4190.308: Computer Architecture (Fall 2018)

Project #5: Optimizing Performance on a Pipelined Y86-64 Processor

Due: December 16th (Sunday), 11:59PM

1. Introduction

In this project, you will learn how to optimize the performance of a program on a pipelined Y86-64 processor. Our target is the pipelined Y86-64 processor implementation called PIPE-Stall which does not support data forwarding. You need to optimize the `bmp_mosaic()` function written in Project #3 so that you can get the most out of the PIPE-Stall processor.

2. The PIPE-Stall processor

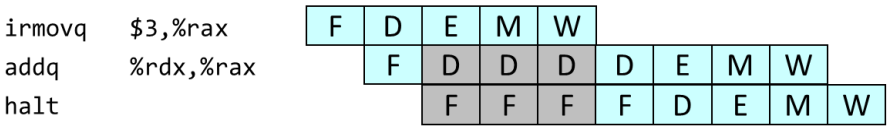
2.1 Supported instructions

In addition to the original Y86-64 instructions, the PIPE-Stall processor supports **`iaddq`**, **`mulq`**, **`divq`**, **`rmmovb`**, and **`mrmovb`** instructions you have implemented in Project #4.

2.2 Characteristics of the PIPE-Stall processor

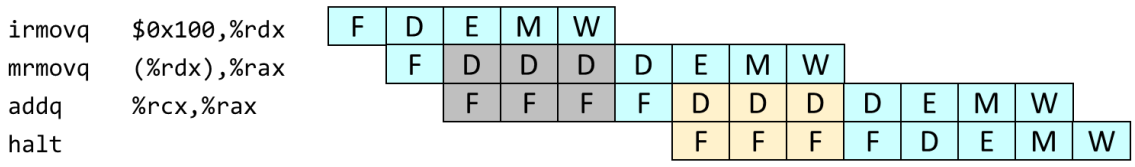
The original PIPE processor described in the textbook uses data forwarding whenever there are data dependencies among instructions. However, the PIPE-Stall processor stalls if there is a data hazard. Some of example cases are shown below.

2.2.1 Normal Data Hazard



Due to the data dependency on the `%rax` register, the pipeline is stalled for 3 cycles (gray boxes) until the `irmovq` instruction writes the value to the `%rax` register in the write-back stage.

2.2.2 Load / Use Data Hazard



The load/use data hazard is treated the same way as the data hazard shown in 2.2.1. The `addq` instruction is stalled for 3 cycles (yellow boxes) until the value read from memory is written into the `%rax` register. In the above example, note that the `mrmovq` instruction is stalled for 3 cycles as well (gray boxes), because there is a data dependency on the `%rdx` register with the previous `irmovq` instruction.

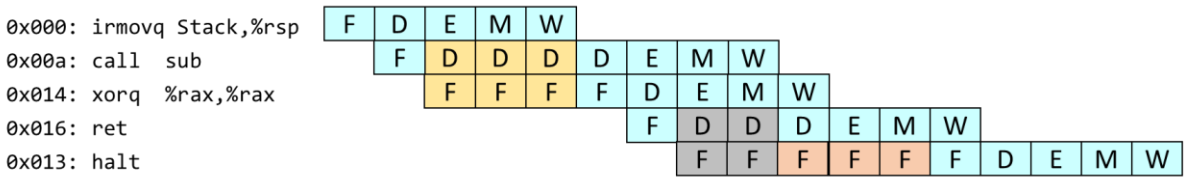
2.2.3 Procedure Call / Return

The `call` and `ret` instructions have a data dependency to each other as both require the access to the `%rsp` register. Also, they have data dependencies to other instructions that manipulate the `%rsp` register. Let us consider the following program.

```
0x000:     irmovq    Stack,%rsp
0x00a:     call      sub
0x013:     halt
0x014: sub:
0x014:     xorq      %rax,%rax
0x016:     ret

           .pos 0x100
0x100: Stack:
```

The above program will be executed in our PIPE-Stall processor as follows:



First, the `call` instruction is stalled for 3 cycles (yellow boxes) until the location of the stack is written into the `%rsp` register by the `irmovq` instruction. The `xorq` instruction in the procedure immediately follows the `call` instruction because we supply the address of "`sub`" (`valC` of the `call` instruction) to the next fetch stage.

Second, the `ret` instruction is stalled for 2 cycles (gray boxes) in the decode stage because it has a data dependency with the previous `call` instruction for the `%rsp` register. It cannot proceed until the `call` instruction writes the modified value to the `%rsp` register.

Finally, once the `ret` instruction resumes its execution, the fetch stage should be stalled until the return address is available (red boxes). The return address becomes available at the end of the

memory stage of the `ret` instruction, and this address is fed back to the fetch stage in the write-back stage of the `ret` instruction.

