CS 211 - Mid-Term Examination - Summer 2005

Answer All Questions

Time: 2 hours

40 points

NAME:

Part A.

Ques.1: (10 points)

For each of the questions, underline the correct statement(s).

- All the region based ILP scheduling methods, when compared to scheduling basic blocks,
  1. always result in larger compilation time
- The size of the executable code may increase when using
  1. speculative instructions
  2. superblock scheduling
- The performance of a pipelined processor is typically measured using
  1. throughput
- The stall cycles due to a branch instruction depends on
  1. the time to compute the branch target
  2. the time to determine branch outcome
- To recover from mispredicted branches in superscalar processors,
  1. branch validation and recovery mechanisms must be provided
- Speculative execution allows
  1. instructions (that are ready) to be executed before they are needed.
- In the study of k-bit branch predictors storing history on n branches (where k bits of history is stored for each branch, and n is number of entries in the branch history table), it was observed that
  1. the performance levelled off after 4K entries in the history table.
  2. a 3-bit history was shown to be enough for most benchmarks
- Predication in EPIC/VLIW processors
  1. eliminates branch effects by executing both branches
The motivation for EPIC/VLIW processors comes from

1. trends in semiconductor technology
2. more parallelism can be extracted by software than by the hardware schedulers

The theoretical limit on the maximum performance (i.e., minimum time) of a piece of code (i.e., set of instructions) is determined by

1. the dataflow limit

Ques. 2: (10 points) Provide short answers for at least FIVE of the following six questions. (You can answer the extra question for extra credit towards your total grade, but state which question is your extra credit question). Your answers must be brief and your answer, for entire question 2, MUST fit on the one page provided (in the next sheet). Any violation of these conditions will result in no credit.

(a) Using a simple example pseudo code illustrate how the speculative model of execution has the potential of increasing processor performance in EPIC processors? An unused slot (time and functional unit) in an ILP processor can be filled with an instruction that is ready to execute but further down in the execution sequence. If the speculation is correct, then the results of that operation would be available by the time that instruction is needed.

```
loop: ADD R1, R2, R3
      MUL R4, R2, R1
      SUB R2, R2, #4
      B NZ loop
      LD R5, 0(R6)
      ....
```

The LD instruction can be executed before the branch in a speculative manner – since the load units are free in the “loop” body, this allows utilization of an unused unit. If the branch is not successful then the speculative LD will be a useful instruction thus reducing the execution cycles.

(b) For the example code below show how internal forwarding works (using the 5 stage MIPS architecture as the example architecture).

```
add R1, R2, R3
mul R3, R1, R2
load R4, 0(R5)
add R6, R4, R3
```

The data computed by the add instruction will be forwarded by the ALU stage as input to the mul instruction thus saving a stall cycle. The second add instruction needs data from the load, and we still need one stall cycle. Once data is loaded from memory it is forwarded to the input of the ALU for the second add instruction.
• (c) For the following code segment, show the true dataflow graph and derive its 'dataflow limit' (i.e., critical path). Are there any false dependencies in the code and if so, how can they be removed?

A: R4 ← R0 + R8
B: R2 ← R0 * R4
C: R4 ← R4 + R8
D: R8 ← R4 * R2
E: R4 ← R1 + R3

• (d) What is Trace scheduling and why does lead to improved performance in ILP processors? Trace scheduling forms a large region of code based on the most likely execution path – i.e., common case. By providing a large region of code to the scheduler, we enable the extraction of more parallel instructions.

• (e) Using an example in pseudo-code show the impact of register allocation on the performance of a program.

```
add r0, r1, r2
mul r1, r3, r2
```

There is a false dependence between the two instructions due to register r1. By allocating a different register, say r4, to the output of the mul instruction we now get two independent instructions that can be executed in one cycle on an ILP processor.

• (f) Using pseudo-code illustrate how the concept of predication works in EPIC processors. For the code below, statements S1 are control dependent on the branch outcome and are executed if the predicate X > 0 is true. Similarly, S2 is executed if a predicate is false.

```
if (x>0) then S1
else S2
```

By setting a predicate p1 true if \( x > 0 \) and a predicate p2 true if \( x \leq 0 \) we can rewrite the code in a predicated manner and execute both branches tied to specific predicates.

```
p1 if x>0
p2 if x<= 0
S1 if p1
S2 if p2
```

Space for Answer Question 2: Your answer MUST fit on this page
Part B: (20 points) Part B requires you to answer four questions—each is worth 4 or 6 points, so pace yourself (and "schedule" your answers accordingly).

