CMU 18-447 Introduction to Computer Architecture, Spring 2013

Midterm Exam 2

Date: Wed., 4/17

Instructor: Onur Mutlu
TAs: Justin Meza, Yoongu Kim, Jason Lin

Name:

Legibility & Name (5 Points):	
Problem 1 (90 Points):	
Problem 2 (35 Points):	
Problem 3 (35 Points):	
Problem 4 (50 Points):	
Problem 5 (50 Points):	
Problem 6 (55 Points):	
Total (320 Points):	

Instructions:

1. This is a closed book exam. You are allowed to have two letter-sized cheat sheets.
2. No electronic devices may be used.
3. This exam lasts 1 hour and 50 minutes.
4. Clearly indicate your final answer for each problem.
5. Please show your work when needed.
6. Please write your initials at the top of every page.
7. Please make sure that your answers to all questions (and all supporting work that is required) are contained in the space required.

Tips:

- **Be cognizant of time.** Do not spend to much time on one question.
- **Be concise.** You will be penalized for verbosity.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.
- **Write legibly.** Show your final answer.

1. Potpourri [90 points]

(a) **DRAM [4 points]**

What is the purpose of the precharge command in DRAM?

To prepare the DRAM bank for the next access.

(b) **TLBs [3 points]**

What does a TLB cache?

Page table entries.

(c) **SIMD Processing [12 points]**

Suppose we want to design a SIMD engine that can support a vector length of 16. We have two options: a traditional vector processor and a traditional array processor.

Which one is more costly in terms of chip area (circle one)?

The traditional vector processor | **The traditional array processor** | Neither

Explain:

An array processor requires 16 functional units for an operation whereas a vector processor requires only 1.

Assuming the latency of an addition operation is five cycles in both processors, how long will a `VADD` (vector add) instruction take in each of the processors (assume that the adder can be fully pipelined and is the same for both processors)?

For a vector length of 1:

The traditional vector processor: 5 cycles

The traditional array processor: 5 cycles

For a vector length of 4:

The traditional vector processor: 8 cycles (5 for the first element to complete, 3 for the remaining 3)

The traditional array processor: 5 cycles

For a vector length of 16:

The traditional vector processor:	20 cycles (5 for the first element to complete, 15 for the remaining 15)
The traditional array processor:	5 cycles

(d) **Page Replacement [16 points]**

Suppose we have a demand paging system with a physical memory of 4 GB and a virtual page size of 4 KB. An ambitious system software designer aims to implement true LRU as the page replacement policy. What is the minimum number of bits required to store the necessary information to enable this policy? Show your work.

$$\log_2\left[(4\,\text{GB}/4\,\text{KB})!\right] = \log_2\left[\left(2^{20}\right)!\right]$$

Where should these bits be stored?

Page table

Suppose another engineer wants to use the Victim-NextVictim policy we discussed in class. What is the minimum number of bits required to store the necessary information to enable this policy? Show your work.

We only need to store the virtual page numbers of the victim and the next victim pages: $2 \times \log_2\left[(4\,\text{GB}/4\,\text{KB})\right] = 40$ bits.

Which of the above policies performs better (circle one)?

True LRU Victim-NextVictim [It depends]

Explain briefly your reasoning:

If the access pattern is not LRU friendly, the Victim-NextVictim policy can perform better. If it is LRU friendly, LRU will perform better. For example, a cyclic reference pattern to a working set that is larger than the physical memory size would lead to better performance with Victim-NextVictim.

(e) **Cache Mystery [15 points]**

SlickMarketing, Inc. markets its newly-designed Super-Duper^{TM} processor as having a 128 KB, 2-way cache with 5-cycle access latency. The cache block size is 64 bytes.

A programmer writes a program that repeatedly accesses (in a loop) only the following two unique cache blocks. The loop is executed for billions of iterations.

A, B

What do you expect the average memory access time to be?

5 cycles (as both blocks are expected to hit in the cache)

In the steady state, the programmer measures the average memory access time to be 7.5 cycles.

What can you say about this processor? Please describe everything you can say, concretely, but be concise. Show your work.

The cache is a poor man's associative cache (or pseudo-associative cache), i.e. the associativity is provided in time, and not space.
Blocks A and B map to the same "set": When A is accessed the access takes 5 cycles. When B is accessed, first the location of A is probed (which leads to a miss), then the location of B is probed, which leads to a hit. The access time to B is 10 cycles. Since A and B are accessed in an alternating fashion, the average memory access latency is the average of 5 and 10 cycles.

(f) **Memory Channels [8 points]**

Assume that we have a byte-addressable processor with a 4 KB page size. The virtual address is 64 bits. The physical memory is 4 GB. The processor employs a 3-level cache hierarchy where cache block size is 256 bytes. The processor has 4 on-chip memory controllers, each controlling an independent memory channel, consisting of a 1 GB DIMM. Consecutive cache blocks in physical memory space are located in consecutive channels.

