adder multiplexers 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 multiplexers 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 has not been executed this may indicate that you have missed a corner case in your test bench. (However, some statements should not 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.

You can find documentation of the coverage window output in Table 4-52 of `/sw/mentor/modelsim_SE_2021.3/modeltech/docs/pdfdocs/modelsim_se_gui_ref.pdf`.

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 multiplexers 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 will implement the control signals for the adder input multiplexers and the carry-in selection multiplexer. In the second part, you will implement the multiplexers control signal for the `MIN` and `MAX` instructions. Note that the `MIN` and `MAX` instructions use the adder to compare the numbers.

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

Note that the calculation of absolute value shall 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 Fig. 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 multiplexers is given in Table 2.2, and a list of input and output signals for the `MIN/MAX` multiplexers is given in Table 2.3.

You shall also write assembler code to test the functionality of your ALU. At a minimum, you shall 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.

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

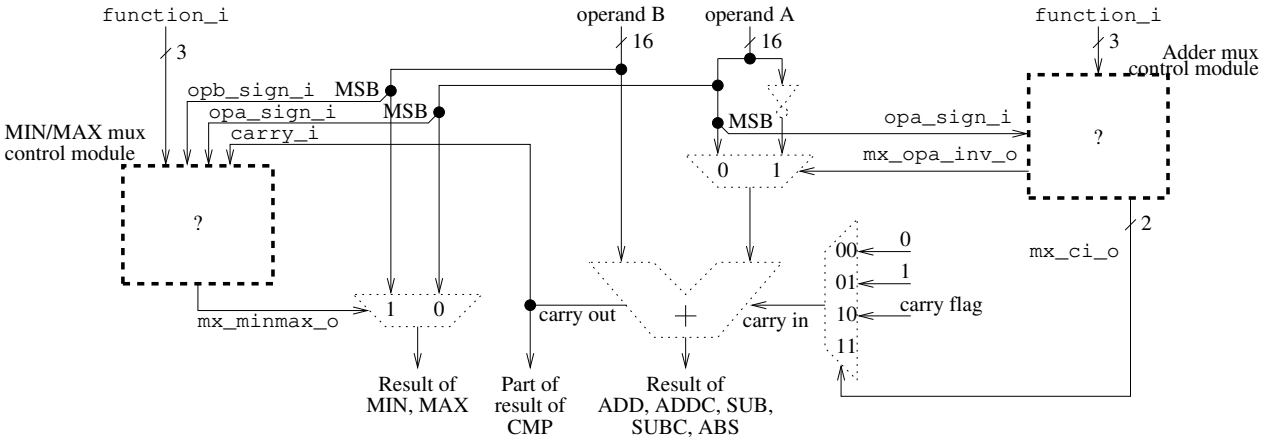

Figure 2.1: The Arithmetic Unit and MIN/MAX modules.

Table 2.1: The functions supported by the ALU.

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

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 2.3: Input and output signals of the `min_max_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

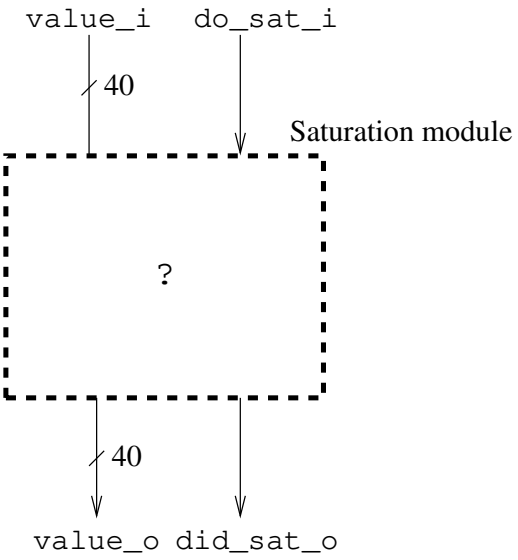

Figure 2.2: The `saturation` module.

Table 2.4: Input and output signals of 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

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 shall be one when saturation is performed and the data is modified by the module, it is zero when the data is unmodified. The output must be purely combinational, there must be no flip-flops in the module.

A schematic overview is given in Fig. 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.

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 Fig. 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`,

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

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`

`movel`, and `postop` instruction. Similarly, the `ADD` operation is used for the `addl` instruction.

If you cannot find the three adders: read the code.

In order to finish this task, you will first need to create a schematic which shows how you plan to modify the datapath.

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 Fig. 2.3.

Because of license issues, you must login to the computer `only-da.ad.liu.se` through ssh to be able to run synthesis.

