KIT-Fakultät für Informatik
Institut für Technische Informatik

Session:2 - Assembly Programs

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.

August 7, 2022

Assembly Programs

1 Week

Motivation and Introduction

The main goal of this session is to get used to assembly programs. Every example shows a basic operation. Examine the code, simulate it with *dlxsim* and answer the corresponding questions. The assembly code is available in the directory "*/home/asip00/Sessions/Session2/*". *Dlxsim* is available in the directory "*asip00/ epp/dlxsimbr_Laboratory/" (default)*. You can copy *dlxsim* to your directory or you can start it from the default directory. For every exercise, the part that starts like "a)", "b)" ... you have to write an answer. This answer should mainly prove that you have understood the problem, so you can make your answers short, but they still have to answer everything, that was asked. Mail your answer to **sajjad.hussain@kit.edu** and use the topic "asipXX-Session1", with XX replaced by your group number.

Exercises

The following exercise correspond to the given assembly files in the "*/home/asip00/Sessions/Session1*" directory.

0. **Preparing your project**

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

   0.2 For each application (C or Assembly), you have to create subdirectories in the "*Application*" directory.

   0.3 Copy a "*Makefile*" file from the "*TestPrint*" subdirectory to each application subdirectory.

   0.4 Set proper parameters and settings in "*env_settings*.

   0.5 For these exercises, we will run basic assembly programs in dlxsim and see the effect of pipeline forwarding (-pf1) or no forwarding (-pf0), like

```
make dlxsim DLXSIM_PARAM="-da0 -pf0"
```

1. **Basic Assembly Instructions**

   1.1 Understand the functionality of every instruction and understand the values of every target register after the program run has completed (see the dlxsim chapter for reading register values in dlxsim simulation). Check the number of cycles needed to execute all instructions.

   (a) What is the reason for this high number of cycles? Which instruction causes that behaviour and why is it doing so?

(b) What is the code/data address space? What is starting and ending address of code/data section?

2. **Memory Access**

(a) Explain the goal of the instruction combination *LSOI /ADDI*. What is generally (so: not only for this specific example) the register value after *ADDI*, what is it after the additional *LSOI*?

(b) Why is it in general not possible to omit the *LSOI* instruction, although it would be possible in this special example?

(c) Read about "forwarding" and "load delay slot" in brownie32 datasheet. Disable NOP after the load instruction, and try executing with –pf0 and –pf1 both.

3. **Branches**

(a) Which high-level control structure (e.g. 'call subroutine' . . .) is implemented in this example code?

(b) What is computed with this example? R24 = *function* (R21, R22);

(c) Look at the *NOP* instructions and explain why they are placed there.

4. **Loops**

(a) What is computed with this example? R23 = *function* (R21, R22). Debug the application step-by-step with the capabilities of dlxsim.

(b) The approach to compute the function in the way this example is doing it has two specific names. Do you know those names? (One is founded by the main operations, while the other is founded historically. You either know the names or you don't. If you don't know both names, you may guess)

(c) In general, how often is the loop maximally executed? How the input data has to look like to get this maximal number of iterations?

(d) Enable and disable the first NOP, and try executing with –pf0 and –pf1 both.

5. **A High Level Structure**

(a) Which high-level control structure do you recognize? Explain the purpose of the instruction-block between "*ADDI R23, R0, $(2)*" and "*JPR R24*".

(b) Why do you have to shift by value 3? Explain it with a close view to the body of the control structure and pay attention to the addressing mode of the DLX processor.

(c) What are the general differences between branch and jump instructions in the DLX instruction set (also have a look at the different instruction formats to find a part of the answer)?

6. **ASM Directive in C**

6.1 You can use assemble instruction directly in c as follows [1,2]:

```
int res=0;
int in1=20;
int in2=30;
int main() {
        __asm__ volatile (
        "add    %[out],         %[op1],         %[op2]\n"
        : [out] "=&r" (res)
```

```
        : [op1] "r" (in1),[op2] "r" (in2)
        );
        return 0;
}
```

6.2 In any asm block, assembly instructions appear first, followed by the inputs and outputs, which are separated by a colon. The assembly instructions can consist of one or more quoted strings. The first colon separates the output operands; the second colon separates the input operands. If there are clobbered registers, they are inserted after the third colon. If there are no clobbered inputs for the asm block, the third colon can be omitted, as Listing 2 shows.

```
int A[10] = {45,23,0,0,0,0,0,0,0,0};
int main() {
        __asm__volatile (
        "add    %[out],         %[op1],         %[op2]  \n"
        : [out] "=&r" (A[2])
        : [op1] "r" (A[0]),     [op2] "r" (A[1])
        );
        return 0;
}
```

6.3 There is no output operand for the sw instruction, hence the outputs section of the asm is empty. None of the registers is modified, so they are all input operands, and the target address is passed in with the input operands. However, something is modified: the addressed memory location.

```
int res [] = {0,0,0,0,0,0,0,0};
int a=45;
int main() {

        __asm__ volatile  (
        " sw    %1(%2), %0          \n"
        :
        : "r"(a), "i"(sizeof(int)),"r"(&res)
        );

        return 0;
}
```

6.4 Branching can be tricky with inline asm. Using labels, the branch-to address can be designated with a unique identifier that can be used as a target branch address.

```
int x, y, z;
int a=45, b=23, c=1;
int main() {
__asm__ volatile (
"               addi    %[out1],        %[op1],         %[op2]                  \n"
"               brnz            %[op3], here                            \n"
"  there:       add     %[out2], %[op1],        %[op2]                    \n"
"  here:        mul     %[out3], %[op1],        %[op2]                          \n"
: [out1] "=&r" (x),     [out2] "=&r" (y),       [out3] "=&r" (z)
: [op1] "r" (a),        [op2] "r" (b) , [op3] "r" (c)
:  "r0" );
        return 0;
}
```

6.5 See "A guide to inline assembly for C and C++" for further details.

(a) Write a C program using assembly instructions to load two consecutive values from an array and store their addition or subtraction at the fourth location of an array depending on the third value of array i.e. 0 means addition and 1 means subtraction.

Next Session: ModelSim Simulation

Readings for the next session: Chapters 2.3, 4 & 5

[1]. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

[2].https://dmalcolm.fedorapeople.org/gcc/2015-08-31/rst-experiment/how-to-use-inline-assembly-language-i
html