University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

EECS152/252A
Spring 2022

J. Wawrzynek
2/22/22

Midterm Exam 1
Solutions

Name: ____________________

Student ID number: ____________________

You have until 11AM to take the exam.

This is a *closed-book, closed-notes* exam. Also no calculators, phones, pads, or laptops allowed.

Each question is marked with its number of points (one point per expected minute of time). Start by answering the easier questions then move on to the more difficult ones. You can use the backs of the pages to work out your answers. Neatly copy your answer to the allocated places. **Neatness counts.** We will deduct points if we need to work hard to understand your answer.

Write the student ID numbers of the person to your left and to your right on this page. If you are sitting on an aisle, just indicate “aisle” for left or right.

Left student ID number: ____________________

Right student ID number: ____________________

Before you turn in your exam, write your student ID number on all pages.

Tuesday 8th March, 2022 14:21

1. Microprogramming [12 pts]

You are asked to write microcode for a new instruction called `SQUARE`, defined as follows:

`R[rd] = R[rs1] * R[rs1]`

This instruction will be implemented on the microcoded single-bus RISC-V machine that we introduced in Problem Set 1, see below.

Unfortunately, your ALU does not support a multiplication operation, but you can perform the square operation as a simple loop of consecutive additions, as follows:

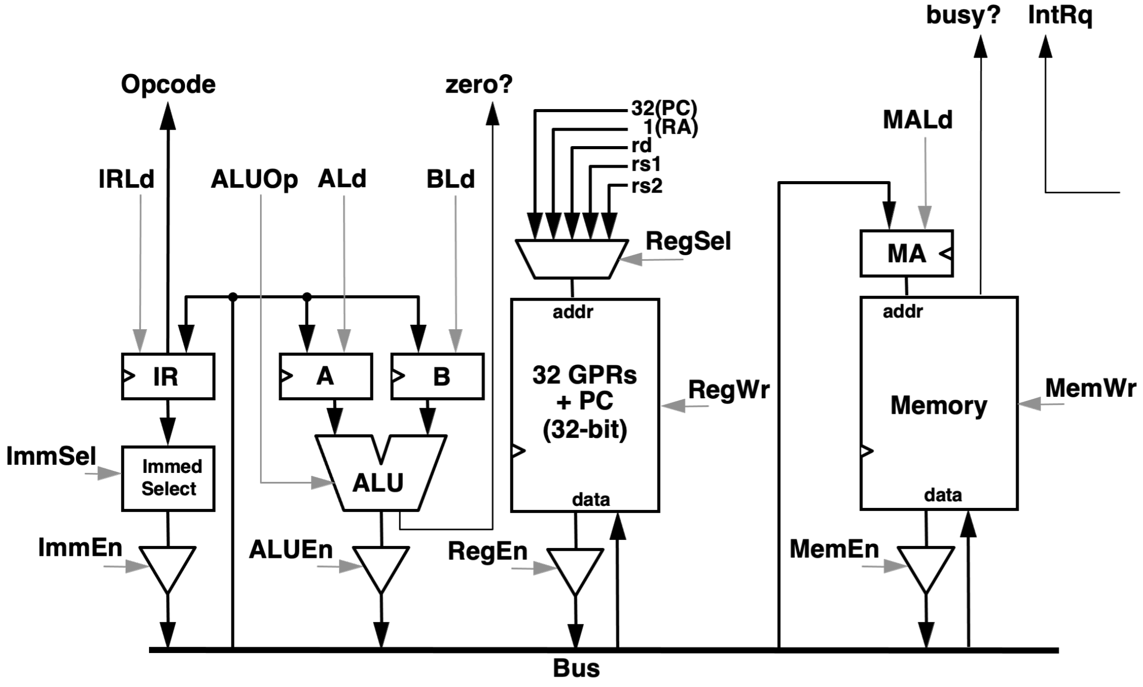
```
// Compute x*x
int result = x;
int count = x - 1;

while (count != 0) {
   result += x;
   count -= 1;
}
```

You are permitted to modify `R[rd]` and `R[rs2]`, but you should not modify any other architectural state. Also, assume that "x" ≥ 1.

Express your answer by filling in the microcode table below. Remember to mark "don't care" entries with a "*". Minimize the total number of micro-instructions you use.

