CS/ECE 752: Advanced Computer Architecture I
Fall 2024

Professor Matthew D. Sinclair

Midterm Exam I SOLUTION
Friday, October 11th, 2024
Weight: 15%

CLOSED BOOK
ONE SHEET OF NOTES (TWO-SIDED) ALLOWED

NAME: ______________________________

DO NOT OPEN EXAM UNTIL TOLD TO DO SO!

Read over the entire exam before beginning. You should verify that your exam includes all of the problems listed in the table below as well as two spare pages for scratch work. Budget your time according to the weight of each question and your ability to answer them. I have provided a large amount of space for you to write your answers, but if that space is not sufficient, please write on the BACK of the SAME sheet. Otherwise, you should use the back of the sheet or the provided extra pages for scratch work. **WRITE YOUR NAME ON EACH SHEET**. You will turn in your cheat sheet with your exam, so please make sure your name is written on it.

Upon announcement of the end of the exam, stop writing on the exam paper immediately. Pass the exam to aisles to be picked up by the proctor (me) or bring them to the proctor up front. The instructor will announce when to leave the room. Failure to follow instructions may result in forfeiture of your exam and will be handled according to the UW Academic Misconduct policy.

Problem	Possible Points	Points
Problem 1	9	
Problem 2	10	
Problem 3	14	
Problem 4	11	
Problem 5	21	
Total	65	

Problem 1 [9 points]

Part A [4 points]

Assume you have a program *P* where 50% of the program can be run in parallel. Your colleague Bob claims that he can get a speedup of 3X on this program with a new CPU architecture he designed. Use Amdahl's Law to analyze the truthfulness of Bob's claim.

Solution:

In the best case, the parallel section reduces to 0, so all that is left is the 50% sequential part of the program. By Amdahl's Law, we know that the maximum speedup we can get on this program is:

$$Speedup_P = \frac{1}{\left((1-0.5)+\left(\frac{0.5}{\infty}\right)\right)} = \frac{1}{0.5} = 2$$

Thus, it is **not** possible to get the speedup Bob is claiming.

Part B [5 points]

Power consumption is becoming increasingly important to consider when designing modern processors. Which of the power components (e.g., of an equation) is most important? And how does technology scaling affect the main two power consumption components? To receive credit, you must justify your answers.

Solution:

Dynamic power is the most important part and usually dominates in modern circuits. It is proportional to the square of the voltage and frequency. Other components are only linearly affected by voltage (leakage, short-circuit) and/or frequency (short-circuit). As a result, dynamic power usually dominates the other two terms.

As transistors shrink, the power per transistor shrinks because of reduced size and capacitance. However, smaller transistors also mean that more transistors can fit in the same area, which increases power density and total power. Moreover, smaller transistors are usually faster, and can run at higher frequencies, which increases power consumption. Thus, total dynamic power tends to increase as transistors shrink, although on a per-transistor basis it gets better.

Static power is consumed even when transistors are inactive – because aren't truly able to be turned all the way off. It is a growing problem as transistor size shrinks, because leakage grows exponentially as threshold voltage shrinks (which, previously, we would decrease as transistor size decreased to help with dynamic power).

Problem 2 [10 points]

Part A [5 points]

For many years CPU pipelines continued to get deeper (i.e., have more stages) nearly every generation. However, in the past 15 years CPU pipelines have started to decrease compared to early generations. Why did this happen? To receive credit, you must justify your answers.

Solution:

Ultimately, the "goodness" of a processor is determined by IPC * clock frequency.

Processor designers took the increasing transistor counts from Moore's Law and used them to design deeper (more pipeline stages), wider (superscalar) processors. These designs improved performance because they allowed more things to be worked on concurrently – especially with out-of-order processors this helped hide long latency operations like cache misses and floating point operations. Having additional pipeline stages is beneficial in that it often allowed us to increase the clock frequency at the same time (because less work had to be done in any given pipeline stage). Thus, superscalar designs improved IPC and deeper pipelining improved clock frequency.

However, these designs ultimately hit the point of diminishing returns due to the limits of ILP. Specifically, since branches typically make up 20-25% of dynamic instructions, if we mispredict a branch the penalty for doing so now becomes much larger – because there are now more stages between fetch and whenever the branch is resolved (and with superscalar those stages typically had multiple instructions in them). For example, with a 4-wide superscalar processor, on average there would be a branch we'd need to predict every single cycle (and in an 8-wide design, on average there would be 2). Although branch predictors like TAGE and Perceptron have helped overcome this challenge to some extent, the penalty (flushing all mispredicted instructions) is still really high and in practice this hurt performance. Thus, companies started pulling back on pipeline depth (from 20-24 stages (Intel Nehalem, 2008) to 14-19 stages in modern processors like Intel Cooper Lake and its successors) – essentially this was the point where the CPI losses (from flushing misspeculated instructions) offset the clock gains from having a deeper pipeline and thus being able to increase clock frequency.