Say that the operating system wants to allocate the physical memory pages of an application to only two DIMMs on memory channels 0 and 1. Describe how the operating system can accomplish this. Be concise and concrete.

Impossible.
Since cache blocks are interleaved across channels by the memory controller, the operating system cannot confine a physical memory page to a single channel.

(g) **Bulk Synchronous Parallel Processing [20 points]**

Assume we have parallelized a program into eight threads. Each thread performs the *exact same computation* on a different part of a very large data set partitioned across the threads. The threads communicate at the end of the parallel portion by synchronizing at a barrier. After that, a single thread runs and aggregates the results generated by each parallel thread. Many parallel computations are structured in a similar way, a model called "Bulk Synchronous Parallel" processing.

You run this program on a multi-core processor with 8 cores, where the only shared resource between the cores is the memory controller and the DRAM system, and measure the following execution time, in seconds, for each thread. There is a single memory controller and it employs the FR-FCFS request scheduling policy.

	T1	T2	T3	T4	T5	T6	T7	T8
Execution time	7	7	10	7	7	7	7	6

What could be three reasons T3 takes longer than others? The first one is given to you.

1. Interference in the memory controller that delays T3 more than others

2. Load imbalance due to task partitioning across threads: T3 inherently has a larger amount of data input set (assigned by the programmer or dynamically)

3. Input set dependence: The data set accessed by T3 causes it to execute for longer (even though all threads have the same amount of input data). For example, due to the nature of its input data values, T3 may suffer from a much higher branch misprediction rate than other threads.

Suppose you would like to tackle the first reason. How would you modify the memory controller to improve the program's performance?

Identify the lagging thread and prioritize it in the memory request scheduler.

Now, say you have another program that works in the same way. You run this program on the same multi-core processor and measure the following execution time, in seconds, for each thread:

	T1	T2	T3	T4	T5	T6	T7	T8
Execution time	10	10	10	10	10	10	10	10

How would you change the scheduling policy in an attempt to improve the program's performance?

Explain concisely.

Rank the threads and perform round-robin shuffling of ranks. Each thread gets the highest priority for N cycles and then gets the lowest priority and makes its way up to the highest rank after $9N$ cycles.

Why can your proposal improve performance? Be concise.

> Reduces interference between threads by ensuring each thread's memory requests are serviced in parallel in different banks.

Can your proposal hurt performance (circle one)?

[YES] NO

Why or why not?

> It can degrade row buffer locality of each thread or of some threads.

(h) **Pointers [12 points]**

Assume we have a byte-addressable computer with a 64-bit virtual address space and a 4KB page size. Assume that the data is laid out such that it is aligned at the data type size boundaries.

The following shows the contents of the valid translations cached in the TLB for the running program:

VPN	PFN
0x0007577123451	0x00011
0x005abcd0123a1	0xC5789

The following shows the contents of a 64-byte L3 cache block starting at address 0xC5789100. The L3 is physically addressed.

```
0005abcd0123a100 [005abcd012300aaa] ffffffffffffffff 00000000C5789100
0000000000011000 00000000C5789808 [005abcd012000000] 000757712340000d
```

(i) Identify and circle the values that are likely to be pointers.

(ii) Assume that a load instruction accesses this cache block. Describe a simple mechanism that detects in hardware which values in the cache block are pointers.

> Compare the most-significant N bits of the *virtual address* of the load with the most significant N bits of each 64-bit-aligned 64-bit value in the cache block. If match, declare the 64-bit value a pointer. If not, predict the 64-bit value is not a pointer.

2. Cache Enigma [35 points]

A processor has a 4-way set-associative L1 cache that can house 4 blocks in total. The access latency to this cache is 1 cycle. The replacement policy is true LRU. The processor is known to not employ any prefetching mechanism.

The processor also has a 16-way set-associative L2 cache that can house 128 blocks in total. The access latency to this cache is 20 cycles.

A programmer writes a test program that in a loop repeatedly accesses only the following data cache blocks (assume billions of iterations are run):

A, B, C, D, E, F

where A, ..., F are different cache block addresses.

In the steady state (i.e., after the loop has executed for a few iterations), the programmer finds out that the average memory access time is 1 cycle.

Then, the programmer writes another program that in a loop repeatedly accesses only the following data cache blocks:

A, B, C, D, E, F, G, H

In the steady state (i.e., after the loop has executed for a few iterations), the programmer finds out that the average memory access time is 20 cycles.

(a) What can you say about this processor? (I.e., what is going on?)

Please describe everything you can say, concretely, but be concise.