Hint: One approach is to store "x" in `B`, temporary values for "count" in `R[rs2]`, and temporary values for "result" in `R[rd]`. (You are not required to follow this convention).

The microcoded machine is identical to the one we introduced in Problem Set 1. For your reference, we have reproduced the single-bus datapath, as well as information about the ALU and μbranch logic.



Here are our standard possible values for the μBR column:

N	next
J	jump
EZ	branch-if-equal-zero
NZ	branch-if-not-zero
D	dispatch
S	spin on `busy?` signal

Finally, here are the operations the ALU supports:

ALUOp	**ALU Result Output**
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B
SLT	Signed(A) <Signed(B)
SLTU	A <B

Q1 Grading Rubric
Total of 12 pts. Full credit assigned if filled out microcode table is both correct and optimized.

- Microcode Sequence (assigned one of the following)
  - i Correct but unoptimized, with one extra line : -1
  - ii Correct but unoptimized, with more than one extra lines : -1.5
  - iii Microcode does not implement the SQUARE function correctly: -3
- Contains invalid microcode or microcode that will result in incorrect behavior : -3
- Filling out the Table Entries (assigned one of the following)
  - i Correct, but missed entries that could be marked as "don't cares": -0.5
  - ii Errors in less or equal to 2 columns : -1
  - iii Errors in more than 2 columns : -3
- Incomplete answers
  - i Significantly incomplete or incorrect: -9
  - ii Completely incomplete : -12

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	µBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	0	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0
SQUARE:	A, B <- R[rs1]	0	rs1	0	1	1	1	*	0	*	0	0	*	0	N	
	R[rd] <- A	0	rd	1	0	*	0	COPY_A	1	*	0	0	*	0	N	
LOOP:	A <- A-1	0	*	0	0	1	0	DEC_A_1	1	*	0	0	*	0	N	
	if (A==0) goto FETCH0 R[rs2] <- A	0	rs2	1	0	*	0	COPY_A	1	*	0	0	*	0	EZ	FETCH0
	A <- R[rd]	0	rd	0	1	1		*	0	*	0	0	*	0	N	
	R[rd] <- A + B	0	rd	1	0	*	0	ADD	1	*	0	0	*	0	N	
	A <- R[rs2] goto LOOP	0	rs2	0	1	1	0	*	0	*	0	0	*	0	J	LOOP

2. Virtual Memory [19 pts]

(a) **Page Table Calculations** [3 pts]

Consider a computer with 256-byte pages, 16-bit virtual addresses, and 16-bit PTEs (4 bits of which are used for valid and protection). The computer uses *two-level hierarchical* page tables. Assume the machine is byte-addressable.

i. How many virtual pages can be addressed by this computer?

Since the virtual addresses are 16 bits wide and the machine is byte-addressable, the total amount of virtual address space is 2^{16} bytes.
Pages size is 256 bytes, so a total of

$$2^{16}/2^{8} = 2^{8} = 256$$

virtual pages can be addressed.

ii. What is the maximum size of the physical memory that can be supported by this computer?

The size of the physical page number (PPN) is $16 - 4 = 12$ bits.
Each of the PPNs can address a single page in memory which is 256-bytes.
Thus, the maximum size of physical memory that this machine can address is

$$2^{12} \times 256 = 1 \text{ megabyte (MB)}$$

iii. Suppose that a running program is currently using 300 bytes of memory. What is the smallest possible number of PTEs and PTPs that can be valid in the page table(s) of this program?

Since the page size is 256 Bytes, there must be at least two valid pages in memory.
This means at least **2 PTEs** must be valid, and since we have a 2-level page table, at least **1 PTP** must be valid.

Q2(a) Grading Rubric

- +1 for each fully correct answer for the sub-questions

(b) **Memory Trace With Linear Page Table** [10 pts]

Suppose that a computer has 8-bit virtual addresses, 32-byte pages, 4-byte PTEs, a *linear (i.e. single-level)* page table, and a 4-entry TLB with LRU replacement policy.

At the beginning, the TLB is empty and the free page list is (in this order): 0x1, 0x6, 0x3, 0x5, 0xA.

The following virtual addresses are accessed by the programmer (in this order): 0x4A, 0x22, 0x15, 0x20, 0x81, 0x62.

