

CS 211 - Mid-Term Examination - Fall 2008. Solutions

Section A.

Ques.1: 10 points For each of the questions, underline or circle the most suitable answer(s).

- The performance of a pipeline processor is typically measured in terms of
 1. latency of the pipeline—partially correct
 2. throughput of the pipeline
- Internal forwarding is a technique in pipelines that
 1. reduces stalls due to RAW dependencies
- Data speculation technique in EPIC processors works by
 1. moving load instruction before a store instruction
- A key difference between EPIC and Superscalar processors is
 1. EPIC requires dependency free code whereas superscalar does not
- In the study of the performance of history based branch predictors (with n history bits and k entries in the branch history table), it was observed that
 1. performance levels off once k exceeds some number of entries.
- If an exception occurs during speculative execution
 1. the exception handling must be deferred until branch outcome is known
- In a speculative execution, or out of order execution, in superscalar processors using the generalized Tomasulo method
 1. writes to registers are deferred until instruction is committed and retired
- One difference between classic VLIW architecture and the EPIC architecture model is
 1. EPIC allows functional units with different latencies but VLIW does not
- Adding a predecoding stage in superscalar processors
 1. increases issue rate by decreasing time taken in the decoding stage
- Out of order execution allows
 1. instructions behind a stalled instruction to proceed to execution
- The theoretical limit on the maximum performance (i.e., minimum time) of a piece of code is determined by

1. the dataflow limit, also known as critical path

Ques. 2: (10 points) Provide short answers for the following questions. There are six parts in this question and you must answer **at least five**. Your answers must be brief and you should be able to answer each question in no more than three sentences (with the exception of the cases where sample code is required). **Your answer, for entire question 2, MUST fit on the one page provided. Any violation of these conditions will result in no credit.**

- (a) What does the Branch target buffer (BTB) store, and how is this information used ?
The branch instruction address (used to do an associative search), branch target address, and the history (prediction) bits.

- (b) Why does speculative execution have the potential of increasing processor performance in ILP processors ? Illustrate your answer by using a simple example in pseudo-code.
Can execute instructions that are ready to execute (i.e., their data is ready) before they are needed by utilizing any unused functional units. Example of an instruction after a branch being executed before the branch condition is evaluated.

- (c) Does internal forwarding always eliminate all stall cycles due to data hazards in a pipeline? You can justify/explain your answer by using a (very) simple pseudo (assembly) code.

No- it does not eliminate all stalls. An example is a Load of a value into a register followed by a use of that register. This will still have a one cycle stall.

- (d) Consider a processor with the following CPI for each type of instruction, and the instruction mix for a particular benchmark. Determine the effective CPI for this benchmark.

	ALU	Load/Store	Cond.Branch	Jumps
CPI	1	1.5	2	1.2
Inst.Mix	40%	20%	30%	10%

Determine the weighted average CPI based on the instruction frequencies: $CPI = (1 * 0.40) + (1.5 * 0.2) + (2 * 0.30) + (1.2 * 0.10)$

- (e) For the following code segment, show the true dataflow graph and derive its critical path assuming all operations take one cycle. Identify any false dependencies in the code if they exist.

```
A: R4 <- R0 + R8
B: R2 <- R0 * R4
C: R4 <- R4 + R8
D: R8 <- R4 * R2
E: R4 <- R1 + R3
```

False dependencies between D and E (WAR) on R4, between B and C on R4, and between C and E (WAW) on R4.

- (f) Your computer vendor offers three optional enhancements for the same price. The speedup of each enhancement and its applicability (usability in pre-enhanced instructions) is given below. (For example, enhancement A can only be applied to 50% of the code – the remaining code has to be executed in un-enhanced mode.) If you can only buy one optional enhancement, which one should you buy ?

	Speedup	Usability
A:	5×	50% of pre-enhanced instructions
B:	10×	40% of pre-enhanced instructions
C:	100×	30% of pre-enhanced instructions

Apply Amdahl's law to get equation for speedup S as:

$$S = 1/(f_{unenhanced}) + (f_{enhanced}/S_{enhanced})$$

The speedup equations are: $S_A = (1/(1-0.5) + (0.5/5)) = 1/0.6$, $S_B = 1/(0.6 + 0.4/10) = 1/0.640$ and $S_C = 1/(0.7 + 0.3/100) = 1/0.703$. Therefore A gives best speedup.

Ques.3: CPU Performance Models.

Consider the following scenario: The students in CS211 have proposed a novel solution to improve the CPU time. By examining the dataflow of many benchmarks, they found that a combined ALU operation can be provided for a small increase in CPI and clock cycle but a decrease in instruction count. Specifically, they proposed that two types of 2-input operand ALU operations with true dependencies (which they termed as “GW dependency instruction pairs”) be combined into one 3-input operand operation called a GW Instruction (GW-Int). For example, ADD R1,R2,R3 ($R1 = R2 + R3$) and MUL R5,R1,R4 ($R5 = R1 * R4$) can be combined into one GW-Int ADMUL R5,R2,R3,R4 operation ($R5 = (R2 + R3) * R4$). Thus, two instructions are replaced by one. However, (1) the effective CPI of these new GW-Int instructions only (for example, the ADMUL) is increased 25% (the original ALU instructions left unchanged will not have an increase in CPI) and (2) the overall processor clock cycle for operations increases by 5%. Note that if a program originally had 10 GW dependent instruction pairs (i.e., 20 instructions) then with the new design it will have 10 instructions replacing the original 20 instructions, and (a) these 10 (and not other ALU operations) will take 25% more CPI and (b) the processor clock cycle is 5% longer for all instructions.

