Introduction to Computer Organization – Fall 2024

Homework 3

Due: @11:55PM, Monday, November 18th

98.11

Name: ________Qiulin Fan_______ Uniqname: ________rynnefan________

1. Submit a pdf of your typed or handwritten homework on Gradescope.

2. Your answers should be neat, clearly marked, and concise. Typed work is recommended, but not required unless otherwise stated. Show all your work where requested, and state any special or non-obvious assumptions you make.

3. You may discuss your solution methods with other students, but the solutions you submit must be your own.

4. **Late Homework Policy:** Submissions turned in by 1:00 am the next day will be accepted but with a 5% penalty. Assignments turned in between 1:00 am and 11:55 pm will get a 30% penalty, and any submissions made after this time will not be accepted.

5. When submitting your answers to Gradescope you need to indicate what page(s) each problem is on to receive credit. The grader may choose not to grade the homework if answer locations are not indicated.

6. After each question (or in some cases question part), we've indicated which lecture number we expect to cover the relevant material. So "**(L7)**" indicates that we expect to cover the material in lecture 7 (but depending on your lecture, it may have been covered later)

7. **The last question is a group question**.
    - You may do it in a homework group of up to two students including yourself (yes, you can do them by yourself if you wish).
    - If you work in a group of two for these questions, list the name of the student you worked with in your assignment. Further, we suggest that you not split these problems up but rather work on the problem as a group.
    - Turn these group questions in as part of your individual submission.
    - **For these questions (and these questions ONLY) you are allowed to copy/paste solutions from the other student in your homework group.**
    - It is an honor code violation if a student is listed as contributing who did not actually participate in working on that problem.

Problem 1: Short answer questions (10 points) (L12-17)

Select True or False for each of the below statements:

1) **[✓ True] / False**: Resolving branches earlier in a pipeline will result in a lower CPI than if resolved in later pipeline stages
2) **True / [✓ False]**: A pipelined datapath requires fewer hardware components than a multi-cycle processor, as it reuses datapath elements
3) **True / [✓ False]**: The single-cycle datapath described in lecture has a faster clock frequency than the multi-cycle datapath.

Select the best choice for each statement:

4) Which has the highest frequency of access? **Disk storage / [✓ register] / cache**
5) Which has the highest access latency? **[✓ Disk storage] / register / cache**
6) Which makes use of both temporal and spatial locality? **Disk storage / register / [✓ cache]**
7) Which is physically located closest to the ALU? **Disk storage / [✓ register] / cache**

Problem 2: Intro to Pipelining (15 points) (L13)

For this problem consider the LC2K code segment below. Notice that there are no hazards.

Address				
0	lw	0	1	6
1	lw	0	2	7
2	add	3	4	5
3	sw	0	1	7
4	add	1	2	3
5	halt			
6	.fill	4		
7	.fill	5		

1) Fill in the timing graph below with the pipeline stage that each instruction is in at the start of each cycle. Recall that the stages are IF, ID, EX, MEM, WB. The first IF has been filled in for you. **[5]**

Instruction \ Cycle #	0	1	2	3	4	5	6	7	8	9
lw 0 1 6	IF	ID	EX	MEM	WB					
lw 0 2 7		IF	ID	EX	MEM	WB				
add 3 4 5			IF	ID	EX	MEM	WB			
sw 0 1 7				IF	ID	EX	MEM	WB		
add 1 2 3					IF	ID	EX	MEM	WB	
halt						IF	ID	EX	MEM	WB

Let the initial values of the registers be as follows:

Register	Value
0	0
1	2
2	-1
3	4
4	7
5	5
6	-2
7	-4

2) For the following 3 parts, fill in the contents of the **pipeline registers** at the point in time between cycle A and B. That is, "Cycle 3-4" means fill in the contents of the pipeline registers **after cycle 3 finishes, and before cycle 4 starts**. If a pipeline register is unused or you don't care about the value, write an "X". *Note that cycle 3-4 doesn't follow cycle 0-1, etc.* **[10]**