There are also secondary impacts of increasing pipeline depth that played a role. Specifically, increasing the pipeline depth introduces more stages between reading and writing registers. This means that there is potentially more pressure on the register file since registers will be "alive" longer. Moreover, either requires deeper, more complex full bypassing support or the partial bypassing support may have to add additional stalls. Additionally, a load-to-use hazard may become longer (more stages) in deeper pipelines – potentially hurting performance if independent work does not exist in the processor. The end of Dennard's scaling also meant that power budgets went up and we weren't able to scale frequency as easy anymore – but the branch prediction/mispeculation issues mentioned above were more prominent.

Part B [5 points]

The Mahlke paper we reviewed suggests that ISAs should only support predication. However, why do most ISAs either support both conditional branches and predicated execution instead of only supporting predication as Mahlke proposed? To receive credit, you must justify your answers.

Solution:

As we discussed in class, there are a number of reasons why most processors have not adopted only predication.

First, because prediction by definition always involves converting some instructions (the ones on the path where the predicate evaluates to false) to NOPs, it arguably has more work to do that branch prediction (conditional branches) in some cases. In particular, in the situation where branch prediction either a) correctly predicts the branch or b) the branch has a large number of instructions, branch prediction should result in less work. In the case of b, this is likely true even if the branch prediction is wrong, although there will be an inflection point. Conversely, predication does better in situations where there is/are relatively few instructions in the branches (e.g., if/else) since if branch prediction is wrong, the overhead will be relatively higher and predication will be converting a small number of instructions to NOPs. Moreover, predication is likely also going to do better in situations where the conditional branch is very difficult to predict correctly (e.g., the branches TAGE and Perceptron cannot get right).

Another reason is that predication requires an extra register (the predicate register) per instruction. These predicate registers consume some of the register file space (or require a larger register file), potentially having unintended consequences (e.g., more register pressure causing stalls). As mentioned above, since modern branch predictors are very good, this overhead is maybe not worthwhile in all cases.

Predication also requires better support from the compiler (e.g., to convert conditional branches into predicate code, manage hierarchical predication, etc.), and makes backwards compatibility in the face of changes to the underlying hardware very difficult. In the CPU world, compatibility was traditionally seen as very important (e.g., because then users did not need to recompile their code for each new processor), so this was somewhat of a knock against predication. However, not all CPU vendors valued backwards compatibility – ARM in particular does not guarantee backwards compatibility across generations of its ISA. ARM found this valuable for the areas it focuses on (e.g., mobile, IOT) where having lots of hardware support for "old" features could cause area to significantly increase in processors with very little space for this. Likewise, in mobile and IOT devices, there is typically not a huge amount of area for branch predictors – and predictors like TAGE often require lots of area to perform well. Thus, in these spaces predication was often seen as a strong/viable alternative to conditional branch prediction, whereas in the desktop/server world area challenges were less acute and backwards compatibility more valuable – meaning supporting only predication was less useful.

Collectively, these reasons highlight why most processors opt for either only conditional branches or both conditional branches and predicated execution – by supporting both, the processor has a solution that works well in all cases (e.g., pick conditional branches and branch prediction in easy to predict branches, pick predication for hard to predict branches). Alternatively, since modern branch predictors like TAGE and Perceptron often have 99% accuracy, the situations where predication is clearly better are few – so supporting only conditional branches may be a better use of resources.

Problem 3 [14 points]

Part A [6 points]

Traditionally CPU vendors have viewed forward (or upward) compatibility as an important feature to maintain in their ISAs. Provide two examples of how companies provide forward compatibility and describe the advantages and disadvantages to providing forward compatibility. To receive credit, you must justify your answer.

Solution:

The idea of forward (upward) compatibility is that we want older (previously produced/manufactured) processors to be able to support new programs, which might have new ISA instructions the older processors may not be aware of. Typically, this was seen as an important feature to provide because programmers would not need to recompile their program to run on the older hardware – it just "worked" transparently to the programmer without them needing to change anything (although the performance usually would be lower than on more modern processors). This was hugely important and essentially created a form of brand loyalty for processor families – companies typically were not willing to jump to a competitor's design if it only provided small performance benefits **and** required recompiling all of their code to get working on that new line of processors. This was especially true when Moore's Law was providing 2X performance every ~24 months – the company could just wait for the next generation design from their current vendor of choice and performance would "automatically" improve.