> The L1 cache can hold only 4 blocks. Therefore, a victim cache with an access latency of 1 cycle is present. This victim cache is accessed in parallel with the access to L1 cache. Cache blocks evicted from the L1 cache are inserted into the victim cache. Based on the access latencies from above, the victim cache can hold either 2 or 3 cache blocks.

(b) Based on the above information, what do you expect the average memory access time of yet another program that in a loop repeatedly accesses only the following data cache blocks?

A, B, C, D, E

Explain:

> 1 cycle. The 5 cache blocks from above can all fit in the L1 cache and victim cache, and thus all cache block accesses are hits (in the steady state).

(c) Again, based on the above information, what do you expect the average memory access time of yet another program that in a loop repeatedly accesses only the following data cache blocks?

A, B, C, D, E, F, G

Explain:

Either 1 or 20 cycles.
If the victim cache holds 2 blocks, the 7 cache blocks from above cannot all fit in the two caches, resulting in all accesses being misses.
If the victim cache holds 3 blocks, the 7 cache blocks from above can all fit in the two caches, resulting in all accesses being hits.

(d) Finally, again, based on the above information, what do you expect the average memory access time of yet another program that in a loop repeatedly accesses only the following data cache blocks?

A, B, C, D, E, F, G, H, I

Explain:

20 cycles. It is impossible for all 9 cache blocks to fit in the two caches, so all accesses are misses in the steady state.

3. GPUs and SIMD [35 points]

We define the *SIMD utilization* of a program run on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays A, B, and C are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 4 instructions in each thread.) A warp in the GPU consists of 64 threads, and there are 64 SIMD lanes in the GPU.

```
for (i = 0; i < 1024768; i++) {
    if (A[i] > 0) {
       A[i] = A[i] * C[i];
       B[i] = A[i] + B[i];
       C[i] = B[i] + 1;
    }
}
```

(a) How many warps does it take to execute this program?

Warps = (Number of threads) / (Number of threads per warp)
Number of threads = 2^{20} (i.e., one thread per loop iteration).
Number of threads per warp = $64 = 2^6$ (given).
Warps = $2^{20}/2^6 = 2^{14}$

(b) When we measure the SIMD utilization for this program with one input set, we find that it is 67/256. What can you say about arrays A, B, and C? Be precise.

A: 1 in every 64 of A's elements are positive.

B: Nothing.

C: Nothing.

(c) Is it possible for this program to yield a SIMD utilization of 100% (circle one)?

YES NO

If YES, what should be true about arrays A, B, C for the SIMD utilization to be 100%? Be precise.

A: Either:
(1) All of A's elements are greater than 0, or
(2) All of A's elements are less than or equal to 0.

B: Nothing.

C: Nothing.

If NO, explain why not.

(d) Is it possible for this program to yield a SIMD utilization of 25% (circle one)?

YES [NO]

If YES, what should be true about arrays A, B, and C for the SIMD utilization to be 25%? Be precise.

A:

B:

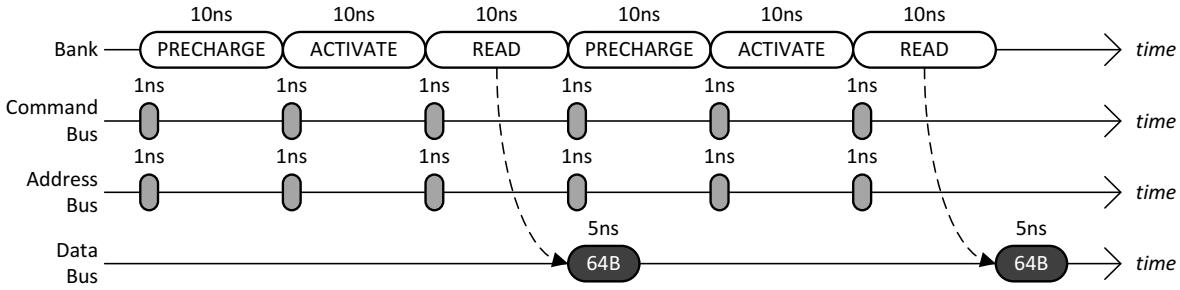
C:

If NO, explain why not.

The smallest SIMD utilization possible is the same as part (b), 67/256, but this is greater than 25%.

4. Memory Scheduling [50 points]

Row-Buffer Conflicts. The following timing diagram shows the operation of a single DRAM channel and a single DRAM bank for two back-to-back reads that conflict in the row-buffer. Immediately after the bank has been busy for 10ns with a `READ`, data starts to be transferred over the data bus for 5ns.