Example: **Cycle 0-1**

IF/ID	
Opcode:	lw
PC Plus 1:	1

ID/EX	
Opcode:	noop
PC Plus 1:	X
regA val:	X
regB val:	X
offset:	X

EX/MEM	
Opcode:	noop
aluResult:	X
regB val:	X

MEM/WB	
Opcode:	noop
writeData:	X

i. Cycle 3-4

IF/ID	
Opcode:	sw
PC Plus 1:	4

ID/EX	
Opcode:	add
PC Plus 1:	3
regA val:	4
regB val:	7
offset:	X

(dest : 5)

EX/MEM	
Opcode:	lw
aluResult:	7
regB val:	X

MEM/WB	
Opcode:	lw
writeData:	4

ii. Cycle 4-5

IF/ID	
Opcode:	add
PC Plus 1:	5

ID/EX	
Opcode:	sw
PC Plus 1:	4
regA val:	0
regB val:	4
offset:	7

EX/MEM	
Opcode:	add
aluResult:	11
regB val:	X

MEM/WB	
Opcode:	lw
writeData:	5

-0.5

iii. Cycle 6-7

IF/ID	
Opcode:	X
PC Plus 1:	X

add (0)

ID/EX	
Opcode:	halt
PC Plus 1:	6
regA val:	X
regB val:	X
offset:	X

EX/MEM	
Opcode:	add
aluResult:	9
regB val:	X

MEM/WB	
Opcode:	sw
writeData:	X

Problem 3: Data Hazards (15 points) (L14)

For this problem, reference the following piece of assembly code. All parts of this problem use the 5-stage LC2K pipeline datapath *as presented in lecture*. Assume all other memory locations are initialized to zero.

```
        lw      0       1       one     //lw1
        lw      0       2       num2    //lw2
        lw      1       3       num1    //lw3
        add     3       3       3       //add1
        lw      1       3       3       //lw4
        add     1       1       2       //add2
        nor     3       3       4       //nor1
        halt                            //halt
one     .fill   1
num1    .fill   5
num2    .fill   3
```

1. Complete the table assuming **detect-and-stall** is used to deal with data hazards. If an instruction stalls in a particular stage, just list out that stage for each stalled cycle. You may not need all the columns. **[7]**

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
lw1	IF	ID	EX	MEM	WB												
lw2		IF	ID	EX	MEM	WB											
lw3			IF	ID*	ID	EX	MEM	WB									
add1					IF	ID*	ID*	ID	EX	MEM	WB						
lw4								IF	ID	EX	MEM	WB					
add2									IF	ID	EX	MEM	WB				
nor1										IF	ID*	ID	EX	MEM	WB		
halt												IF	ID	EX	MEM	WB	

2. Complete the table assuming **detect-and-forward** is used to deal with data hazards. You may not need all the columns. **[8]**

−1.14

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
lw1	IF	ID	EX	MEM	WB												
lw2		IF	ID	EX	MEM	WB											
lw3			IF	ID	EX	MEM	WB										
add1				IF	ID*	ID	EX	MEM	WB								
lw4						IF	ID	EX	MEM	WB							
add2							IF	ID	EX	MEM	WB						
nor1								IF	ID	EX	MEM	WB					
halt									IF	ID	EX	MEM	WB				

IF

(stall)

Problem 4: CPI of Pipelines (15 points) (L15)

Say you have an LC2K program with the following characteristics:

- 35% lw
- 20% sw
- 15% add/nor
- 30% beq
- 25% of register-writing instructions are immediately followed by an instruction dependent on it. You may assume that it is true no matter what instructions are involved.
- 70% of branches are not taken

1. What would be the expected CPI of your program using **detect-and-stall** to resolve data hazards and **detect-and-stall** for control hazards? Assume we are using the 5-stage pipeline described in class and the stalling is handled by the pipeline. Clearly show your work. **[7]**

   Reg writing instructions: 35%+15% = 50%
   So data hazard: 0.5 * 2 * 0.25 = 0.25
   Control hazard: 0.3 * 3 =0.9
   Omit the extra 4 cycles at the end.

