ASIP Laboratory - Session 7

Paul Georg Wagner Majd Mansour

June 28, 2017

Exercise 1: Creating the First CoSy Compiler

The first task of this session was to create a CoSy compiler for the existing basis CPU that has been used in the previous sessions. In order to do this, we copied the ASIPMeister project directory from the previous session and simply executed the `makeCoSy` script. At first, the script resulted in an error on the remote workstation that is used for the compiler creation, because the `~/.bash_profile` file that needs to be sourced was missing. After restoring this file, the compiler was generated without any more errors.

Exercise 2: Compiling and Simulating the Application

After the CoSy compiler for the basis CPU has been generated, it should be used for compiling C code for the architecture. For this, we first copied the provided `arrayloop.c` file to a new project directory. Then we compiled the C code using gcc and executed the resulting binary to get the correct output of this code that can be used for later comparison.

In order to compile the C code for the basis CPU using the created CoSy compiler, the required library files have to be present in the project directory. As it is beneficial to avoid copying of code, we used symbolic links to achieve this. Afterwards the `arrayloop.c` file can be compiled to assembly using `make sim`.

We then simulated the resulting assembly file using both dlxsim and ModelSim. Unlike the gcc version, when compiling for the FPGA the code uses the provided library to write the result on the LCD. However, both dlxsim and ModelSim redirect this to an output file. Ultimately the output for both dlxsim and ModelSim was the same as the gcc output. This indicates that the created CoSy compiler is working as desired.

Exercise 3: Extending the CPU with a custom instruction

The main part of this session consisted of extending the basis CPU with three new instructions, namely (1) `avg rd, rs0, rs1`, (2) `swap rd, rs1` and (3) `minmax rd0, rd1, rs0, rs1`.

1. Implementing `avg rd, rs0, rs1`

   The `avg rd, rs0, rs1` instruction should calculate the average of two registers `rs0` and `rs1` and store the result in register `rd`. In dlxsim this instruction is defined as an `ARITH_3PARAM` instruction of the R_R instruction format. As opcode the value `0x17 = 000 0001 0111` is used in the `func` field.

   In order to implement this instruction, we first created a new instruction definition in the ASIPMeister project with the respective values (see figure 1). The instruction implementation is done by using the ALU in the execution phase to add the two input registers `rs0` and `rs1`, and then using the dedicated shifter (SFT0) resource to shift the result one bit to the right. This corresponds to a division by two, which results in the (integer) average of the two input values. As usually, the result is written back to register `rd` in the WB phase. The micro operation description is given in figure 2.

