Session:5 - Adding New Instructions

Customized Embedded Processor Design

Application Specific Instruction-Set Processors- ASIP
Lab (Prakitikum)

Responsible/Author:
MSc. Sajjad Hussain

Supervisors:
MSc. Sajjad Hussain, Dr.-Ing. Lars Bauer, Prof. Dr.-Ing. Jörg Henkel

Chair of Embedded Systems,
Building 07.21, Haid-und-Neu-Str. 7,
76131 Karlsruhe, Germany.

November 2, 2023

Adding New Instructions

1 Week

Motivation and introduction

In this session, we will implement some custom instructions for an application to speed up the execution time. Moreover, even when the compiler uses the new instructions, they might not be used in all optimization levels. For that, we will also introduce the feature, which is used to add inline assembly to the application. By using inline assembly, you can force the usage of custom instructions or you can optimize bigger blocks (e.g. application hot spots) in hand written assembler. For every part, that starts like “a)”, “b)” . . . you have to mail the answers and asked files/tables to **sajjad.hussain@kit.edu** and use the topic “asipXX-Session4”, with XX replaced by your group number.

Exercises

1. **Preparing your project**

   1.1 You can use the same project as in the last session, and just create a separate application subdirectory for each example. You can start a fresh project as in the Session 1, but this would be time consuming.

   1.2 For the C application, you have to create subdirectory in the “*Application*” directory (e.g. “arrayloop”), and copy your application from “*/home/ces-asip00/Sessions/Session5/loop.c*” to here.

   1.3 Copy a “*Makefile*” file from the “*TestPrint*” application subdirectory to each application subdirectory.

   1.4 Set proper parameters and settings in “*env_settings*.

   1.5 For these exercises, we will be using pipeline forwarding (-pf1) option which is the default one.

   1.6 Make sure that you already have VHDL files and GNU tools in your project’s meister directory.

2. **Compiling and Simulating the Application**

   2.1 First, you have to compile the application using gcc compiler to compare with the later results from dlxsim and ModelSim. To compile with GCC, comment the line “*#define ASIP*” and compile the application like “*gcc loop.c*”. For *gcc* you can forward the printed output to a file, e.g. “*a.out > output_gcc.txt*” (‘*a.out*’ is the default name of the binary that is created when you compile a program. The file ‘*output_gcc.txt*’ will contain the printed array.

   2.2 Now, compile “loop.c” using “make sim”. However, for compiling it, you first need to provide the required libraries, i.e. “*lib_lcd_dlxsim”* (also for dlxsim/*ModelSim*), “*loadStoreByte”*, and “*string”*.

2.3 After compiling, simulate the application in *dlxsim* and *ModelSim* and compare whether the printed results are the same compared to a *gcc*-compiled version. The *gcc* version will print the arrays on the screen and *dlxsim* and *ModelSim* will print them to a *virtual* LCD. For *dlxsim* you can forward the LCD output to a file, using the "*-lf*" parameter, e.g. "*make dlxsim DLXSIM_PARAM="-da0 –pf1 -lf**output_dlxsim.txt***" writes output to the file "*output_dsim.txt*". *ModelSim* automatically writes to the file "*lcd.out*".

2.4 To compare, whether the files generated from gcc, dlxsim & ModelSim are identical, you can use command-line tools like "*diff output_gcc.txt output_ModelSim.txt*" or graphical tools like "*kompare*" or "*kdiff3*".

2.5 Print statements should be commented out for a fair comparison. Then simulate in dlxsim and ModelSim.

(a) How many cycles do you need for execution in dlxsim and ModelSim (without printing)?

3. Adding a new instruction to *dlxsim Simulator*

1. Create a project directory for this session by copying the directory "*/home/ces-asip00/-epp/ASIPMeisterProjects/TEMPLATE_PROJECT/*" and renaming it (e.g. *brownieAVG*).
2. You have to create another subdirectory for our application in the "*Application*" directory (e.g. "arrayloopAVG"). Copy "loop.c" here and then you start putting new instructions here. Print statements should be comment out for a fair comparison.
3. But before modifying "loop.c", create a simple C or assembly file to test different custom instruction (e.g. AVG) into an application subdirectory i.e. *TestAVG*.
4. Copy a "*Makefile*" file from the "*TestPrint*" application subdirectory to each application subdirectory.
5. Copy the provided *ASIPMeister* CPU file "*browstd32.pdb*" from "/home/ces-asip00//Sessions/Session into your project directory, and rename it "*browstd32AVG.pdb*"
6. Set proper parameters and settings in "*env_settings*" as discussed in Figure 2-5 in the Laboratory Script. Specially the followings:

```
export PROJECT_NAME=brownieAVG
export CPU\_NAME=browstd32AVG
export ASIPMEISTER_PROJECTS_DIR=${HOME}/ASIPMeisterProjects
export DLXSIM_DIR=/home/ces-asip00/epp/dlxsimbr_Laboratory
```

