Peter Woodman
0127119
CSE 410 - H. Levy
Book and Programming Assignment 3

1. *Latency* has to do with how long it takes an instruction to complete once it has begun to execute. *Throughput* is how many instructions can get executed in a given block of time. *Pipelining* attempts to increase throughput, often at the cost of added latency, by splitting up instruction execution into smaller discrete stages so that one can execute the next instruction before the first one has completely finished. In this way, throughput is increased. For example, if an instruction takes 1 μs to complete, but it can be divided up into 5 discrete steps each taking .2 μs, at the end of 1 μs, we'll still have only executed 1 instruction, but after 2 μs, we'll have executed 6, and 5 more for each μs after, assuming no stalls. This is much better than one instruction per μs.

2. **Data hazards**
   This refers to the situation where, in order to execute, an instruction needs data from an instruction currently in the pipeline. An example of this is two add instructions in a row, the second one depending upon the result of the first.

   Structural Hazards
   This kind of dependency emerges when two instructions need to use the same part of the hardware at the same time, such as two instructions needing to read from memory at the same time, or write to a register at the same time. In MIPS, if i understand correctly, you will not run into these, as MIPS was designed for pipelining- they are simply design considerations that should be easy to satisfy completely for the processors implementing this instruction set.

   Control/Branch Hazards
   This emerges when a branch instruction starts execution, and one cannot be sure whether the branch will be taken or not until after the code has executed. This is a narsty one, as this kind of hazard will occur at nearly every branch in a program- we don't know the result of a branch until it is computed, otherwise it wouldn't really be a branch!

3.
```
(1)  add $3, $4, $2
(2)  sub $5, $3, $1
(3)  lw  $6, 200($3)
(4)  add $7, $3, $6
```

   Lines 2, 3 and 4 all depend on the result of line 1, i.e. the contents of register 3. Line 4 also depends on the result of line 3, the load instruction. Without taking any other instruction sinto account, line 4 is the only line that will cause a bubble that is not solvable by data forwarding. In line 1, the result of the `add` op out of the ALU can be fed back into itself to avoid the hazard. Same with line 3- the result can be fed back in from MEM/WB. Line 4 will have no problem reading register 3, but it will have a problem reading register 6, as we need to wait for the LW instruction to complete the memory phase before we'll be able to know the contents of register 6.

   However, code reordering could allow one to avoid stalls with this code. Line 4 needs to know the contents of register 6 by its ALU phase- in the code above, without reordering, when line 4 is on the ALU phase, line 3 is in the Data Memory stage. If we swap lines 3 and 2, neither 3 or 2 will be affected at all, but we will be able to forward the value of register 6 directly from the Data Memory phase to the ALU for line 4's `add` and avoid any stalls.

-quad.s-

```
# third speem assignments
# write a routine to calculate the result of ax^2 + bx + c.
    # Program 3
    # Name: Peter Woodman
    # Class: CSE410
    # Date: 04242005

# COMMENTS! COMMENTS! AHEM!
#
# So, I'll admit it, I found the instructions to be a bit ambiguous and tricky.
# I get the feeling this is the intent, though. However, I feel the need to
# explain myself in case I've done something weird. As the procedure I use,
# "quad", does not use anything but unsaved temp registers, I do not save
# anything to the stack (except the result- this is a very odd promise this
# routine asks for). I also use "pseudoroutines"- I consider anything with
# _%routine% at the beginning of it to be a part of "routine", and operating
# in the same register space. I don't know if I'm doing something naughty
# here or not (other than wasting some instructions with jumps- do tell me if
# I'm asking for trouble by doing this, though!
#  thx,
#   peter.
#

    .data
str1: .asciiz " y="
str2: .asciiz "a="
str3: .asciiz " b="
str4: .asciiz " c="
str5: .asciiz " x="
endstr: .asciiz "No more quads, exiting.."
array: .word 3 5 2 5 1 7 23 6 7 9 1 15
length: .word 12
newline: .asciiz "\n"

.align 2

    .globl  main
    .text

main:
    # we make magic to our stack in first!!

    la    $s0, array    # the hot numbers for munching!
    lw    $s1, length   # how many for munching??

_main_loop:
    addi  $s1, $s1, -4
    bltz  $s1, _main_end

    lw    $t0, 0($s0)   # for "a" being
    lw    $t1, 4($s0)   # "b"!
    lw    $t2, 8($s0)   # "c"!
    lw    $t3, 12($s0)  # the "x"! what a powerfull stuff!
```

```
    addi $s0, 16

    subu  $sp, $sp, 16      # we want for making arguments a frame in "stack"?
    sw    $t0, 12($sp)      # why is it "stack"? could not be "register"?
    sw    $t1, 8($sp)       # we are having the 4 register "argument register"
    sw    $t2, 4($sp)
    sw    $t3, 0($sp)

    jal   _main_print_args

    jal   quad              # ooh! our very first procedure! a proud parent now!

    li    $v0, 4            # do we do it OK? look and see!
    la    $a0, str1
    syscall

    li    $v0, 1
    lw    $a0, 0($sp)
    syscall

    li    $v0, 4
    la    $a0, newline
    syscall

    j     _main_loop

quad:
    lw    $t0, 12($sp)      # ha ha! what a lark it is! (a)
    lw    $t1, 8($sp)       # a silly thing to be using such a stack! (b)
    lw    $t2, 4($sp)       # asshat! like molasses the memory pouring! (c)
    lw    $t3, 0($sp)       # ok we are GO! (d)

    add   $t9, $zero, $t2   # we keep "y" here! you ask "(wh)y"? i say "because"

    mul   $t4, $t1, $t3
    add   $t9, $t9, $t4

    mul   $t4, $t3, $t3     # here we make (x x x)! very silly!
    mul   $t4, $t4, $t0
    add   $t9, $t9, $t4

    sw    $t9, 0($sp)       # write to top of "stacked"

    jr    $ra               # go home now!!


_main_print_args:

    li    $v0, 4
    la    $a0, str2
    syscall

    li    $v0, 1
    add   $a0, $t0, $zero
    syscall
```

```
    li    $v0, 4
    la    $a0, str3
    syscall

    li    $v0, 1
    add   $a0, $t1, $zero
    syscall

    li    $v0, 4
    la    $a0, str4
    syscall

    li    $v0, 1
    add   $a0, $t2, $zero
    syscall

    li    $v0, 4
    la    $a0, str5
    syscall

    li    $v0, 1
    add   $a0, $t3, $zero
    syscall

    jr $ra

_main_end:
    li    $v0, 4
    la    $a0, endstr
    syscall

    li    $v0, 10
    syscall

.end
```

-program output-

```
pjjw@clango:~/Documents/coursework/cse410/hw3$ spim quad.s
SPIM Version 7.1 of January 2, 2005
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/local/share/xspim/exceptions.s
a=3 b=5 c=2 x=5 y=102
a=1 b=7 c=23 x=6 y=101
a=7 b=9 c=1 x=15 y=1711
No more quads, exiting..
```