Determine the area of the MAC unit by synthesizing it using the command

```
shell> make synth_macdp
```

to run the synthesis script. You may need to run the command

```
shell> module add synopsys/dc2019.03
```

before it 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. Similar to

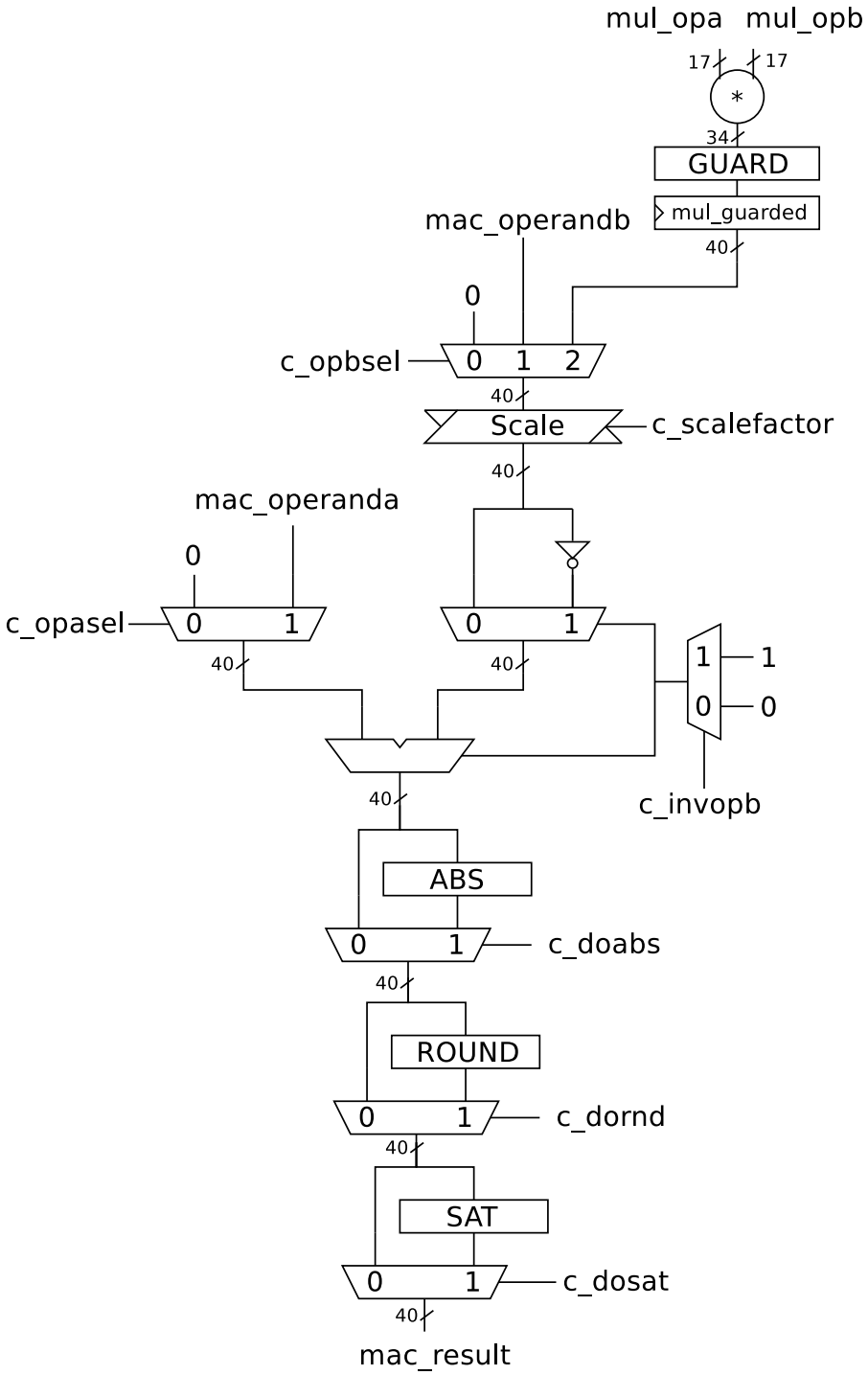

Figure 2.3: The unoptimized `mac_dp` module. (Not shown: logic for the control table and the overflow flag.)

the other modules in this lab, you should, at a minimum, aim for 100% 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.

Lab 3 – Program Flow

3.1 Introduction

In this lab you will learn more about what special considerations are needed when designing logic for program flow instructions.

3.2 General Description

After finishing lab 2 all building blocks of the processor are completed. It is now time to put all the parts together. In a pipelined processor this is not as easy as it first might seem. The parts must not only fit together in the topological aspect of connecting the right wire to the right input. The parts must also fit together in time. In Senior this task is complicated by the fact that there are three different pipeline depths in the processor, the pipeline depth depends on the instruction being executed. A pipeline will also make all kinds of program flow instructions troublesome as will be shown later in this exercise.

During all execution, instructions are fetched from the program memory (PM) using the program counter (PC) as the address. An instruction decides the values of all the control signals in the processor. An instruction usually has fewer bits than there are control signals in a processor, thus the instruction must be converted into its control signals before it is of any use. This is the task of the instruction decoder. The instruction decoder, situated in pipeline stage P2 in the processor according to Fig. 3.1, has two sub modules; `id_decode_logic` and `id_pipeline_logic`. Verify this by open `instruction_decoder.v` (only a Verilog version of this file is available). Your first task is to briefly describe what the two modules do. It is important that you understand how the pipeline in Fig. 3.1 works to complete this lab.

In lab 2 you have been running the entire processor, but not necessarily had to investigate it in any detail. In this lab it is advised that you familiarize yourself a bit more with the processor and the system supporting it, see Section 0.9.