Ques. 3 (5 points)
Simulate the execution of the following piece of code using Tomasulo’s algorithm—there are five instructions labelled A through E. Show the contents of the reservation station (RS) entries, register file busy, tags (tag is the RS ID number—recall that tag is set to 0 if operand ready, else set to RS ID producing result/operand), and data fields for each cycle. Indicate which instruction is executing in each functional unit in each cycle. Also indicate any result forwarding across a common data bus by circling the producer and consumer and connecting them with an arrow. (If it makes it easier for you, you may want to 'replicate' the base diagram used in the book and notes for each cycle). If the notation is confusing, then use the notation you are most familiar with. Assume the following:

- dual issue (i.e., you can issue/dispatch two instructions in each cycle)
- latency is 2 cycle for add and 3 for multiply
- an instruction can begin execution in the same cycle that it is dispatched assuming all dependencies are satisfied.
- in-order issue of instructions
- Assume initially R1=10, R3=10, R0=6, R2=3.5, R4=10, R8=7.8

A: R1 ← R1 * R3
B: R4 ← R0 + R8
C: R2 ← R0 * R4
D: R4 ← R4 + R1
E: R8 ← R4 * R2

First note that the dataflow limit is 8—this is the lower bound on how fast the computation can be completed. We are assuming no limits on the number of reservation stations—for add and mul—but let’s assume ONE add unit and ONE mult unit. The analysis will change if we change these assumptions.

We have a dual dispatch processor—we can dispatch two instructions in one cycle.

- Cycle 1: Dispatch A and B. Both operands for A and B are ready, so issue A to res. station for Mul and B to res. station for Add.
  B is waiting in reservation station for Mul. since it needs output of A (i.e., operand to be sent to R4).
- Cycle 2: Dispatch C and D to reservation stations. Both have to wait for earlier instructions so they are not issued.
- Cycle 3: B completes, this means the operands for C are ready. So issue C but we have to wait for Mul to be free (since A is using it and we assume one Mul unit). Note that D still waits still A has not completed and it needs data generated by A. We can dispatch E if we have at least 2 res. stations for the Mul unit.
• Cycle 4: A completes execution. This means the Mult unit is free so we can start executing C. Since operands are ready for D, we start execution of D in the Add unit.

• Cycle 5: C,D are still executing, and E is waiting in the res. station for Mult.

• Cycle 6: D completes and C is still executing since Mult needs 3 cycles to complete, and E waits since it also needs result from C.

• Cycle 7: C completes, so we can start executing E.

• Cycle 10: E completes and we are done executing the code segment.

space for answer to Ques.3

Ques.4: (4 points) List all the dependencies (true, anti, output – i.e., RAW, WAR, WAW) in the following code fragment (which has four statements in the loop body). Indicate whether the true dependencies are loop carried or not. Can you rewrite the loop so that it is parallel (i.e., remove loop carried dependencies) ? (You have to explain your answer – no credit will be given for simple Yes or No answer.)

... 

for (i=2; i<100; i=i+1){
    a[i] = a[i] + b[i];  /* S1 */
    c[i-1] = a[i] + b[i];  /* S2 */
    a[i-1] = 2*b[i];  /* S3 */
    b[i+1] = 2*b[i];  /* S4 */
}

• anti-dependence from S1 to S2 on a

• true dependence from S1 to S2 on a

• loop carried true dependence from S4 to S1 on b

• loop carried true dependence from S4 to S3 on b

• loop carried true dependence from S3 to S3 on b

• loop carried output dependence from S3 to S3 on a

For loop to be parallel each iteration must be independent (i.e., no loop carried dependencies), which is not the case here, therefore it cannot be parallelized. Because dependencies 3,4,5 are true loop carried dependencies they cannot be removed – therefore loop cannot be made parallel.
Ques.5: (6 points)

Consider the handling of branches in dynamic ILP processors using hardware solutions such as branch prediction buffers. Assume that you have a 2-bit branch history table, and a finite state machine based branch predictor. First, give a 2-bit finite state machine based branch predictor. (Use the standard notation – i.e., N means Not Taken and is represented by a 0, and T means branch taken and is represented by a 1. For example, a state NN/N represents history of NN, predicts N=not taken, and goes to state NT/N if the actual outcome is a T=taken branch.)

