LSU EE 4720 **Homework 4** Solution **Due: 28 April 2010**

Problem 1: A deeply pipelined MIPS implementation is constructed from our familiar five-stage pipeline by splitting `IF`, `ID`, and `ME` each into two stages, but leaving `EX` and `WB` as one stage. The total number of stages will be eight, call them `F1`, `F2`, `D1`, `D2`, `EX`, `Y1`, `Y2`, and `WB`. In this system branches are resolved at the end of `D2` (rather than at the end of `ID`). Assume that all reasonable bypass paths are present.

(*a*) Provide a pipeline execution diagram of the code below for both the 5-stage and this new implementation, for enough iterations to compute the IPCs.

Solution appears below. Note that for the 8-stage system the second iteration, starting in cycle 9, starts with the processor in the same state as the third iteration, starting in cycle 18, and so the second iteration can be used to compute IPC. The execution rate is $\boxed{\frac{5}{18-9} = \frac{5}{9}\,\text{insn/cycle}}$ for the eight-stage system. By a similar argument the rate for the five-stage system is $\boxed{\frac{5}{12-6} = \frac{5}{6}\,\text{insn/cycle}}$.

```
# Solution

# Eight-Stage Pipeline
#
LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
 addi r2, r2, 4   F1 F2 D1 D2 EX Y1 Y2 WB
 lw r1, 0(r2)        F1 F2 D1 D2 EX Y1 Y2 WB
 add r3, r3, r1         F1 F2 D1 D2 ----> EX Y1 Y2 WB
 bne r5, r4 LOOP           F1 F2 D1 ----> D2 EX Y1 Y2 WB
 addi r5, r5, 1               F1 F2 ----> D1 D2 EX Y1 Y2 WB
                                 F1 ----> F2x
                                          F1x
LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
 addi r2, r2, 4                              F1 F2 D1 D2 EX Y1 Y2 WB
 lw r1, 0(r2)                                   F1 F2 D1 D2 EX Y1 Y2 WB
 add r3, r3, r1                                    F1 F2 D1 D2 ----> EX Y1 Y2..
 bne r5, r4 LOOP                                      F1 F2 D1 ----> D2 EX Y1..
 addi r5, r5, 1                                          F1 F2 ----> D1 D2 EX..
                                                            F1 ----> F2x
                                                                     F1x
LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
 addi r2, r2, 4                                                          F1 F2..


# Five-Stage Pipeline
#
LOOP: #           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 addi r2, r2, 4   IF ID EX ME WB
 lw r1, 0(r2)        IF ID EX ME WB
 add r3, r3, r1         IF ID -> EX ME WB
 bne r5, r4 LOOP           IF -> ID EX ME WB
 addi r5, r5, 1               IF ID EX ME WB
LOOP: #           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 addi r2, r2, 4                  IF ID EX ME WB
 lw r1, 0(r2)                       IF ID EX ME WB
 add r3, r3, r1                        IF ID -> EX ME WB
 bne r5, r4 LOOP                          IF -> ID EX ME WB
 addi r5, r5, 1                                IF ID EX ME WB
LOOP: #           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 addi r2, r2, 4                                    IF ID EX ME WB
```

(*b*) Suppose the 5-stage MIPS runs at 1 GHz. Choose a clock frequency for the 8-stage system for which the time to execute the code above is the same as for the 5-stage MIPS.

First, express the execution rates for the two systems above in instructions per second. Let ϕ_5 and ϕ_8 denote the clock frequencies of the 5-stage and 8-stage systems. The execution rate in instructions per second is the CPI times the clock frequency: $\text{CPI} \times \phi = \frac{\phi}{\text{IPC}}$. Solving $1.8\phi_8 = 1.2\phi_5$ for ϕ_8 yields $\boxed{\phi_8 = \frac{1.8}{1.2}\phi_5 = 1.5\,\text{GHz}}$.

(*c*) Consider two ways to make a 7-stage system from the 8-stage system. In method ID, the two ID stages (`D1` and `D2`) are merged back into one (or if you prefer, the ID stage was never split in the first place). In method ME, the two ME stages (`Y1` and `Y2`) are merged back into one (or were never split).

Which method is better, and why? Assume that the eight-stage system runs at 1.8 GHz. Consider both the likely impact on clock frequency (remembering that you are at least senior-level computer engineering students) and the benefit for code execution (don't just consider the code above, argue for what might be typical code).

As mentioned in class a number of times, it is the memory stage which would take the most time. Here are sample stage timings with memory taking the most: `IF`, 1 ns; `ID`, 0.4 ns; `EX`, 0.56 ns; `ME`, 1 ns; and `WB`, 0.2 ns. For the 5-stage system memory determines the clock frequency. On the 8-stage system, assuming the memory stages timings are split, clock is determined by the `EX` stage.

Based only on clock frequency it would be better to merge the `D1` and `D2` stages, since that would not impact clock frequency (the original `ID` could handle 1.8 GHz). In contrast, if the memory stages were merged the clock would be back to 1 GHz and so there would be no performance gain.

Splitting stages can introduce stalls or squashes. Because memory was split, the 8-stage system suffers two stall cycles on a load/use pair. Because `IF` and `ID` were split there are two squashes after each taken branch.

Merging `Y1` and `Y2` would eliminate one stall cycle for each load/use pair. However, scheduling can eliminate many such stalls. Merging `D1` and `D2` would eliminate one squash on every taken branch. Since many taken branches can't be avoided and occur about 1 out of every 12 instructions in integer code, it would be better to merge the `D1` and `D2` based on code considerations.