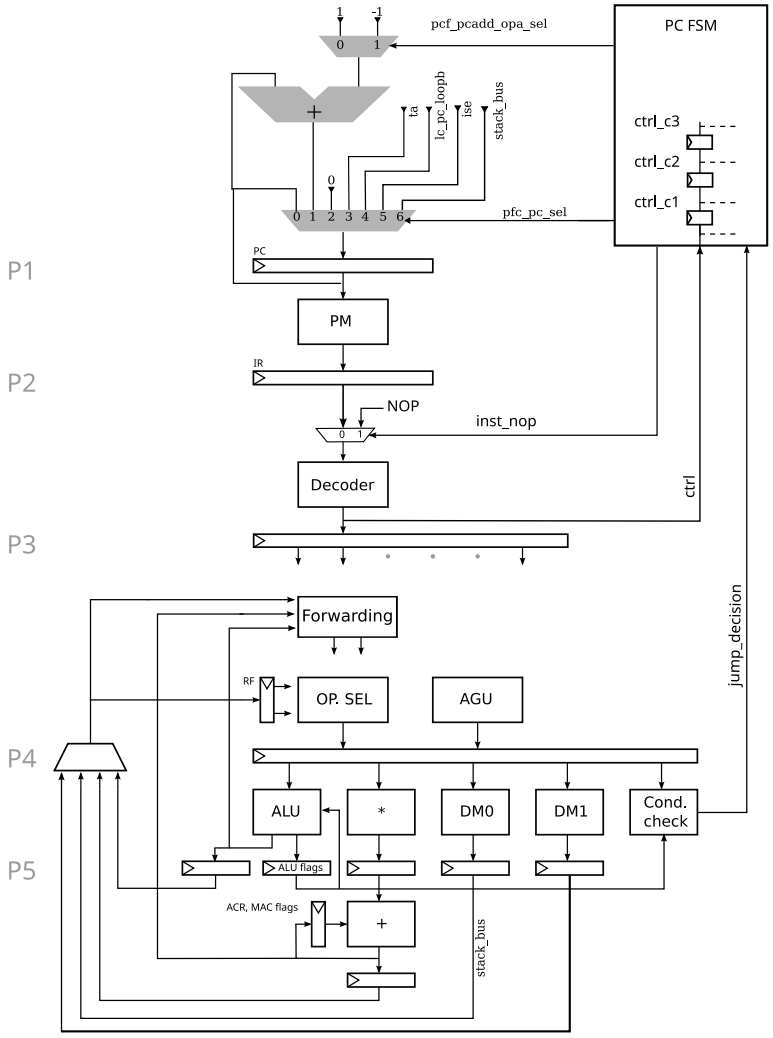

Figure 3.1: The normal pipeline, including PFC details.

3.3 Task 1: Instruction Decoding Hardware

What is the function of the modules `id_pipeline_logic` and `id_decode_logic`?.

Their functionality is not far from their names.

Jumps, calls, and returns are essentially the same operation and will be referred to as just jumps if the distinction is not important.

Table 3.1: Pipeline table for jump instruction.

P0 **PC+**	**P1** **PC(instr)**	**P2** **IR**	**P3**	**P4**
1	0(A)	—	—	—
2	1(B)	A	—	—
3	2(JMP 0)	B	A	—
4	3(C)	JMP 0	B	A
5	4(D)	C	JMP 0	B
0	5(E)	D	**C**	JMP 0
1	0(A)	E	**D**	**C**
2	1(B)	A	**E**	**D**
3	2(JMP 0)	B	A	**E**
4	3(C)	JMP 0	B	A
5	3(C)	C	JMP 0	B
and so on				

3.4 Understanding Delay Slots

Due to the pipeline the actual PC update in a jump cannot and will not happen immediately. The previous instruction must have reached and set the ALU flags in pipeline stage P4 to make conditional jumps work correctly. MAC flags will be set one cycle later and if using them to jump on the programmer must ensure they are set correctly. Waiting for the correct flags to be set means that a number of instructions after the jump instructions will be fetched and executed. The question is then what to do with them. There are two extreme options. The first extreme option is to let the processor execute them as they are fetched, and it is the programmers responsibility to make sure that the instructions after the jump instruction are useful. Instructions after the jump are said to reside in the jumps delay slots. The safe way to handle delay slots is to insert only `NOP`:s in all slots. The other extreme option is to let the processor deal with this and always insert `NOP` instructions in place of the fetched instructions. In Senior you can chose what you want to do with the `ds` directive. Thus when using `ds1` with a `jump` instruction the instruction following the `jump` will execute but the other two instructions will be forced to `NOP`:s by the processor. This is the reason for the `pfc_inst_nop_o` signals from the PC FSM. If it is set high, the decoder will see a `NOP` instruction instead of the instruction fetched from PM.

A special case is the `ret` instruction where the return address is fetched from top of the stack. In that case the jump address is available in stage P5 instead of P4 as normal, and because that, an extra NOP must be inserted in by the hardware.

3.5 Pipeline Table

To fully understand what is going on in the pipeline of a processor it is useful do draw a pipeline table as shown in Table 3.1. This example shows how a jump instruction is executed. In the P0 column the next PC value is listed, in P1 the actual PC value together with the instruction at that

Table 3.2: Pipeline table for jump instruction with `NOP`s introduced.