   CPI = 1+0.25 +0.9 =2.15

2. What would be the expected CPI of your program using **detect-and-forward** to resolve data hazards and **predict-not-taken** for control hazards? Assume we are using the 5-stage pipeline described in class. Clearly show your work. **[8]**

   Detect-and-forward: Among reg-writing instructions with dependent instruction following, only need to consider lw to stall one cycle.

   So Data Hazard: 0.35 * 1 *0.25 =0.088
   Control Hazard: 0.3 * (1-0.7) * 3 = 0.27
   Omit the extra 4 cycles at the end.

   CPI = 1 + 0.088 + 0.27 = 1.358

Problem 5: Cache Overhead (15 points) (L19)

A new company has proposed a number of different cache layouts for their system, and you've been asked to come in and calculate the overhead for each of the different caches. Their system uses a cache with 4 KB of data storage capable of addressing exactly 16 GB of byte-addressable memory (you should assume addresses are only as large as needed to access this amount). Stores will be handled by write-back and allocate-on-write policies. Please be sure to show the work for your calculations.

1. The first design is a fully associative cache with a block size of 16 bytes, how many bytes of overhead would the cache keep in total (including any necessary tag bits, valid bits, dirty bits, or LRU bits)? **[5]**

   Cache size: 4kb = 4 * 2^10 bytes = 2^12 bytes
   number of blocks: Cache size / Block size = 2^12 / 2^4 = 2^8
   Address size: log_2(2^4 * 2^30) = 34 bits
   Block offset = log_2(block size) = log_2(2^4) = 4 bits
   LRU size: log_2(number of blocks) = log_2(2^8) = 8 bits
   So tag size = 34 - 4 = 30 bits

   Thus total bits per block = tag size + 1(valid) + 1(dirty) + LRU size = 40 bits
   Overhead per block: 40/8 = 5 bytes
   Total overhead = 5 * 256 = 1280 bytes

2. Their next design utilizes a direct mapped cache with 64 different cache lines. How many bytes of overhead would the cache keep in total (including any necessary tag bits, valid bits, dirty bits, or LRU bits)? **[5]**

   Cache size = 2^12 bytes
   Number of cache lines = 64 = 2^6
   Block size = Cache size / Number of cache lines = 2^12 / 2^6 = 2^6 bytes
   Block offset = log_2(block size) = 6 bits
   Index size = log_2(Number of cache lines) = 6 bits
   Tag size = 34 - 6 - 6 = 22 bits

   Total overhead per cache line = 22 + 1(dirty) + 1(valid) + 0(LRU) = 24 bits = 3 bytes
   Total overhead = 3 bytes/line * 64 lines = 192 bytes

3. Finally, they've suggested a 2-way set associative cache with 4 different sets. How many bytes of overhead would the cache keep in total (including any necessary tag bits, valid bits, dirty bits, or LRU bits)? **[5]**

   Block size = cache size / (num_sets * associativity) = 2^12 / 8 = 2^9
   Block offset = log_2(Block size) = 9 bits
   Index size = log2(Number of sets) = 2 bits
   LRU bits = log2(Associativity) = 1 bit
   Tag size = 34 - 9 - 2 = 23 bits
   Total overhead per cache line = 23 + 1(valid) + 1(dirty) + 1(LRU) = 26 bits

   num of lines = num_sets * associativity = 8
   So total overhead = 26 * 8 bits = 26 bytes

Problem 6: Cache Performance (15 points) (L19)

