CSE 471 Autumn 2001

Computer Design and Organization

Midterm

Wednesday November 7th

NAME : ______________________________

Do all your work on these pages. Do not add any pages. Use back pages if necessary. Show your work to get partial credit.

This exam is worth 40 points. After each question, you will find the number of points it is worth. You should spend approximately x minutes on a question worth x points. That will leave you with 10 minutes to read the statement of the problem and to look over your work.

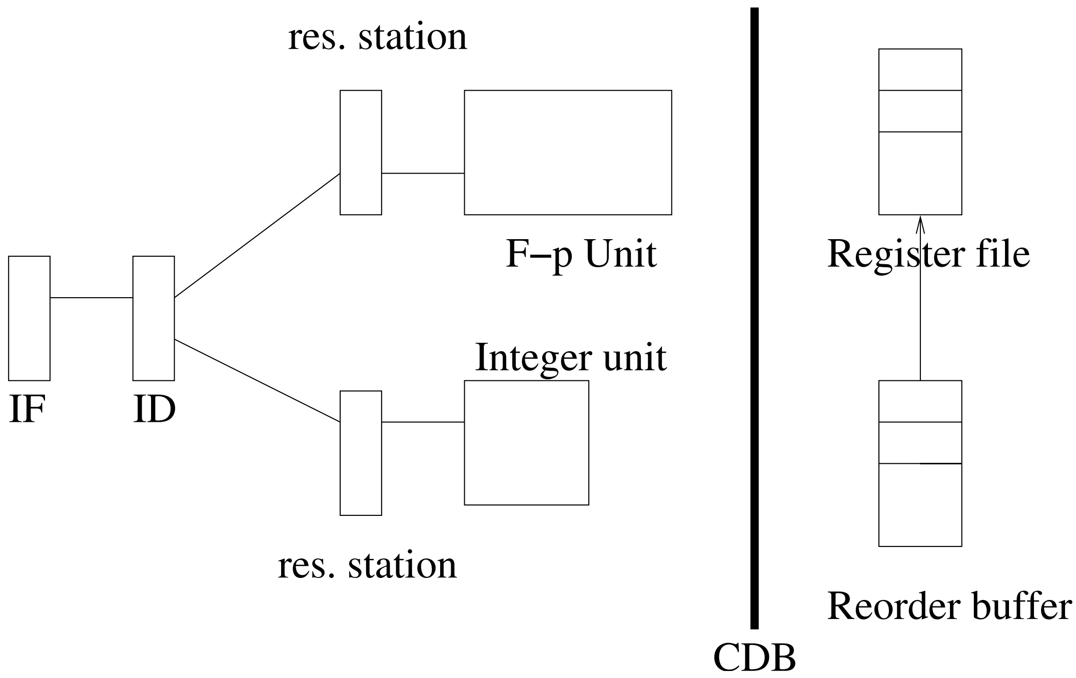
1. 20 points
    - (a) 1 point ______________
    - (b) 3 points ______________
    - (c) 3 points ______________
    - (d) 2 points ______________
    - (e) 3 points ______________
    - (f) 2 points ______________
    - (g) 4 points ______________
    - (h) 2 points ______________
2. 20 points ______________
    - (a) 4 points ______________
    - (b) 2 points ______________
    - (c) 3 points ______________
    - (d) 8 points ______________
    - (e) 3 points ______________

The figure below sketches the skeleton of an out-of-order processor that can dispatch, issue, and execute 32-bit instructions out-of-order but requires a commit step to complete them in order. It has two functional units: one for integer, load/store and branches, and the other for floating-point operations. Each of the two units has one reservation station. There is a reorder buffer and a register file. Communication between various units use a Common Data Bus (CDB).

Many of the features of this *single issue* processor are left undefined on purpose; you'll have to define some through your answers to the questions that follow.

The stages through which an instruction go through are:

1. Fetch (IF in the figure).
2. Decode, issue, and dispatch (ID). In this stage, structural hazards are detected (the flow of instructions stalls if there is a structural hazard), renaming takes place, reservation station(s) are filled etc.
3. Execute: The integer unit takes 2 cycles to execute; the floating-point unit takes 4. The units are pipelined.
4. Commit.