P0 **PC+**	**P1** **PC(instr)**	**P2** **IR**	**P3**	**P4**
1	0(A)	—	—	—
2	1(B)	A	—	—
3	2(JMP 0)	B	A	—
3	3(C)	JMP 0	B	A
3	3(C)	C	JMP 0	B
0	3(C)	C	**nop**	JMP 0
1	0(A)	C	**nop**	**nop**
2	1(B)	A	**nop**	**nop**
3	2(JMP 0)	B	A	**nop**
4	3(C)	JMP 0	B	A
5	3(C)	C	JMP 0	B
and so on				

address in the program memory is listed, in P2 to P4 the instruction is simply propagated through the pipeline. Here it is clearly seen how, in P3 and P4, three extra instructions are executed (C, D, and E), that is if nothing is done about it, i.e. inserting `NOP`s as described earlier. If instead `NOP`s are introduced, we will get a pipeline table as in Table 3.2. Note that C is still fetched as we cannot decode the `JMP` until after it has reached P2. So the instruction in P2 can affect the next instruction in P1.

If you find it troublesome completing the PC FSM, it is recommended that you draw a pipeline table for all program flow instructions, i.e., `JMP`, `CALL`, `RET`.

3.6 Task 2: Pipeline Table of a Small Program

In Listing 3.1 a small program is defined. Your task is to fill out a pipeline table as described in Section 3.5. Detailing how the instructions progress in the pipeline. You can refer to the instructions as the line number they are defined on in Listing 3.1. Set PC to zero for the first instruction (on line 2). When doing this exercise look at the pipeline of the processor, shown in Fig. 3.1, and think about what instructions are in each pipeline stage.

Also answer the following questions:

Which instructions get executed and which instructions get flushed?

What are the differences between a return and any other jump instruction?

The value of the return PC must be fetched from the stack.

Listing 3.1: A small program, `lab3_ex.asm`

```
PC
        .code
0:       set r1,0x0001
1:       set r2,0x0001
2:       sub r0,r1,r2
3:       jump.eq start
4:       add r13,r1,r2
     start
5:       add r1,r1,r2
6:       jump ds2 l1
7:       add r1,r1,r2
8:       add r2,r1,r2
9:       add r3,r1,r2
     l1
10:      call ds3 f1
11:      add r4,r1,r2
12:      add r5,r1,r2
13:      add r6,r1,r2
14:      move r7,r2

15:      out 0x12,r0

     f1
16:      move r8,r1
17:      ret ds1
18:      add r9,r1,r2
19:      add r10,r1,r2
20:      add r11,r1,r2
21:      add r12,r1,r2
```

3.6.1 Task 2: Summary

Create a pipeline table of the program in Listing 3.1 and answer the questions.

3.7 Task 3: Program Counting

Before any instruction decoding or pipelining can take place the instruction to use must be fetched. This is done by taking the value of the program counter as an address into the program memory. In a processor with no flow control, the program counter logic is easy. Just increment it by one instruction each clock cycle. However, in a useful processor some form of flow control must be implemented. Before solving this task, the program counter will only increment its value by one each clock cycle and executing any form of program flow instructions will result in undefined behavior.

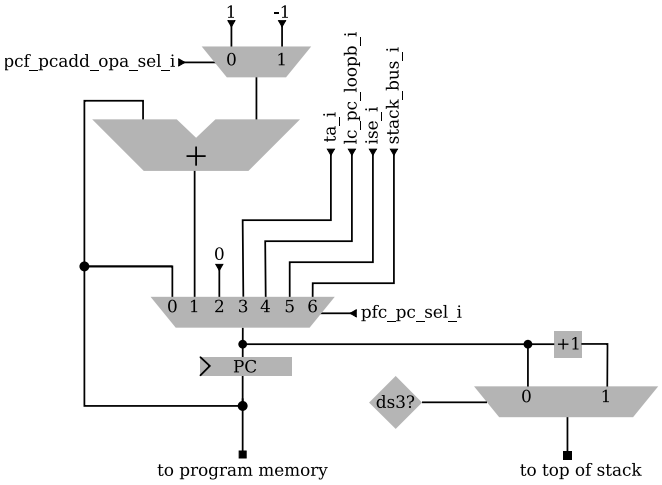

Figure 3.2: Next PC selection block diagram and logic for generating the return address in case of a `call` instruction

3.7.1 The PC FSM

Your task is to complete the finite state machine responsible for selecting what the next value of the program counter shall be. Large parts of the FSM has been implemented in `pc_fsm.v(hd)`. Your task is to complete the FSM. You need to make sure that the FSM jumps to the correct state and also in each state sets the correct output signals.

As shown in Fig. 3.1, the PC FSM takes inputs from the instruction decoder (the `ctrl` signal) and the condition checker (the `jump_decision` signal). It outputs two signals, `pc_add_opa_sel` and `pc_sel` to the program counter (next PC block). The program counter will update the PC value according to these two control signals from PC FSM.

In order to complete this task you must understand the architecture of the program counter (`program_counter.v`), shown in Fig. 3.2 with descriptions of the control signal shown in Table 3.3. Worth noting about the program counter architecture is that two of the output signals from the PC FSM is used to select the next PC.

Figure 3.2 also shows how the return address is calculated for `call` instructions, which is good to know when you are to complete the FSM.