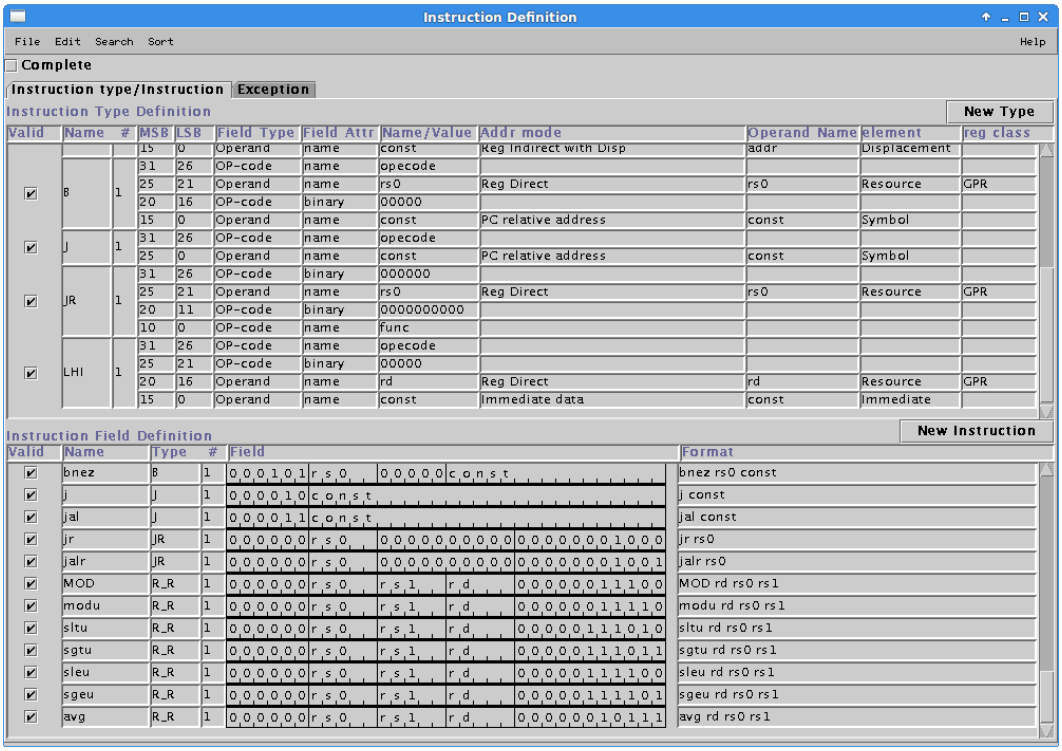

Figure 1: Instruction definition of `avg rd, rs0, rs1`.

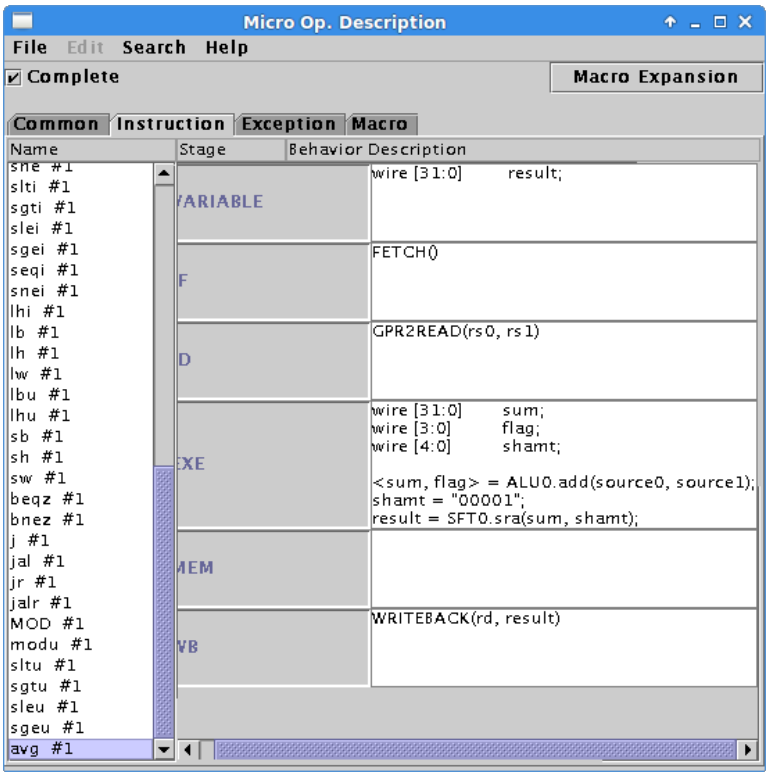

Figure 2: Micro operation definition of `avg rd, rs0, rs1`.

