4190.308: Computer Architecture (Spring 2018)

Project #5: Optimizing performance on a pipelined Y86-64 processor

Due: June 19th (Tuesday), 11:59PM

1. Introduction

In this project, you will learn about how to design and implement a pipelined Y86-64 processor and how to optimize the performance of a program on it. This project is organized into two parts. In Part A, you will implement the simulator of the pipelined Y86-64 processor which does not support data forwarding (called PIPE-Stall). In Part B, you need to optimize the `bmp_grid()` function written in Project #4 so that you can get the most out of the PIPE-Stall processor.

2. Part A: Implementing the PIPE-Stall processor

2.1 New instructions

First, you should extend the pipelined Y86-64 processor to support new instructions, `iaddq`, `mulq`, `rmmovb`, and `mrmovb`, as in Project #4. Their behaviors and encodings are exactly same as in Project #4.

2.1.1 `iaddq : rB ← rB + V`

```
iaddq V, rB
```

The `iaddq` instruction adds the 64-bit constant value `V` to register `rB`. The condition codes should be set accordingly.

2.1.2 `mulq : rB ← rB * rA`

```
mulq rA, rB
```

The `mulq` instruction multiplies `rA` and `rB` and stores the result to register `rB`. The condition codes should be set accordingly.

2.1.3 `rmmovb` : M_1`[rB+D] ← LSB(rA)`

```
rmmovb rA, D(rB)
```

The `rmmovb` instruction stores the LSB (Least Significant Byte) of the register `rA` into the memory address `rB + D`, where `D` is a 64-bit constant. No condition codes are affected.

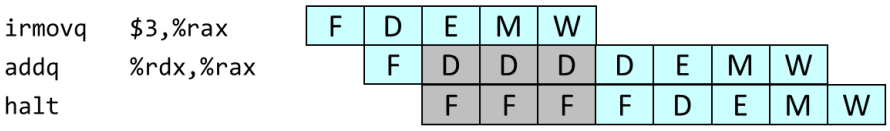
2.1.4 `mrmovb` : `rA` ← `Sign_Extend(M1[rB+D])`

The `mrmovb` instruction reads a single byte from the memory address `rB + D` and stores it into the LSB (Least Significant Byte) of the register `rA`. Other remaining bits in register `rA` are set to the sign bit of the original value read from the memory. No condition codes are affected.

2.2 Stalling the pipeline in PIPE-Stall

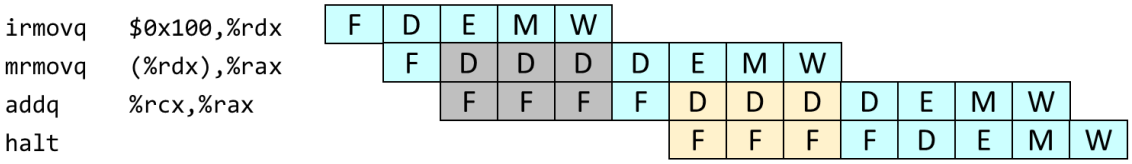
The original PIPE processor described in the textbook uses data forwarding whenever there are data dependencies among instructions. However, your task is to make the pipeline work without using any data forwarding. Our pipelined Y86-64 processor PIPE-Stall stalls whenever there is a data hazard. Some 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.

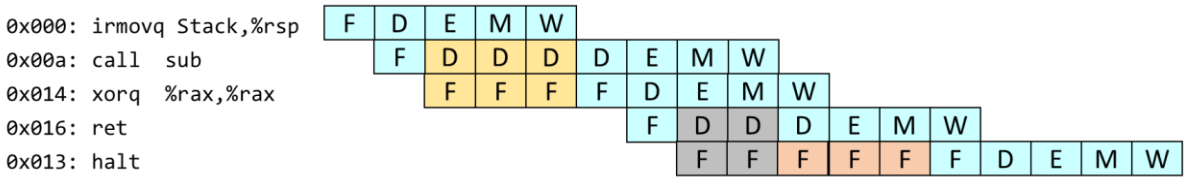
2.2.3 Procedure Call / Return

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.

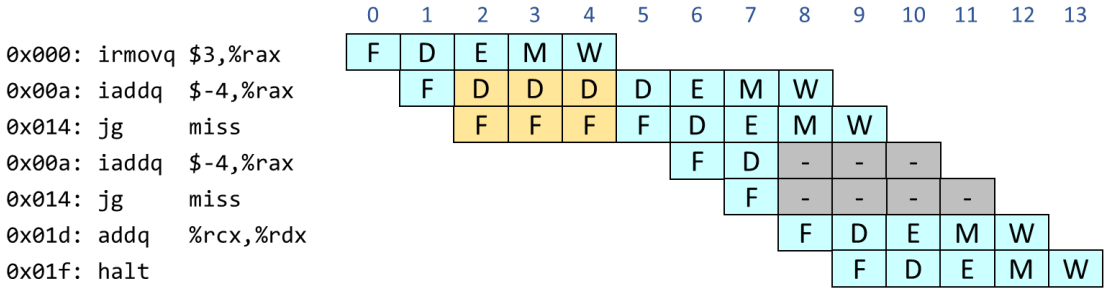
2.2.4 Mispredicted branch

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

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

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.