Control Signal Naming Convention

The control signals names in the modules is named as follows: `ctrl_`$\xi\Delta\gamma$. Where ξ is `_i` or `_c`μ, where μ is an integer representing how much the control signals has been delayed locally in the module. Δ is the control signal name delimiter, ` ` in Verilog or `_` in VHDL, and γ is the control signal name. For example, from Table 3.4, `ctrl_c2_PFC_RET` (VHDL), corresponds to the signal `ctrl_i_PFC_RET` delayed two clock ticks.

Furthermore, you must understand how the PC FSM works. The PC FSM is of Mealy type, which means that both the next state and the output is a function of the current state and the input

Table 3.3: Next PC selection signal descriptions

Signal	Width	Description
ise_i	16	Interrupt address (not used)
lc_pc_loopb_i	16	Start address of repeat loop (not used)
ta_i	16	Target address of a jump or call
stack_bus_i	16	Top of hardware stack, used for returns
pfc_pcadd_opa_sel_i	1	Increment/decrement PC
pfc_pc_sel_i	3	Next PC multiplexer selection

Table 3.4: PC FSM input signals. The signal ctrl_i is divided into 4 fields.

Signal	Width	Description
jump_decision_i	1	High if jump shall be taken
lc_pfc_loope_i	16	End address of repeat loop (not used)
lc_pfc_loop_flag_i	1	High if loopn register is 0 (not used)
pc_addr_bus_i	16	Current PC value (not used)
ctrl_i	6	Control signals from the instruction decoder
ctrl_i_PFC_REPEAT_X	1	High if repeat of only x instructions (not used)
ctrl_i_PFC_JUMP	1	High if executing a jump instruction
ctrl_i_PFC_DELAY_SLOT	2	Number of delay slots after a jump
ctrl_i_PFC_RET	1	High if executing a return instruction

signals. The general architecture of a Mealy machine is shown in Fig. 3.3.

The state transition graph for the PC FSM is shown in Figure 3.4. Worth noting are the five different paths going from S0. There are four different paths for jumps depending on how many delay slots are used. Finally, there is one path for normal PC increment.

There are two subtasks in getting the PC FSM to work. One task is to get all the transitions right and the other task is to make sure that each state outputs the correct values. It is not necessary to solve one task before the other, it might be easier to solve them together, one instruction at a time. To solve both these subtasks you must understand the meaning of the input and output signals to/from the PC FSM, the signals are listed in Tables 3.4 and 3.5.

It is important to note that all modules involved in program counting get their control signals non-pipelined from the instruction decoder. This in effect means that the loop controller and PC FSM gets their control signals in pipeline stage P2.

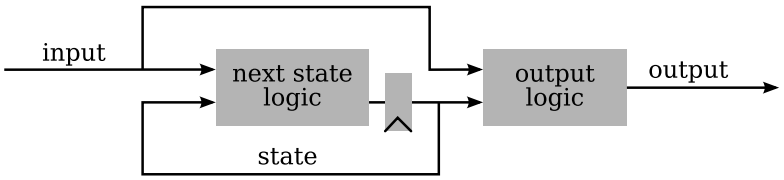

Figure 3.3: General architecture of a Mealy-type FSM.

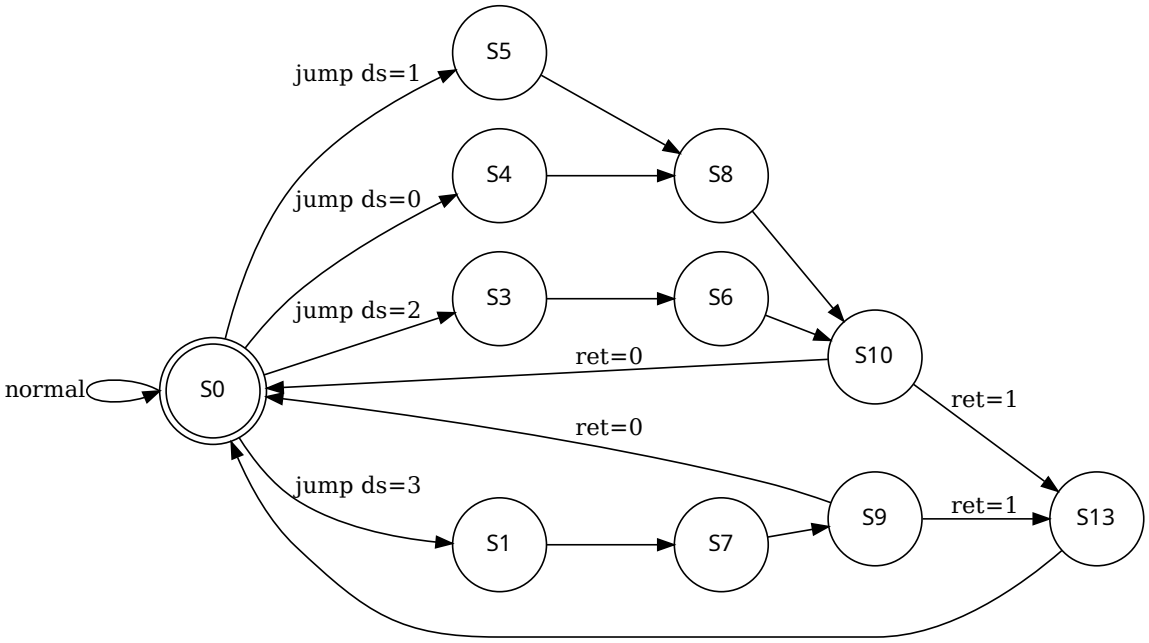

Figure 3.4: The PC FSM.