It's 1987 and Apple has just released the Macintosh II featuring a Motorola 68030 processor. The 68030 contains a state of the art **128 byte** data-cache and uses **byte-addressable** memory. Having taken 370, you know there's more to cache design than just size, so you write a C program to help you determine the block size, number of blocks per set, and number of sets in the cache. You have a function ACCESS(char *) that loads the memory at the given pointer. Assume all variables other than the data[] array are stored in registers and that the cache has been cleared before each call to miss_rate(...). Assume data[0] is at the start of the cache block.

```
#define CACHE_SIZE XX
double miss_rate(int stride) {
      char data[CACHE_SIZE * 2]; // note array is 2*cache size

      for (int i = 0; i < stride; i++){
            for (int j = i; j < (CACHE_SIZE * 2); j += stride) {
                  ACCESS(data + j)
            }
      }
}
```

1. In order to test your code, you first run it on a **256 B fully-associative cache** with **32 B blocks**. You set the CACHE_SIZE macro to 256. Fill in the miss rate for each stride in the following table. Only answers in the table will be graded. Fractions are OK. **[6]**

Stride	Miss rate
1	1/32
2	1/16
4	1/8
8	1/4
16	1/2
32	1
64	1/32
128	1/32

You now run your program on the Motorola 68030 processor with a **128 byte** data-cache. You set the CACHE_SIZE macro to 128 and collect the following data

Stride	Miss rate
1	1/16
2	1/8
4	1/4
8	1/2
16	1
32	1
64	1/16

Using the above data answer the following questions and justify your answers.

2. What is the block size in bytes? **[3]**

   16 bytes. This is because the missing rate reaches 1 for the first time when stride = 16, hitting a new block for each stride.

3. What is the associativity of the cache? **[3]**

   Answer: 4.

   Fully associative cache is first ruled out because considering at stride = 32, cache would hold all 8 blocks without eviction, and the miss rate could not be 1.

   If it is 1-way(direct-mapped): Consider data[224] that is always mapped to the same block, the next access will always hit, so the miss rate cannot be 1. Contradicts.

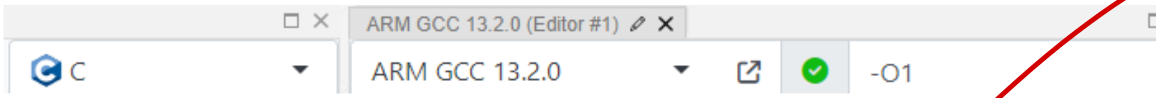
   If it is 2-way: At stride = 64, all data[0], data[64], data[128], data[192] would map to the same set so the miss rate will be 1. Contradicting with the miss rate 1/16.

   Thus 4-way. 4 way causes no contradiction. At stride 32, accesses data[0], …, data[224] all map to the same set. However, the set can only hold 4 blocks, so some blocks will be evicted, leading to a miss rate of 1; and at stride 64, accesses data[0], data[64], data[128], data[192] all map to the same set, but since there are only 4 blocks and the set can hold all 4, no evictions occur after the first access. This explains the miss rate returning to 1/16.

4. How many sets are there in the cache? **[3]**
   Number of sets = cache size / (block size * associativity) = 128 / (16*4) = 2

Problem 7: Looking at a real compiler with a slightly different ISA (Group, 15 points) (L6)

Go to https://godbolt.org/. Select the ARM GCC 13.2.0 complier, the C programming language, and set the compiler options to "–O1" (turning on the optimizer). Note, this is a 32-bit version of ARM and among other differences, the registers are listed with an r rather than an X (so r1 rather than X1). Enter the following code and then answer questions in parts a through d. If you've set things correctly, the website bar should look something like this:



```
#include<stdio.h>
int fib(int n)
{
    int a,b;
    if (n <= 1)
        return n;
    a=fib(n-2);
    b=fib(n-1);
    return(a+b);
}

int main ()
{
    int n = 7;
    printf("%d",fib(n));
    return 0;
}
```

You might find https://developer.arm.com/documentation/ddi0597/2023-09/Base-Instructions?lang=en useful as a reference. Also, be sure to notice just how useful the colorization is.

