

Principles in Computer Architecture

CSE 240A Homework One

October 15, 2012

*Only Problem Set Two will be graded. **Turn in only Problem Set Two** on October **26**, 2012 (**Friday**) by **11:00am**.*

*Submit homework via csemoodle <csemoodle.ucsd.edu> as a **PDF**. Other formats may or may not be viewed and evaluated accurately.*

Problem Set One

1. Hennessy & Patterson (5th Ed.) 1.1 (*Chip Manufacturing*)
2. Hennessy & Patterson (5th Ed.) 1.8 (*Moore's Law*)
3. Hennessy & Patterson (5th Ed.) 1.14 (*Speedups*)
4. Hennessy & Patterson (5th Ed.) A.8 (*Instruction Encoding*)
5. Hennessy & Patterson (5th Ed.) A.18 (*ISA Styles*)
6. Hennessy & Patterson (5th Ed.) C.4 (*Hazards of Pipelining*)
7. Hennessy & Patterson (5th Ed.) C.5 (*Addressing Modes*)
8. Hennessy & Patterson (5th Ed.) C.6 (*Register-memory Pipelining*)
9. Hennessy & Patterson (5th Ed.) C.13 (*Scoreboards*)

Problem Set Two

1. Power

Your company's internal studies show that a single core system is sufficient for the demand on your processing power. You are exploring, however, whether you could save power by using two cores.

- (a) Assume your application is 100% parallelizable. By how much could you decrease the frequency and get the same performance?
- (b) Assume that the voltage may be decreased linearly with the frequency. Using the dynamic power equation (described in the book), how much dynamic power would the dual-core system require compared to the single-core system?
- (c) Now assume that the voltage may not be decreased below 40% of the original voltage. This voltage is referred to as the “voltage floor”, and any voltage lower than that will lose the state. What percent of parallelization gives you a voltage at the voltage floor?
- (d) Using the dynamic power equation described in the book, how much dynamic power would the dual-core system require from part (a) compared to the single-core system when taking into account the voltage floor?

2. Pipelines

Assume that a new addressing mode is being added to the classical 5-stage MIPS pipeline described in your textbook. The addressing mode utilizes two registers and one immediate constant. The two registers correspond to *base address* and *offset*, while the immediate constant corresponds to an index. When used by a load/store instruction, the following format is used: **ld R1, imm(R2)R3**. The traditional 5-stage pipeline has been modified to include a second memory stage, M2, thus resulting in the following pipeline stages: **F D E M1 M2 W**. When executing a load/store instruction that uses this addressing mode, the effective address is computed in two consecutive cycles E and M1: the E stage computes **R2+imm** while the M1 stage adds the value of **R3** to the sum. The single integer ALU is used for all the effective address computations.

- (a) Please describe the new pipeline hazards introduced by this addressing mode.
- (b) What new forwarding paths need to be introduced to the pipeline in order to minimize the impact of the introduced pipeline hazards?
- (c) For the following code fragments, please identify the number of pipeline stalls assuming that all of the newly required forwarding paths have been added to the pipeline.

```

1.
ld R1, (r8)20
ld R4, (R1)r2
st R5, (R1)10
sub R4, R4, R5

2.
add R1, R1, R7
ld R4, 10(R1)R2
add R5, R4, R7
st R5, (R1)20

```

3. Exceptional Scoreboard

The fundamental rule of exception handling is that the instructions issued after the faulting instructions should not be allowed to change the processor into an **unrecoverable state** when an exception occurs. One simple way to ensure this result is through *precise exceptions*, which disallows any subsequent instructions' writing their results if a previous instruction has not finished execution. Obviously, the same semantically correct result that precise exceptions deliver can be accomplished by other means. We are asking you to explore these concepts within the context of the scoreboard implementation. Recall that each instruction undergoes **four steps** in the scoreboard: *issue*, *read operands*, *execution*, *write results*. In this process, the three types of hazards - WAW, RAW and WAR - are checked at the *issue*, *read operands*, and *write results* stages, respectively.

For the rest of the problem, you can use the following code fragment to analyze the various conditions.

```

add.d f2, f6, f8
div.d f0, f2, f4
sub.d f4, f6, f10
mul.d f6, f8, f2
add.d f0, f2, f8

```

- (a) For the above code fragment, assume the **div.d** instruction, which typically has a latency of 40 cycles, may cause an exception during its execution. In this situation, are any of the instructions **sub.d**, **mul.d**, or **add.d**, allowed to enter the *write results* stage? If you think an instruction cannot write its result, can it be issued if the corresponding functional unit is free? Give your reasons for each instruction.
- (b) The traditional scoreboard handles WAW, RAW and WAR hazards in the following ways:

WAW: at *issue* stage, an instruction can be issued if no other active instruction has the same destination register.

RAW: at *read operands* stage, the functional unit can proceed to read the operands if no earlier issued active instruction is going to write any of the source operands.

WAR: at *write results* stage, a completing instruction cannot be allowed to write its result when there is an earlier issued active instruction that has not read its operands and one of the operands is the same register as the result of the completing instruction.

We want to incorporate exception handling into the scoreboard. However, we do *not* want to *unnecessarily delay the issue of instructions*. Given this condition, do any of the above three rules need to be modified in order to handle exceptions in a scoreboard? If you think a rule needs to be modified, please specify the corresponding new hazard checking mechanism. Otherwise, please give a reasoning as to why no modification is needed.

- (c) It seems that the incorporation of exception handling introduces significant performance overhead for the traditional scoreboard. To minimize this overhead, a clever engineer has suggested to insert several “**exception flags**” for each instruction. Fundamentally, an exception flag indicates whether an exception condition has been cleared, even though the corresponding instruction hasn’t finished its execution yet.

Using these “**exception flags**”, do you think the conventional hazard checking mechanism, presented in part (b), needs to be modified? Again, we do not want to unnecessarily delay the issue of instructions. If you think a rule needs to be modified, please specify the corresponding new hazard checking mechanism. Otherwise, please give a reasoning as to why no modification is needed.

- (d) As mentioned before, the basic mechanism to implement *precise exceptions* is to force instructions to write their results in **program order**. However, *in-order writeback* will cause a large number of the entries of the instruction status table to be occupied by the completed instructions, thus limiting the efficiency of hardware utilization.

To overcome this limitation, instead of the *in-order writeback* policy, an engineer has suggested to just use the writeback policy you developed in (b). Additionally, (s)he suggests that the instruction, once its results are written, should be kicked out of the instruction status table to free the corresponding entry. Do you think this suggestion is able to achieve the same semantically correct result that *precise exceptions* deliver? Clearly write down your reasoning.