UChicago CMSC-22200 Computer Architecture, Autumn 2016
Lab 2: Pipelined ARM Simulator

Instructor: Prof. Yanjing Li
TAs: Xin-Chuan (Ryan) Wu, Maria Hyun, Mohammad Reza Jokar, Gushu Li

Assigned: Thur., 10/06, 2016
Due: **11:59 pm, Thur., 10/20, 2016**

Introduction

For this assignment, you will write a C program which is a pipelined ARM simulator for a limited subset of the ARMv8 instruction set (the same set that you implemented in Lab1; see the last two pages of the writeup for the list of required instructions). This pipelined simulator will not only model the functional behavior of each instruction, but also enable pipelined instruction processing.

Your task is to implement the 5 pipeline stages you learned from class. The five stages are Instruction Fetch, Instruction Decode, Execute, Memory Access, and Register Writeback. Your simulator must be able to handle all possible branch cases, stall cases and bypass cases. We do not require exception/interrupt handling in this lab.

The source code for the Lab 2 is provided in the course website. In the `src/` directory, we provide two files (`shell.c` and `shell.h`) that already implement the shell and another two files, (`pipe.h pipe.c`) where you will implement the pipelined simulator – `pipe.h and pipe.c are the only files that you are allowed to change.`

Use the simulator that you developed in Lab1 as a starting point to implement the pipelined simulator. Your simulator in Lab 1 can also be used as a reference to verify that your pipelined simulator functions correctly. In addition, you should write your own input files that cover all data and control dependency scenarios to test the pipeline (use the same method as Lab1 to create input files and check the results).

Register File and Memory Timing Model

Register file (RF): You may assume combinational read (i.e., output of the read data port is a combinational function of the RF contents and the corresponding read select port) and synchronous write (i.e., the selected register is updated on the positive edge clock transition when write enable is asserted).

Memory: you may assume a "magic" memory where read and write operations can take effect immediately (using the mem_read_32 and mem_write_32 functions).

The Simulation Routine

As mentioned, there are two files, (`pipe.h pipe.c`) where you will implement the pipelined simulator. In the `pipe.h` file, we have an struct named CPU_ State, same as Lab 1, representing the current state of the processor. We also define CURRENT_STATE and RUN_BIT as Lab 1.

```
typedef struct CPU_State {
/* register file state */
int64_t REGS[ARM_REGS];
int FLAG_N;        /* flag N */
int FLAG_Z;        /* flag Z */
int FLAG_V;        /* flag V */
int FLAG_C;        /* flag C */

/* program counter */
uint64_t PC;
```

```
} CPU_State;

/* CURRENT_STATE is the current arch. state; */
/* Register values and flags are initialized to 0
and PC is initialized to the address of the first instruction in memory */
CPU_State CURRENT_STATE;

/* RUN_BIT is initialized to 1;
need to be changed to 0 if the HLT instruction is encountered */
int RUN_BIT;
```

Memory access is done with the following two functions, defined in `shell.c`, exactly the same as Lab 1. Use these two functions for all memory-related operations. You're not allowed to change these two functions.

```
uint32_t mem_read_32(uint64_t address);
void mem_write_32(uint64_t address, uint32_t value);
```

To enable pipeline execution, you must define your own data structures to represent the pipeline registers between different pipeline stages, and assign values to these pipeline registers as instructions pass through the pipeline. Everytime after an instruction finishes the Writeback stage, CURRENT_STATE should be modified accordingly to reflect the effects of this instruction. Use good coding practices – for example, define a data structure like below to hold all pipeline registers between the fetch and decode stage (similarly for other stages).

```
typedef struct Pipe_Reg_IFtoDE {
  //pipeline register 1
  //pipeline register 2
  // ...
  //pipeline register n
} Pipe_Reg_IFtoDE;
```

The simulation terminates upon the execution of the HLT instruction (i.e., changing the RUN_BIT from 1 to 0). Note that, when the HLT instruction is encountered, all instructions that are still outstanding in the pipeline must be completed before the simulation is terminated.

In the `pipe.c` file, you should implement the 5 pipeline stages using 5 functions that are called by the pipe_cycle function. You can add any code, helper functions, etc. to the 5 pipe_stage_* functions, but you are not allowed to change their prototypes. You can also add any code to the pipe_cycle function, but you are not allowed to change its prototype. Also you should keep the 5 pipe_stage_* function calls in the pipe_cycle function.

```
void pipe_cycle()
{
    pipe_stage_wb();
    pipe_stage_mem();
    pipe_stage_execute();
    pipe_stage_decode();
    pipe_stage_fetch();
}
```

Steps to Complete This Lab

We suggest that you should develop the pipeline simulator in 3 steps. Partial credits will be given upon completion of each step.