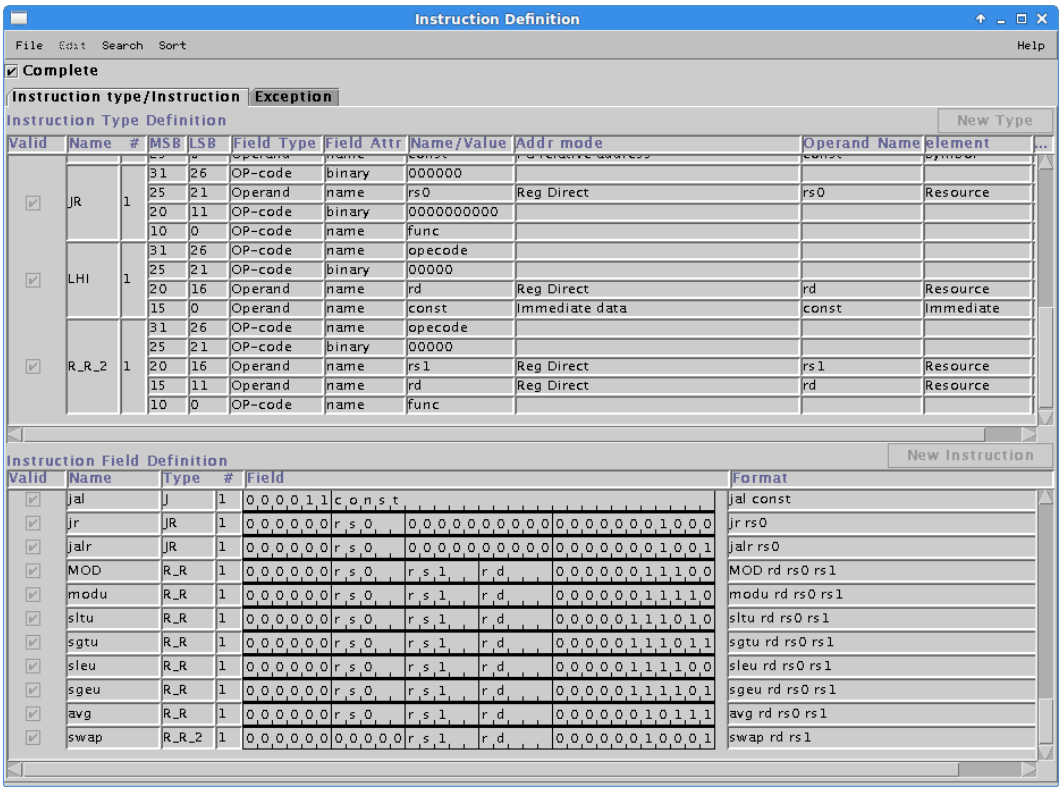

Figure 3: Instruction definition of `swap rd, rs1`.

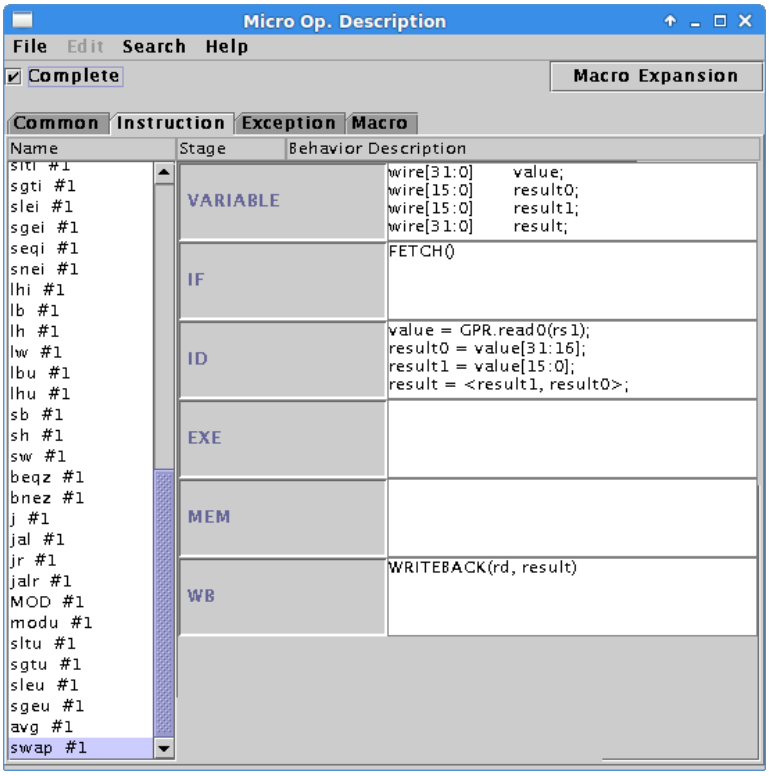

Figure 4: Micro operation definition of `swap rd, rs1`.

2. Implementing `swap rd, rs1`

   The `swap rd, rs1` instruction should swap the highest and lowest 16 bits of the input register `rs1` and store the result in register `rd`. In dlxsim this instruction is defined as an `ARITH_2PARAM` instruction. As opcode the value `0x11 = 000 0001 0001` is used in the `func` field.

   Since in the ASIPMeister project the only existing R_R instruction format required two input registers, we had to create a new instruction format R_R_2 without the `rs0` field. Then we could define the `swap rd, rs1` instruction (see figure 3). The micro operation implementation of this instruction is shown in figure 4 and simply consists of loading the input register `rs1` and swapping the bits. This does not even require the ALU, hence the EXE phase for this instruction is empty.

3. Implementing `minmax rd0, rd1, rs0, rs1`

   Finally, the `minmax rd0, rd1, rs0, rs1` instruction was implemented, which stores the minimum and maximum of two input registers `rs0` and `rs1` in the output registers `rd0` and `rd1`. The opcode for this instruction is defined as the value `0x1F = 000 0001 1111` in the `func` field. However, since we need to define a fourth register operand in the instruction format, we cannot use the entire eleven bits for the `func` field. Instead we decided to define a new instruction format R_R_3 that – unlike the R_R and R_R_2 format – uses a distinct opcode in the most significant six bits and omits the `func` field altogether. Hence the resulting R_R_3 format consists of six bits opcode and four times five bits for the four operand registers. The remaining least significant six bits are set to 0 (see figure 5). The new `minmax` instruction of this R_R_3 format then uses the unused opcode `110000`.