Fill out the following tables:

- Table 1, describing the TLB hits, page fault, and physical addresses of different memory accesses.
- Table 2, describing the linear page table at the end of the memory access trace. (Assume the page table begins at address 0x0). Leave page table entries blank if they are invalid.
- Table 3, describing the TLB at the end of the memory access trace.

We have already filled in the beginnings of the tables for you.

Table 1: Memory Access Trace

Virtual Address	**Page table index**	**TLB hit/miss**	**Page hit/fault**	**Physical address**
0x4A	0x2	miss	hit	0x8A
0x22	0x1	miss	fault	0x22
0x15	0x0	miss	fault	0xD5
0x20	0x1	hit	(hit)	0x20
0x81	0x4	miss	fault	0x61
0x62	0x3	miss	hit	0x402

Table 2: Page Table Entries

Address	**PTE (excluding valid bit)**
0x00	0x6
0x04	0x1
0x08	0x4
0x0c	0x20
0x10	0x3
0x14	
0x18	0x2
0x1C	

Table 3: TLB

VPN	0x2 → 0x3	0x1	0x0	0x4
PPN	0x4 → 0x20	0x1	0x6	0x3

Q2(b) Grading Rubric

- -1 : for partially correct answer on each of the virtual address accesses
- Partial credit is assigned if all table entries are correct except for the translated PA for each VA access

- -2 : for significantly incorrect answer on each of the virtual address accesses

(c) **Two-Level Page Table** [2 pts]

A computer has two-level page tables. All page tables (at either level) hold four 8-byte entries. Pages are 32 bytes long. Virtual addresses are 9 bits wide. Currently, a program is running with only one page allocated. This page has a VPN of 0xA and a PPN of 0x3. The base page table starts at address 0x0, and the second-level page table starts at address 0x20. Given this information:

i. Can we know, from the information above, which entries in the base page table are valid? If so, which entries are valid and what are the PTPs/PPNs stored in those entries?

Yes. The **third/index 2/address 0x10** entry in the first-level page table is valid, with a PTP of **0x1**. We also accept answers of PTP **0x20**.

Note: the page size is 32 Bytes and the PTP/PTEs are 8 Bytes, so the third entry in the first-level page table with base address 0x0 is 0x10.

ii. Can we know, from the information above, which entries in the second-level page table are valid? If so, which entries are valid and what are the PPNs stored in those entries?

Yes. The **third/index 2/address 0x30** entry in the second-level page table is valid, with a PTE of **0x3**.

Q2(c) Grading Rubric

- +1 : For each fully correct sub-question.
- +0.5 : If it contains correct answer on which entry is valid or what is the PTP/PTE stored in that entry
- *Note:* We accepted answers that contain wrong addresses (e.g. 0x8 for the entry in the first-level PT), as long as it contains one of the key words (third entry, index 2).

(d) **Qualitative Short Question** [4 pts]

i. Suppose you increased the length of your virtual addresses. How would this affect the speed of your page table walker? Explain your reasoning. Assume that the page size and the page table size (in bytes) do not change.

Since the page and page table size is fixed, increasing the VA length will lead to more levels in the hierarchical page table.
The hardware page table walker (PTW) will thus be slower in performance due to increased numbers of memory accesses required for each page table walk.

Note: Given a fixed page size and page table size, increading the VA width inevitably leads to more levels in the hierarchical page table. Answers that contain this observation are assigned full credit.

ii. Under what conditions would you prefer a linear page table over a hierarchical page table?

- When the machine has a small virtual address space or has a small number of virtual pages total (say, due to large page sizes).
  Answer must explicitly mention the size of the VA space or the virtual memory's total number of pages/PTE.

- When it is critical to have very fast page table walks, and/or when the total size of page tables in physical memory is not a concern.
- Also, linear page tables are preferred in the (highly unlikely) case that your program uses nearly all the pages in its virtual address space.

iii. What is one advantage and one disadvantage of replacing a virtually-indexed, physically-tagged cache (VIPT) with a virtually-indexed, virtually-tagged cache (VIVT)?

+ In case of a hit, you can avoid reading physical addresses out of the TLB.
(Unfortunately, you may still need to read the TLB for protection bits).
- More likely or harder to handle alaising problems from shared physical pages.
- Cache must be flushed on context switches

Note: VIPT can also access the TLB in parallel with cache, so an answer not specifying that we avoid TLB access itself does not receive full credit.