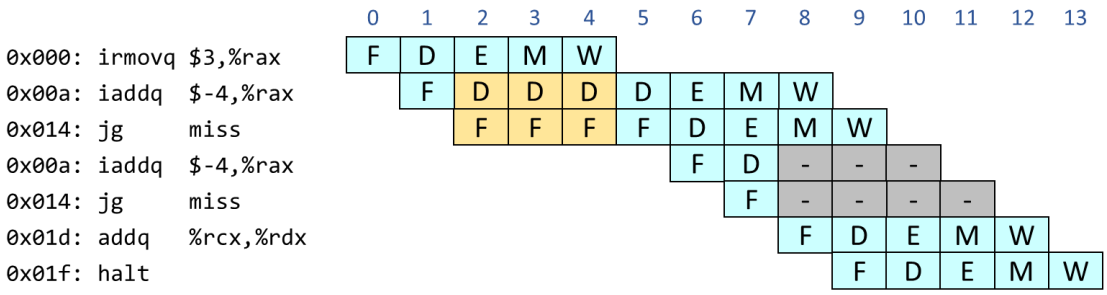
2.2.4 Mispredicted branch

The mispredicted branch is handled in the same way as the original PIPE processor. We also use the always-taken prediction, so the next instructions in the branch target are fetched immediately. The branch outcome is known at the end of the execute stage in the conditional branch instruction.

```
0x000:     irmovq    $3,%rax
0x00a: miss:
0x00a:     iaddq     $-4,%rax
0x014:     jg        miss
0x01d:     addq      %rcx,%rdx
0x01f:     halt
```

When the branch is mispredicted, the following two instructions are turned into the `nop` instructions. Consider the following example.

The following diagram shows how the above program is executed in our PIPE-Stall processor.



The `iaddq` instruction is stalled for 3 cycles (yellow boxes) due to the data dependency on the `%rax` register with the previous `irmovq` instruction. As soon as the `jg` instruction is fetched on cycle 5, the next `iaddq` and `jg` instructions are fetched on cycle 6 and 7, respectively, assuming that the conditional branch is taken. However, when the first `jg` instruction reaches the execute stage on cycle 7, it is known that the branch is not taken. Hence, two instructions fetched on cycle 6 and 7 are turned into the `nop` instructions on cycle 8. Meanwhile, the original `jg` instruction supplies the address of the next instruction in the memory stage so that the `addq` instruction is fetched on cycle 8.

3. Optimizing the performance of bmp_mosaic() on PIPE-Stall

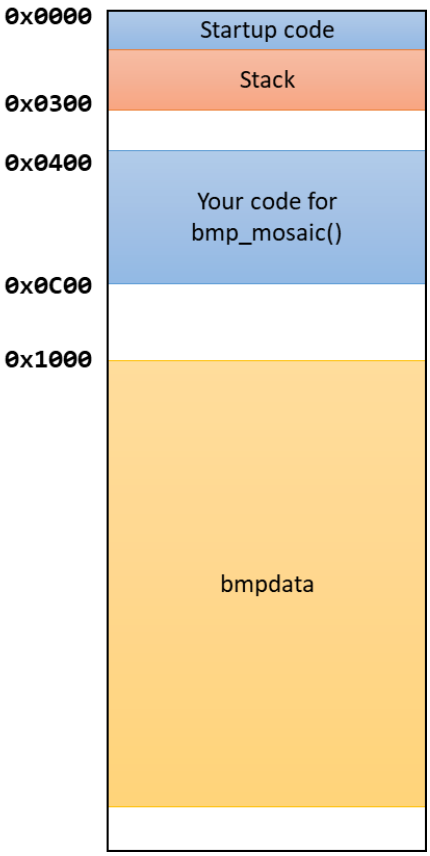
3.1 Rewriting `bmp_mosaic()` for PIPE-Stall

Your task is to rewrite the `bmp_mosaic()` function you have written in Project #3 to optimize its performance on the PIPE-Stall processor. The prototype of `bmp_mosaic()` is same as the one in Project #3:

```
void bmp_mosaic (unsigned char *imgptr, long long width, long long height,
                 long long size);
```

As in Project #3, four arguments are passed in `%rdi`, `%rsi`, `%rdx`, and `%rcx` registers, respectively. There is no limitation in the register use. You can freely use all the registers available in the Y86-64 architecture (e.g., `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, `%r8` ~ `%r14`). Remember that there is no `%r15` in Y86-64.

The following figure shows the memory layout when your program is running. When the power is turned on, the PIPE-Stall processor begins its execution by fetching an instruction at `0x0000`. The startup code first sets the stack pointer, initializes registers with arguments for `bmp_mosaic()`, and calls the `bmp_mosaic()` function which is located at `0x0400`. The image data is stored in a memory region starting at `0x1000`. Due to this layout, the maximum stack size is limited to about 709 bytes (`0x300` – startup code size).