However, the downside to forward compatibility is that it requires careful forethought, and often is inefficient/slow (discussed more in next paragraph). For example, reserving sets of trap and nop opcodes, overloading traps (e.g., via firmware patches), or adding ISA hints may be inefficient, and also potentially could be expanded many times over the years to maintain compatibility – increasing bloat. In particular, mobile/IOT designs do not have significant area to devote to implementing such measures. Thus, companies like ARM that focus in these spaces often do not provide forward compatibility.

To provide forward compatibility, companies leveraged multiple options, including emulating support for new instructions in low-level support, allowing software (e.g., compilers) to detect support for new instructions, or redefining reserved opcodes. For example, if a processor detects a new instruction it does not have support for, it can trap to the operating system and have it run. Or, it could have software/the compiler transform the code using the instruction added with a newer generation processor into an equivalent form of code that performs the same action, but with more steps (e.g., CUDA converts TensorCore operations on older GPUs into a sequence of instructions to implement equivalent functionality like dot products). Finally, by reserving opcodes that were not defined at the time the older processor was released, the older processor can either trap to the OS to implement the new instructions that use them or the company can release firmware patches that specify what to do for these new instructions. Sidenote: reserving opcodes only works for a while until your ISA runs out of reserved opcodes (especially in RISC ISAs).

Part B [8 points]

Branch predictors use information about the past branch behavior to predict future behavior. Some branch predictors use *local* information, while others use *global* information. Explain **how** and **why** hybrid (aka tournament) branch predictors utilize this information to improve branch prediction accuracy. To receive credit, you must justify your answer.

Solution:

The key idea behind tournament branch predictors is that neither local nor global information (in isolation) is sufficient to accurately predict the behavior of some branches in our program. Thus, tournament predictors utilize both and have them compete against one another. To do this, they implement an additional level of logic (then "Chooser") which determines for a given branch whether the tournament predictor believes the local (e.g., simple BHT) or global (e.g., correlated) predictor is the right one based on a threshold (i.e., in the past for this branch was local or global right more often)? Thus, in general it is able to outperform either local or global branch predictors.

Local and global information:

Local branch predictors generally only use information about this specific branch (i.e., the PC or a subset of the PC) to determine which direction the branch should take. This tends to work well for branches in code like loops – i.e., highly biased branches that are taken (or not taken) much more often than the alternative (e.g., in a loop the branch is not taken N-1 times and then taken 1 time). Thus, the PC is used to index into the branch history table, where a bit(s) store the prediction for this PC's branch based on the recent history. There is the potential for aliasing in this approach when we use a subset of the PC bits, if multiple branches have the same bits for the subset of the PC.

Global branch predictors can be implemented in several ways. But, in general, they use a branch history register (BHR) that keeps track of the last m global (dynamic) branches – i.e., not just for this specific branch. This tends to work well for branches that are correlated – e.g., if two separate branches are taken and a third branch is only taken if both of the prior ones are taken. Each of those branches has a bit in the BHR that represents whether that branch was taken or not, and these m-bits are then used to index into the pattern table (which, like the local predictors, has a prediction of whether we should take this branch or not). Depending on implementation, global branch predictors may also have multiple pattern tables indexed by PC, and multiple global history registers indexed by PC (or part of it). The one we discussed in HW3 (and in class) though has a single BHR and multiple pattern tables indexed by PC (or part of it).

Thus, compared to local predictors, the global predictor requires bits both for the global BHR (and there may be multiple BHR in some designs, indexed by PC) and the pattern table(s), which use the PC if there are multiple pattern table(s) (as discussed above). Some predictors also hash the BHR with the PC (e.g., gshare). In comparison, the local approach only uses the PC to index its table(s). Finally, both local and global branch predictors use saturating counters (e.g., 2-bit saturating counter from Smith's paper) to store the recent history.

Problem 4 [11 points]

Part A [8 points]

In class we frequently discussed superscalar, out-of-order processors. Is it possible to have superscalar, in-order processors? If so, what limitations would a superscalar, in-order processor have relative to a superscalar, out-of-order processor? If not, what prevents this from being possible? To receive credit, you must justify your answer.