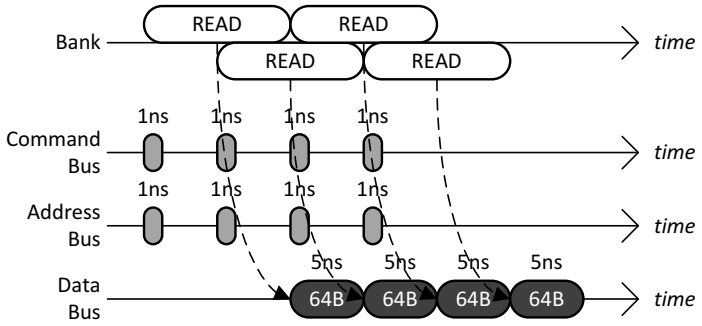
(a) Given a long sequence of back-to-back reads that always conflict in the row-buffer, what is the data throughput of the main memory system? Please state your answer in **gigabytes/second**.

64B/30ns = 32B/15ns = 32GB/15s = 2.13GB/s

(b) To increase the data throughput, the main memory designer is considering adding more DRAM banks to the single DRAM channel. Given a long sequence of back-to-back reads to all banks that always conflict in the row-buffers, what is the minimum number of banks that is required to achieve the maximum data throughput of the main memory system?

30ns/5ns = 6

Row-Buffer Hits. The following timing diagram shows the operation of the single DRAM channel and the single DRAM bank for four back-to-back reads that hit in the row-buffer. It is important to note that row-buffer hits to the same DRAM bank are pipelined: while each `READ` keeps the DRAM bank busy for 10ns, up to at most **half** of this latency (5ns) can be overlapped with another read that hits in the row-buffer. (Note that this is different from Lab 6 where we unrealistically assumed that row-buffer hits are non-pipelined.)



(c) Given a long sequence of back-to-back reads that always hits in the row-buffer, what is the data throughput of the main memory system? Please state your answer in **gigabytes/second**.

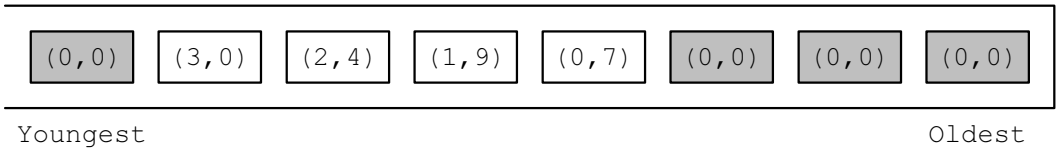
64B/5ns = 64GB/5s = 12.8GB/s

(d) When the maximum data throughput is achieved for a main memory system that has a single DRAM channel and a single DRAM bank, what is the bottleneck that prevents the data throughput from becoming even larger? **Circle** all that apply.

BANK COMMAND BUS ADDRESS BUS DATA BUS

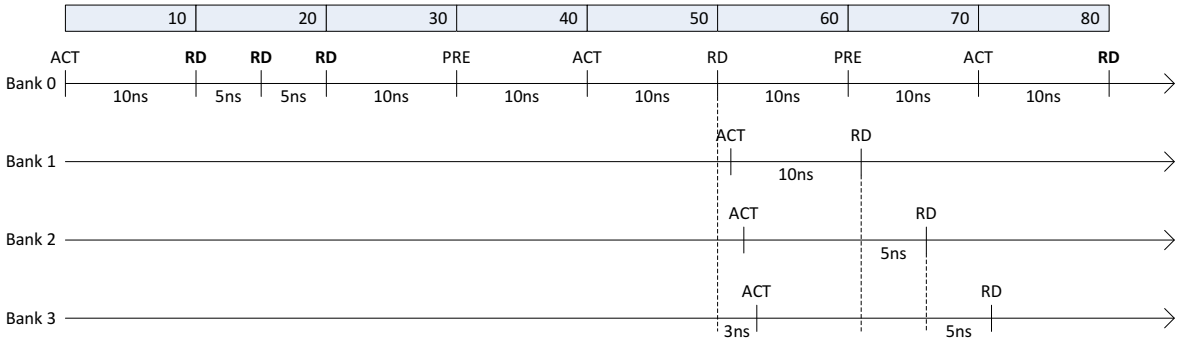
Memory Scheduling Policies. The diagram below shows the memory controller's *request queue* at time 0. The shaded rectangles are read requests generated by thread $T0$, whereas the unshaded rectangles are read requests generated by thread $T1$. Within each rectangle, there is a pair of numbers that denotes the request's (*BankAddress*, *RowAddress*). Assume that the memory system has a **single** DRAM channel and **four** DRAM banks. Further assume the following.

- All the row-buffers are **closed** at time 0.
- Both threads start to stall at time 0 because of memory.
- A thread continues to stall until it receives the data for all of its requests.
- Neither thread generates more requests.