1. (20 points) (This question continues on the next 3 pages)
Associated with the processor is a Branch Prediction Unit. It consists of:

- An untagged Branch Prediction Buffer (BPB) of 1024 2-bit saturating counters.
- A direct-mapped Branch Target Buffer (BTB) of 256 entries.

(a) (1 point)
The BPB is accessed using an index formed by XORing the PC (Program Counter) bits (19+x-1,19) with PC bits (3+x-1,3). What is the value of x?

x = 10 (10 bits needed to access a 1K table)

(b) (3 points)
How many bits are needed for each entry of the BTB. You can assume that the two rightmost bits of the 32-bit PC are always 0.

The target address is 30 bits.

The tag of the entry is 32 -2 - 8 = 22 bits long (subtract 2 because rightmost 2 bits are always 0 and the index is 8 to access 256 entries)

Consider now the flow of a branch instruction through the pipeline.

(c) (3 points)
What actions related to branch prediction occur during the IF stage?

The BPB is accessed through the above 10 bit index and the BTB is accessed through an 8 bit index.

If the BPB returns NT, we just pass the prediction along to the next stage.

If the BPB returns T and there is a miss in the BTB, we just pass the prediction along to the next stage.

If the BPB returns T and there is a hit in the BTB, we set the PC to the target address found in the corresponding entry in the BTB. We pass the info (T, target address) to the next stage(s) so they can be verified later.

(d) (2 points)
What actions related to branch prediction/execution occur during the ID stage?

In all cases, the instruction is decoded, the target address is computed, and an entry is reserved in the reorder buffer.

If the instruction is not a branch, nothing else related to branch prediction/execution has to be done.

If the instruction is a branch, we save the current register renaming map. We pass the (prediction, target address) to the Integer Execution unit control. If the branch was predicted NT that's all what's needed.

If the branch was predicted T and there was a hit in the BTB, nothing else needs to be done.

If the branch was predicted T and there was a miss in the BTB, the instruction at the computed branch address is fetched. The instruction currently in the IF stage is squashed.

In case of a predicted taken branch and hit in the BTB, PC modification has been done in IF and although verification of the target address, i.e., comparison between the predicted and the computed ones, could be done at this stage we delay it until until the next stage.

(e) (3 points)
What actions related to branch prediction/execution occur at the end of the EX stage?

*The condition is computed. Then there are 4 cases, the cross-product of (correct,incorrect) * (NT,T). In all cases the 2-bit pointer corresponding to this branch will be updated according to the outcome. Also, the outcome of the prediction will be passed to the reorder buffer entry corresponding to this branch.*

Case 1: Correct prediction for NT. Nothing more to do.

Case 2: Correct prediction for T. Now we need to check if the computed and predicted addresses are the same. If not, consider that you have an incorrect prediction for the reorder buffer sake and fetch the instruction at the corrected address that you also enter in the BTB. If yes (or if there had been a BTB miss) (re)enter the adddress in the BTB (it might write the same thing that was already there but it does not cost you anything to do it).

Case 3: Incorrect prediction for NT. Fetch the instruction at the correct address. (Re)enter it in the BTB.

Case 4: Incorrect prediction for T. Fetch the instruction at the correct address given by the old PC that you have carried along all this time.

(f) (2 points)
What actions related to branch execution occur during the commit stage?

If the branch was predicted correctly, nothing has to be done. The branch will be retired (i.e., its position in the reorder biuffer will be freed) when it becomes the head of the reorder buffer.

If the branch was predicted incorrectly, all instructions in the reorder buffer between the branch and the tail of the buffer must be squashed or nullified. There are several possible implementations. For example, you can free all these reorder buffer entries and the reservation stations that hold tags corresponding to these entries and nullify the results of the functional units whose results should go in these entries. You also need to restore the register renaming map.

(g) (4 points)
The BPB is now replaced by a 2-level branch predictor. The global history is recorded by 2 shift registers of 10 bits each. The Pattern History Table consists of 1024 2-bit saturating counters. The branch predictor is thus an SAg(10,2).

