CS 520: Computer Architecture and Organization
Fall 2015: Prof. Timothy N. Miller
Homework 2
Deadline: Monday October 5

Problem 1

Consider the processing of N instructions on an instruction pipeline with K stages.

- A fraction ***i*** of the instruction fetches result in an I-cache miss and each miss requires ***B*** cycles to deliver the required instruction. A cache hit requires the usual one cycle of pipeline latency in the Fetch stage, while a miss introduces ***B*** bubble cycles.
- A fraction ***m*** of all instructions are LOADs and a fraction ***q*** of these LOAD instructions experience a D-cache miss. A D-cache miss requires ***M*** cycles to fetch the data from the RAM into the D-cache. A cache hit requires the usual one cycle of pipeline latency in the MEM stage, while a miss introduces ***M*** bubble cycles.
- A fraction ***n*** of all instructions are STOREs and a fraction ***p*** of these STORE instructions experience a D-cache miss. A D-cache miss requires ***M*** cycles to fetch the data from the RAM into the D-cache (even for stores). A cache hit requires the usual one cycle of pipeline latency in the MEM stage, while a miss introduces ***M*** bubble cycles.
- Another fraction ***b*** of the instructions are branches and a fraction ***t*** of the branches are taken (that is, result in a non-consecutive instruction access). Each such taken branch introduces a ***S***-cycle bubble.
- A fraction ***d*** of the instructions are stalled for an extra ***R*** cycles due to data-dependencies.

Compute the effective CPI realized in the course of processing this instruction stream, assuming that each pipeline stage (ideally) has a delay of a single clock cycle when it is not blocked. Provide adequate explanations for your answer and state any assumptions you have made.

Problem 2

Consider the following instruction for implementation on the 5-stage APEX pipeline:

MUL Rx, Ry, Rz

where *Ry* and *Rz* are source registers that contain 32-bit operands. This instruction generates the product of the 32-bit contents of *Rz* and *Ry* and writes the 64-bit result into two neighboring registers specified by *Rx*. If *x* is even, then the lower 32 bits of the product are store in *Rx*, and the upper 32 bits are stored in *R(x+1)*. If *Rx* is odd, then the lower 32 bits are stored in *Rx*, and the upper 32 bits are stored in *R(x-1)*. As an example, the instruction MUL R4, R7, R8 computes the product of the contents of R7 and R8 and writes the 64-bit product into registers R4 and R5.

For this problem, you are required to modify the APEX datapath shown on Page 34 of the notes to incorporate a separate multiply ALU into the EX stage (in parallel to the existing ALU) and make other changes as needed to support the MUL instruction. Assume that the multiply function unit has a single cycle latency. Do not add any more stages to the pipeline, nor should you assume that instructions can spend more than a single cycle in any stage following the D/RF stage.

Questions:

a) List and briefly explain all the changes you need to make, including the use of additional latches, multiplexers, register file ports, etc.

b) Draw a diagram similar to that shown on Page 34 of the notes to show the various information that flows from one stage of the APEX pipeline to the following stage. Assume that we are fetching the MUL R4, R7, R8 instruction from address 2048 in cycle T and:
   1) R7 contains the value 7FFFFFFF (in hexadecimal), stored in 2's complement (1 bit sign, 31-bit magnitude)
   2) R8 contains the value FFF00000 (in hexadecimal, 2's complement)

c) Describe, using a table (similar to Page 32), what happens in each stage and what directives are given by the control logic within each stage as this MUL instruction moves through the pipeline.

d) In a few sentences, describe an alternative approach that does not require adding ports to the register file.

Important: The logic for computing the neighbor of *Rx* does NOT require an adder or subtractor. Using an adder and/or subtractor will result in points being deducted.

Problem 3

Consider the introduction of the following new instruction to the APEX ISA:

ADDSH <dest> <src1> <src2> #shift_amount

Here, *src1* and *src2* are registers that provide input operands and *dest* is the destination register. The literal included in the instruction, *shift_amount* specifies a 5 bit positive value that indicates the number of bit positions over which the sum has to be logically shifted to the left. The semantics of the instruction is as follows in the register transfer language that we have used:

<dest> ← SHL ((<src1> + <src2>), shift_amount)

where SHL is a logical left shift operation performed by a dedicated hardware shifter that shifts the value of the sum by the number of bit positions specified in *shift_amount*.

The shift operation performed by the dedicated hardware shifter requires one cycle to perform a shift operation (including latching delays). The addition operation, as before, requires an ALU with a delay of a single clock cycle (including latching delays). This shifter has an input operand and a shift amount as input, as well as control inputs that indicate the direction of shifting and whether the shift operation is a logical shift or an arithmetic shift. The shifter has a single output that delivers the shifted result.
To incorporate this instruction into the APEX pipeline, with a single cycle delay for each stage, the operations for this instructions are performed in two execution stages, connected in series, EX1 and EX2. All other stages remain as in the original design. The stage EX1 is the original ALU capable of performing the addition, while EX2 includes only the shifter (described above). All instructions pass through these 6 stages and many instruction (such as an ADD or a LOAD) only *pass through* the EX2 stage, with no operations done on the ALU result as is spends one cycle going through EX2.

Questions:

a) Draw a detailed diagram to show the latches, multiplexers, the shifter, the ALU etc. and other artifacts, if any, that are used to implement the ADDSH instruction, as well as the ADD and LOAD instructions (whose semantics are exactly as defined in the original design).

b) Describe what happens in each stage, including how the decoded information is used, as the (i) ADDSH, (ii) ADD and (iii) LOAD instruction are processed by the 6-stage pipeline.

c) Without any additional hardware, describe how to perform *just* a shift of src1 if:
   1) R0 always reads zero.
   2) R0 is just a regular register, but the destination register is R0.