   In order to properly implement this instruction, we need to be able to write back two values at once in the WB phase. With the old register file definition this was barely possible, since only one write port has been defined. Hence we modified the GPR register file and added a second write port, giving us access to a new `GPR.write1()` function. The micro operation implementation of the `minmax` instruction then consists of reading the input values, using the ALU to compare the values and write back the two values in the WB phase. The implementation is shown in figure 6.

4. Testing the new instructions

   In order to test the new instructions, we first used small test programs written directly in assembly. We compiled the test programs using `make sim` and then simulated them using ModelSim. The outputs of the test programs showed that the implemented instructions work as desired.

   Then we replaced the `avg`, `swap`, `min` and `max` macros that have been defined in `arrayloop.c` with SINAS inline assembly in order to force the CoSy compiler to use the new instructions when compiling this program. The used inline assembly is shown in listing 1. After compiling the modified `arrayloop.c` with `make sim`, the created assembly file contained the new CPU instructions. The compiled project has then been simulated using ModelSim, and once more the resulting output matched the original output. Hence the new CPU instructions also work when being generated by the CoSy compiler.

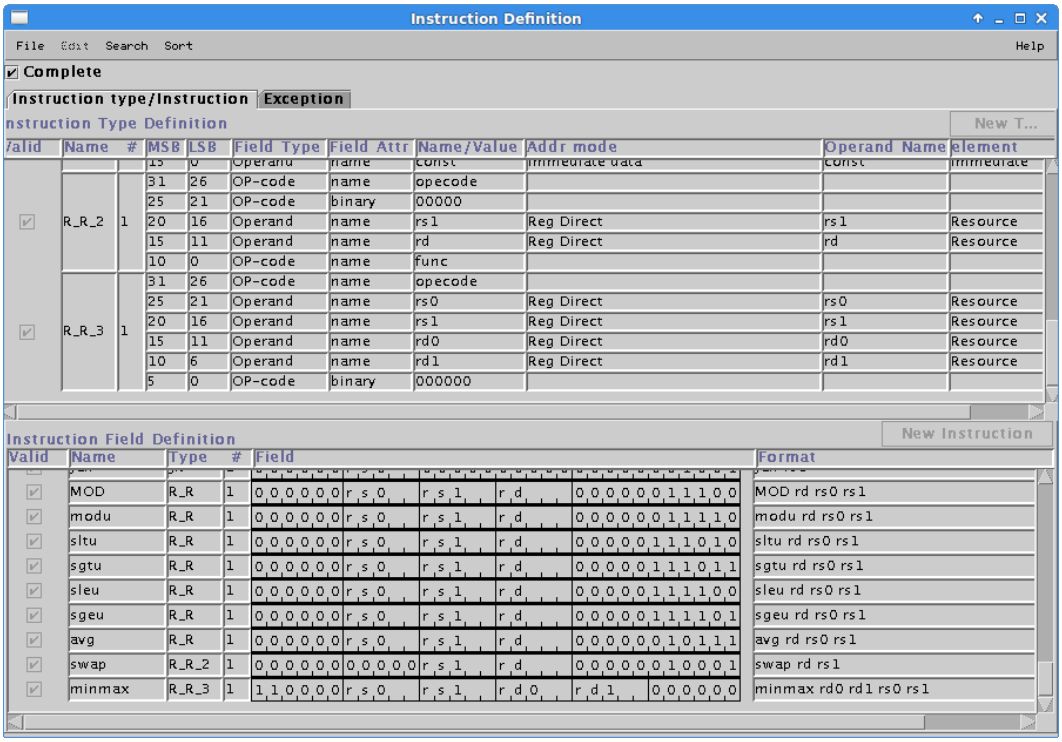

Figure 5: Instruction definition of `minmax rd0, rd1, rs0, rs1`.