In summary, based on both clock frequency impact and stall/squash benefit, it would be better to merge the `D1` and `D2` stages.

Problem 2: Itanium is a VLIW ISA designed for general-purpose use. Being a VLIW ISA (as defined in class) its features were chosen to simplify superscalar implementations. The questions below are about such features, read the Intel Itanium Architecture Software Developer's Manual Volume 1, Section 3 for details and concentrate on Sections 3.3 and 3.4. The manual is linked to the course references page, `http://www.ece.lsu.edu/ee4720/reference.html`. Use this copy to be sure that section and table numbering used here match.

(*a*) Section 3.3 mentions four types of functional unit, and where an instruction using a particular unit can be placed in a bundle (see Table 3-9 and 3-10).

Suppose an Itanium implementation fetches one bundle per cycle. Indicate the maximum number of execution units of each type needed. (That is, there would be no advantage of having more than this number.) Assume that the units all have latency 1 or else are fully pipelined.

The table below shows the types of execution units that can be used by each slot, as well as the maximum number of units of each type that any bundle can need. The table was constructed by examining table 3-10, and noting that the L-unit/X-unit instructions really use an I or B unit.

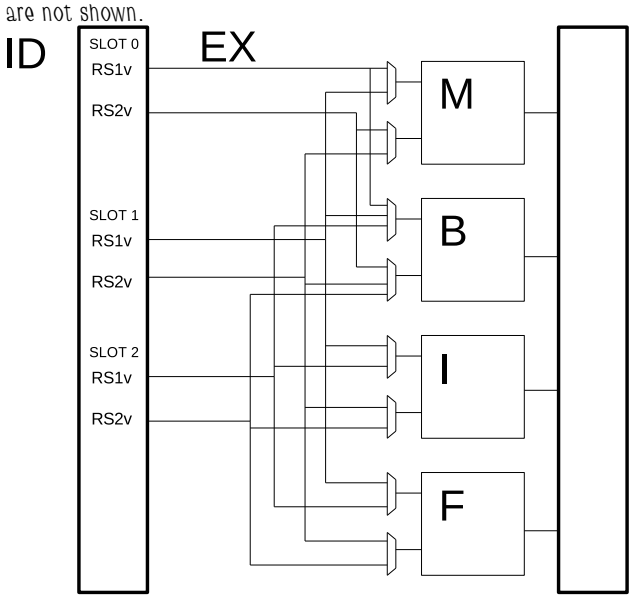
The bottom of the table shows the maximum number of times a unit will appear in a bundle. There would be no benefit in having more than that number. For example, there would be no point in having three M-units (assuming fully pipelined) because an instruction in slot 2 would never use it.

```
         Execution Unit
Slot 0:  M  B
Slot 1:  M  B I F
Slot 2:     B I F*
Max:     2  3 2 1   <- Maximum useful number of units.
```

(*b*) For this problem suppose the implementation had the minimum number of units of each type, one. Sketch the pipeline execute stages, and show connections to each of the FU inputs. There should be three sets of source operands flowing down the pipeline. Some (or all) of the execute units should have multiplexors at

their inputs to select operands from one of the three instructions in a bundle. Show the multiplexors, and based on the slot restrictions show the minimum number of inputs.

Solution shown below. The ID stage provides register values for instructions in each slot (RS1v, etc). Note that if there were no restrictions on which instruction could appear in each slot then each mux would have three inputs. Also note that bypass paths are not shown.



(*c*) Notice in Table 3-10 that there is no template with a stop right after Slot 0 and right after Slot 1. Provide a possible reason for this.

With at most one internal stop a bundle can be issued in at most two pieces. With two internal stops the hardware would have to allow three pieces, one for each slot, to move separately. The added hardware cost would not be worth the small benefit in code size.

Suppose there was a template with two such stops (as described above), call this ISA Itanium-stop-stop. Why might code compiled for Itanium-stop-stop be smaller than code compiled for Itanium?

Because the compiler (or human) would no longer need to insert nops (as a last resort) to comply with the one-internal-stop restriction.

Consider Itanium and Itanium-stop-stop implementations that fetch one bundle per cycle (same as in the prior problems). Explain why Itanium-stop-stop might be no faster than Itanium.

Bundle placement and execution (assuming five stages) are shown for code for the regular Itanium and Itanium stop-stop. The last instruction, `or`, executes at the same time on both systems, so stop-stop has no execution time advantage. (If the hardware complexity results in a lower clock frequency then it would have performance disadvantage.)

```
// Itanium Scheduling
// Bundle 1 - Template 03          0  1  2  3  4  5  6  7
nop                 // Slot 0      IF ID EX ME WB
add r1 = r2, r3 ;;  // Slot 1      IF ID EX ME WB
sub r4 = r1, r5 ;;  // Slot 2      IF ID -> EX ME WB
// Bundle 2 - Template 02
nop                 // Slot 0         IF -> ID EX ME WB
xor r6 = r4, r7 ;;  // Slot 1         IF -> ID EX ME WB
or  r8 = r6, r9     // Slot 2         IF -> ID -> EX ME WB

// Itanium Stop-Stop

// Bundle 1 - Template 03          0  1  2  3  4  5  6  7
// Bundle 1 - Template SS-1
add r1 = r2, r3 ;;                 IF ID EX ME WB
```

```
sub r4 = r1, r5 ;;                  IF ID -> EX ME WB
xor r6 = r4, r7 ;;                  IF ID ----> EX ME WB
// Bundle 2 - Template SS-2
or  r8 = r6, r9                        IF ----> ID EX ME WB
```