Step 1: Implement the 5 stages of the pipeline in the `pipe.c` file without worrying about any data or control dependencies. Remember that in each cycle the simulator run each stage once.

Step 2: You should enable data dependency handling in this step. Provide bypass between stages to avoid the unnecessary stalls and make sure that the result is correct. Remember that in some scenarios, the processor has to stall even if bypass is implemented.

Step 3: You should also handle control dependency in this step, which completes the entire pipeline implementation. When there is a branch instruction at the decode stage, the simulator should stall the pipeline by inserting bubbles until the branch target is resolved, and squash the instruction at the fetch stage if that instruction turns out to be the incorrect target.

We provide a few test inputs and their outputs to test your code for each of the three steps. In order to test that your simulator is working correctly for all scenarios, you should also write more test cases on your own. You can use the rdump/mdump command to verify that the state of the machine is updated correctly after each cyle.

Lab Files

From the course website, you will find a source code distribution with three subdirectories `src/`, `inputs/` and `outputs/`.

In `src/`, we are providing you with the simulator skeleton as described above. You can compile the simulator with the provided `Makefile`.

In `inputs/`, we have written some input files for you. You should write more input files in order to be confident that your simulator is correct.

In `outputs/`, we are providing you with the outputs of the given input files.

Handin

You will be working in groups of 2, and only one copy of the code needs to be submitted for one group, but `please make sure you write down both names and cnetid at the top of the pipe.c file.`

You should electronically hand in your code (all files in the `src/` directory) into lab2 folder of your own SVN repository. Contact the TAs if your handin SVN repository does not exist. Your code should be readable and well-documented. In addition, please turn in additional test cases that you used in `inputs/` subdirectory.

You should stick with the same partner as Lab1, and we will take the submission from the student whose name comes first in alphabetical order.

Key Reminders

1. You only need to modify two files, (`pipe.h pipe.c`) and you are not allowed to change shell.h and shell.c. You must use mem_read_32 and mem_write_32 functions to implement loading and storing 8-bit, 16-bit, and 64-bit data, and you are not allowed to modify mem_read_32 and mem_write_32. You should implement the 5 pipe_stage_* functions without changing their prototypes, and you are not allowed to change the pipe_cycle function prototype. Also you should keep the 5 pipe_stage_* function calls in the pipe_cycle function.

2. Follow the 3 steps as mentioned previously to complete this lab. After completing all 3 steps, you will have a fully-functioning pipeline simulator that is able to handle all data/control dependency scenarios.

3. You should write more input files in order to be confident that your simulator is correct.

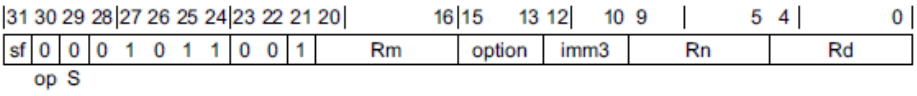
4. Use the same method as Lab1 to assemble a ARM program (with script asm2hex).

LAB2 Required Instructions: (Only need to implement 64-bit variant)

Except for LSL and LSR, you can assume the shift amounts is 0 for all instructions.

ADD (Extended Register, Immediate)
ADDS (Extended Register, Immediate)
CBNZ
CBZ
AND (Shifted Register)
ANDS (Shifted Register)
EOR (Shifted Register)
ORR (Shifted Register)
LDUR
LDURB
LDURH
LSL (Register, Immediate)
LSR (Register, Immediate)
MOVZ
STUR
STURB
STURH
STURW
SUB (Extended Register, Immediate)
SUBS (Extended Register, Immediate)
MUL (by element)
SDIV
UDIV
HLT
CMP
BL
BR
B
BEQ
BNE
BGT
BLT
BGE
BLE

1. ADD extended instruction format listed in the reference manual as below



But if you compile ADD Xd, Xn, Xm, you will get a machine code bit_21 is 0. When you compile the format ADD Xd, Xn, Xm, `<extend> <amount>`, the bit_21 is 1. Please make sure ADD Xd, Xn, Xm can work with your simulator. The same thing happens in ADDS, SUB, and SUBS.

2. ADD extended instruction format is shown like this.

```
ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

In lab2, we don't expect you to implement the {`, <extend> #<amount>`} part. You're only required to

implement ADD Xd, Xn, Xm. ADDS, SUB and SUBS are the same.

3. For add function, though the reference manual use addWithCarry in the description, you don't need to implement addWithCarry. Simple add is fine.

4. For ADD immediate, defines as below

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case shift of
    when '00' imm = ZeroExtend(imm12, datasize);
    when '01' imm = ZeroExtend(imm12:Zeros(12), datasize);
    when '1x' ReservedValue();
```

No need to implement ReservedValue().

5. For load/store, no need to do CheckSPAlignment().