7. Now we start adding new instructions to our processor to speed up the execution. These new instructions are "*avg rd, rs0, rs1*", "*swap rd, rs*", "*rot rd, rs0, amount*" and "*minmax rdMin, rdMax, rs0, rs1*". First, implement the new instructions into *dlxsim*, as explained in the Chapters 3.2.2 and 3.2.3 of the Laboratory Script. Use the instruction format and opcodes as below:

```
OpcodeInfo opcodes[]
//name     class                      op        mask                other flags    rangeMask
{"avg",        ARITH_3PARAM, 0x6c1,    0x1ffff,  0x20,    0 ,       0xffff8000 },
{"swap",      ARITH_2PARAM, 0x541,    0x1ffff,   0x20,    0 ,       0xffff8000 },
{"minmax", ARITH_4PARAM, 0x981,  0xfff,                0x20,    0 ,       0xffff8000 },
{"rot",    ARITH_3PARAM,  0x8,    0x3f,              0,  CHECK_LAST|IMMEDIATE_REQ, 0xffff8000 },
{"bgeu",      BRANCH_2OP, 0x11,   0x3f,  0,        0 ,      0 }***
*** Not needed in this session.
```

8. Therefore, you have to copy *dlxsim* to your local home (to be able to modify it) and you have to configure the "*env_settings*" to use your local *dlxsim* (see Figure 2-5 in the Laboratory Script).

(b) Write a small assembly code to test your new instructions in dlxsim. You can use the Session2 assembly language program as the reference.

4. **Extending the CPU with a custom instruction**

   4.1 In your new CPU, implement the new instruction "*avg rd, rs0, rs1*", "*swap rd, rs*", "*rot rd, rs amt*" and "*minmax rdMin, rdMax, rs0, rs1*" as they are used in the application. This new instruction "*minmax*" computes both the minimum and the maximum of two inputs *rs0* and *rs1* and write them simultaneously to two registers (*rdMin* and *rdMax*).

   Hint:

   i. You **can use the opcode and instruction formats as indicated in the figure below.**
   ii. Do not implement the "*swap"* instruction as it is written in the C-code. Think what this instruction is doing and implement it without any shifts! Test the new instructions with a small assembly code in *ModelSim*.
   iii. For instructions that return two values, you need to change # of GPR write ports. Normally, only one register is forwarded from EXE or WB stage, now you have to add new resources (Forwarding units) to forward two register values.
   iv. Remember, that you cannot use a hardware resource twice in the same cycle, e.g. you cannot use the ALU twice in the EXE stage. Additionally, using it in two different pipeline stage significantly complicates the whole CPU design (just think about the required wiring).
   v. Remember that your new instruction has to support forwarding as well.

   4.2 Generate the VHDL Files.

   4.3 Please remember that for new custom instructions defined in ASIPmeister to be used automatically with C Compiler, you have to implement relevant "**CKF Prototype**" in ASIPmeister. Follow instructions at section 4.11.C in ASIPmeister tutorial and 11.2 in ASIPmeister user manual.

   4.4 Generate GNU Tools for your new processor.

5. **Compiling and Simulating the Application with custom instructions**

   5.1 You can test your custom instructions with small assembly programs.

   5.2 After testing the new instruction with a small assembly code, use *inline assembly* in the application "*loop.c*" for using the new instructions, see Chapter 8.2.3 in the Laboratory Script.

