Computer Architecture Practical 1 – Pipelining

Issued: Monday 28 January 2008
Due: Friday 15 February 2008 at 4.30pm (at the ITO)

This is the first of two practicals for the Computer Architecture module of CS3. Together the practicals make up for 25% of the final mark for the module. This practical consists of pen-and-paper exercises. Assessment of this practical will be based on the correctness and the clarity of the solution. This practical is to be solved individually to assess your competence on the subject. Please bear in mind the School of Informatics guidelines on plagiarism. You must return your solutions to the ITO before the due date and time shown above.

Problem 1 – [10 Marks]

Use the following code fragment:

```
loop: L.D    F4,  0(R2)
      L.D    F6,  0(R3)
      MUL.D  F8,  F4,F0
      MUL.D  F10, F6,F2
      ADD.D  F12, F8,F10
      DADDUI R2,  R2,8
      DADDUI R3,  R3,8
      DSUBU  R5,  R4,R2
      BNEZ   R5,  loop
```

Assume that the initial value of R4 is R2+7992, that R2 and R3 contain the base addresses of some arrays, and that F0 and F2 contain some data values pre-loaded prior to the loop. For this exercise assume the standard five-stage integer pipeline and the MIPS FP pipeline as described in the lectures. Assume the latencies and initiation intervals shown in the table below. If structural hazards are due to write-back contention, assume the earliest instruction gets priority and other instructions are stalled.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory	1	1
FP add	3	1
FP multiply	6	1
FP divide	24	24

a. Show the timing of this instruction sequence for the MIPS FP pipeline without any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle "forwards" through the register file. Assume that the branch is handled by flushing the pipeline. If all memory references hit in the cache, how many cycles does this loop take to execute?

b. Show the timing of this instruction sequence for the MIPS FP pipeline with normal forwarding and bypassing hardware. Assume that the branch is handled by predicting it as not taken. If all memory references hit in the cache, how many cycles does this loop take to execute?

Problem 2 – [10 Marks]

A machine is called "underpipelined" if additional levels of pipelining can be added without changing the pipeline-stall behaviour appreciably. Suppose that the five-stage MIPS integer pipeline was changed to four stages by merging EX and MEM and lengthening the clock cycle by 50%. Assume that branches are resolved in the ID stage in both cases. How much faster would the conventional MIPS pipeline be versus the underpipelined MIPS on integer code only? Make sure you include the effect of any change in pipeline stalls assuming that, in the application mix running on the five-stage pipeline, 4% of instructions stall due to branches and 5% of instructions stall due to loads.

Problem 3 – [10 Marks]

Here is an unusual loop. First, list the dependences and then rewrite the loop so that it is parallel.

```
for (i=1; i<100; i=i+1) {
   a[i] = b[i] + c[i];   /* S1 */
   b[i] = a[i] + d[i];   /* S2 */
   a[i+1] = a[i] + e[i]; /* S3 */
}
```

Problem 4 – [20 Marks]

Assume the pipeline latencies shown in the table below, and a one-cycle delayed branch. Unroll the following loop a sufficient number of times to schedule it without any delays. Show the schedule after eliminating any redundant overhead instructions. Despite the fact that the loop is not parallel, it can be scheduled with no delays.

```
loop: L.D   F0, 0(R1)
      L.D   F4, 0(R2)
      MUL.D F0, F0, F4
```

```
ADD.D F2, F0, F2
DADDI R1, R1, -8
DADDI R2, R2, -8
BNEZ  R1, loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Problem 5 – [30 Marks]

In this Exercise, we will look at how a common vector loop runs on a variety of pipelined versions of the MIPS integer and FP pipelines. The loop is the so-called DAXPY loop (double-precision aX plus Y) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, $Y = a * X + Y$, for vector length 100:

```
loop: L.D    F2, 0(R1)   ; load X(i)
      MUL.D  F4, F2, F0  ; multiply a*X(i)
      L.D    F6, 0(R2)   ; load Y(i)
      ADD.D  F6, F4, F6  ; add a*X(i) + Y(i)
      S.D    0(R2), F6   ; store Y(i)
      DADDUI R1, R1, 8   ; increment X index
      DADDUI R2, R2, 8   ; increment Y index
      DSGTUI R3, R1, 800 ; test if done
      BEQZ   R3, loop    ; loop if not done
```

where `DSGTUI` is the instruction *double set greater than unsigned integer*, which sets the result register to true if the value of the source register is greater than the unsigned immediate value. For the questions below, assume that the integer operations issue and complete in one clock cycle (including loads) and that their results are fully bypassed. Ignore the branch delay. Use the FP latencies from Problem 4 in this Practical. Assume that the FP units are fully pipelined.

a. Assume the single-issue MIPS pipeline described above. Do not perform any re-ordering of instructions. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution on the first iteration of the loop. How many clock cycles does each loop iteration take?

b. Unroll the code to make four copies of the body and schedule it for the single-issue MIPS pipeline described above. When unrolling and scheduling, you should optimise the code to eliminate as many stalls as possible. How many clock cycles does each loop iteration take?

c. Consider the original (non-unrolled) DAXPY code. Assume a hardware with Tomasulos algorithm with one integer unit, three FP adders, and two RP multipliers. Show the state of the reservation stations and register-status tables (as in the slides of Lecture 7) when the DSGTUI writes its result on the CDB. Do not include the branch.

d. Assume a superscalar architecture that can issue two independent operations in a clock cycle (including two integer/load-store operations). Unroll the DAXPY code to make four copies of the body and schedule it. Assume one fully pipelined copy of each functional unit (including the integer unit). When unrolling and scheduling, you should optimise the code to eliminate as many stalls as possible. How many clock cycles will each iteration take?

Problem 6 – [20 Marks]

Consider a predicated instruction set where all instructions have been augmented with predication. So, for instance, a predicated ADD would have the following syntax:

```
(p1) ADD R1,R2,R3
```

which would be executed to completion if p1 is TRUE and would be discarded if p1 is FALSE. Assume the RISC architecture is extended with 8 predicate registers (p1, ..., p8). Consider also that a special compare instruction is used to set the predicate registers as follows:

```
CMP.EQ p1,p2=R1,R2
```

which would set p1 to TRUE and p2 to FALSE if R1=R2, and p1 to FALSE and P2 to TRUE if R1!=R2. Consider the following code fragment:

```
      DSUB  R1, R13, R14
      BNEZ  R1, L1
      DADDI R2, R2,1
      SD    0(R7), R2
      J     L2

L1:   MUL.D F0, F0, F2
```

```
     ADD.D F0, F4, F0
     S.D   0(R8), F0

L2: ...
```

a. Using predicated instructions, write this code fragment as a single basic block. List all the data and control dependences in the original code fragment and in your predicated version.

b. Assume the 5-stage MIPS integer pipeline and the FP MIPS pipeline with the latencies of Problem 4. Branches and jumps are resolved in the ID stage and the (possibly) incorrectly fetched instruction is discarded. Assume full bypassing is supported for both integer and FP register operands. Further assume that predicated instructions obtain the values of the predicated registers in the beginning of the ID stage (and stall if they are not yet available) and are either allowed to continue or are turned into NOPs by the end of the ID stage. Finally, assume that the compare instruction described above generates the predicate results at the end of the ID stage (it only requires a simple comparator) and bypassing is supported for predicated register operands from ID back to ID. Show the timing of the original instruction sequence and your predicated version when each path is executed. What is the performance improvement (if any) of the predicated version for each execution path?

Nigel Topham 2008