How is the direction of a branch predicted?
When and how are the history registers being updated?

The PHT contains the T/NT prediction. At IF time, it is accessed using 1 of the 2 shift registers. The selection of the shift register is done by using, for example, bit 30 of the PC (for "odd-PC" and "even-PC" branches).

Since we have a single-issue machine and only a small chance of encountering a second branch before we know the outcome of the first (at the end of the EX stage), we update the shift register used in the IF stage for this branch at the end of the EX stage. We shift left and record a 1 if the branch was taken, a 0 otherwise in the least significant bit. The counters in the PHT will be also updated at that time

(h) (2 points)
What is needed to transform the 2-level predictor into a *gshare* predictor?

Instead of using the shift register to access the PHT, we now XOR the shift register with 10 bits of the PC

2. (20 points) (This question continues on the next 2 pages)

The processor shown on page 2 has a MIPS-like (or DLX-like) ISA and is the subject of this question.

(a) (4 points)
Give examples of sequences of instructions (instructions can be written in the form $R1 \;<-\; R2 + R3$) exhibiting respectively RAW, WAW, and WAR hazards. Which of these hazards is independent of the underlying architecture?

In the examples below, register R1 is the one on which there is a dependency.

```
     RAW               WAR                 WAW
R1 <- R2 + R3      R2 <- R1 * R3      R1 <- R2/R3
R4 <- R1 + R5      R1 <- R4 + R5      R1 <- R4 + R5
```

RAW hazards happen regardless of the underlying architecture. WAW and WAR can be eliminated with schemes such as register renaming

(b) (2 points)
The instruction stream stalls in the presence of structural hazards. Indicate two sources of structural hazards.

Structural hazards detected at the ID stage and that stall the instruction stream are:

- *Full reorder buffer*
- *The unit to which an instruction was to be issued has a full reservation station*

Conflict on the CDB is not really a structural hazard. It does not prevent the instruction stream to progress: one result will be broadcast and if none of the above 2 conditions exists, new instructions will be fetched.

(c) (3 points)
Which units in the Figure should broadcast their results on the CDB and which should listen to that broadcast?

The functional units, integer and floating-point, should broadcast their results on the CDB.

The broadcast should be listen to by the reservation stations and the reorder buffer.

(d) (8 points)
There are several possibilities for implementing register renaming in this processor. Describe one particular implementation. Be sure to specify what happens at the various stages, how you allocate/deallocate resources etc.

One possible implementation is to use the reorder buffer (ROB) to be an extension of the logical file as well as a keeper of the "in-order" list of instructions. Assume a 3 operand instruction (for branches, see Question 1 and there are variations for load/store).

At decode/issue time, the following actions take place:

- *Grab an entry at the tail of ROB, say ROBx (if none, we have a stall).*
- *Rename the result register with ROBx and enter this mapping in the register renaming table*
- *Get the names (from the registe map) or values of source registers.*
- *Issue to a reservation station (assuming one is free, otherwise stall) the values or names of the source registers as well as the destination ROBx of this instruction.*

At execute time, wait till both operands have values. Then you can start execution.

At end of execute stage, broadcast (result, ROBx) on the CDB. The corresponding entry in the ROB, i.e., ROBx, as well as reservation stations that have an operand tagged with ROBx will grab the broadcast value.

In the commit stage, nothing happens for this instruction until ROBx is at the head of the buffer and it has grabbed its result in a previous broadcast. When these conditions are fulfilled, the result is sent to the correct logical register (whose name is gotten from the map). The head of the buffer is moved to the next entry, thus freeing ROBx.

(e) (3 points)
In the processor shown in the figure, throughput could be enhanced if completion out-of-order was allowed. Why is it not a good idea to allow out-of-order completion?

Out-of-order completion presents problems for predicted branches, precise exceptions, and interrupts.

For example, if an instruction were allowed to commit its result to a logical register and it was in program order after a mispredicted branch, one would have to restore the old value of the result register. The same type of problem would arise if we wanted to have precise exceptions and restartable interrupts.