

EE 533 Lab #6

Quad HW-Threaded ARM ISA Compatible Processor Core in NetFPGA

Young Cho - youngcho@isi.edu

Part 1: ARM ISA-compatible Processor

Integrate additional functional units to your processor core Datapath from the end of the previous lab to execute RISC instructions. This processor must incrementally support a subset of ARM ISA that can run assembly programs generated by an ARM C compiler. ISA should include all the instructions that use the functions defined by your ALU, branch instructions, and load/store instructions (if needed).

Ultimately, you must write and execute a simple bubble sort program for your processor that sorts N numbers randomly arranged in the data memory and demonstrates the correctness of the processor using NetFPGA.

An example bubble sort program is as follows:

sort.c

```
int main() {
    int array[10] = {323, 123, -455, 2, 98, 125, 10, 65, -56, 0};
    int i, j, swap;

    for (i = 0 ; i < 10; i++) {
        for (j = i+1 ; j < 10 ; j++) {
            if (array[j] < array[i]) {
                swap = array[j];
                array[j] = array[i];
                array[i] = swap;
            }
        }
    }
}
```

ARM C/C++ cross-compiler Installation

First, you must install and use ARM C/C++ compiler freely available here:

<https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads>

There are several versions of the compiler for different operating systems. Download and install one compatible with your development computer or virtual machine based on your preference. Once you successfully install the software, you should be able to easily get to a terminal that allows you to run the ARM gcc compiler.

For example, starting the script in Windows 11 version of the ARM compiler v14.2.1 will open a terminal to execute the compiler by typing “arm-none-eabi-gcc-14.2.1.exe”. The names of the compiler will differ depending on the version.

Assembly code for Bubble Sort

Write a bubble sort C program like the example above and save it to a folder where the gcc is executable. Then translate the program into an ARM assembly program with command similar to the following:

```
> arm-none-eabi-gcc -S sort.c
```

If all goes well, there should be a newly generated equivalent assembly program named “sort.s”. This is the assembly program that you must convert into machine code, like the following example, which your processor should be able to execute.

The goal of this step is to understand what the compiler generates and how it is used by the hardware. Test out the compiler to see what the output assembly code looks like. Write a program that compares assembly code and the generated binary code and allows you to reverse engineer the association between sections of the code.

Since you control your design, you are not constrained to anything (or at least not too many.) So, make your design as ambitious as you can and want. You can even have vector instructions. This is a computer architecture course, NOT A REAL-LIFE situation where you are usually motivated by cost and performance, so be creative, but make sure you can complete your design by the end of the due dates.

You may study the ARM ISA to identify and use the fields for all your instruction types. Otherwise, you may come up with your fields. Your approach to your design doesn’t matter as long as the core executes the ARM compiler-generated code correctly. On the other hand, your report must specify the ISA fields and how they are used to execute the assembly code. Furthermore, you must submit details on how assembly programs are converted into machine codes using your code generation scripts and programs. You must describe and demonstrate how your analysis scripts/program works.

At the end of this step, your team must identify a subset of ISA that your processor needs to integrate to execute a generated simple bubble sort ARM assembly program.

```

.cpu arm7tdmi
.arch armv4t
.fpu softvfp
.eabi_attribute 20, 1
.eabi_attribute 21, 1
.eabi_attribute 23, 3
.eabi_attribute 24, 1
.eabi_attribute 25, 1
.eabi_attribute 26, 1
.eabi_attribute 30, 6
.eabi_attribute 34, 0
.eabi_attribute 18, 4
.file "sort.c"
.text
.section .rodata
.align 2
.LC0:
.word 323
.word 123
.word -455
.word 2
.word 98
.word 125
.word 10
.word 65
.word -56
.word 0
.text
.align 2
.global main
.syntax unified
.arm
.type main, %function
main:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame =
56    @ frame_needed = 1,
uses_anonymous_args = 0
    push {fp, lr}
    add fp, sp, #4
    sub sp, sp, #56
    ldr r3, .L8
    sub ip, fp, #56
    mov lr, r3
    ldmia lr!, {r0, r1, r2, r3}
    stmia ip!, {r0, r1, r2, r3}
    ldmia lr!, {r0, r1, r2, r3}
    stmia ip!, {r0, r1, r2, r3}
    ldm lr, {r0, r1}
    stm ip, {r0, r1}
    mov r3, #0
    str r3, [fp, #-8]
    b .L2
.L6:
    ldr r3, [fp, #-8]
    add r3, r3, #1
    str r3, [fp, #-12]
    b .L3
.L5:
    ldr r3, [fp, #-12]