Table 3.5: PC FSM output signals.

Signal	Width	Description
`pfc_pc_add_opa_sel_o`	1	What to add to PC (see Fig. 3.2)
`pfc_pc_sel_o`	3	Final PC selection (see Fig. 3.2)
`pfc_inst_nop_o`	1	If high, inserts a NOP instruction
`pfc_lc_loopn_sel_o`	1	If high, decrement the loop counter register (not used)

PC FSM Transitions

As can be noted in your skeleton file `S0` will only jump to itself. This is obviously not correct, and you must select which state to jump to from `S0`.

Look for the "What is the next state?" comment.

PC FSM Output

All the output signals from the PC FSM are listed in Table 3.5 and it is your task to make sure they are set correctly in each state.

3.7.2 Testing the PC FSM

There are a number of assembler files prepared for you to test the RTL code you have written. These are listed in Table 3.6. It is suggested that you run the test programs in the order they are

listed in Table 3.6 since the later programs might assume that some program flow instructions are working. To run the test programs, just follow the procedure outlined in Section 0.7.

Table 3.6: PC FSM test programs.

File	Description
`pfc_jump.asm`	Tests `jump`
`pfc_cond_jump.asm`	Tests conditional `jump`
`pfc_call.asm`	Tests `call` and `ret`

3.7.3 Debugging Hints

If you encounter any bugs in your PC FSM it is much easier to debug them if you create your own test programs. (For example, by copying one of the test benches provided by us and removing everything but one test.)

Another common issue in this lab is that you may create a combinational loop which is fairly hard to find. If your simulation time suddenly stops advancing you have probably managed to create a combinational loop.

Look at Fig. 3.1 and determine whether you are setting the control signal for the NOP multiplexer correctly.

We have also noticed that missing signals in the sensitivity list in a process is a relatively common issue which can cause all sort of weird bugs.

3.7.4 Task 3: Summary

Complete the HDL code for the PC FSM as outlined above. Then test it using the test programs listed in Table 3.6.

3.8 What to Answer

When you have completed all lab tasks, demonstrate your design, show the code you have written, and be prepared to answer questions on how your design works.

Lab 4 – Prototyping ASIP Instructions

The purpose of this lab is to acquaint yourself with the structure of a simple instruction set simulator. You should know how you can simulate some of the effects of a pipeline in a simple simulator even though the pipeline itself is not implemented in the simulator. You should know how you can use such a simulator to quickly evaluate a proposed ASIP instruction, as one of the most important choices when constructing an ASIP is to choose what to accelerate in specialized instructions.

4.1 Introduction to Motion Estimation

In this lab, we will investigate how to accelerate a motion estimation algorithm commonly used in video coding applications. By dividing frames into smaller blocks and sending the translation (apparent motion) of each block it is possible to compress a video signal significantly. (In practice it is also necessary to send additional correction information as information about 2D motion is insufficient to reconstruct most video sequences adequately.)

The idea behind motion estimation is to first select a block in the new frame and then compare it with a number of blocks in the old frame. The metric which we will use in this lab is the sum of absolute difference (SAD). The comparison is done by taking the absolute value of the difference between each pixel in the old and new block. These values are then added to each other. The block with the lowest total sum is selected as the location for the new block in the old frame. If the lowest sum is 0, we have found a perfect match. However, most of the time we will not be able to find perfectly matching blocks in the new frame due to for example noise and non 2D-motion such as scaling and rotation. This procedure is repeated for each block in the new frame.

This idea is illustrated in Fig. 4.1. In this case only two blocks are investigated and the block on the right is the best match. In the lab we will investigate many possible locations around the original location. As test data we are using two frames from the `src6_ref__625.yuv` test stream from the Video Quality Expert Group. The video sequence was clipped to 176×144 pixels to reduce runtime and memory usage.

The pseudo code in Listing 4.1 shows how motion estimation using SAD can be done.

Listing 4.1: Pseudo code for motion estimation.

```
1 for each block in the image{ // 4x4 blocks
2   best_sad = Inf;
3   for each candidate position{
4     sad = compare_blocks(candidate_block, target_block);
```

```
5       if (sad < best_sad) {
6         best_sad = sad;
7         best_block = candidate_block;
8       }
9     }
10    output_position(best_block);
11  }
12
13  // This is the kernel algorithm which we are interested in
14  compare_blocks(a,b){
15    sum = 0;
16    for each pixel p { // 16 pixels
17      difference = a[p] - b[p];
18      sum += abs(difference);
19    }
20    return sum;
21  }
```

