Engineering 304

Lab #2: NIOS CPU – Machine Code & Assembled vs. Compiled Code

Purpose
The purpose of this one-week lab is to learn about machine codes and compare hand-written assembly and C-code-compiled assembly code. The microprocessor will be an Altera NIOS II (soft core) microprocessor.

Overview
In this lab, you will work with machine codes and write a program to fill an array with the elements of the Fibonacci sequence.

Part I – Machine Codes
In this part of the lab, you will be working with the "machinecode.s" assembly file which is essentially the "sampleprog.s" file with an extra instruction. A spreadsheet file is available for doing the analysis work on converting assembly to machine code as directed below.

A. Download the provided files from Moodle or the S: drive, place them in a new directory for this lab's project on your H: drive.
B. Begin by examining the NOP (add r0, r0, r0) instruction. Using the information in the NIOS CPU Reference Manual, Section II.8 (found on the shared drive or Moodle), determine the binary pattern (machine code) for the NOP instruction, particularly how the various fields are encoded.
    - Record your work in the "NIOS Instruction Analysis" spreadsheet.
C. Determine the true equivalent instruction for the pseudo instruction "movi r10, 23". Create the binary pattern for this instruction. (Hint page 8-67).
    - Record your work in the spreadsheet.
D. Power-up and setup a DE2 board using the system files from Moodle or the S: drive. Start up the debugger with a new project and then compile and load the machinecode.s file onto the DE2 board (use an Assembly project, ignore the .c file for now). Test the program to see if it adds the array properly. Refer to Lab SW01 if you want a refresher on how to setup an Altera project and download the NIOS CPU image to the board.
    - Enter the calculated total value in the spreadsheet.
E. Next, enter the hex number corresponding with the machine code for the MOVI instruction into the memory tab of the debugger so that you replace the machine code for the NOP instruction with the new code for the MOVI instruction (from Step C). You will need to know where in memory the NOP instruction had been placed and then you will replace it by double-clicking on the machine code for the NOP instruction and typing the hex digits as the new machine code. You will not see anything change in the disassembly tab, but you will be able to see the effect of the MOVI instruction as r10 is changed by executing the new instruction. To rerun the program, use the "restart" action. If you re-"load" the program, you will wipe out the changes you made to the NOP instruction in the memory tab.
    - The debugger should now have the modified instruction substituted into its memory.
F. Determine the code for the instruction "beq r9, r10, PROG_END" which will replace the instruction "beq r9, r0, PROG_END." Notice that PROG_END is just a label or simply an address where the last instruction of the program is located. A branch operation adds an offset to PC+4 to make the program counter point to the desired instruction. For example, if your branch

instruction is found at location 12 and your branch target is at location 36, then the offset would need to be 20 so that the PC would be changed to 12+4+20=36. It is the offset that is stored in the immediate field of the branch instruction. What offset would be needed to jump from the location of the BEQ instruction to the location of the instruction at the PROG_END label? It may be helpful to examine the disassembly tab in order to complete this part.

- Record the offset and machine code in the spreadsheet.

G. Enter the new machine code for the BEQ instruction as a replacement of the instruction "beq r9, r0, PROG_END" using the memory tab.

- The debugger should now have the modified instruction substituted into its memory.

H. What effect will putting this instruction have on your program? What will the final summation be for this latest version of your program? Test it out and record the results.

- Record the results in the spreadsheet.

I. Extra Credit: Using the memory-edit feature of the memory tab, change the last branch instruction to point to the start of the program (MAIN_PROG_INIT) instead of to itself (PROG_END). Verify it works. Hint: you will have to calculate a new branch target offset that is negative (in twos-complement format).

- Record for extra credit submission:
  i. Assembly code replacing "br PROG_END" for start of program
  ii. Hex and Binary machine code for the new instruction
  iii. Explanation of how this changes the program execution

Part II – Create and Run a Fibonacci Program

Your goal for this part of the lab is to write a simple program that will fill a 40-element integer data array with the values of Fib(0) through Fib(39). The sequence Fib(n) is defined such that Fib(0)=0, Fib(1)=1, and for n > 1, Fib(n) = Fib(n-1) + Fib(n-2). Perform the following set of steps:

A. Create a pseudo-code program or a flow chart for generating this sequence. It is advised that Fib(0) and Fib(1) are initialized outside a loop and the remaining Fib(n) values are generated in a loop (either for or while), by accessing the previously determined values. Do not use recursion at this time.

- Record your non-recursive / loop-based pseudo code for future submission

B. Use manual analysis or research to determine the expected values for Fib(0) to Fib(39). It is important to know the correct results for comparison to your own solution – this comparison can verify the accuracy of your program.

- Record the expected values of the Fibonacci sequence for testing

C. Create an assembly program for generating this sequence, compile the program, and test it. Note: the "sampleprog.s" program from last lab is one that reads from a global array found in memory and should be a good example to follow for handling memory operations and updating the global pointer to sequence through an array in memory. Instead of using the ".word" assembler directive to create and initialize your array in data memory, you should use the ".skip N" directive that allocates N bytes of data memory without initializing the values.

- Verify correctness and save your Assembly Fibonacci program for submission

D. Create a C program for generating this sequence, compile the program, and test it. Note: you should create your fib array as a global variable, not local to the main program and you should not include any I/O (references to stdio, printf, cout, etc). All of the looping and fib calculations can be done in the "main()" function. For this lab, do not create any additional functions besides the main() function. A template file called "main.c" is provided in the folder as a starting point. You will need to change the program type to

"C Program" instead of "Assembly" when you setup the project in the debugger. You will use the debugger to check your results.

- Verify correctness and save your C Fibonacci program for submission

E. Recompile your C-code, but this time specify different optimization levels using the "-Ox" compiler option (the compiler options are found in debugger program setup dialog box - written as "dash - capital o" followed by a number). Compare your original compiled code with the code from the various optimization levels.("-O1" with "-O0", and "-O6").

F. Compare the number of instructions executed by your main function for the four versions of the code (your hand-written assembly, -O0, -O1, and –O6). Clearly document how you arrived at your answer (#non-loop_instructions + (loop_count.*#loop_instructions))

Version	# non-loop instructions	# loop instructions	Total executed in the loop code	Total main function instructions executed
Assembly				
-O0				
-O1				
-O6				

Hand In (a printed copy if possible):

Part I:

- The excel worksheet that includes the updated sum resulting from replacing the BEQ instruction.
- If you did the extra credit, add a page describing how your determined the new branch instruction.

Part II:

Combine the following in a Word document (be sure to label the items in the document):

- A printout of your pseudo code/flowchart (this may also just be comments in your code)
- A printout of your assembly code (see printing instructions from the previous lab)
- A screenshot of your assembly-code debugging session disassembly window (use "snip")
- A screenshot of the memory showing the Fib array for the assembly version (include the memory addresses in your screenshot)
- A printout of your C code
- Screenshots of your C-code debugging sessions which show the disassembly of your C-code for various optimization levels
- A screenshot of the memory showing the Fib array for your C -O1 version
- Your calculations of the number of instructions executed in each version of the code (see above table example) – this can be done on the disassembly screen shots so you can identify the loops and the non-loop code.
- The completed table for Part II comparing the difference C optimization levels and Assembly
- A written comparison of the disassembled C code and your hand-written assembly that addresses questions like:
    - Which is better?
    - How much code does each use (# of instructions)?
    - How many instructions are executed in each?
    - What is different about register use?
    - Are there differences in how the loop structure is built?
    - Did you hand-write assembly code which was better than the highly optimized compiled code?