```

```

    lsl r3, r3, #2
    sub r3, r3, #4
    add r3, r3, fp
    ldr r2, [r3, #-52]
    ldr r3, [fp, #-8]
    lsl r3, r3, #2
    sub r3, r3, #4
    add r3, r3, fp
    ldr r3, [r3, #-52]
    cmp r2, r3
    bge .L4
    ldr r3, [fp, #-12]
    lsl r3, r3, #2
    sub r3, r3, #4
    add r3, r3, fp
    ldr r3, [r3, #-52]
    str r3, [fp, #-16]
    ldr r3, [fp, #-8]
    lsl r3, r3, #2
    sub r3, r3, #4
    add r3, r3, fp
    ldr r2, [r3, #-52]
    ldr r3, [fp, #-12]
    lsl r3, r3, #2
    sub r3, r3, #4
    add r3, r3, fp
    str r2, [r3, #-52]
    ldr r3, [fp, #-8]
    lsl r3, r3, #2
    sub r3, r3, #4
    add r3, r3, fp
    ldr r2, [fp, #-16]
    str r2, [r3, #-52]
.L4:
    ldr r3, [fp, #-12]
    add r3, r3, #1
    str r3, [fp, #-12]
.L3:
    ldr r3, [fp, #-12]
    cmp r3, #9
    ble .L5
    ldr r3, [fp, #-8]
    add r3, r3, #1
    str r3, [fp, #-8]
.L2:
    ldr r3, [fp, #-8]
    cmp r3, #9
    ble .L6
    mov r3, #0
    mov r0, r3
    sub sp, fp, #4
    @ sp needed
    pop {fp, lr}
    bx lr
.L9:
    .align 2
.L8:
    .word .LC0
    .size main, .-main
    .ident "GCC: (Arm GNU Toolchain
14.2.Rel1 (Build arm-14.52)) 14.2.1
20241119"

```

One suggestion is to build the register files out of a single block-memory. The upper 2 bits of the memory address for the block memory can be used in place of the Demultiplexer and Multiplexer in the above register file design, and the remaining bits can indicate the register number. Since there are only two read/write ports and you need the third for the writeback, you can simply double-clock the memory to read at one cycle and write during the other cycle.

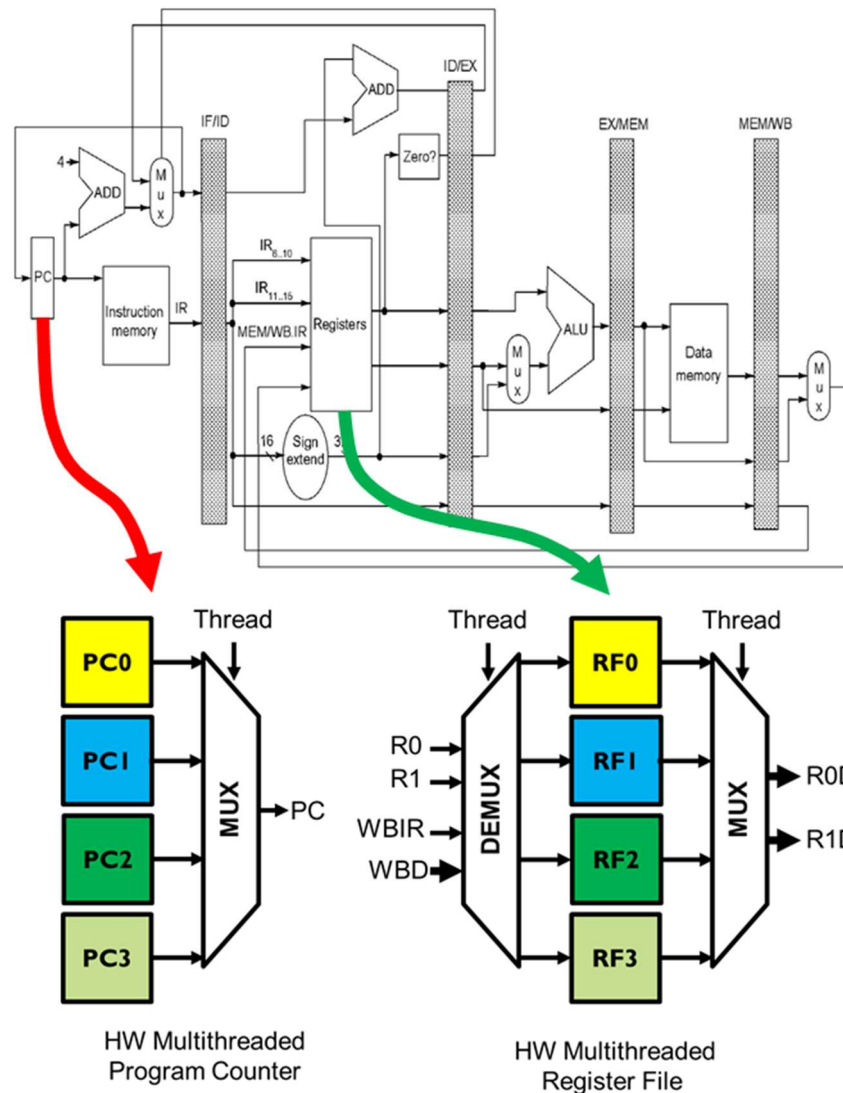


Figure 2: An example of a hardware modification for a Zero Overhead Multiple Threads

Software Verification

Write and compile four independent programs in C. Then execute them as four parallel threads and verify their operation. In your report, you will need to describe the design of your software and hardware to support the parallel execution of your programs. You will need to get the hardware working with your software on NetFPGA and demonstrate multiple network packet-processing threads running on your processing core.

1. Submission and Demonstration

- Draw a high-level design of the datapath. The figure should depict the communication between the components.
- Include the following in your report:
 - Screen Capture of Schematics
 - Generated Verilog Files
 - GitHub records and descriptions per team member

- The transcript of a sequence of commands typed to the interface
- The sort code as a readable assembly code
- The internal memory dump that shows the array data before and after sort program execution
- Software examples of Multiple threads
- Program and data upload/download procedures
- Screenshots of the results
- Demonstration YouTube video must also show your interface to your processor via software/hardware registers.