We provide two sets of answers. The correct way to solve the problem is to model contention in the banks as well as in all of the buses (address/command/data). This answer is what immediately follows. However, no student modeled it to this level of detail. Therefore, we decided to give full credit if you modeled contention in only the banks correctly. This answer is given in the answer boxes.

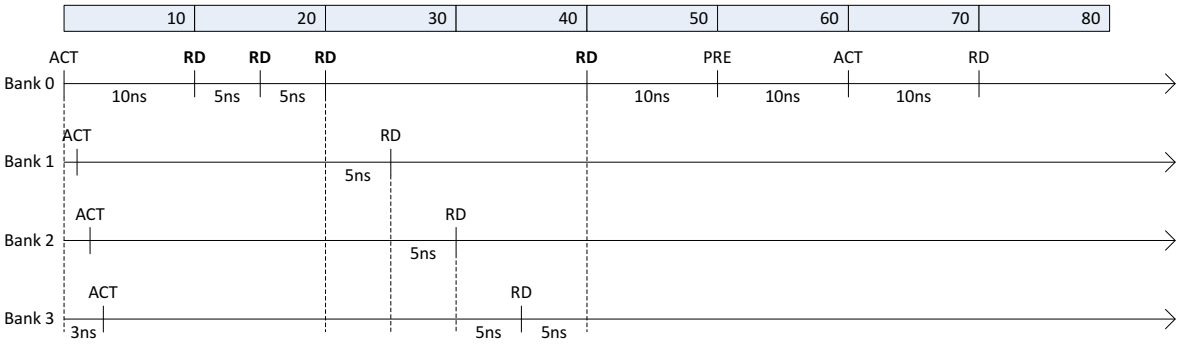
(e) For the `FCFS` scheduling policy, calculate the memory stall time of $T0$ and $T1$.



T0: (10 + 5 + 5 + 10 + 10 + 10) + 10 + 10 + 10 + 10 + 5 = 95ns

T1: (10 + 5 + 5 + 10 + 10 + 10) + 1 + 10 + 5 + 5 + 10 + 5 = 86ns

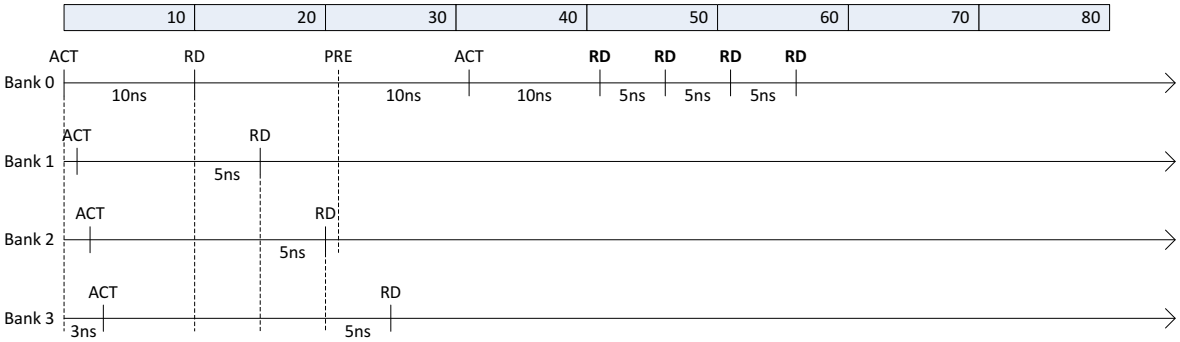
(f) For the `FR-FCFS` scheduling policy, calculate the memory stall time of $T0$ and $T1$.



T0: (10 + 5 + 5 + 5 + 5 + 5 + 5) + 10 + 5 = 55ns

T1: (10 + 5 + 5 + 5 + 5 + 5 + 5) + 10 + 10 + 10 + 10 + 5 = 85ns

(g) For the PAR−BS scheduling policy, calculate the memory stall time of $T0$ and $T1$. Assume that all eight requests are included in the same *batch*.



T0: (10 + 5 + 5) + 1 + 10 + 10 + 5 + 5 + 5 + 10 + 5 = 71ns

T1: (10 + 5 + 5) + 5 + 10 + 5 = 40ns

(e) For the `FCFS` scheduling policy, calculate the memory stall time of $T0$ and $T1$.

$T0$:

Bank 0 is the critical path for both threads.