Q2(d) Grading Rubric

- -1 : Incorrect answer or unclear reasoning for sub-question (i)
- -1 : Incorrect answer or unclear reasoning for sub-question (ii)
- -1 : Incorrectly identified or explained an advantage of VIVT over VIPT (iii)
- -1 : Inorrectly identified or explained a disadvantage of VIVT over VIPT (iii)

3. Caches [16 pts]

(a) **Cache Sub-Blocks** [8 pts]

We will consider a cache optimization known as “sub-blocking” (also called “sectored caches”):

- The number of sets and ways is unchanged.
- Each cache block is broken up into smaller “sub-blocks”.
  - Each sub-block has its own valid bit.
  - On a cache miss, only the cache sub-blocks accessed by the user’s program are loaded in.
    - Other sub-blocks remain in the “invalid” state until they are also loaded in.
- Make sure you understand that “sets” are not “sub-blocks”!

Suppose that we have an 8 KB, two-way set associative cache with 4-word (16-byte) cache lines. The replacement policy is LRU. *Each cache block is broken up into four smaller sub blocks.*

We will evaluate the following two loops:

```
// Loop A
sum = 0;
for (int i = 0; i < 128; i++)
  for (int j = 0; j < 32; j++)
    sum += buf[i*32 + j];

// Loop B
sum = 0;
for (int j = 0; j < 32; j++)
  for (int i = 0; i < 128; i++)
    sum += buf[i * 32 + j];
```

i. What is the number of misses for Loop A and for Loop B with the sectored cache?

For both Loop A and B, all **4096** (32×128) memory accesses will miss.
In a sectored cache, only a single word (sub-block) is loaded at a time
and none of the loaded sub-words are accessed again in both Loop A and B.

ii. What is the number of misses for Loop A and for Loop B if the cache is *not sectored* (i.e. no sub-blocks)?

- Loop A: there are only compulsory misses upon the access to the first word in each cache line. This is once every 4 words and the total number of misses for Loop A is **1024**.
- Loop B, we have all **4096** accesses resulting in a miss. Since memory is accessed in a stride of 32 words, the loop cannot utilize the full cache (only sets 0, 8, 16, . . . , etc. are used) as discussed in Problem Set 2 Question 2. The LRU policy evicts cache lines (in both ways) of the selected sets before the next access to those lines can happen.

iii. Qualitatively explain whether our sectoring scheme has improved the average memory access time of Loop A and Loop B. If so, why? If not, why not?

- Loop A: the sectoring scheme has **not** improved/decreased the AMAT since
  i hit time is unchanged,

ii miss rate has increased by a factor of four,

iii miss penalty has decreased by at most a factor a four (and possibly less than that).

- Loop B: the sectoring scheme has **improved/decreased** the AMAT since

A. hit time is unchanged,

B. miss rate also remains the same (all accesses miss in either case),

C. miss penalty has decreased because only sub-blocks need to be loaded upon misses.

Q3(a) Grading Rubric

- -1 : Incorrect answer or unclear reasoning for each Loop for sub-question (i) (2 pts total)
- -1 : Incorrect answer or unclear reasoning for each Loop for sub-question (ii) (2 pts total)
- -2 : Incorrect answer or unclear reasoning for each Loop for sub-question (iii) (4 pts total)

Note: We tried not to "carry forward" errors from (i) to (ii). For (ii), we accepted any answer where A's miss rate was reduced by a factor of 4, and where B's miss rate was unaffected.

Note: We tried not to "carry forward" errors from (i)/(ii) to (iii). For (iii), we accepted any answer which made sense given the student's responses to (i) and (ii).

(b) **Memory Access Time Tradeoffs** [2 pts]

Suppose that we removed an outer-level cache to free up area on our chip. With this new area, we doubled the size of our L1 cache.

Suppose that this optimization worsened the L1 hit time from 1 cycle to two cycles, and increased the miss penalty from 50 cycles to 100 cycles. *Before* this optimization, the L1 miss rate was 10%.

What does the new miss rate have to be for our new optimization to improve the average L1 cache access time?