The performance of your code will be measured by the total number of cycles to complete the given task. Note that our PIPE-Stall processor stalls for 3 cycles whenever there is a data dependency between instructions. Also, it has 2-cycle penalty for mispredicted branch and 3-cycle penalty for `ret` instruction. Considering these characteristics of the PIPE-Stall processor, you have to optimize the performance of `bmp_mosaic()`. You may make any semantics preserving transformations to the `bmp_mosaic()` function such as reordering instructions. You may also find it useful to read about **loop unrolling** in Section 5.8 of the textbook. Loop unrolling is a program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration.

3.2 Evaluation

To receive any credit here, your code must be correct first. The half of your credit (50 points) will depend on the test cases which check whether your code is correct or not. Once you pass all the test cases, you will get different amount of credits depending on the performance of your code. We will express the performance of your code in units of *cycles per pixels* (*CPP*). That is, if the simulated code requires *C* cycles to change *N* pixels in a BMP file, then CPP is *C*/*N*.

Since some cycles are used to set up the call to `bmp_mosaic()` and to set up the loops, you will get different CPP values for different combinations of image heights, image widths, and square sizes. We will therefore evaluate the performance of your function by computing the average of the CPPs for different parameters. If your average CPP is *c*, then your remaining credit *S* will be determined as follows:

c <= 20.0	*S* = 50 points + 10 points bonus
20.0 < *c* <= 25.0	*S* = 50 points
25.0 < *c* <= 30.0	*S* = 45 points
30.0 < *c* <= 35.0	*S* = 40 points
35.0 < *c* <= 40.0	*S* = 35 points
40.0 < *c* <= 50.0	*S* = 30 points
50.0 < *c* <= 60.0	*S* = 25 points
60.0 < *c* <= 70.0	*S* = 20 points
70.0 < *c* <= 80.0	*S* = 15 points
80.0 < *c* <= 100.0	*S* = 10 points
100.0 < *c* <= 150.0	*S* = 5 points
c > 150.0	*S* = 0 points

3.3 Verifying your code

You can use the sequential Y86-64 simulator (`ssim`) you have implemented in Project #4 to verify the logical correctness of your code. For this purpose, we provide a sample image data along with the corresponding simulator output (`result.out`). The output generated by `ssim` (with the option `-s`) for the given image data should match the content of the `result.out` file. (Try "`make test`" to compare the result.) The actual number of cycles taken on the PIPE-Stall processor will be available when you run your code on the grading server. (We also encourage you to implement the PIPE-Stall simulator by modifying the `pipe-full.hcl` file. It was one of the project assignments last semester!)

3.4 Restrictions

- The code size of `bmp_mosaic()` should be less than or equal to 2048 bytes. The `bmp_mosaic()` function starts at the address `0x400`. Therefore, the address of your code should be within `0x0C00`.
- There is no restriction in the register usage. You can freely use any of Y86-64 registers. Also, you can use stack for temporary storage, but the maximum stack size is limited to 709 bytes.
- The total number of cycles in the PIPE-Stall simulator is set to 10,000,000 cycles. If your program runs longer than this limit, it will be terminated.
- Your `bmp_mosaic()` implementation should work for BMP images of any size.
- Your `bmp_mosaic()` implementation should work for any positive value of "`size`" less than image width & height.
- Your `bmp_mosaic()` implementation should leave the bytes in the padding area untouched.
- This time, the total number of submissions is limited to 50 times.

4. Skeleton codes

The following skeleton codes are provided for this project.

`Makefile`	The main `Makefile` for this project. You need to set YAS and SSIM to point to the locations of `yas` and `ssim`, respectively (Use `yas` and `ssim` in Project #4).
`bmpmain.ys`	The Y86-64 assembly file which contains the startup code and the sample image data.
`bmpmosaic.ys`	The Y86-64 assembly file for implementing `bmp_mosaic()`. You should submit this file.
`result.out`	The sample output. When you give the "`-s`" option to the simulator, it will automatically dump the memory locations whose values are changed into the file named `memory.out`. The contents of `memory.out` should be identical to this file. (cf. run "`make test`")

5. Hand in instructions

- You only need to submit the `bmpmosaic.ys` file to the grading server at http://sys.snu.ac.kr

6. Logistics

- You will work on this assignment alone.
- If you have any questions, please feel free to post them in the QnA board.
- Only the assignments submitted before the deadline will receive the full credit. 25% of the credit will be deducted for every single day delay.
- You can use up to 5 ***slip days*** during this semester. Please let us know the number of slip days you want to use in the QnA board in the submission site within 5 days after the deadline.
- Any attempt to copy others' work will result in heavy penalty (for both the copier and the originator). Don't take a risk.

This is your last project. Thanks for all your hard work during this semester.

Jin-Soo Kim
Systems Software & Architecture Laboratory
Dept. of Computer Science and Engineering
Seoul National University