2.3 Implementing the PIPE-Stall simulator

For Part A, you will be mainly working in `./misc` and `./pipe` directories. To implement the PIPE-Stall processor, you need to change the following files:

`./misc/isacore.c`	This file contains the codes that simulate the operation of each instruction. In order to support `mulq`, `rmmovb`, and `mrmovb` instructions, you need to change this file. However, you can reuse the `isacore.c` file used in Project #4.

`./pipe/psimcore.c`	This file implements the core logic of the PIPE-Stall simulator. This file should be also changed to support `mulq`, `rmmovb`, and `mrmovb` instructions in the same way as you've done in Project #4.
`./pipe/pipe-stall.hcl`	This file describes the control logic of the PIPE-Stall simulator in HCL (Hardware Control Language). To stall the pipeline, you need to change the pipeline control logic at the end of this file. For your reference, the control logic of the full PIPE simulator (with data forwarding) is available in the `./pipe/pipe-full.hcl` file. You can start from this file to understand the implementation of the pipelined Y86-64 processor we have covered in classes. Also, you need to generate the control signal "`mem_byte`" correctly to make `rmmovb` and `mrmovb` instructions work.

2.4 Part A Evaluation (50 points)

Part A is worth 50 points. You have to pass the following tests to receive the credit.

- **40 points** for regression tests for Y86-64 instruction set
  The `./ptest` directory contains various scripts that generate systematic regression tests of the different instructions, the different jump possibilities, and different hazard possibilities. These scripts are very good at finding bugs in your simulator. To run the tests, perform "`make`" in the `./ptest` directory. The source codes of failed test cases (`*.ys` files) are stored in the same directory after running the tests. Please refer to the `./ptest/README` file for further details. **No partial credits**. Your PIPE-Stall simulator should pass all the test cases to receive any credit.
- **10 points** for additional tests to verify the correctness of the `iaddq`, `mulq`, `rmmovb`, and `mrmovb` instructions

3. Part B: Optimizing the performance of bmp_grid() on PIPE-Stall

3.1 Rewriting `bmp_grid()` for PIPE-Stall

Your task in Part B is to rewrite the `bmp_grid()` function you have written in Project #4 to optimize its performance on the PIPE-Stall processor. For this part, you will be working in the `./bmpgrid` directory.

```
void bmp_grid (unsigned char *imgptr, long long width, long long height,
               long long gap);
```

As in Project #4, 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 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_grid()`. You may make any semantics preserving transformations to the `bmp_grid()` 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 Restrictions

- The code size of `bmp_grid()` should be less than or equal to 1024 bytes. The `bmp_grid()` function starts at the address `0x400`. Therefore, the address of your code should be less than `0x800`.
- Your `bmp_grid()` implementation should work for BMP images of any size.
- Your `bmp_grid()` implementation should work for any positive value of "`gap`".
- Your `bmp_grid()` implementation should leave the bytes in the padding area untouched.

3.3 Part B Evaluation (50 points)

To receive credit here, your code must be correct first. We will run several test cases to check out whether your code is correct or not. If you don't pass these tests, you don't receive any credit. 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_grid()` and to set up the loops, you will find that you will get different values of the CPP for different combinations of image heights, image widths, and gap values. 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 score S for this part will be determined as follows:

c <= 5.0	S = 50 points + 10 points bonus
5.0 < c <= 5.5	S = 50 points
5.5 < c <= 6.0	S = 45 points
6.0 < c <= 6.5	S = 40 points
6.5 < c <= 7.0	S = 35 points
7.0 < c <= 7.5	S = 30 points
7.5 < c <= 8.0	S = 25 points
8.0 < c <= 8.5	S = 20 points
8.5 < c <= 9.0	S = 15 points
9.0 < c <= 9.5	S = 10 points
9.5 < c <= 10.0	S = 5 points
c > 10.0	S = 0 points

4. Skeleton codes

The following skeleton codes are provided for this project. These skeleton codes are based on the Y86-64 toolset developed by textbook authors available at http://csapp.cs.cmu.edu/3e/sim.tar

`README`	The original README file in the Y86-64 toolset.
`README.SNU`	This file summarizes some of changes made for this project.
`Makefile`	This is a top-level Makefile used by the GNU `make` utility.
`simguide.pdf`	An official guide to the Y86-64 simulators.
`misc/`	This directory contains the files for Y86-64 assembler (`yas`) and instruction simulator (`yis`).
`pipe/`	This directory contains the files for implementing the PIPE Y86-64 simulator.
`bmpgrid/`	This directory contains a template file for implementing `bmp_grid()`. In this directory, you can perform "`make test`" to see if your code produces the correct result.
`y86-code/`	This directory has the sample codes written in Y86-64. We have added several test programs for new instructions such as `iaddq1.ys`, `iaddq2.ys`, `mulq1.ys`, `mulq2.ys`, `mrmovb.ys`, and `mrmovb.ys`.
`ptest/`	This directory contains automatic testing scripts for individual instructions. For more details, please refer to the `./ptest/README` file.

5. Hand in instructions

- You need to submit `./misc/isacore.c`, `./pipe/pipe-stall.hcl`, `./pipe/psimcore.c` and `./bmpgrid/bmpgrid.ys` files only. You can do this by performing "`make handin`" in the top directory. It will generate the `pa5.tar.gz` file in the `./handin` directory. Upload this file to the submission site (`http://sys.skku.edu`).
- **As in Project #4, the total number of submissions will be limited to 10 times.**

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 1 week 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.

Have fun!

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