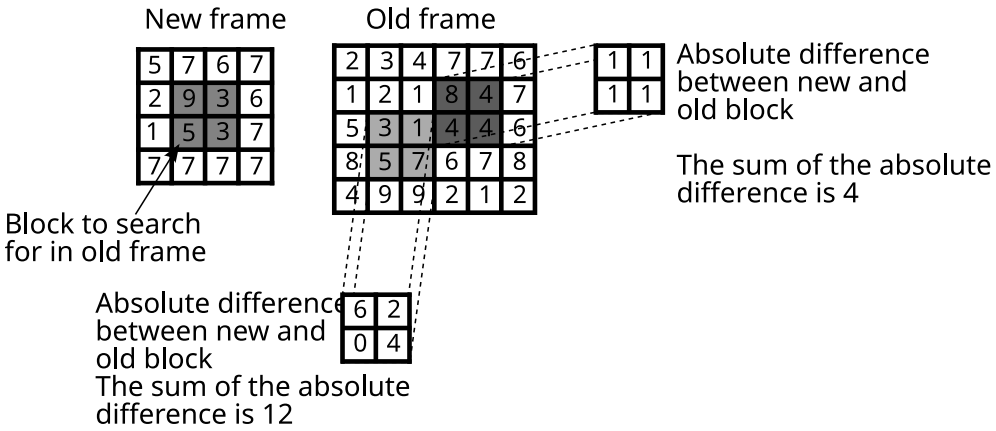

Figure 4.1: Block search using sum of absolute difference.

If you are interested in seeing the motion compensation in action, use the MATLAB script `motion.m`. There is going to be a huge black rectangle around the motion compensated image as we for simplicity reasons do not care about the borders, as special care must be taken not to search outside the borders of the original image. While it is of course possible to implement code to deal with this we do not bother about this to make `sad.asm` easier to understand for you.

4.2 Assembly Code

For this lab, the entire assembly program for a motion estimation algorithm has already been written. The program divides the images into blocks of 4×4 pixels and does an exhaustive search for the best match in a large region around each 4×4-block in the original frame.

You can find the program in `sad.asm`. The kernel part is listed below in Listing 4.2.

Listing 4.2: The kernel part of `sad.asm`.

```
1         repeat  sad_kernel_end,16
2         ld0     r0,(ar3++)      ; Load displacement in image
3         nop
4         ld0     r2,(ar0,r0)     ; Load pixel in original image
5         ld1     r1,(ar1,r0)     ; Load pixel in new image
6         nop
7
8         sub     r1,r1,r2        ; Calculate difference
9         abs     r1,r1           ; Take absolute value
10
11        add     r4,r4,r1        ; Sum of absolute difference
12 sad_kernel_end
```

To understand the kernel part, you need to know what the address registers are used for:

- `ar0` is a pointer to the upper left corner of the original block
- `ar1` is a pointer to the upper left corner of a block in the new image
- `ar3` is a pointer to a table with displacement values that are used to access the individual pixels in the 4×4-blocks (16 entries all in all)

4.3 Possible ASIP Instructions

Your task in this lab is to evaluate how this SAD kernel can be accelerated through the use of two custom instructions.

4.3.1 `accel_sad`

The first instruction you will implement is called `accel_sad`. It takes two registers as arguments. An example is given below:

```
1         accel_sad rD,rA
```

- Load `val1` from memory 0 at location `ar0` + `rA`
- Load `val2` from memory 1 at location `ar1` + `rA`
- Calculate the absolute difference between `val1` and `val2`

- Accumulate the absolute difference in a special SAD accumulation register (`sr31`)
- Write the current value of the SAD accumulation register into register `rD`. (The same value you just wrote into `sr31`.)

The implementation shall be pipelined so that it is possible to issue `accel_sad` instructions directly after each other.

The binary encoding of the `accel_sad` instruction is:

```
11000000 00dddda aaaa0000 00000000
```

where `ddddd` is the destination register (`rD`) and `aaaaa` is the source operand register (`rA`). (The destination and source register are located in the same place of the binary code as for other Senior instructions.)

This instruction is to be implemented in the Senior processor by putting it in the pipeline stage after the memory access. Full register forwarding must be used for the result.

4.3.2 `repeat_sad`

This instruction works just like the regular repeat instruction except that it will abort the repeat loop early if the current block is obviously worse than the best block which we have found to date.

This is done by keeping the current best SAD value in a special register (`sr30`) and comparing the value of the SAD accumulation register with `sr30`. If the SAD accumulation register is larger or equal, we abort the loop.

When implemented in the Senior processor, the PC FSM must use combinational logic to compare `sr30` and `sr31`.

4.4 The Simplified Senior Simulator

In this lab you will work with a simplified simulator of Senior where only the instructions required in this lab have been implemented. The error checking is also not quite as rigorous as the one in `srsim`. You will find the main part of the simulator in `sim.c`.

4.4.1 Quick Tour of `sim.c`

In the beginning of `sim.c`, there are several global variables used by the simulator to keep track of the values of the various registers in the processor. Perhaps the most important are `rf` and `rf_busy`. The latter keeps track of if a particular register is busy or not. `rf_busy` is used to simulate the effect of the pipeline even though `sim.c` is not a pipeline true simulator. For example, in the following sequence, `rf_busy` is used to make sure that `r9` is not accessed during the nop:

```
1        set    r9,53
2        nop
3        add    r2,r9,r3
```

Most of the other global variables are self-explanatory.

