COMP206 **Homework 2** Assigned: October 1, 2003
Prof. Montek Singh Due: October 8, 2003

Note: This homework assignment is in two parts, Part I and Part II. Part I consists of problems from the textbook (Hennessy/Patterson 3rd ed., or "HP3") whose solutions appear at the end of the book. These problems are for self-study only, and will not be graded, and therefore need not be submitted as part of your written work. Part II consists of problems that will count towards your grade for this homework.

Part I (not graded):

1. **Architectural Tradeoffs.** Do HP3 Problem 2.2.
2. **MIPS: Assembly Language Programming.** Do HP3 Problem 2.8.
3. **MIPS Pipeline: Hazards, Stalls and Forwarding.** Do HP3 Problem A.2. *Note:* In part (a), "branch is handled by flushing the pipeline" means that the branch policy is "always stall."
4. **Overhead of Control Hazards.** Do HP3 Problem A.3.
5. **Architectural Tradeoffs.** Do HP3 Problem A.4.

Part II (to be graded):

1. (10 points) **Architectural Tradeoffs.** Do HP3 Problem 2.4. Note: part (a) is already solved in Appendix B, so no credit will be given for it, but be sure to answer all of the remaining questions.
2. (15 points) **MIPS: Assembly Language Programming.** Do HP3 Problems 2.8 and 2.9, but *use the following C fragment instead:*

   ```
   for(i=1; i<=100; i=i+1)
     A[i] = B[i] - A[i] + C;
   ```

   Note: Be sure that your assembly code updates the final values of all the variables exactly as the C fragment does. Feel free to use any optimizations you would like, in order to make the code more efficiet. Be sure that every line of assembly code you write is properly commented.
3. (20 points) **Instruction Set Example.** Consider the ultimate in simple instruction sets: the "Single Instruction Computer" (abbreviated SIC). SIC has only one instruction: Subtract and Branch if Negative, or `sbn` for short. The `sbn` instruction has three operands, each consisting of the address of a word in memory:

   ```
   sbn a, b, c   # Mem[a] = Mem[b] - Mem[a]; if (Mem[a] < 0) goto c
   ```

The instruction will subtract the number in memory location a from the number in location b and place the result back in a, overwriting the previous value. If the result is greater than or equal to 0, the computer will take its next instruction from the memory location just after the current instruction. If the result is less than 0, the next instruction is taken from memory location c. SIC has no registers and no instruction other than `sbn`. Make the following assumptions in this problem:

- You have available spare memory words that can be used for temporary results; give symbolic names to these locations, and use these symbolic names in the generated code.
- You can initialize the contents of memory locations with compile-time constants. Once again, use mnemonic symbolic names for such constants (*e.g.,* `zero, one,` etc.).
- Use "." to indicate the address of the current instruction, and ". + k" to indicate the address of the k-th instruction after the current instruction.

(a) Write a sequence of SIC instructions to copy a number from location a to location b.

(b) Generate SIC code for the C program fragment

```
c = a - b;
```

where a, b, and c are distinct memory locations holding integer values. Ignore overflow, but correctly handle negative as well as positive operands.

(c) Generate SIC code for the C program fragment

```
c = a div b;
```

where a and b are non-negative integers in distinct memory locations, and a `div` b represents the quotient when a is divided by b. Assume that b is non-zero.

(d) What is the instruction count of the SIC code for unsigned integer division from part (c) above?

4. (20 points) **MIPS Integer Pipeline: Hazards, Stalls and Forwarding.** Consider the following program fragment to compute values for variable D. All variables are memory-resident, integer-valued, and located at distinct memory addresses.

```
D = (A*X+B)*X+C;
```

A compiler for a MIPS machine generates the following code for this program fragment: (The memory addresses are symbolic names.)

```
1:  LW R1, X
2:  LW R2, A
3:  MULT R10, R1, R2
4:  LW R3, B
5:  ADD R10, R10, R3
6:  MULT R10, R10, R1
7:  LW R4, C
8:  ADD R10, R10, R4
9:  SW D, R10
```

(a) Enumerate all of the data hazards that exist among the 9 instructions. Classify them as RAW, WAR, and WAW hazards.

(b) Draw the precedence graph for the program fragment.

(c) Assume a five-stage linear pipelined implementation of MIPS with the two additional assumptions (both admittedly unrealistic): (i) all operations spend a single cycle in the EX stage, and (ii) there are no forwarding paths whatsoever.[1] Represent the execution of the program above with the help of a space-time diagram, and determine the number of clock cycles it will take to complete execution. Clearly indicate all stalls.

(d) Still assuming no forwarding, rearrange the order of instructions in this program to reduce execution time without violating any precedence constraints. Draw a space-time diagram to represent the execution of the rearranged program.

(e) Now assume that all reasonable forwardings are available. Draw a space-time diagram for the execution of the *original* program.

(f) Still assuming that all reasonable forwardings are available, draw a space-time diagram for the execution of the *rearranged* program.

(g) Summarize the execution times of the four program executions in a two-dimensional table. Label the y-axis with the lack (or availability) of forwarding, and the x-axis with the lack (or availability) of compile-time scheduling.

5. (20 points) **MIPS: Forwarding and Stalling (for Branch Instructions).** The HP3 textbook mainly focuses on forwarding and stalling, where the "consuming" instruction (*i.e.,* the destination instruction, which reads a register) is an ALU operation. In this problem, you are to indicate how to implement forwarding and stalls in MIPS, where the consuming instruction is a branch instruction.

   Assumption: Assume the optimized branch hardware as shown in Figure A.24, which allows branches to be completed at the end of the ID stage.

   (a) *Branches: Bypassing.* Write a short MIPS code sequence (3 instructions or fewer!), where the first instruction writes to register R1, and the last instruction is the branch instruction "`BEQZ R1, Exit`" (which reads the new R1 value), such that the new value of R1 can be bypassed to BEQZ in the pipeline without stalling.[2] Also, draw a simplified figure of the MIPS pipeline, showing BEQZ when it is in the ID stage, the other instructions in their corresponding stages, and sketch how the bypassing hardware is implemented, which forwards the new result from the "producer" instruction to the "Zero Detect Unit" in ID.

   (b) *Branches: Stalling.* Repeat part (a), but now illustrate a case where the final branch instruction must stall. Again, your MIPS code sequence should be 3 instructions or fewer, where the last instruction is: "`BEQZ R1, Exit`".

   (c) *Branches: Complete Stall and Forwarding Control Tables.* Create two small tables, similar to those in the HP3 textbook (see Figs. A.21 (stalls) and A.22 (forwarding)), to specify the "stall" and "forwarding" control for all the different cases where "BEQZ R1, Exit" is the consuming instruction (reads a value of R1). Your two tables should specify the symbolic conditions for all cases, involving a branch as the consumer instruction, where stalling or bypassing are required.

[1] Note, however, that a combination of a register write and a register read in the same clock cycle "forwards" through the register file, because the latter occurs on a later clock edge.

[2] Recall that BEQZ stands for "*B*ranch if the operand is *EQ*ual to *Z*ero."

Note: Your tables should list the symbolic conditions for the *optimized* branching hardware of Figure A.24. The symbolic conditions listed in the textbook (Tables A.21 and A.22) are applicable only to the *unoptimized* branching hardware of Figure A.18!

Hint: The solution requires very few table entries!

6. (15 points) **Delayed Branches.** Do HP3 Problem A.7.