Let the new miss rate be $0.1x$.

$$1 + 0.1 \times 50 \geq 2 + (0.1 \times x) \times 100 \implies x \leq 0.4$$

Thus, the new miss rate has to be less or equal to 40% of the original miss rate to improve the average L1 access time.

Q3(b) Grading Rubric

- -1: At least partially correct set up of the AMAT equations (including hit time) but incorrect final answer.
- -2: Significantly incorrect reasoning.

(c) **Short Qualitative Questions** [6 pts]

i. How do write buffers affect write miss penalties in a write-through cache? (A write miss occurs when the program attempts to STORE to an address that is not in the cache). Explain your reasoning.

Write buffers **reduce/improve** write miss penalties, because the cache is no longer blocked on a write miss. This improvement is especially prominent in write-through caches, where the cache line must be written to the next level cache on every write miss.

ii. How do write buffers affect read miss penalties? (A read miss occurs when the program attempts to LOAD from an address that is not in the cache). Explain your reasoning.

This question was ambiguous as to whether the cache was write-through or write-back, so we accepted multiple answers:

In writeback caches, write buffers **reduce/improve** read miss penalties by reducing the cost of evicting dirty lines.

In writethrough caches, read misses do not cause evictions. This means that write buffers can **increase/worsen** read miss penalties, because the write buffer must be flushed or checked for data when a read miss happens. However, if we presume that write-buffers are unlikely to hold the results of a read miss, then we can check the write buffer in parallel with the next-level cache, so that there is **almost no effect** on the read-miss penalty.

Note: We gave partial credit for answers which essentially treated the write buffer as a victim cache.

iii. What advantages does adding a victim cache have over simply extending the size of an existing cache?

Victim caches typically have a higher **associativity** than the L1.

Another advantage of victim caches is that it's very easy to add just a few entries to your total cache capacity with them. Extending a direct-mapped L1 is difficult unless you double the number of it's entries.

Q3(c) Grading Rubric

- (i) Claim that write buffers increase write miss penalties in a write-through cache: -2 points
- (i) Incorrect reasoning for why write buffers decrease write miss penalties: -1 point
- (ii) Incorrect reasoning for how write buffers affect read miss penalties: -2 points
- (iii) Incorrect advantage of a victim cache, or an advantage that would have applied equally well to just extending the cache size, or an advantage that would have applied equally well to just adding another outer-level cache: -2 points

4. Iron Law [6 pts]

(a) Suppose we could add complex instructions (like polynomial instructions or memory-to-memory instructions) to the RISC-V Base ISA. How would this affect processor performance and hardware complexity? Refer to elements of the Iron Law in your answer.

- Instructions/Program is potentially **decreased**. The same subroutine may be expressed with fewer complex instructions.
- CPI is likely **increased**. Complex instructions will need to be cracked down into multiple micro-ops which can result in more stalls in the pipline due to dependencies. If the complex instructions are not craked down into micro-ops, then the pipeline will need to stall to execute variable-latency complex instructions.
- It can also be argued that CPI is **unchanged** or **decreased**, if the program code size is reduced significantly to improve the instruction cache miss rate.
- Hardware complexity and cycle time (time/cycle) is **increased**. Additional logic for decoding the complex instructions and issuing micro-instructions will increase the hardware complexity of the processor front-end, potentially worsening the critical path delay.
- The overall impact on processor performance depends on how much code size reducetion the complex instructions offer and the frequency of these instructions appearing in the workload.

(b) Suppose we extended the RISC-V Base ISA with new 8-bit instructions. (RISC-V Base ISA instructions are 32 bits). Programmers would be permitted to freely mix 32-bit and 8-bit instructions in their code. How would this affect processor performance and hardware complexity? Refer to elements of the Iron Law in your answer.

- Instructions/Program is likely to be **unchanged**. Adding 8-bit instructions to the ISA itself does not change the total number of instructions in a given program.
- CPI is potentially **decreased**. While the number of instructions are unchanged, the size of the program can be reduced if 8-bit instructions are used in place of 32-bit instructions. The reduction in code size improves the instruction cache miss rate, and thus decreases CPI.
- Hardware complexity and cycle time (time/cycle) is **increased**. Additional hardware to support 8-bit instructions is required that can potentially increase cycle time.
- The overall impact on processor performance depends on the balance between reduced CPI and increased hardware complexity.