(a) First derive the CPU time equation for the old design and the new design assuming an ideal CPI of 1 for all instructions in the old design, and assuming that $x\%$ of GW dependent instructions ($x\%$ of the original set of instructions) have been converted to the new GW instructions. What is the minimum percentage of the original instructions that must be eliminated for the new design to have a lower CPU time (derive the equation).

(b) Now consider following instruction mix, and original effective CPI for each type of instruction. What is the smallest percentage (of the entire code) of GW dependent instructions that must be converted (to the new GW-Int form) for the new design to have a better performance than the old design.

	ALU	Load/Store	Cond.Branch	Jumps
CPI	1	2.0	2	1.5
Inst.Mix	40%	20%	30%	10%

(a)

Let $CPU_{old} = IC_{old} * CPI_{old} * Clk_{old}$. If $x\%$ have been converted to the new instructions, note that for every two such old instructions we have 1 new instruction with the higher CPI. Therefore the new CPU time can be expressed as:

$$CPU_{new} = [(1 - x) * IC_{old} * CPI_{old} + (x/2) * IC_{old} * 1.25 * CPI_{old}] * 1.05 * Clk_{old}$$

For the new design to have less time, $CPU_{new}/CPU_{old} < 1$. Cancelling out the common terms $IC_{old}, CPI_{old}, Clk_{old}$ we get:

$$[(1 - x) + (x/2) * 1.25] * 1.05 < 1$$

(b)

Using the earlier formula, now plug in the actual instruction mix with the effective CPI numbers.

$$CPU_{old} = (0.4 * 1.0 + 0.2 * 2 + 0.3 * 2 + 0.1 * 1.5) * IC_{old} * Clk_{old} = 1.55 * IC_{old} * Clk_{old}$$

If we convert instructions to the new type, note that only the ALU instructions will be converted. Assume that $y\%$ of the ALU instructions have been converted to the new form; i.e., the overall code that has been converted will be $x = 0.4 * y$ since the ALU is 40% of the overall code. The new CPU time is given by:

$$CPU_{new} = [(0.4(1-y)*1.0+0.4*(y/2)*1.0*1.25)+0.2*2+0.3*2+0.1*1.5]*IC_{old}*Clk_{old}*1.05$$

Cancel out the common terms, and set $CPU_{new}/CPU_{old} < 1$ to solve for y .

Ques.4: 5 points Tomasulo and Branch Prediction.

(a)

- In cycle 1, with a 4-issue window we can issue up to 4 instructions. However, there is a structural hazard if we issue A,B,C,D since there are only 2 reservation stations for ADD. Therefore instructions A,B,C are issued. A and B have their operands ready, therefore they are issued to Add1 and Mult1 with values and are ready for execution. C is issued to Add2 with tags Add1 and Mult1 (since it needs the outputs of A and B).
- In cycle 2, we cannot issue E since D has not been issued (and E needs output of D and C). A completes execution. B is still executing.
- In cycle 3, A has completed and released its functional unit. D can be issued to Add1 with values R0 and R2 (which was generated by A) and D starts execution (it will complete in cycle 4 and release the unit in cycle 5). E can be issued to Mult2. B completes execution in Mult1 and will release the unit in cycle 4.

(b)

An examination of the code reveals that two branches B1 and B2 are co-related – if one is taken then the other is not taken. Therefore a correlating branch predictor can give an improved performance compared to a standard 2-bit predictor.

Ques.5: Loop unrolling and MIPS pipeline scheduling.

(a)

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

- true RAW dependency between S1 and S2 on $a[i]$
- false WAR dependency between S1 and S1 on $a[i]$
- false WAR dependency between S2 and S2 on $b[i]$
- false WAP dependency between S3 and S3 on $c[i]$
- true RAW loop carried dependency between S3 and S1 on $c[i + 1]$

Since the loop as written has a loop carried true dependency it cannot be unrolled.

Upon further examination, note that we can move the statement $c[1] = 2 * c[1]$ inside the loop to start as the first instruction. Thus we can change S3 to updating $c[i]$ and move it to be the first statement in the loop. This removes the loop carried dependency and makes it possible to unroll the loop.

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

(b)

There are RAW dependencies between one instruction and the next in sequence. Forwarding will get rid of the dependencies between consecutive ALU operations (add, mul) but the load followed by add will still have a stall cycle since the result of the load will not be available till after the memory access stage.

Ques.6: Scheduling ILP/EPIC Processors

- How can control speculation be used to schedule the code above ? Indicate the speculative instructions (and the type) and indicate what speculation check instructions are needed.
 - In the first cycle, the arithmetic units are free since the first instruction is a Load (and others need this data). Therefore, we can speculatively execute `MULT R10,R10,R10` in the first cycle before the branch has executed. There is no correction code required since R10 is not used anywhere else in the code.
- How can predication be used to schedule the above code ? Indicate the predicated instructions.
 - Predication results in both branches being executed. In the case above, set a predicate P1 to be true if R4 and R2 are equal. Set predicate P2 to be true if they are not equal. Instructions at Next are predicated with P1, instructions after branch (in sequence) are predicated with P2.
- How can data speculation (i.e., advanced loads) be used to schedule the above code ? Indicate the speculative instructions (and the type) and indicate what speculation check instructions are needed.
 - The fourth instruction is a Load `R3, (R2 + 100)`, and it can be moved before the Store instruction (the third instruction) using data speculation. The correction code (if there was an intervening write to the same location) would be to reload the data – i.e., Load `R3, (R2 + 100)`.
- How can data and control speculation be used to schedule the above code ? Indicate the speculative instructions (and the type) and indicate what speculation check instructions are needed.
 - We can move the Load instruction at Next to execute before the branch. This would mean control speculation. However, if we move the load to execute before the Store `(R7), R2` then this would also imply data speculation. The correction code would be to reload the data.