For the sample code shown below, which has five branches B1, B2, B3, B4, and B5 compute the accuracy of the branch prediction method (using your branch prediction algorithm that you defined in part (i)) for (1) each branch and (2) for the overall program – i.e., what is the percentage of branch mis-predictions (or you can compute the percentage of correct predictions). The history bits for each branch are shown in the line. For the program below, do you think a 2-level branch predictor (correlating branches) can perform better? Why.

```
{
  int i

  i = 0;
  do {
    x := 50 - i;
    y := i - 50;
    if (x > 0) { /* branch B1: History bits in table = NT */
      /* some straight line code that doesn’t change any of i,x,y,z ..*/
    }
    if (y >= 0) { /* branch B2: History bits in table = NN */
      /* some straight line code that doesn’t change any of i,x,y,z..*/
    }
    i = i+1;
  } while (i < 100); /* branch B3: history bits = TN */
  if (z>0) { /* branch B4: history bits = TT , assume z=10 */
    /* some straight line code that doesn’t change z */
  }
  if (z=<0) { /* branch B5: history bits = NN */
    /* some straight line code that doesn’t change z */
  }
}
```

Let’s consider the branch predictor in the figure.

- Branch b1 is executed 100 times. Initially the value of $x = 50$ and the branch is taken. It is taken 50 times, before failing at $i = 50$ ($x = 0$). The predictor predicts N when $i = 0$ which is wrong; the predictor next state is TT which then predicts T for the next iteration onwards. At iteration $i = 50$ the predictor predicts T but the actual outcome is N and the next state is TN. At iteration $i = 51$, the prediction again is T but the actual
outcome is N and the next state NN (from where it will predict correctly to be N). Thus, there are a total of 3 mispredictions out of 100.

- Branch b₂ is executed 100 times. Initially, at \( i = 0 \) the value of \( y = -50 \). The predictor predicts N and the outcome is N. This continues till iteration \( i = 50 \). The prediction is N but the actual outcome is T and the next state is NT. At \( i = 51 \) the prediction is again N but outcome is T and the next state is TT. Thus, it has 2 mispredictions out of 100.

- Branch b₃ is executed 100 times. First time at \( i = 1 \) the prediction is T and the outcome is T and the next state is NT. Next, at \( i = 2 \) the prediction is N and the outcome is T and the next state is TT. At the last iteration, \( i = 101 \), the prediction is T and the outcome is N. Thus, it has 2 mispredictions out of 100.

- Assume \( z = 10 \). Branch b₄ is executed once. The prediction is T and the outcome is T.

- Branch b₅ is executed once. The prediction is N and outcome is N.

- The total mispredictions rate is 8/302.

- b₄ and b₅ are related – if one is taken the other is not. Thus a corelating branch predictor works well for these two branches.

**Space for answer to Ques.5**

**Ques.6:** (5 points) For the sample code below (shown as a dependency graph – a DAG) show the schedule generated by the list scheduling algorithm for an ILP processor with 2 load units and 2 floating point arithmetic units. First, show the node ranking function that you choose. The latencies for each operation are shown in the figure. Show all steps in the schedule. What
is the computation being performed by the code? What is the dataflow limit (i.e., lower bound
on the computation time)? Will speculation and predication help in speeding up this code −
explain?

**Space for Answer to Ques. 6**

The computation being performed is the equation \(((a + b) * (c - d)) + (e/f))

The critical path is the path that has divide in it i.e., it has length 7 and this is the dataflow
limit.

Your exact schedule depends on what ranks you have given the nodes. Let us consider
the following rank/priority list: (5, 6, 9, 3, 4, 1, 2, 7, 8, 10, 11). With this rank the schedule is as
follows, where Load1 and Load2 refer to the two load units and FP1 and FP2 refer to the two
floating point units, and at each time we show the label of the node being scheduled:

<table>
<thead>
<tr>
<th>Time</th>
<th>Load1</th>
<th>Load2</th>
<th>FP1</th>
<th>FP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

Prediction or speculation will not help with this code since there are no branches in the
code.