Q4 Grading Rubric

- For each sub-question, incorrect or unclear answer/reasoning for
  1. Insturctions/Program : -1
  2. CPI : -1
  3. Hardware complexity and cycle time : -1
  4. Overall processor performance : -1

5. PIPELINING [15 pts]

For this question, consider the six-stage pipeline below, designed for a *single-accumulator ISA*:

Fetch	**D**ecode	**A**ddress generation	**M**emory	**E**xecute	**W**riteback

This is similar to the accumulator ISA described in Problem Set 1. Note the accumulator is read during the DECODE stage, and written to in the WRITEBACK stage. In this ISA, the only way to reference memory locations is with immediate offsets from the PC. There is no register-indirect addressing.

There are three types of instructions in this ISA: Arithmetic, Load/Store, and Branch instructions.

Arithmetic instructions load one operand from memory, apply it to the value in the accumulator, and store the result in the accumulator. E.g: ADD 112 (add the value stored in PC+112 to the accumulator).

Load/store instructions load or store the value in the accumulator to/from memory. E.g. Load 112 (load the value stored in PC+112 to the accumulator).

Branch instructions check if the accumulator is zero or not zero, and branch accordingly. Branches are resolved in the EXECUTE stage of the pipeline. In case of a mispredict, we flush the pipeline of earlier instructions. E.g: Branch-if-Zero 112 (branch to PC+112 if the accumulator value is 0).

(a) **Hazards** [8 pts]

i. Which bypass paths are necessary to minimize stalls caused by RAW hazards in this design?

- Bypass paths from the end of Memory (M), Execute (E), and Write-back (W) stages to end of Decode (D).
- Bypass paths from the end of Memory (M), and Execute (E) stages to end of Address-gen (A).
- Bypass paths from the end of Execute (E) stage to end of Memory (M).
- *Note:* The W to D path is optional if we assume that an accumulator read being performed in the same cycle as a write will return the value of the write.

An alternative, but equally correct, solution is as follows:

- Bypass paths from the beginning of Execute (E), Writeback (W) stages to beginning of Address-gen (A).
- Bypass paths from the beginning of Execute (E), and Writeback (W) stages to beginning of Memory (M).
- Bypass paths from the beginning of Writeback (W) stage to beginning of Execute (E).

Note: This is a very aggressive bypassing strategy, and in practice, you may prefer to remove some of these paths to reduce cycle times or hardware complexity.
Note: We will not accept answers which bypass from the end of one stage to the beginning of the combinational logic of another stage, as this unnecessarily increases the clock time.

ii. Are WAR, WAW, or control hazards possible in this design? If so, what hardware interlocks must be added to resolve those hazards?

WAR and WAW hazards are impossible in this design because instructions are completed in-order. The accumulator is updated only at write-back.
Control hazards, however, are possible in case of branch mispredictions. Note that branch mispredictions can occur indepedent of the particular branch prediction mechanism. In case of a mispredict, the pipeline needs to be flushed as stated in the problem.

However, an additional hardware interlock is required to resolve control hazards in this pipeline. Consider:

```
BNZ 200 // to be taken
STORE 100
ADD 4
```

When the branch instruction (`BNZ`) is resolved at the Execute stage, in the same cycle, the ensuing STORE is in the Memory stage. It is possible that the STORE writes the current accumulator value to memory before it is killed by the pipeline flush. Therefore, we need **at least a single cycle hardware interlock** between a branch instruction and a STORE instruction following immediately.
Alternatively, we can insert interlocks after a branch instruction until the branch is resolved at Execute (freeze the pipeline). Both answers are acceptable.

Q5(a) Grading Rubric

- Subquestion (i): total of 4 pts
  - i Missing exactly one of the five required bypass paths: -1
  - ii Missing exactly two or three of the five required bypass paths: -2
  - iii Missing four or more of the five required bypass paths: -3
  - iv Exactly one bypass path that is not helpful (e.g. from Address-gen to Decode): -1
  - v Two or more bypass paths that are not helpful: -2
- Subquestion (ii): total of 4 pts
  - i Claimed that WAR or WAW hazards are possible -1
  - ii Claimed that control hazards are impossible -2
  - iii Did not identify interlock which stalls stores -1

(b) **Pipeline Diagram** [3 pts]

Consider the code below which runs on the pipelined CPU. Remember that all instructions use *PC-relative* offsets.