Solution:

Yes, it is possible to have in-order, superscalar processors. For example, we could have a 4-wide processor that executes instructions in-order – in this situation, we need to consider things like rigid or fluid pipeline design, stalling when an older instruction(s) in our pipeline stage stalls, and stalling within a "group" of 4 instructions when there are dependencies within those instructions. Handling memory accesses also becomes more complicated, because we could potentially have 4 instructions all trying to load to/store from memory at the same time. If two of these instructions are trying to load to and store from the same address in the same cycle, this is especially complicated to ensure the ordering is done right (or stalls are added to prevent this). Ultimately, these sequential dependencies represent a significant limitation relative to out-of-order processors though – OOO processors can identify independent work and execute those instructions while earlier ones may be stalling due to dependencies, whereas in the superscalar, in-order processor we're stuck stalling on those dependencies and thus doing no useful work in that timeframe.

Part B [3 points]

Your colleague Bob suggests that you can improve performance by adding a new widget *W1* that removes all WAW dependencies or widget *W2* that removes all WAR dependencies. **Assume the processor Bob suggests adding *W1* or *W2* to an in-order, 1-wide processor.** If you only have enough time and space in your product design cycle to add *W1* or *W2*, is it more important to add *W1*, *W2*, or is neither worth adding? Why? To receive credit, you must justify your answer.

Solution:

Since we are using an in-order, non-superscalar processor, WAR and WAW hazards are irrelevant. Thus, neither W1 nor W2 is worth adding.

Problem 5 [21 points]

This problem concerns Tomasulo's algorithm (with reservation stations) with a reorder buffer (similar to the P6 design we discussed in class), with the following changes/additions/clarifications.

- Assume only the following functional units (and 1 functional unit of each type):

Functional Unit	Cycles in EX
Integer Add	1
Integer Mul	3
Integer Div	7

- The processor can issue and commit/retire at most one instruction per cycle.
- The CDB supports only one broadcast per cycle.
- Assume you have an unlimited number of reservation stations and reorder buffer entries.
- An instruction waiting for data on the CDB can move to EX in the cycle after the CDB broadcast.
- An instruction waiting to write to the CDB holds its execution unit until it gets the CDB; i.e., it prevents other instructions needing the same functional unit from beginning execution.
- Assume that integer instructions also follow Tomasulo's algorithm (analogous to the floating point instructions) so they can be issued out of order and the result from the integer functional unit is also broadcast on the CDB and forwarded to dependent instructions through the CDB.

Complete the blank entries in the following table using the above specifications. For each instruction, fill in the cycle numbers in each pipeline stage (C stands for complete, S stands for issue, X stands for execute, and R stands for retire) and indicate where its source operands are read from (use RF for register file, ROB for reorder buffer, and CDB for common data bus). The entries for the first instruction and for the issue stage are filled in for you (Op means operand). Entries with dots should be ignored. Assume that at the start of the code snippet below, all registers have up-to-date values.

NOTE: If you make a mistake and the subsequent cell is correct given your mistake, you will not be penalized for cascading errors.

(tables on subsequent pages)

This problem uses the following assembly language convention for *all* instructions (note: # in an immediate means the immediate is in base-10 format):

```
        opcode          dest, src1, [src2]
```

Part A [5 points]

List all **true** and **false** data dependencies from the code snippet in the table in Part B (on the next page).

Solution:

RAW:

- R2: Instruction 1→2
- R2: Instruction 1→3
- R7: Instruction 4→5
- R7: Instruction 4→7
- R5: Instruction 5→6
- R5: Instruction 5→8
- R6: Instruction 6→8

WAW:

- R2: Instruction 1→3
- R5: Instruction 5→8

WAR:

- R6: Instruction 1→8
- R1: Instruction 2
- R5: Instruction 5
- R5: Instruction 5→8

Part B [16 points]

Complete the blank entries in the following table.

Solution:

	S	Op1	Op1 source	Op2	Op2 source	X	C	R
MUL R2, R6, R12	1	R6	RF	R12	RF	2	5	6
ADD R1, R1, R2	2	R1	RF	R2	CDB	6	7	8
DIV R2, R2, R4	3	R2	CDB	R4	RF	6	13	14
ADDI R7, R4, #4	4	R4	RF	…	…	5	6	15
ADD R5, R5, R7	5	R5	RF	R7	CDB	7	8	16

	S	Op1	Op1 source	Op2	Op2 source	X	C	R
MUL R6, R3, R5	6	R3	RF	R5	CDB	9	12	17
ADDI R9, R7, #8	7	R7	ROB	…	…	8	9	18
ADD R5, R5, R6	8	R5	ROB	R6	CDB	13	14	19

(Optional helper tables on next 2 pages)

You may find the following structures useful (but do not have to fill them out):

ROB							
ht	#	Insn	R	V	S	X	C

Reservation Stations								
#	FU	busy	op	T	T1	T2	V1	V2

Map Table	
Register	

CDB	
T	Value