ICS Homework 4

April 20, 2020

1 Organization

1.1 Hazard

```
# demo.ys
0x000:    irmovq stack,%rsp
0x00a:    call p
0x013:    irmovq $5,%rsi
0x01d:    halt

0x020:.pos 0x20
0x020:p:  irmovq $-1,%rdi
0x02a:    ret  #below will not be executed
0x02b:    irmovq $1,%rax
0x035:    irmovq $2,%rcx
0x03f:    irmovq $3,%rdx
0x049:    irmovq $4,%rbx

0x100:.pos 0x100
0x100:stack:
```

1. During executing the above example, how many hazards will happen? Please point them out.
   SOLUTION:
   There are 3 hazards.
   Data hazard between "irmovq stack,%rsp" and "call p". %rsp is written by the previous instruction and is read by the latter instruction.
   Data hazard between "call p" and "ret". %rsp is written by the previous instruction and is read by the latter instruction.
   Control hazard due to the "ret" instruction.

2. How could the above data hazards be handled? Please describe in detail.
   SOLUTION:
   Data hazards are handled by forwarding.
   For the first data hazard, the call instruction at stage **Decode** will get the value of %rsp through **e_valE** in the **Execute** stage of irmovq instruction.
   For the second data hazard, the ret instruction at the **Decode** stage will get the value of %rsp through **M_valE** in the **Memory** stage of the call instruction.

3. What is the difference between **stall** and **bubble**?
   SOLUTION:
   A pipeline stall means input memory of a stage remains the same as the previous cycle. A pipeline bubble means input memory of a stage is the same as a nop instruction. When an instruction stalls in a stage, a bubble is injected into its subsequent stage.

1.2 Control Combination

1.2.1

Write a Y86-64 assembly-language program that causes combination A (Figure 4.67 in ICSAPP) to arise and determines whether the control logic handles it correctly.
SOLUTION:

```
# Code to generate a combination of not-taken
# branch and ret
        irmovq stack, %rsp
        irmovq rtnp, %rax
        pushq %rax      # Set up return pointer
        xorq %rax,%rax  # Set Z condition code
        jne target      # Not taken (First
                        # part of combination)
        irmovq $1,%rax  # Should execute this
        halt
target: ret             # Second part
                        # of combination
        irmovq $2,%rbx  # Should not execute
        halt
rtnp:   irmovq $3,%rdx  # Should not execute
        halt
.pos 0x40
stack:
```

This program is designed so that if something goes wrong (for example, if the ret instruction is actually executed), then the program will execute one of the extra irmovq instructions and then halt. Thus, an error in the pipeline would cause some register to be updated incorrectly. This code illustrates the care required to implement a test program. It must set up a potential error condition and then detect whether or not an error occurs.

1.2.2

Write a Y86-64 assembly-language program that causes combination B (Figure 4.67 in ICSAPP) to arise and completes with a halt instruction if the pipeline

operates correctly.
SOLUTION:
The following test program is designed to set up control combination B. The simulator will detect a case where the bubble and stall control signals for a pipeline register are both set to zero, and so our test program need only set up the combination for it to be detected. The biggest challenge is to make the program do something sensible when handled correctly.

```
# Test instruction that modifies %esp followed
# by ret
        irmovq mem,%rbx
        mrmovq 0(%rbx),%rsp # Sets %rsp to point
                            # to return point
        ret                 # Returns to
                            # return point
        halt
rtnpt:  irmovq $5,%rsi      # Return point
        halt
.pos 0x40
mem:    .quad stack         # Holds desired stack
                            # pointer
.pos 0x50
stack:  .quad rtnpt         # Top of stack: Holds
                            # return point
```

This program uses two initialized words in memory. The rst word (mem) holds the address of the second (stackthe desired stack pointer). The second word holds the address of the desired return point for the ret instruction. The program loads the stack pointer into %rsp and executes the ret instruction.

2 System Software

2.1 Signal handler

One TA writes the following code about a user-defined signal handler on a x86-64 Linux machine. Read this C program and answer the questions below.

```
#include <signal.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

void handler(int sig)
{
        printf("hello\n");
}
```

```c

int main(void)
{
        signal(SIGINT, handler);
        kill(getpid(), SIGINT);
        while(1);
        return 0;
}
```

1. Does the function 'handler' run in user mode or kernel mode?
   SOLUTION:
   User mode.

2. What's the output of this program? When we type a 'ctrl-c', what will happen? How to use *kill* command to stop this program?
   SOLUTION:
   hello
   When we type a 'ctrl-c', it will print one more 'hello'.
   First find the pid $\$pid$ of this program. And then use '$kill - 9\ \$pid$'

3. When we run this program in gdb, can we see the same result as Q2? If not, explain why and show how to fix it to get the same result in gdb as Q2.
   SOLUTION:
   No.
   Because gdb will stop the debugging program immediately whenever an error signal happens. Although SIGINT does not indicate an error in the program, it is normally fatal so it can carry out the purpose of the interrupt: to kill the program. Gdb will stop our program and not pass this signal to our program. (see: https://sourceware.org/gdb/onlinedocs/gdb/Signals.html)
   We can change these settings with the handle command like

   ```
   handle SIGINT nostop print pass
   ```