PC	**Instruction**	**Comment**
0	`Add 100`	// Result of Add is 0
4	`Store 100`	
8	`Load 96`	
12	`Branch-if-zero 8`	// This branch is taken
16	`Nop`	// This Nop is skipped over by the previous Branch-if-zero
20	`Add 80`	
...		
100	`0x0`	// This address stores data, rather than instructions
104	`0x0`	// This address also stores data, rather than instructions

Assume that the pipeline always predicts that branches are *not* taken. In other words, even though the Branch-if-zero above was taken, the CPU initially predicted that it was not taken.

Also, assume that the pipeline above is *fully bypassed.* This means that you can bypass results from the end of any stage to the end of any earlier stage.

Fill in this pipeline diagram of these instructions. The first row has been filled in for you:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`Add 100`	F	D	A	M	E	W														
`Store 100`		F	D	A	A	M	E	W												
`Load 96`			F	D	D	A	M	E	W											
`Branch-if-zero 8`				F	F	D	A	M	E	W										
`Add 80`										F	D	A	M	E	W					

Explanations: there are stalls in the pipeline, despite it being fully-passed.

1. Between the first ADD and STORE. The STORE must wait in the address-generation stage until the updated accumulator value is ready to be stored.
   This induces a single cycle stall.
2. Branch resolution occurs at Execute.
   Therefore, the next correct instruction (ADD 80) can only be fetched at the cycle after the branch instruction completes Execute.

Q5(b) Grading Rubric

- Total of 3 pts. Full credit if correctly filled in the pipeline diagram.
  i Did not delay the pipeline by at least one cycle for the Add → Store dependency: -1
  ii Began the Fetch of the final Add before the M stage of the Branch: -1
  iii Began the Fetch of the final Add before the W stage of the Branch: -0.5
  iv Used fewer bypassing paths than we mentioned were possible: -0.5
  v Some other mistake not mentioned above: -1

(c) **Exceptions** [4 pts]
Assume that branch, load, and store instructions can all cause exceptions. Exceptions may be detected in the Address-Gen, Memory, and Execute stages. Under these conditions, the pipeline above does *not* support precise exceptions, unless new hardware modifications are made.

i. Propose a hardware modification which would make exceptions precise for this pipeline.

```
ADD 100 // exception (e.g. overflow) at E
STORE 200
```

Imprecise exceptions can occur in this pipeline if

1. a previous instruction (e.g. ADD) causes an exception at Execute, but at the same time,
2. the immediately following STORE has already written the current accumulator value into memory.

In the same cycle, the STORE instruction is in Memory stage when the ADD is in Execute. The STORE may have irrevocably written to memory before the exeception is identfied.

An example hardware modification to enable precise exceptions would be to store to memory (perform the write operation) only after the previous instruction has completed execution.
Another idea would be to add write buffers that will delay the actual memory write until the STORE has been committed (passed the commit point).

ii. Suppose that the hardware designer refuses to add support for precise exceptions. What could the programmer do when running on this six-stage pipeline to eliminate the possibility of imprecise exceptions?

Without any hardware modifications, the programmer could simply insert one NOP before every STORE instruction to ensure precise exceptions can be supported.
Note: Simply re-ordering instructions may not be sufficient, because your code isn't guaranteed to have enough non-exception-causing instructions for you to insert in front of every store.

Q5(c) Grading Rubric

- Sub-question (i): total 2 pts.
  - i Correctly identified when imprecise exceptions can occur, but the proposed modification does not enable precise exceptions : -1
  - ii Identified an unnecessarily expensive hardware change (like stalling all instructions in the Decode stage if an earlier instruction exists in the pipeline): -0.25
  - iii Significantly incomplete or incorrect : -2
- Sub-question (ii): total 2 pts.
  - i Correctly identified when imprecise exceptions can occur, but the proposed change to software does not enable precise exceptions : -1
  - ii Identified an unnecessarily expensive software change (like inserting if-statements befor every single instruction that can trigger an exception): -0.25
  - iv Significantly incomplete or incorrect : -2

Note: Although the question doesn't ask you for the exact circumstances under which precise exceptions occur, we still allowed students to recover some partial credit if they correctly identified those circumstances, even if their proposed hardware modifications are incorrect.