T0 = Closed + Pipelined-Hit + Pipelined-Hit + Conflict + Conflict + Data
= (ACT+RD)+(RD/2)+(RD/2)+(PRE+ACT+RD)+(PRE+ACT+RD)+DATA
= 20ns + 5ns + 5ns + 30ns + 30ns + 5ns
= 95ns

$T1$:

T1 = Closed + Pipelined-Hit + Pipelined-Hit + Conflict + Data
= (ACT+RD)+(RD/2)+(RD/2)+(PRE+ACT+RD)+DATA
= 20ns + 5ns + 5ns + 30ns + 5ns
= 65ns

(f) For the `FR-FCFS` scheduling policy, calculate the memory stall time of $T0$ and $T1$.

$T0$:

Bank 0 is the critical path for both threads. First, we serve all four shaded requests since they are row-buffer hits. Lastly, we serve the unshaded request.

T0 = Closed + Pipelined-Hit + Pipelined-Hit + Pipelined-Hit + Data
= (ACT+RD)+(RD/2)+(RD/2)+(RD/2)+DATA
= 20ns + 5ns + 5ns + 5ns + 5ns
= 40ns

$T1$:

T1 = Closed + Pipelined-Hit + Pipelined-Hit + Pipelined-Hit + Conflict + Data
= (ACT+RD)+(RD/2)+(RD/2)+(RD/2)+(PRE+ACT+RD)+DATA
= 20ns + 5ns + 5ns + 5ns + 30ns + 5ns
= 70ns

(g) For the PAR-BS scheduling policy, calculate the memory stall time of $T0$ and $T1$. Assume that all eight requests are included in the same *batch*.

$T0$:

First, we serve all four unshaded requests in parallel across the four banks. Then, we serve all four shaded requests in serial.

T0 = Closed + Conflict + Pipelined-Hit + Pipelined-Hit + Pipelined-Hit + Data
= (ACT+RD)+(PRE+ACT+RD)+(RD/2)+(RD/2)+(RD/2)+DATA
= 20ns + 30ns + 5ns + 5ns + 5ns + 5ns
= 70ns

$T1$:

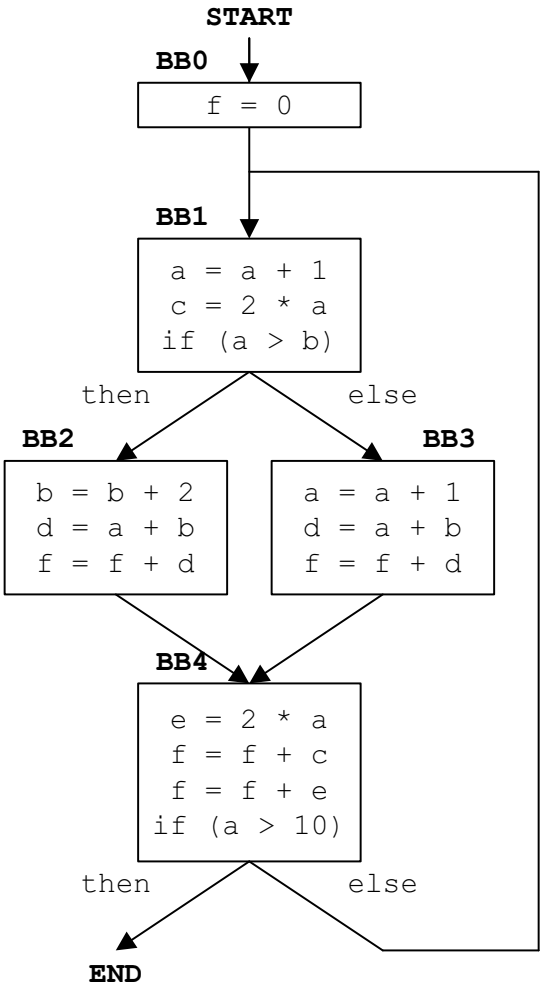
T1 = Closed + Data
= (ACT+RD)+DATA
= 20ns + 5ns
= 25ns

5. Code Optimizations [50 points]

Assume an **in-order**, **non-pipelined** machine. The ISA for this machine consists of only the following five instructions that never generate exceptions. In this syntax, x is a register, whereas y and z are either registers or constants. Note that this machine is able to execute branches and jumps **instantaneously**.

Instruction Name	Syntax	Cycles
Move	$x = y$	1
Addition	$x = y + z$	2
Multiplication	$x = y * z$	5
Branch-If-Greater-Than	`if` $(y > z)$	0
Jump	–	0

Consider the following assembly program that is written in the machine's ISA. For any initial values of registers a and b, the purpose of the program is to compute the final value and store it in register f. After the program terminates, it is considered to have executed correctly if and only if the values stored in registers a, b, and f are correct.



(a) How many cycles does the machine take to execute the assembly program, when a and b are initialized to 0 and 1, respectively?