Do not modify the global variables directly. Only use the functions below.

`advance_cycles()`

This function is used to tell the simulation that a certain number of clock cycles have passed. It is used in the main simulation loop each time an instruction is fetched. It is also used in `advance_pc()` whenever a jump occurs which will stall the processor for a number of cycles.

`repeat_sad_stop()`

This is a helper function which tells the repeat loop to stop repeating in a certain number of clock cycles if it is using the `repeat_sad` mode.

`sx()`

Sign-extend a number with a certain number of bits.

`get_opa()` and `get_opb()`

`get_opa()` returns the 16-bit value of `opa` by looking at the binary instruction code. The function also checks if it is legal to access this register in this clock cycle by looking at `rf_busy`. `get_opb()` works similarly but may also return an immediate value instead of a value from the register file.

`set_reg()`

This function writes a value to the specified register in the register file. You must also specify in how many clock cycles the user is allowed to use the value. (Basically, how many `nop` instructions the user will have to issue between the current instruction and the instruction which is trying to use the written value.)

`sr_read()` and `sr_write()`

These functions are used to read and write the special registers.

`abs16()`

Takes the absolute value of a 16-bit two's-complement number.

`insn_...()`

All functions that implement the execution of a certain instruction group begins with `insn_`. For example, `insn_iterative_op` implements all iterative instructions.

`advance_pc()`

Advances the program counter one step taking into account both jumps, delay slots and loops.

`run_insn()`

Fetches and executes one instruction. The main simulation loop calls this function until an error is flagged or a specified number of instructions have been executed.

`load_images()`

This function is responsible for loading the example images used by the SAD. (In a production simulator this would not be hard coded but configurable through a configuration file.)

`mem0_read()` and `mem1_read()`

There are a number of support functions in files besides `sim.c`. For the lab, the most important support functions are located in `memory.c`. `mem0_read()` is used to read a specified memory address in memory 0. `mem1_read()` is used to read a specified memory address in memory 1.

4.4.2 Instruction Decoding in the Simulator

The instruction decoding starts in `run_insn()` which is using a switch statement to look at the type field of the instruction to decide which kind of instruction it is. For example, when a program flow instruction is encountered `insn_pfc()` is called, which in turn is using a switch statement to decide what kind of PFC instruction it is dealing with.

You can follow the program flow for accelerated instructions to figure out where to add your changes when implementing `accel_sad`.

4.5 `Makefile` Targets

The following targets are defined in the `Makefile`:

- `sim`: Build the simulator
- `clean`: Remove all generated files
- `test`: Assemble `sad.asm` and run the simulator on the resulting hex file
- `prof_report`: Generate a profiling report which outputs how many times each line in the source code was executed.

4.6 Lab Preparation Tasks

Investigate and understand the structure of `sim.c`.

Draw a hardware schematic where you show how your SAD unit could be implemented. (Do not forget the signal to the PC FSM for `repeat_sad`!) You need this figure to know how many registers there are in the SAD unit and where they are located in the pipeline before you can finish the lab.

Draw a pipeline table where you show what happens in the various pipeline stages when using your `accel_sad` instruction (based on the hardware schematic you have drawn) in conjunction with `repeat_sad`. For an example of what a pipeline table looks like, see Tables 14.5–14.7 in the textbook.

Use the pipeline figure and your hardware schematic to figure out the delay values you will use in `set_reg()` and `repeat_sad_stop`.

The lab assistant will ask you to show and explain your drawings during the lab examination!

4.7 Lab Tasks

In this lab you do not have to modify the RTL code at all, we will instead modify a simple instruction set simulator by adding a couple of instructions to it. You should only have to modify the function responsible for executing accelerated instructions in `sim.c`.

The simulator outputs the results in `results.hex`. The output when running your accelerated assembly program should be identical to the output of an unmodified `sim.c` and `sad.asm`. It is a good idea to run the simulator immediately before making any modifications to `sim.c` and `sad.asm` and save `results.hex` as `reference.hex`.

You can compare two files in a shell prompt using the command

```
shell> diff file1 file2
```

If there are no differences, this command will print nothing. Otherwise, the differences will be printed to the terminal.

The most important task of this lab is to evaluate the `accel_sad` and `repeat_sad` instructions in a realistic fashion. This means that the delay values that you use in the `set_reg()` and the `repeat_sad_stop()` function should be based on your modifications to the processor pipeline. If you set these delay values too large, your evaluation of these instructions is likely to be too pessimistic. On the other hand, if you set these values too small, your evaluation will be too optimistic.

Note that checking your `results.hex` for correctness is not a guarantee that your delay values are correct.

You will have to figure out the delays yourself by knowing where the SAD unit is located in the pipeline, as described in Section 4.3.1.

Support for the `repeat_sad` instruction is almost finished. The easiest way to get it working is to call `repeat_sad_stop()` if appropriate when executing the `accel_sad` instruction. (Once again with the correct value...)

After implementing these instructions, you should modify `sad.asm` to take advantage of the new instructions and measure the speed improvement of using `accel_sad` alone and in conjunction with `repeat_sad`.

Do not forget that you must write appropriate values to `sr30` and `sr31` in the appropriate place around the SAD kernel when modifying `sad.asm`.