   5.3 There are two methods to insert a explicitly insert a custom instruction inside your C code:

   i. Using __builtin_brownie32__ABC directive: This method has a bug while using with the instruction that has zero return values or instructions that return more than one value. Examples:

```
Int a,b,c,d;
c = __builtin_brownie32_AVG(a,b);
d = __builtin_brownie32_SWAP(a);
```

   ii. Using __asm__ directive: This can be used for all the cases.

```
Branch Instruction
Int a,b,c,d,e;
__asm__ volatile (
"bgeu %[my_op1], %[my_op2], _here2\n"
"_there2: sub %[my_out], %[my_op1], %[my_op2]\n"
"_here2: add %[my_out], %[my_op1], %[my_op2]\n"
: [my_out] "=&r" (e)
: [my_op1] "r" (a),[my_op2] "r" (b)
);
__asm__ volatile (
"minmax %[my_out1], %[my_out2], %[my_op1], %[my_op2]\n\t"
: [my_out1] "=&r" (c), [my_out2] "=&r" (d)
: [my_op1] "r" (a), [my_op2] "r" (b)
);
```

5.4 After modifying the application code by the *inline assembly* stuffs for a particular custom instruction, compile the application using “*make sim*”, make sure that the result is still correct (*diff* the *ModelSim* output with gcc-compiled version) and find out, whether the new instructions have been used or not.

Note: you have to check the generated assembly code to be sure that the new custom instructions are used in the code.

5.5 Now you have to determine the number of cycles for executing the application to compute the speedup against the old CPU with the old compiler. To determine the number of cycles you have to remove (i.e. comment out) the loop for printing the results! Otherwise, this loop is the dominating hotspot and you will not notice a significant speedup when using the new assembly instructions!

5.6 Simulate the application with ModelSim.

(c) How many cycles do you need for execution in dlxsim and ModelSim? Attach assembly programs you used to test custom instructions with this mail.

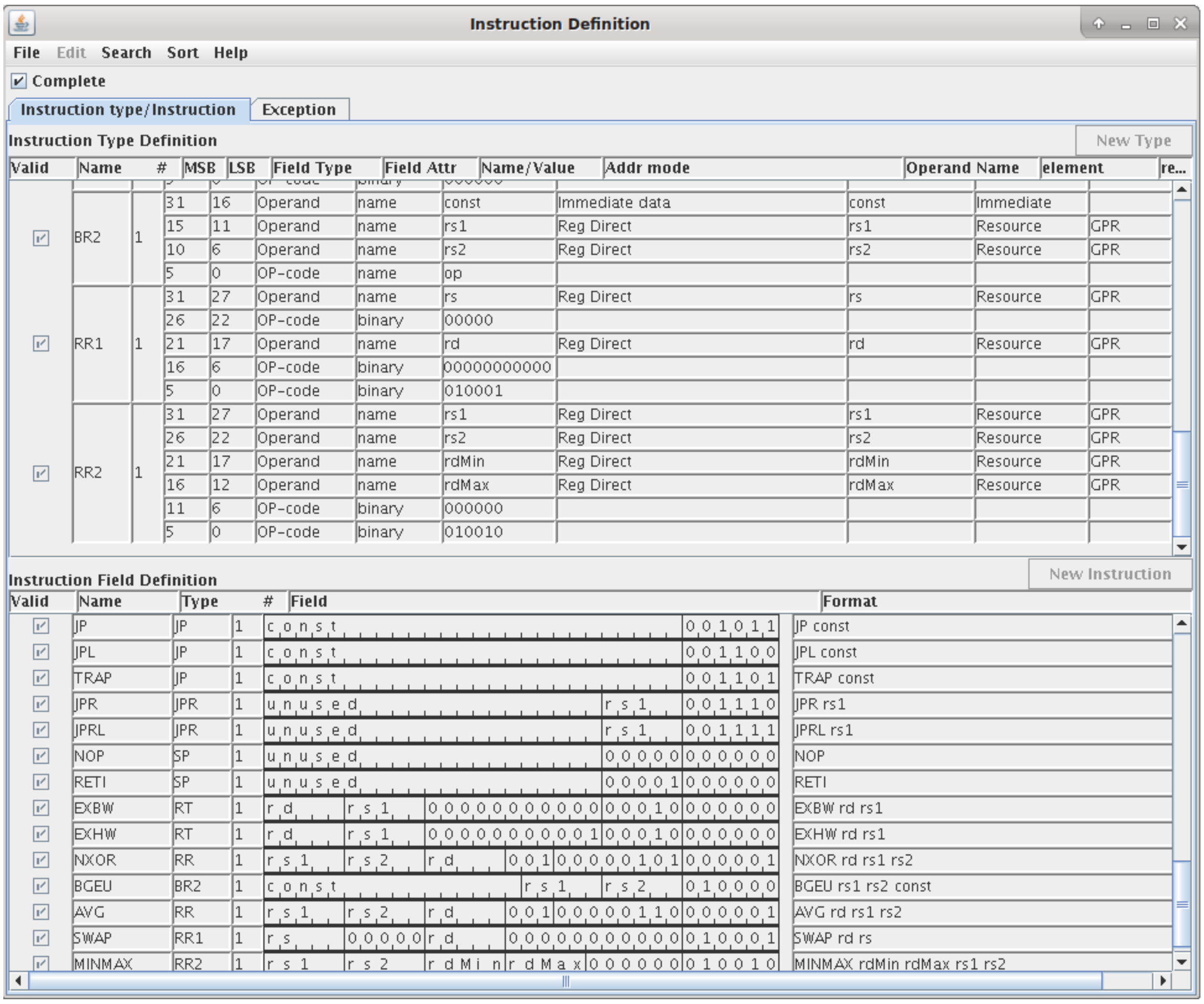
(d) What is the speedup (i.e. #Cycles without custom instructions / #Cycles with custom instructions)?