In this program, the two registers a and b are racing to become larger than the other. `If` ($a > b$) `then` b is incremented so that it can "catch up" to a; `else` a is incremented so that it can "catch up" to b. However, this game of "leap frog" is rigged in favor of a, because a is guaranteed to be incremented by 1 at every iteration of the loop. Therefore, when b is trying to catch up, it can only close the gap by 1, even though b itself is incremented by 2. On the other hand, when a is trying to catch up, it always closes the gap by 2. To give b more turns to catch up, the majority of the loop iterations execute `BB2`, rather than `BB3`.

Initially: (a, b) = (0, 1)

Iteration #1:
After BB1: (1, 1)
Before BB4: (2, 1)

Iteration #2:
After BB1: (3, 1)
Before BB4: (3, 3)

Iteration #3:
After BB1: (4, 3)
Before BB4: (4, 5)

Iteration #4:
After BB1: (5, 5)
Before BB4: (6, 5)

Iteration #5:
After BB1: (7, 5)
Before BB4: (7, 7)

Iteration #6:
After BB1: (8, 7)
Before BB4: (8, 9)

Iteration #7:
After BB1: (9, 9)
Before BB4: (10, 9)

Iteration #8:
After BB1: (11, 9)
Before BB4: (11, 11)
Exit loop.

Each iteration executes 8 instructions: 6 additions and 2 multiplications. Per-iteration execution time: 6*2 + 2*5 = 22 cycles. Initializing f takes 1 cycle.

Answer: 8-iterations * 22-cycles/iteration + 1-cycle = 177 cycles.

You decide to make simple optimizations to the program to reduce its execution time. The optimizations involve only **removing**, **modifying**, and/or **moving** some of the already existing instructions. However, there are two restrictions: you may **not** move instructions out of the loop and you may **not** add completely new instructions.

(b) Show the optimized assembly program.

```
BB0:
f = 0

BB1:
a = a + 1
c = a + a /* CHANGE MULTIPLICATION TO ADDITION */
if (a > b)

BB2:
b = b + 2
~~d = a + b~~ /* MOVE */
~~f = f + d~~ /* MOVE */

BB3:
a = a + 1
~~d = a + b~~ /* MOVE */
~~f = f + d~~ /* MOVE */

BB4:
d = a + b /* MOVE */
f = f + d /* MOVE */
e = a + a /* CHANGE MULTIPLICATION TO ADDITION */
f = f + c
f = f + e
if (a > 10)
```

(c) How many cycles does the machine take to execute the optimized assembly program, when a and b are initialized to 0 and 1?

Each iteration executes 8 instructions: 8 additions.
Per-iteration execution time: 8*2 = 16 cycles.

Answer: 8-iterations * 16-cycles/iteration + 1-cycle = 129 cycles.

After learning about *superblocks*, you decide to optimize the program even further to reduce its execution time. In order to form the superblock(s), assume that you run the program once beforehand when a and b are initialized to 0 and 1. During this profile run, if a branch is biased in either direction by more than **60%**, it is included in the superblock(s). However, there are two restrictions: you may **not** move instructions out of the loop and you may **not** unroll the loop.

(d) Show the superblock-optimized assembly program. **Circle** the superblock(s).

```
BB0:
f = 0

BB1:
a = a + 1
c = a + a
if (a > b)

BB2:
b = b + 2

BB2-TAIL:
d = a + b
f = f + d
~~e = a + a~~ /* REMOVE UNNECESSARY COMPUTATION */
f = f + c
f = f + c /* SUBSTITUTE c for e */
if (a > 10)

BB3:
a = a + 1

BB3-TAIL:
d = a + b
f = f + d
e = a + a
f = f + c
f = f + e
if (a > 10)
```

The superblock consists of `BB1`, `BB2`, and `BB2-TAIL`. Recall from (a) that the branch is biased towards BB2, not BB3.

(e) How many cycles does the machine take to execute the superblock-optimized assembly program, when a and b are initialized to 0 and 1?

5 out of 8 iterations take BB2. Those iterations execute 7 instructions: 7 additions. Execution time: 7*2 = 14 cycles.

3 out of 8 iterations take BB3. Those iterations execute 8 instructions: 8 additions. Execution time: 8*2 = 16 cycles.

Answer: 5-iterations * 14-cycles/iteration + 3-iterations * 16-cycles/iteration + 1 = 119 cycles.

(f) If you had used *traces* to optimize the program instead of superblocks, would the execution time increase, decrease, or stay the same compared to (e)? Choose one and explain briefly why.

Increase. Requires fix-up code.

(g) If you had used *hyperblocks* to optimize the program instead of superblocks, would the execution time increase, decrease, or stay the same compared to (e)? Choose one and explain briefly why.

Increase. Cannot perform the optimizations done with superblocks since we do not know the outcome of the predicated instructions. Also, useless instructions from one of the branch paths are executed due to predication.