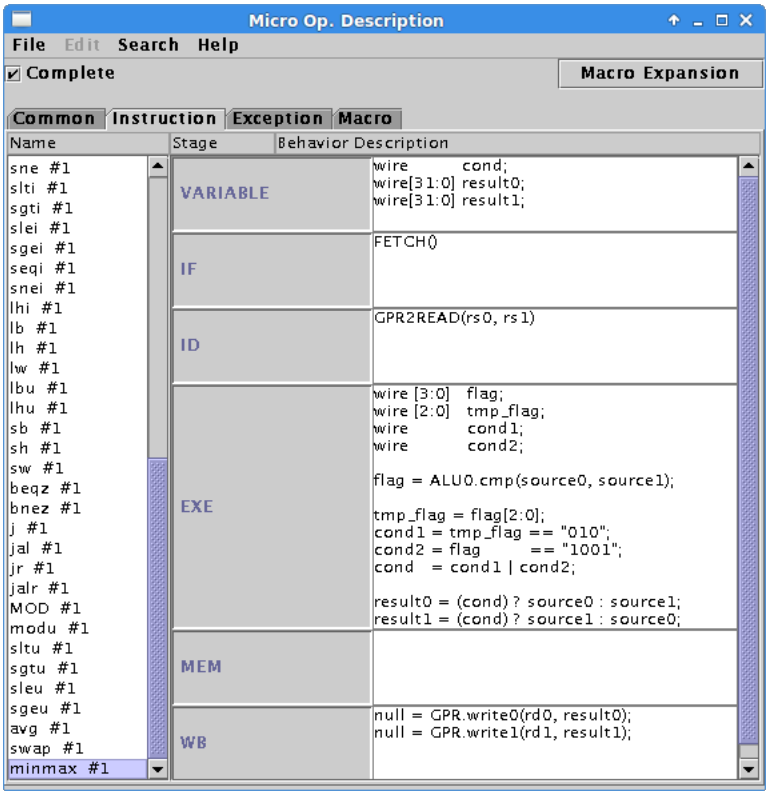

Figure 6: Micro operation definition of `minmax rd0, rd1, rs0, rs1`.

Listing 1: SINAS inline assembly in modified arrayloop.c

```
asm int avg(int a, int b) {
    nop nop nop
    avg @{}, @{a}, @{b}
    nop nop nop
}

asm int swap(int a) {
    nop nop nop
    swap @{}, @{a}
    nop nop nop
}

asm int min(int a, int b) {
    @[
    .barrier
    .scratch temp
    .restrict temp:reg
    ]
    nop nop nop
    minmax @{}, @{temp}, @{a}, @{b}
    nop nop nop
}

asm int max(int a, int b) {
    @[
    .barrier
    .scratch temp
    .restrict temp:reg
    ]
    nop nop nop
    minmax @{temp}, @{}, @{a}, @{b}
    nop nop nop
}
```

5. Benchmarking the new instructions

   Finally, the amount of CPU cycles should be determined for the original (dlx_basis) and the modified (dlx_avg) version of `arrayloop.c`, in order to calculate the speedup achieved by implementing the new CPU instructions. This requires to remove the for-loop that controls the output in `arrayloop.c`, since it generates a lot of additional instruction cycles, biasing the final result. Furthermore we tested different CoSy compiler optimization levels (O0 to O4) and used both dlxsim and ModelSim to determine the amount of required instruction cycles to finish the program. The results of the benchmark are displayed in table 1.

   As usual, the ModelSim simulation generally shows a slightly different amount of cycles than the dlxsim simulation. Also the used compiler optimization level has a great impact on the number of required instructions, bringing it down from almost 20.000 cycles to barely 4.000 in the original and from 15.000 to 3.000 in the optimized version. Interestingly however, the highest optimization level of O4 even has a slightly negative influence on the instruction count, making the optimization levels of O2 and O3 the most efficient for this problem. Most importantly however, the optimized versions using the new CPU instructions all require less cycles than the original versions. This results

Optimization level		Cycles dlx_basis	Cycles dlx_avg	Speedup
-O0	ModelSim	19796	14894	1,33
	dlxsim	19811	14909	1,33
-O1	ModelSim	19796	14894	1,33
	dlxsim	19811	14909	1,33
-O2	ModelSim	4116	2869	1,43
	dlxsim	4111	2864	1,44
-O3	ModelSim	4116	2869	1,43
	dlxsim	4111	2864	1,44
-O4	ModelSim	4121	3164	1,3
	dlxsim	4116	3159	1,3

Table 1: Benchmark of original and optimized `arrayloop.c` using different optimization levels.

in speedups ranging from 1,3 with optimization level O4 to a maximum of approximately 1,4 with optimization level O2 and O3.