Collaborator: Zifei Bai

1) Copy the assembly code for the function "fib". Comment each line of assembly for that function explaining what it does. You are going to have to look up some of the instructions online. We are not looking for things that just restate what the assembly instruction does (such as "stores register 2" or "copies register 3 to memory") but instead explains it in context ("puts the argument onto the stack" or "calls the function fib"). **[7]**

```
fib:
        push    {r3, r4, r5, lr} // push args onto stack (callee save)
        mov     r4, r0 // reg be placeholder for reg0, function parameter n
        cmp     r0, #1 // compare n and immediate value #1
        ble      .L1   // if true, link to L1, return func
        subs     r0, r4, #2 // calculate new value of n, pass it to new fib
        bl       fib // call new fib on reg 0, parameter n-2
        mov      r5, r0 // reg 5 is placeholder for new reg0 (n- 2)
        subs     r0, r4, #1  // calculate new value of (n-1), pass to new fib
        bl       fib // call new fib on reg 0,  parameter n-1
        add      r0, r0, r5 // add reg0(n-1) and reg5(n-2) to reg0
.L1:
        pop      {r3, r4, r5, pc} // remove the args from stack
```

2) What will be the maximum stack depth of this program (in bytes)? Include main and briefly justify your answer. **[2]**

4*4*7+2*4 = 120B
The maximum depth for fib function is 7 because fib() is called 7 times in main(). fib() puts 4 registers on the stack every time it is called and every register has 4B, so it is 4*4*7 = 112. We also have 2 registers on the stack, which is 2*4 = 8. 112 + 8 = 120B.

✓

3) Explain how arguments are being passed, where the return value is being placed, and how caller/callee save issues are being resolved. **[2]**

−0.25

From ARM's ABI, args are passed in reg0 for fib(), and the return value is also reg0. The program uses callee save to solve the issue. The function pushes all the registers it would use in the future and removes them after the function returns.

(r3, r4, r5 are callee saved)

✓

4) Write a short C program which computes combinations recursively[1] and give it to the same compiler. Briefly provide the C code, the assembly code, and explain how arguments are passed. **[4]**

```
int combination(int n, int k) {
   int x, y;
```

[1] The recursive definition to calculate n choose k can be defined as follows (for n, k non-negative integers):
C(n,k) = 1 If k = 0 or n = k;
Else C(n,k)=C(n-1, k) + C(n-1, k-1)

```
   if(k == 0 || n == k) return 1;
   return combination(n - 1, k) + combination(n - 1, k - 1);
}
```

```
combination:
        cmp     r1, r0
        it      ne
        cmpne   r1, #0
        bne     .L8
        movs    r0, #1
        bx      lr
.L8:
        push    {r4, r5, r6, lr}
        mov     r6, r1
        subs    r4, r0, #1
        mov     r0, r4
        bl      combination
        mov     r5, r0
        subs    r1, r6, #1
        mov     r0, r4
        bl      combination
        add     r0, r0, r5
        pop     {r4, r5, r6, pc}
```

The arguments of combination(n, k) are passed on reg0(n) and reg1(k).
Reg6 stores the original value of reg1(k) during the recursive runs. On the first recursive call, reg4 contains the original value of reg0(n), and assigns reg0 as reg4 - 1, now reg0 is (n-1). Then, the updated reg0 and reg1 were passed, and the first part, combination(n-1, k), was finished.
After the first part, reg5 holds the return value r0. Now we focus on the second part, combination(n, k-1). Reg0 holds (n-1), reg6 contains the original value of reg1(k), and assigns reg1 as reg6 - 1, now reg1 is (k-1). Then, the updated reg0 and reg1 were passed, and the second part, combination(n-1, k-1), was finished.