

# ICS Homework 4

March 18, 2021

## 1 Organization

### 1.1 Hazard

```
1 # demo .ys
2 0x000:    irmovq stack,%rsp
3 0x00a:    call p
4 0x013:    irmovq $5,%rsi
5 0x01d:    halt
6
7 0x020:.pos 0x20
8 0x020:p:  irmovq $-1,%rdi
9 0x02a:    ret  #below will not be executed
10 0x02b:    irmovq $1,%rax
11 0x035:    irmovq $2,%rcx
12 0x03f:    irmovq $3,%rdx
13 0x049:    irmovq $4,%rbx
14
15 0x100:.pos 0x100
16 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.

## 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.

```
1  #include <signal.h>
2  #include <sys/types.h>
3  #include <stdio.h>
4  #include <unistd.h>
5
6  void handler(int sig)
7  {
8      printf("hello\n");
9  }
10
11 int main(void)
12 {
13     signal(SIGINT, handler);
14     kill(getpid(), SIGINT);
15     while(1);
16     return 0;
17 }
```

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

```
1 handle SIGINT nostop print pass
```