6. Running Ahead [55 points]

Consider the following program, running on an in-order processor with no pipelining:

```
LD  R1  ← (R3)     // Load A
ADD R2  ← R4, R6
LD  R9  ← (R5)     // Load B
ADD R4  ← R7, R8
LD  R11 ← (R16)    // Load C
ADD R7  ← R8, R10
LD  R12 ← (R11)    // Load D
ADD R6  ← R8, R15
```

Assume that all registers are initialized and available prior to the beginning of the shown code. Each load takes 1 cycle to execute, and each add takes 2 cycles to execute. Loads A through D are all cache misses. In addition to the 1 cycle taken to execute each load, these cache misses take 6, 9, 12, and 3 cycles, respectively for Loads A through D, to complete. For now, assume that no penalty is incurred when entering or exiting runahead mode.

Note: Please show all your work for partial credit.

(a) For how many cycles does this program run *without* runahead execution?

42 cycles.

Table 1: Execution timeline for Problem 6(b), using runahead execution with no exit penalty and no optimizations

Cycle	Operations		Enter/Exit Runahead Mode	Runahead Mode?	Is Additional Runahead Instruction?
1	Load 1 Issue		enter RA		
2	Add 1			Yes	Yes
3	Add 1			Yes	Yes
4	Load 2 Issue			Yes	Yes
5	Add 2			Yes	Yes
6	Add 2			Yes	Yes
7	Load 1 Finish	Load 3 Issue	exit RA	Yes	Yes
8	Add 1				
9	Add 1				
10	Load 2 Issue, already issued		enter RA		
11	Add 2			Yes	Yes
12	Add 2			Yes	Yes
13	Load 2 Finish	Load 3 Issue, already issued	exit RA	Yes	Yes
14	Add 2				
15	Add 2				
16	Load 3 Issue, already issued		enter RA		
17	Add 3			Yes	Yes
18	Add 3			Yes	Yes
19	Load 3 Finish	Load 4 Issue, dependent on R11 from Load 3. Load 3 finishes at the end of cycle 19, so R11 isn't available now.	exit RA	Yes	Yes
20	Add 3				
21	Add 3				
22	Load 4 Issue		enter RA		
23	Add 4			Yes	Yes
24	Add 4			Yes	Yes
25	Load 4 Finish		exit RA	Yes	
26	Add 4				
27	Add 4				

(b) For how many cycles does this program run *with* runahead execution?

27 cycles. See Table 1.

(c) How many additional instructions are executed in runahead execution mode?

14 instructions. See Table 1.

(d) Next, assume that exiting runahead execution mode incurs a penalty of 3 cycles. In this case, for how many cycles does this program run *with* runahead execution?

4 runahead periods in total. $27 + 4 * 3 = 39$ cycles.

(e) At least how many cycles should the runahead exit penalty be, such that enabling runahead execution decreases performance? Please show your work.

4 cycles. $(27 + 4 \times X > 42)$

(f) Which load instructions cause runahead periods? Circle all that did:

Load A Load B Load C Load D

For each load that caused a runahead period, tell us if the period generated a prefetch request. If it did not, explain why it did not and what type of a period it caused.

Load A:	This period generated prefetches.
Load B:	Short runahead period. / Overlapping runahead periods.
Load C:	Dependent loads. / Short runahead period.
Load D:	No loads to prefetch in this period. (Useless runahead period)

(g) For each load that caused a runahead period that did not result in a prefetch, explain how you would best mitigate the inefficiency of runahead execution.

Load A:	N/A. Prefetch was generated.
Load B:	Do not enter runahead mode if only a few cycles are left before the corresponding cache miss will be filled. Do not enter runahead mode if the subsequent instruction(s) have already been executed during a previous runahead period.
Load C:	Predict the value of cache-miss address loads.
Load D:	Predict if a period will generate useful cache misses, and do not execute that period if it's deemed useless.

(h) If all useless runahead periods were eliminated, how many additional instructions would be executed in runahead mode?

Since only the first runahead period generated prefetches, only that runahead period will be executed. According to Table 1, 6 instructions are executed in the first runahead period.

How does this number compare with your answer from part (c)?

Less.

(i) Assume still that the runahead exit penalty is 3 cycles, as in part (d). If all useless runahead execution periods were eliminated (i.e., runahead execution is made *efficient*), for how many cycles does the program run with runahead execution?

One runahead period, so $27 + 1 * 3 = 30$ cycles.

How does this number compare with your answer from part (d)?

Less.

(j) At least how many cycles should the runahead exit penalty be such that enabling *efficient runahead execution* decreases performance? Please show your work.

16 cycles. $(27 + 1 \times X > 42)$

Stratchpad

Stratchpad

Stratchpad