Chapter 4

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.)

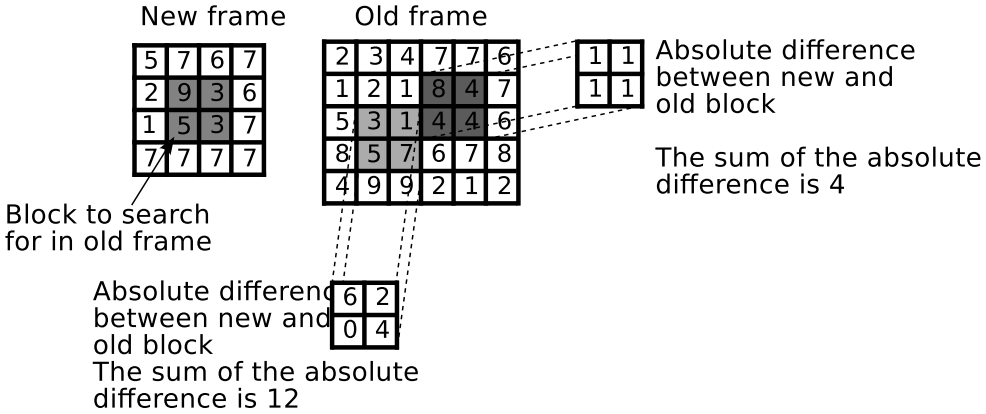

Figure 4.1: Block search using sum of absolute difference

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 Figure 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 176x144 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)

```
for each block in the image{ // 4x4 blocks
  best_sad = Inf;
  for each candidate position{
    sad = compare_blocks(candidate_block, target_block);
    if (sad < best_sad) {
      best_sad = sad;
      best_block = candidate_block;
    }
  }
  output_position(best_block);
}

// This is the kernel algorithm which we are interested in
compare_blocks(a,b){
  sum = 0;
  for each pixel p { // 16 pixels
    difference = a[p] - b[p];
    sum += abs(difference);
  }
  return sum;
}
```

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 4x4 pixels and does an exhaustive search for the best match in a large region around each 4x4 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`)

```
        repeat  sad_kernel_end,16
sad_kernel_start
        ld0     r0,(ar3++)      ; Load displacement in image
        nop
        ld0     r2,(ar0,r0)     ; Load pixel in original image
```

```
6           ld1     r1,(ar1,r0)     ; Load pixel in new image
7           nop
8
9           sub     r1,r1,r2        ; Calculate difference
10          abs     r1,r1           ; Take absolute value
11
12          add     r4,r4,r1        ; Sum of absolute difference
```

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 4x4 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:

```
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 should be implemented in the Senior processor by putting it in the pipeline stage after the memory access. Full register forwarding should 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 should combinationally look at the difference between `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 a number of global variables used by the simulator to keep track of the values of the various registers in the processor. Perhaps the most important is `rf` and `rf_busy`. The later 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:

```
set    r9,53
nop
add    r2,r9,r3
```

Most of the other global variables should be pretty self explanatory.

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 function 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 should 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.)

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. Fpr example, when a 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 in the Makefile are defined:

- 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

1. Investigate and understand the structure of `sim.c`.
2. Draw a hardware schematic where you show how your SAD unit could be implemented. (Don't 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.
3. You should also draw a pipeline table[1] 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`.
4. 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 don't 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 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`.

4.7.1 Lab Hints

Don't forget that you have to write appropriate values to `sr30` and `sr31` in the appropriate place around the SAD kernel when modifying `sad.asm`.

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 don't care about the borders, as special care has to be taken not to search outside the borders

[1] For an example of what a pipeline table looks like, see Tables 14.5-14.7 in the textbook.

of the original image. While it is of course possible to implement code to deal with this we don't bother about this to make `sad.asm` easier to understand for you.

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 unix using the command `diff file1 file2`. If there are no differences this command will print nothing, otherwise the differences will be printed to the screen.