(e) Attach the assembly and/or C file you used to test the custom instruction and to test the given application.

(f) Attach the modified ASIPmeister.pdb file.

Next Session: Bubble Sort – Simulations and Optimization

Readings for the next session: All the Chapters, especially 4 & 8, ASIPMeister Tutorial & Manual



Instruction Type Definition

Valid	Name	#	MSB	LSB	Field Type	Field Attr	Name/Value	Addr mode	Operand Name	element	re...
✓	BR2	1	31	16	Operand	name	const	Immediate data	const	Immediate	
			15	11	Operand	name	rs1	Reg Direct	rs1	Resource	GPR
			10	6	Operand	name	rs2	Reg Direct	rs2	Resource	GPR
			5	0	OP-code	name	op				
✓	RR1	1	31	27	Operand	name	rs	Reg Direct	rs	Resource	GPR
			26	22	OP-code	binary	00000				
			21	17	Operand	name	rd	Reg Direct	rd	Resource	GPR
			16	6	OP-code	binary	00000000000				
			5	0	OP-code	binary	010001				
✓	RR2	1	31	27	Operand	name	rs1	Reg Direct	rs1	Resource	GPR
			26	22	Operand	name	rs2	Reg Direct	rs2	Resource	GPR
			21	17	Operand	name	rdMin	Reg Direct	rdMin	Resource	GPR
			16	12	Operand	name	rdMax	Reg Direct	rdMax	Resource	GPR
			11	6	OP-code	binary	000000				
			5	0	OP-code	binary	010010				

Instruction Field Definition

Valid	Name	Type	#	Field	Format
✓	JP	JP	1	const | 001011	JP const
✓	JPL	JP	1	const | 001100	JPL const
✓	TRAP	JP	1	const | 001101	TRAP const
✓	JPR	JPR	1	unused | rs1 | 001110	JPR rs1
✓	JPRL	JPR	1	unused | rs1 | 001111	JPRL rs1
✓	NOP	SP	1	unused | 00000 | 000000	NOP
✓	RETI	SP	1	unused | 00001 | 000000	RETI
✓	EXBW	RT	1	rd | rs1 | 00000000000 | 00010 | 000000	EXBW rd rs1
✓	EXHW	RT	1	rd | rs1 | 00000000001 | 00010 | 000000	EXHW rd rs1
✓	NXOR	RR	1	rs1 | rs2 | rd | 001 | 00000101 | 000001	NXOR rd rs1 rs2
✓	BGEU	BR2	1	const | rs1 | rs2 | 010000	BGEU rs1 rs2 const
✓	AVG	RR	1	rs1 | rs2 | rd | 001 | 00000110 | 000001	AVG rd rs1 rs2
✓	SWAP	RR1	1	rs | 00000 | rd | 00000000000 | 010001	SWAP rd rs
✓	MINMAX	RR2	1	rs1 | rs2 | rdMin | rdMax | 000000 | 010010	MINMAX rdMin rdMax rs1 rs2

Figure 1: Instruction Definitions and OpCodes